



ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΙΓΑΙΟΥ

ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΙΓΑΙΟΥ

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΠΛΗΡΟΦΟΡΙΑΚΩΝ ΚΑΙ
ΕΠΙΚΟΙΝΩΝΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

ΔΙΑΤΡΙΒΗ

για την απόκτηση Διδακτορικού Διπλώματος
του Τμήματος Μηχανικών Πληροφοριακών και
Επικοινωνιακών Συστημάτων

Γεωργίας Φραντζέσκου

**Η ΜΕΘΟΔΟΣ SCAP: ΜΙΑ ΠΡΟΣΕΓΓΙΣΗ
ΕΜΠΕΙΡΙΚΗΣ ΤΕΧΝΟΛΟΓΙΑΣ ΛΟΓΙΣΜΙΚΟΥ
(THE SOURCE CODE AUTHOR PROFILES (SCAP)
METHOD: AN EMPIRICAL SOFTWARE
ENGINEERING APPROACH)**

Συμβουλευτική Επιτροπή:

Εξεταστική Επιτροπή:

Πρόεδρος:

Πρόεδρος:

Στέφανος Γκρίτζαλης
Αναπληρωτής Καθηγητής
Πανεπιστημίου Αιγαίου

Στέφανος Γκρίτζαλης
Αναπληρωτής Καθηγητής
Πανεπιστημίου Αιγαίου

Μέλη:

Μέλη:

Κωνσταντίνος
Λαμπρινουδάκης
Επίκουρος Καθηγητής
Πανεπιστημίου Αιγαίου
Ευστάθιος Σταματάτος
Λέκτορας
Πανεπιστημίου Αιγαίου

Σωκράτης Κάτσικας
Καθηγητής
Πανεπιστημίου Πειραιώς
Σπυρίδων Λυκοθανάσης
Καθηγητής
Πανεπιστημίου Πατρών
Νικήτας Νικητάκος
Καθηγητής
Πανεπιστημίου Αιγαίου

Διομήδης Σπινέλλης
Αναπληρωτής Καθηγητής
Οικονομικού Πανεπιστημίου Αθηνών

Κωνσταντίνος Λαμπρινουδάκης
Επίκουρος Καθηγητής
Πανεπιστημίου Αιγαίου

Ευστάθιος Σταματάτος
Λέκτορας
Πανεπιστημίου Αιγαίου

Table of Contents

ΕΥΧΑΡΙΣΤΙΕΣ.....	3
ACKNOWLEDGEMENTS.....	4
ΠΕΡΙΛΗΨΗ.....	5
EXECUTIVE SUMMARY.....	7
CHAPTER 1. INTRODUCTION.....	8
1.1. STATEMENT OF THE PROBLEM.....	8
1.2. MOTIVATION.....	10
1.3. CHALLENGES.....	11
1.4. CONTRIBUTION OF THE THESIS.....	12
1.5. THESIS OUTLINE.....	14
CHAPTER 2. SURVEY OF RELATED WORK.....	16
2.1. INTRODUCTION - DEFINITIONS.....	16
2.2. AUTHORSHIP ATTRIBUTION METHODS FOR NATURAL LANGUAGES.....	18
2.2.1 <i>Analytical techniques used on natural language authorship identification</i>	22
2.2.2 <i>Keselj's Approach to Natural Language Authorship Identification</i>	24
2.3. AUTHORSHIP ATTRIBUTION METHODS FOR PROGRAMMING LANGUAGES.....	26
2.3.1. <i>Features for Source Code Authorship Identification</i>	30
2.3.2. <i>Features for Executable Code Authorship Identification</i>	35
2.3.3 <i>Analytical techniques used on programming authorship identification</i>	37
2.4. SUMMARY.....	41
CHAPTER 3. THE SCAP APPROACH.....	43
3.1. DESCRIPTION OF THE SCAP METHOD.....	45
3.2. EMPIRICAL STUDY – HYPOTHESES.AND METHOD.....	49
3.2. EVALUATING THE EFFECTIVENESS OF THE SCAP METHOD.....	51
3.2.1. <i>Comparison of SCAP and Keselj's approach on MacDonell Data</i>	51
3.2.2. <i>Performance of SCAP and Keselj's approach on A Different Programming Language</i>	54
3.2.3. <i>Performance of SCAP and Keselj's approach on Comment-free Source Code</i>	55
3.2.4. <i>Performance of SCAP and Keselj's approach on Difficult Student Data</i>	56
3.2.5. <i>Dealing with many authors using the SCAP and Keselj's approach</i>	57
3.2.6. <i>Performance of SCAP and Keselj's approach on Comments</i>	58
3.2.7. <i>The Significance of Training Set Size</i>	59
3.3. IMPLEMENTING SCAP TO LANGUAGES THAT REPRESENT DIFFERENT PROGRAMMING STYLE.....	61
3.3.1. <i>The Common Lisp data set</i>	62
3.3.2. <i>The Java data set</i>	63
3.4. SUMMARY.....	64
CHAPTER 4. SIGNIFICANCE OF HIGH-LEVEL PROGRAMMING FEATURES.....	66
4.1. PROGRAM FEATURES AND SOURCE CODE AUTHORSHIP IDENTIFICATION.....	66
4.1.1. <i>Comments</i>	68
4.1.2. <i>Programming Layout features</i>	71
4.1.3. <i>Identifiers</i>	74

4.1.4. <i>Programming Structure features</i>	77
4.2. DATASETS AND INITIAL EMPIRICAL ANALYSIS – HYPOTHESES AND METHOD.....	78
4.2.1. <i>The Common Lisp data set</i>	80
4.2.1. <i>The Java data set</i>	80
4.3. SIGNIFICANT FEATURES FOR THE COMMON LISP DATA SET.....	81
4.3.1. <i>Contribution of Comments</i>	81
4.3.2. <i>Contribution of Layout</i>	82
4.3.3. <i>Contribution of Identifiers</i>	83
4.4. SIGNIFICANT FEATURES FOR THE JAVA DATA SET.....	87
4.4.1. <i>Contribution of Comments</i>	87
4.4.2 <i>Contribution of Layout</i>	88
4.4.3 <i>Contribution of Identifiers</i>	89
4.5 SUMMARY OF PERFORMANCE.....	93
4.6 SUMMARY.....	95
CHAPTER 5. THE SIGNIFICANCE OF USER DEFINED IDENTIFIERS IN JAVA SOURCE CODE AUTHORSHIP IDENTIFICATION.....	98
5.1. JAVA IDENTIFIERS AND SOURCE CODE AUTHORSHIP IDENTIFICATION.....	100
5.2 EMPIRICAL ANALYSIS - OUR APPROACH.....	101
5.3 DATA SETS ANALYSED.....	102
5.3.1 <i>The Open Source Java data set</i>	104
5.3.2 <i>The Student Java data set</i>	104
5.4 SIGNIFICANCE OF IDENTIFIERS.....	105
5.4.1 <i>Contribution of Simple Identifiers</i>	106
5.4.2 <i>Contribution of Class Identifiers</i>	108
5.4.3 <i>Contribution of Method Identifiers</i>	110
5.4.4 <i>Contribution of all User defined Identifiers</i>	112
5.5 SUMMARY OF PERFORMANCE.....	114
5.6 SUMMARY.....	116
CHAPTER 6 CONCLUSIONS	118
6.1. DESCRIPTION AND COMPARISON OF PREVIOUS STUDIES.....	119
6.2. DEVELOPMENT OF A NEW APPROACH TO SOURCE CODE AUTHORSHIP IDENTIFICATION	120
6.3. THE SIGNIFICANCE OF HIGH-LEVEL PROGRAMMING FEATURES IN SOURCE CODE AUTHORSHIP IDENTIFICATION	121
6.3.1 <i>The significance of user-defined identifiers in Java source code authorship identification</i>	123
6.4. FUTURE WORK	124
REFERENCES.....	126

ΕΥΧΑΡΙΣΤΙΕΣ

Τέσσερα χρόνια πέρασαν για να φτάσω να γράψω αυτή τη σελίδα την οποία ονειρευόμουν από τη στιγμή που ξεκίνησα αυτή την προσπάθεια. Έμαθα πολλά όχι μόνο για την επιστήμη μου αλλά και για μένα, τη ζωή και τους ανθρώπους γύρω μου.

Ευχαριστώ πρώτα απ' όλα το Θεό που με βοήθησε να ολοκληρώσω αυτή τη διδακτορική διατριβή.

Είμαι απερίοριστα ευγνώμων στον επιβλέποντα καθηγητή μου κ. Στέφανο Γκρίτζαλη. Με τη δική του συνεχή καθοδήγηση, την ηθική του στήριξη και ενθάρρυνση όταν τα πράγματα δυσκόλευαν και την άμεση ανταπόκρισή του στην επίλυση των προβλημάτων που αντιμετώπισα, μπόρεσα να ολοκληρώσω αυτή την προσπάθεια. Αισθάνομαι πολύ τυχερή που με εμπιστεύθηκε, αφιερώνοντας μου τον πολύτιμο χρόνο του και μεταδίδοντάς μου γνώση.

Θερμές ευχαριστίες στο Δρα Ευστάθιο Σταματάτο. Χωρίς τη δική του καθοδήγηση, στο συγκεκριμένο ερευνητικό πεδίο, δε θα ήταν δυνατή η ολοκλήρωση της διατριβής αυτής.

Ευχαριστώ πολύ τους Prof. Stephen Mac Donell και την Dr. Carole Chaski που αν και βρίσκονταν χιλιάδες μίλια μακριά με βοήθησαν ουσιαστικά με τα σχόλιά τους, τις παρατηρήσεις τους αλλά και τη διόρθωση παραγράφων των επιστημονικών άρθρων.

Ευχαριστώ το Δρα Στέλιο Γεωργίου για την πολύτιμη βοήθειά του στο στατιστικό μέρος της διατριβής.

Θερμές ευχαριστίες στον Prof. Stephen Mac Donell, στον Επίκ. Καθ. Δημήτρη Φωτάκη και στον Αναπλ. Καθ. Παναγιώτη Αδαμίδα για τα δεδομένα που μου έδωσαν για την εκτέλεση των πειραμάτων.

Ένα μεγάλο ευχαριστώ στο προσωπικό του Κέντρου Εξυπηρέτησης Πληροφορικής Σάμου για την άμεση ανταπόκριση σε οποιοδήποτε αίτημά μου.

Ευχαριστώ την καθηγήτρια Αγγλικής Φιλολογίας κα Αλίκη Μαμουλή για τη βοήθεια και τις διορθώσεις στο κείμενο της διατριβής.

Και τέλος, θα ήθελα να ευχαριστήσω και να τιμήσω το σύζυγό μου Ανδρέα Βούβουνα για τη συμπαράσταση και την κατανόηση που έδειξε αυτά τα τέσσερα χρόνια.

ACKNOWLEDGEMENTS

Four years have passed and I have reached the point where I can now write this page; a page that I have been dreaming since I first started this task. I have learned many things not only as far as my science is concerned but also about myself, life and the people around me.

First of all, I would like to thank God that helped me to complete this doctoral thesis.

I am extremely grateful to my supervisor Prof. Stefanos Gritzalis. This thesis has been successfully completed with his continuous guidance, moral support and encouragement when things were difficult. His direct response to the problems I had faced was valuable. I feel very lucky that he trusted me, dedicating precious time and knowledge.

I would also like to thank Dr. Efsthathios Stamatatos. Without his guidance in the particular research field of this thesis, it would not have been possible for me to complete it.

I am so grateful to Prof. Stephen Mac Donell and Dr. Carole Chaski for their guidance. They have also helped me substantially with their comments, their observations and the correction of our scientific articles, although they were thousands of miles away.

Special thanks to Dr. Stelios Georgiou for his support in the statistical part of my thesis.

I would like to express my appreciation to Prof. Stephen Mac Donell, Prof. Dimitris Fotakis, and Prof. Panagiotis Adamidis for providing the data sets used in the experiments of this dissertation.

I would also like to thank the people in the HelpDesk of Samos for their prompt response to my requests.

Moreover, thanks to the teacher of English literature Alike Mamouli for editing this thesis.

Finally, I would like to thank and honour my husband Andreas Vouvous for the support and understanding during these four years.

Περίληψη

Σήμερα, σε μια ευρεία ποικιλία περιπτώσεων, ο προσδιορισμός του συγγραφέα πηγαίου κώδικα παρουσιάζει εξαιρετικό ενδιαφέρον. Τέτοιες περιπτώσεις μπορούν να περιλαμβάνουν διαφωνίες όσον αφορά το συγγραφέα, απόδειξη για την ταυτότητα του συγγραφέα στο δικαστήριο, επιθέσεις στο διαδίκτυο υπό μορφή ιών (viruses), τρωικών αλόγων (trojan horses), λογικών βομβών, απάτη κλπ. Η ανάλυση με σκοπό την αποκάλυψη του συγγραφέα πηγαίου κώδικα είναι το ερευνητικό πεδίο που προσπαθεί να αναγνωρίσει το συντάκτη ενός προγράμματος, με δεδομένο ένα σύνολο προκαθορισμένων υποψηφίων συντακτών και με τη θεώρηση κάθε προγράμματος ως γλωσσικά και στιλιστικά αναλυτέας οντότητας. Ο καθορισμός του συγγραφέα ενός προγράμματος βασίζεται συνήθως στην ανάλυση δειγμάτων προγραμμάτων του ίδιου.

Μερικά από τα σημαντικότερα ανοικτά ερευνητικά ζητήματα στον τομέα του προσδιορισμού συγγραφέα προγραμμάτων πηγαίου κώδικα είναι:

- Η εξάρτηση από τη γλώσσα προγραμματισμού των μετρικών που χρησιμοποιούνται στην ανάπτυξη μοντέλων ικανών να διαχωρίσουν τα χαρακτηριστικά διαφόρων συγγραφέων προγραμμάτων.
- Η διαδικασία επιλογής αυτών των μετρικών, η οποία δεν είναι προφανής.
- Ο προσδιορισμός των χαρακτηριστικών της γλώσσας προγραμματισμού, τα οποία συμβάλλουν στην αναγνώριση του συγγραφέα ενός προγράμματος, καθώς επίσης και η μέτρηση της συμβολής τους.

Προκειμένου να αντιμετωπιστούν αυτά τα ζητήματα, η παρούσα διατριβή εστιάζει:

- Στην ανάπτυξη μιας νέας προσέγγισης για τον προσδιορισμό και την ταξινόμηση συγγραφέα προγραμμάτων πηγαίου κώδικα, η οποία αποκαλείται «Προσέγγιση SCAP - Προφίλ συγγραφέων πηγαίου κώδικα», η οποία είναι ιδιαίτερα αποτελεσματική και ανεξάρτητη γλώσσας προγραμματισμού, αφού βασίζεται σε χαμηλού επιπέδου πληροφορίες. Τα πειράματα σε διαφορετικές γλώσσες προγραμματισμού, όπως Java, C++ και Common Lisp και ποικίλης δυσκολίας (6 έως 30 υποψήφιοι συγγραφείς) καταδεικνύουν την αποτελεσματικότητα της προτεινόμενης προσέγγισης.
- Στον προσδιορισμό των χαρακτηριστικών υψηλού επιπέδου που συμβάλλουν στην αναγνώριση του συγγραφέα ενός προγράμματος, χρησιμοποιώντας ως εργαλείο τη μέθοδο SCAP. Εξετάζονται

ποικίλα χαρακτηριστικά των γλωσσών Java και Common Lisp, ενώ η σημασία του κάθε χαρακτηριστικού στην αναγνώριση του συγγραφέα ενός προγράμματος μετριέται μέσω μιας ακολουθίας πειραμάτων στην οποία αφαιρούμε ένα χαρακτηριστικό τη φορά. Παρότι αυτή η μελέτη έδειξε ότι οι Java προσδιοριστές (identifiers), οι οποίοι καθορίζονται από τον προγραμματιστή, δεν επηρεάζουν την ακρίβεια ταξινόμησης, σε αυτό το σημείο επιχειρείται μία εξειδικευμένη μελέτη προκειμένου να ελεγχθεί εάν αυτό το συμπέρασμα ισχύει εάν εξετάσουμε κάθε τύπο προσδιοριστή ξεχωριστά.

Executive Summary

Nowadays, in a wide variety of cases, source code authorship identification has become an issue of major concern. Such situations include authorship disputes, proof of authorship in court, cyber attacks in the form of viruses, trojan horses, logic bombs, fraud, and credit card cloning etc. Source code authorship analysis is the particular field that attempts to identify the author of a computer program, given a set of predefined author candidates, by treating each program as a linguistically and stylistically analyzable entity. This is usually based on the analysis of other program samples of undisputed authorship by the same programmer.

Some of the major open research issues in the field of source code authorship identification are:

- Dealing with the programming language-dependence of the software metrics used to develop models that are capable of discriminating among several authors.
- Dealing with the selection process of these software metrics - a non trivial task.
- Identifying the language features that contribute to authorship identification and measuring the significance of their contribution.

In order to address these issues, the focus of this dissertation is on:

- The development of a new approach to source code authorship identification and classification, called the SCAP (Source Code Author Profiles) approach, which is both highly effective and language-independent, since it is based on low level information. Experiments on data sets of different programming-languages (Java, C++ or Common Lisp) and varying difficulty (6 to 30 candidate authors) demonstrate the effectiveness of the proposed approach.
- Identifying the high level features that contribute to source code authorship identification using as a tool the SCAP method. A variety of features are considered for Java and Common Lisp and the importance of each feature in determining authorship is measured through a sequence of experiments in which we remove one feature at a time. At this stage, while this study has indicated that programmer -defined Java identifiers do not influence classification accuracy a separate set of experiments has been performed, in order to check whether this conclusion holds when we examine each type of such identifiers separately.

Chapter 1. Introduction

1.1. Statement of the problem

With the increasingly pervasive nature of software systems, cases arise in which it is important to identify the author of a usually limited piece of programming code. Such situations include cyber attacks in the form of viruses, Trojan horses and logic bombs, fraud and credit card cloning, code authorship disputes, and intellectual property infringement.

But why do we believe it is possible to identify the author of a computer program? 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. Does the same apply to programming? Could we identify programming constructs that a programmer uses all the time? Spafford and Weber (1993) suggested that a field they called software forensics could be used to examine and analyze software in any form, be it source code for any language or executable programs, to identify the author. Spafford and Weber wrote the following of software forensics:

“It 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”

This identification process is also analogous to attempting to find characteristics in humans that can be used later to identify a specific person. Eye and hair colouring, 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 contribute in the identification of a corresponding programmer.

The closest parallel is found in computational linguistics. Authorship analysis in natural language texts, including literary works has been widely debated for many years, and a large body of knowledge has been developed. Authorship analysis on computer software, however, is different and more difficult than in natural language texts.

Several reasons make this problem difficult. Programmers reuse code, programs are developed by teams of programmers, and programs can be altered by code formatters and pretty printers.

Identifying the authorship of malicious or stolen source code in a reliable way has become a common goal for digital investigators. Spafford and Weber (1993) have suggested that it might be feasible to analyze the remnants of software after a computer attack, through means such as viruses, worms or Trojan horses, and identify its author through characteristics of executable code and source code. Zheng et al. (2003) proposed the adoption of an authorship analysis framework in the context of cybercrime investigation to help law enforcement agencies deal with the identity tracing problem.

Researchers addressing the issue of code authorship have tended to adopt a methodology comprising two main steps (Krsul and Spafford, 1995; MacDonell et al. 2001; Ding and Samadzadeh, 2004). The first step is the extraction of apparently relevant software metrics and the second step is using these metrics to develop models that are capable of discriminating between several authors, using a statistical or machine learning algorithm. In

general, the software metrics used are programming language-dependent. Moreover, the metrics selection process is a non trivial task.

Our objective is to provide a language independent methodology to source code authorship attribution. Additionally we aim to provide the features of a piece of code that contribute to correct authorship attribution.

1.2. Motivation

Three basic areas can benefit considerably by our current work:

1. Authorship disputes: The legal community is in need of solid methodologies that can be used to provide empirical evidence to show that a certain piece of source code is written by a particular person.

2. The academic community: It is considered unethical for students to copy programming assignments. While plagiarism detection can show that two programs are similar, 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 programmer who wrote a particular piece of code from a set of several programmers, so as to better evaluate their work and avoid future disputes over the authorship of projects.

1.3. Challenges

The task of identifying the author of a piece of code seems a difficult task at first glance. Convincing arguments can be given about the intractability of this problem. Consider the following examples of potential problems with the identification of authors:

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

Programming characteristics of programmers tend to change and evolve. Education is only one of many factors that have an effect on the evolution of programming styles. Not only do software engineering models impose particular naming conventions, parameter passing methods and commenting styles; they also impose a planning and development strategy. The waterfall model (Ghezzi et al 1991), 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 companies, projects or tools.

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 rarely the case.

The authorship identification process in computer software can be made reliable for a subset of the programmers and programs written in the same language. Programmers that are involved in high security projects or programmers that have been known to break the law are attractive candidates for classification. Patterns of behaviour are all around us. Likewise for

programming, we can ask: which are the programming constructs that a programmer uses all the time? Could we hide the provenance of a piece of code by changing a certain programming feature?

1.4. Contribution of the Thesis

Authorship Identification in natural language texts including literary works has been widely debated for many years and a lot of studies and methodologies have been developed. More recently the widespread use of software systems made software authorship identification an issue of concern.

Although source code is much more formal and restrictive than spoken or written languages, there is still a large degree of flexibility for programmers to develop their own programming styles (Krsul, and Spafford, 1995). Consequently, the task of software code authorship identification is similar to written text authorship identification (Sallis et al, 1996). Thus, the approaches and methodologies used for traditional textual analysis and forensics can be transferred to software analysis (Kilgour et al., 1998).

With this in mind there are two questions addressed in this Thesis?

- Could we find a methodology in source code authorship identification which is more effective and accurate than the existing methodologies?
- Which are the features of the source code that contribute to effective authorship identification?

Based on the two questions above, the contribution of the Thesis could be divided in the following two categories.

- Development of a new methodology for source code identification named the SCAP approach. Low level information was used so as to achieve quantification of the programming style of each author.

- A number of empirical studies have been carried out in order to identify the high level features that contribute to source code authorship identification.

In more detail the contribution of the Thesis based on the above categories is the following:

A review of the studies related to source code authorship identification has been conducted, identifying the methodologies followed so far as well as some of the advantages and weaknesses of these methods. (Frantzeskou and Gritzalis 2004). A new approach to source code identification has been developed named the SCAP (Source Code Author Profile) approach, which is language-independent and highly effective. The SCAP method is an extension of a method applied to text authorship identification (Frantzeskou. et al 2005a, Frantzeskou et al 2007a). A comparison between the two approaches was carried out on two data sets written in Java and C++ (Frantzeskou. et al 2005a, Frantzeskou et al 2005b). The significance of training size has been examined in (Frantzeskou et al 2005a). The forensic significance of the SCAP approach was outlined in (Frantzeskou et al 2007a). Satisfactory results have been obtained by testing the SCAP approach under different circumstances: we used a data set with code written by students during a Java introductory course (Frantzeskou et al 2006a). The data sets we used showed the effectiveness of our approach with a limited number of candidate authors (6 to 8). It has been demonstrated the effectiveness of the proposed method when dealing with dozens of candidate authors (30 candidate authors) (Frantzeskou et al, 2006a). In addition, the role of comments to source code authorship identification has been examined (Frantzeskou et al 2006b). Finally we have performed more detailed experiments to demonstrate that the SCAP approach is language independent with two data sets using two different styles of programming language, Java

which uses objects, and Common Lisp, which uses a functional/imperative programming style (Frantzeskou. et al 2007b).

The second part of our work involved a study in order to assess the impact that the high level programming features (for example comments, identifiers) have on the accuracy of authorship attribution. The results of this study have been demonstrated on two data sets, one written in Java and another in Common Lisp (Frantzeskou et al 2007b). A different study was carried out in order to assess the significance that Java identifier types have on source code authorship attribution using as a tool the SCAP approach (Frantzeskou et al 2007c).

1.5. Thesis Outline

This thesis is structured as follows. Chapter 2 contains a review of past research efforts in the area of natural and programming languages authorship, Chapter 3 describes our approach to source code authorship identification called the SCAP approach. The same section contains an empirical study which demonstrates that our method is both highly effective and language-independent. Chapter 4 begins with a detailed description of the high level features that might influence source code authorship identification. The rest of this chapter details another empirical study in order to examine which high level programming features contribute to authorship identification, and to what degree. This study used programs written in Java and Common Lisp. Chapter 5 describes the types of Java user-defined identifiers that are thought to influence source code authorship identification. It also includes a third empirical study in order to examine which (if any) identifiers contribute to authorship identification and to what degree. This study used two different Java data sets. The conclusions of this dissertation

can be found in chapter 6, in which we summarize the achievements of our study and we propose future work directions.

Chapter 2. Survey of Related Work

2.1. Introduction - Definitions

Although source code is much more grammatically and syntactically restrictive than natural languages, there is still a large degree of flexibility when writing a program (Krsul and Spafford 1995) and the general methodology of authorship attribution applies to texts in both natural and programming languages. Authorship identification methodology for natural or programming languages can be formulated as follows: Given a set of writings of a number of authors, assign a new piece of writing to one of them. The problem can be considered as a statistical hypothesis test or a classification problem. The essence of this classification is identifying a set of features that remain relatively constant for a large number of writings created by the same person. Once a feature set has been chosen, a given writing can be represented by an n -dimensional vector, where n is the total number of features. Given a set of pre-categorized vectors, we can apply many analytical techniques to determine the category of a new vector created based on a new piece of writing. Hence, the features set and the analytical techniques may significantly affect the performance of authorship identification.

In the following sections we review the literature on authorship attribution for both natural and programming languages, based on the perspectives described above and giving at first the concept definitions related to this study.

An *Author* is defined by Webster (Merriam-Webster 1992) as one that writes or composes a literary work," or as one who originates or creates." In the context of software development the author or programmer is someone

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 collection of instructions that describes a task, or set of tasks, to be carried out by a computer. More formally, it can be described as an expression of a *computational method* written in a programming language language (Knuth 1997).

A *programming language* is an artificial language that can be used to control the behavior of a machine, particularly a computer. Programming languages, like human languages, are defined through the use of syntactic and semantic rules, to determine structure and meaning respectively. Programming languages are used to facilitate communication about the task of organizing and manipulating information, and to express algorithms precisely (Abelson and Sussman; 1992 McLennan and Bruce,1987).

Programming style refers to a set of rules or guidelines used when writing the source code for a computer program. It is often claimed that following a particular programming style will help programmers quickly read and understand source code conforming to the style as well as helping to avoid introducing faults (McConnell, S., 1993).

Authorship analysis is defined as the application of the study of linguistic style, usually to written language often used to attribute authorship to anonymous or disputed documents. Correspondingly, Source code authorship analysis is the process of examining the characteristics of a piece of code in order to draw conclusions on its authorship (Abbasi and Chen 2005). More specifically, the problem can be broken down into the following sub-fields.

1. *Author identification.* The aim here is to decide whether some piece of code was written by a certain programmer. This goal is accomplished by comparing this piece of code against other program samples written by that author. This type of application area has a lot of similarities with the corresponding literature where the task is to determine that a piece of text has been written by a certain author.

2. *Author characterisation.* This application area determines some characteristics of the programmer of a piece of code, such as cultural educational background and language familiarity, based on their programming style.

3. *Plagiarism detection.* This field attempts to find similarities among multiple sets of source code files. It is used to detect plagiarism, which can be defined as the use of another person's work without proper acknowledgement.

4. *Author discrimination.* This task is the opposite of the above and involves deciding whether some pieces of code were written by a single author or by some number of authors. An example of this would be showing that a program was probably written by three different authors, without actually identifying the authors in question.

2.2. Authorship Attribution Methods for Natural Languages

The earliest studies into natural language authorship attribution include those by Mendenhall (1887), Yule (1938, 1944) and Zipf (1932). Mendenhall (1887) studied the authorship of Bacon, Marlowe and Shakespeare by comparing word spectra or characteristic curves, which were graphic representations of the arrangement of their word length and the relative frequency of their occurrence. He suggested that if the curves remained constant and were particular to the author, this would be a good method for

authorship discrimination. Zipf (1932) focussed his work on the frequencies of the different words in an author's documents. He determined that there was a logarithmic relationship, which became known as Zipf's Law, between the number of words appearing exactly r times in a text, where ($r = 1; 2; 3 : : :$) and r itself. Yule (1938) initially used sentence length as a method for differentiating authors but concluded that this was not completely reliable. He later created a measure using Zipf's findings based on word frequencies, which has become known as Yule's characteristic K . He found that a word's use is probabilistic and can be approximated with the Poisson distribution.

The Federalist papers are a series of articles written in 1787 and 1788 to persuade the citizens of New York to adopt the Constitution of the United States of America. There are 85 articles in total, with agreement by the authors and historians that 51 were written by Alexander Hamilton and 14 were written by James Madison. Of the remaining articles, five were written by John Jay, three were jointly written by Hamilton and Madison and 12 have disputed authorship between Hamilton and Madison. This authorship attribution problem has been visited numerous times since the original study of Mosteller and Wallace (1964), with a number of different techniques employed. Using four different techniques to compare the texts under examination, the original study compared frequencies of a set of function words selected for their ability to discriminate between two authors. The techniques used by Mosteller and Wallace included a Bayesian analysis, the use of a linear discrimination function, a hand calculated robust Bayesian analysis and a simplified word usage rate study. Mosteller and Wallace came to the conclusion that the twelve disputed papers were written by Madison. Subsequently, many researchers have confirmed the good discriminating capability of function words (Baayen et al. 1996; Burrows 1989; Holmes & Forsyth, 1995; Tweedie & Baayen, 1998). Rooted from linguistic research, part

of speech (POS) and punctuation usage are other important syntactic features which have been applied to authorship research.

Another kind of lexical features used in authorship attribution was the vocabulary richness measures. These features include the number of words that occur once (*hapax legomena*) and twice (*hapax dislegomena*), as well as several statistical measures defined by previous studies (Yule 1944, Holmes 1992).

In other attribution studies, Shakespeare has been compared with Edward de Vere, the Earl of Oxford (Elliott and Valenza, 1991b), John Fletcher (Lowe and Matthews, 1995) and Christopher Marlowe (Merriam, 1996). Elliott and Valenza used incidences of badge words, fluke words, rare words, new words, prefixes, suffixes, contractions and a number of other tests to build a Shakespeare profile for comparison with other authors. Lowe and Matthews used frequencies of five function words and a neural network analyser, while Merriam used some function words and principal component analysis.

As punctuation is not guided by any strict placement rules (e.g., comment placement), punctuation will vary from author to author. Chaski (1997) has shown that punctuation can be useful in discriminating authors. It is well known that punctuation has the potential of being a successful attributor of authorship, but as Chaski (1997) points out, it has only really been successful when combined on its own with an understanding of its syntactic role in a text. Chaski developed software for the purpose of punctuation-edge counting, lexical frequency ranking and part-of-speech tagging and demonstrated (Chaski 2001) that if punctuation were syntactically classified, it had a better performance in authorship attribution than simple punctuation mark counting.

Stamatatos et al. (2001) introduced a fully automatic method to extract syntax-related features and a better performance was achieved compared to

pure lexical-feature-based approaches. As a recently explored feature type, structural features attracted more attention. People have different habits when organizing an article. These habits, such as paragraph length, use of indentation, and use of signature, can be strong: authorial evidence of personal writing style. This is more prominent in online documents, which have less content information but more flexible structures or richer stylistic information. De Ve1 et al. (2001) proposed to use structural layout traits and other features for e-mail authorship identification and achieved high identification performance. In Zheng et al. (2003), approximately 10 content-specific features were introduced in a cybercrime context and the results showed that they were helpful in improving the author-identification accuracy.

Due to the international nature of the Internet, it is of critical importance to study authorship identification in a multilingual context. Writing style features are largely language dependent. For instance, Chinese has no explicit word boundaries. Consequently, the features and feature extraction techniques for Chinese are very different from those for English. Stamatatos et al. (2001) conducted authorship identification on Greek newspaper articles. He proposed a computer-based feature extraction approach using a natural language processing tool. But Greek is to some extent similar to English because they share similar linguistic characteristics, such as the existence of word boundaries. Keselj et al. (2003) conducted experiments on Greek, English and Chinese data to examine the performance of authorship attribution across different languages. They identified unique linguistic characteristics of Chinese and concluded that some character-based features such as n-gram should be used to avoid word-segmentation problems. They also noted that the Chinese vocabulary is much larger than the English vocabulary, which may give rise to sparse data problem. They examined the

n-gram language model on Greek newspaper articles, English documents, and Chinese novels. In all three languages the best accuracy achieved was 90%. But the performance for Chinese writings was not as good as that for English writings. Multiple-language support of authorship technology is an important new research direction in this field in the light of the continuous globalization of Internet applications.

Compression-based classification is a non-standard approach to authorship attribution and has been used by many researchers (Khmelev, and Teahan, 2003; Benedetto et al, 2002; Frank et al 2000). Compression programs build a model or dictionary of the files they process. Thus compression can be used to “train” classifiers on the labelled documents for each class. Classification of a new document is done by compressing it multiple times, each time using a different class model or dictionary obtained during “training”. The new document is assigned to the class that yielded the highest compression rate. A main attraction of compression-based methods for classification is that they are extremely easy to apply. However, compression based classification methods have drawbacks (such as slow running time), and not all such methods are equally effective.

2.2.1 Analytical techniques used on natural language authorship identification

In early studies, most analytical tools used in authorship Analysis were statistical univariate methods. The pioneering study by Mendenhall (1887) was based on histograms of word-length distribution of various authors. Another popular attribution tool of characterizing the stationary distribution of words or letters is a Naive Bayes (NB) classifier of Mosteller and Wallace (1964) developed during their long work over disputed Federalist Papers. Their systematic work not only provided solid evidence to clarify the disputation but also grounded this field.

The CUSUM statistics procedure is another tool applied to authorship analysis by Farrington (1996). The essence of this procedure is to create the cumulative sum of the deviations; of the measured variable and plot that in a graph to compare among authors. This technique showed some success and even became a forensic tool to assist experts conducting authorship analysis. Nevertheless, Holmes (1998) found that the CUSUM analysis was unreliable because the stability of that test over multiple topics was warranted. Univariate methods have another constraint in that they can only deal with one or more features. These constraints called for the application of the multivariate approaches.

Burrows (1987) first employed principle component analysis (PCA) on the frequency of function words. PCA is capable of combining many measures and project them into a graph. The geographic distance represents the similarity between different authors' style. The good results encouraged many follow-up studies based on the multivariate method. Cluster analysis and discriminant analysis were introduced to this field later by Holmes (1992) and Ledger and Merriam (1994). Mutually supportive results obtained by a variety of multivariate methods have further validated the effectiveness of multivariate approaches.

The advent of powerful computers instigated the extensive use of machine learning techniques in authorship analysis. Tweedie et al. (1996) used a feedforward neural network, also called multilayer perception, to attribute authorship to the disputed Federalist Papers. Radial basis function (RBF) networks were applied by Lowe and Matthews (1995) to investigate the extent of Shakespeare's collaboration with his contemporary, John Fletcher, on various plays. More recently, Khmelev and Tweedie (2002) presented a technique for authorship attribution based on a simple Markov Chain, the key idea of which is using the probabilities of the subsequent letters as

features. Diederich et al. (2000) introduced Support Vector Machines (SVM) to this field. Experiments were carried out to identify the writings of seven target authors from a set of 2,265 newspaper articles written by several authors covering three topic areas. This method detected the target authors in 60 to 80% of the cases. A new area of study is the identification of email authors based on message content. De Vel et al. (2001) used SVM to classify 150 e-mail documents from three authors. In this experiment an average accuracy of 80% was achieved. A variant of Exponentiated Gradient algorithm was examined and showed that this algorithm outperforms other popular classifiers such as NB and Ripper.

In general machine learning methods achieved higher accuracy than did statistical methods. The machine learning methods can deal with a larger set of features with fewer requirements on mathematical models or assumptions. Meanland (1995) also noted that machine learning methods were tolerant to noise and nonlinear interactions among features. Besides techniques, parameters such as the number of authors to be identified and the number of messages used to train the classification model also can impact the performance of authorship identification.

2.2.2 Keselj's Approach to Natural Language Authorship Identification

The SCAP method extends Keselj et al's 2003 work, so it is important to describe this particular method. It is worth to note that the basic idea of the method was originally introduced in Canvar and Trenkle (1994). In Keselj et al's 2003 work, the text is decomposed into character-level n-grams (using a Perl text processing program by Keselj 2003). An n-gram is an n-contiguous sequence and can be defined on the byte, character, or word level. For the Roman alphabet's 26 graphemes, 676 character-level bi-grams are thus possible, although not all of these possible bi-grams will be instantiated in any

given text due to the phonotactic constraints of any particular natural of programming language; for instance, English permits [xa] as in [Xavier] but not [xb], although the bi-gram [xb] may occur in a mathematical equation or programming variable name.

Keselj et al 2003 defines an author profile “to be a set of length L of the most frequent n-grams with their normalized frequencies.” The profile of an author is, then, the ordered set of pairs $\{(x_1; f_1); (x_2; f_2), \dots, (x_L; f_L)\}$ of the L most frequent n-grams x_i and their normalized frequencies f_i . The normalized frequency f_i is obtained by taking the ratio between the actual frequency of a given n-gram and the total number of n-grams located in an author’s profile. Keselj et al 2003 determine authorship based on the dissimilarity between two profiles, comparing the most frequent n-grams. Identical texts will obviously have an identical set of L most frequent n-grams, and thus have zero dissimilarity. Different texts will be more or less similar to each other, based on the amount of most-frequent n-grams which they share. It is important to note that the normalized frequencies constitute the author profile in Keselj et al’s 2003 approach.

The original dissimilarity measure used by Keselj et al. 2003 in text authorship attribution is a form of relative distance:

$$\sum_{n \in profile} \left(\frac{f_1(n) - f_2(n)}{\frac{f_1(n) + f_2(n)}{2}} \right)^2 = \sum_{n \in profile} \left(\frac{2(f_1(n) - f_2(n))}{f_1(n) + f_2(n)} \right)^2 \quad (1)$$

where $f_1(n)$ and $f_2(n)$ are either the normalized frequencies of an n-gram n in the two compared texts or 0 if the n-gram does not exist in the text(s). In this formula, the absolute difference of a given n-gram is divided by the average frequency in order to tackle the sparse data problem. Thus for example, the difference of 0.1 for an n-gram with frequencies 0.9 and 0.8 in two profiles will be less weighted than the same difference for an n-gram with

frequencies 0.2 and 0.1. A text is classified to the author, whose profile has the minimal distance from the text profile, using this measure. Hereafter, this distance measure will be called Relative Distance (RD).

2.3. Authorship Attribution Methods for Programming Languages

On the evening of 2 November 1988, someone infected the Internet with a *worm* program. Spafford (1989) conducted an analysis of the program using three reversed-engineered versions. Coding style and methods used in the program were manually analyzed and conclusions were drawn about the author's abilities and intent. Following this experience, Spafford and Weeber (1993) suggested that it might be feasible to analyze the remnants of software after a computer attack, such as viruses, worms or trojan horses, and identify its author. This technique, called software forensics, could be used to examine software in any form to obtain evidence about the factors involved. They investigated two different cases where code remnants might be analyzed: executable code and source code. Executable code, even if optimized, still contains many features that may be considered in the analysis such as data structures and algorithms, compiler and system information, programming skill and system knowledge, choice of system calls, errors, etc. Source code features include programming language, use of language features, comment style, variable names, spelling and grammar, etc.

Cook and Oman (1989) used "markers" based on typographic characteristics to test authorship on Pascal programs. The experiment was performed on 18 programs written by six authors. Each program was an implementation of a simple algorithm and it was obtained from computer science textbooks. They claimed that the results were surprisingly accurate.

Longstaff and Shultz (1993) studied the WANK and OILZ worms which in 1989 attacked NASA and DOE systems. They have manually analyzed code structures and features and have reached a conclusion that three distinct authors worked on the worms. In addition, they were able to infer certain characteristics of the authors, such as their educational backgrounds and programming levels. Sallis et al (1996) expanded the work of Spafford and Weber by suggesting some additional features, such as cyclomatic complexity of the control flow and the use of layout conventions.

An automated approach was taken by Krsul and Spafford (1995) to identify the author of a program written in C. The study relied on the use of software metrics, collected from a variety of sources. They divided over 50 metrics into three categories: programming layout metrics, programming style metrics, and programming structure metrics. Programming layout metrics includes such fragile metrics as comment placement, indentation, bracket placement, and while lines. These metrics can be easily altered by a code formatter and pretty printer. Also, the text editor used to compose the program can modify these metrics by changing the format to its default or to a preferred layout. Programming style metrics are related to the code layout metrics, but are more difficult to change. Such metrics include variable length, comment length, naming preference, and preference of loop statements. Programming structure metrics are assumed to be dependent on programming experience and the ability of the programmer. Example metrics in the category of style metrics are mean number of lines of code per method/function, data structure usage and preference, and the cyclomatic complexity number (McCabe, 1976). These features were extracted using a software analyzer program from 88 programs belonging to 29 programmers. A tool was developed to visualize the metrics collected and help select those metrics that exhibited little within-author variation, but large between-author variation. Although so many

measurements were collected, many were eliminated and a smaller set remained for the final analysis (Krsul and Spafford, 1995). It can be argued that the information hidden in the unselected measurements was ignored. A statistical approach called discriminant analysis was applied on the chosen subset of metrics to classify the programs by author. The experiment achieved 73% overall accuracy.

Other research groups have examined the authorship of computer programs written in C++ (Kilgour et al., 1998); (MacDonell et al. 2001), a dictionary based system called IDENTIFIED (integrated dictionary- based extraction of non-language-dependent token information for forensic identification, examination, and discrimination) was developed to extract source code metrics for authorship analysis (Gray et al., 1998). In these studies 26 authorship-related metrics were extracted from 351 source code programs, written by 7 different authors. Satisfactory results were obtained for C++ programs using case-based reasoning, feed-forward neural network, and multiple discriminant analysis (MacDonell et al. 2001).

Ding and Samadzadeh (2004), investigated the extraction of a set of software metrics of a given Java source code that could be used as a fingerprint to identify the author of the Java code. They divided over 50 metrics into three categories: programming layout metrics, programming style metrics, and programming structure metrics. The contributions of the selected metrics to authorship identification were measured by a statistical process, namely canonical discriminant analysis, using the statistical software package SAS. A set of 56 metrics of Java programs was proposed for authorship analysis. Forty-six groups of programs were diversely collected. Classification accuracies were 62.7% and 67.2% when the metrics were selected manually while those values were 62.6% and 66.6% when the metrics were chosen by SDA (stepwise discriminant analysis).

Lange and Mancoridis (2007) proposed a technique in which code metrics are represented as histogram distributions. 18 different metrics have been considered in order to represent the style of an author. The most likely author for a given piece of code is found by measuring the differences between histogram distributions of code under scrutiny with those associated with code from a pool of known developers. Their method has been demonstrated using a very large data set comprising twenty developers each authored 3 projects. The definition of success was to classify 40 projects correctly. A genetic algorithm was used in order to find good metric combinations. The accuracy results was 55% in choosing the single nearest match and 75% accuracy in choosing the top three ordered nearest matches.

Kothari et al (2007) used a combination of style and text based metrics in order to represent each programmer's style. The text based metrics used were the 4-grams located in a piece of code and their corresponding frequencies. The calculated metrics were then presented through a filtering tool in order to determine, for each developer, which metrics are most effective in their characterization. These filtered metrics represented the developer's profile. For a given piece of unidentified piece of code all metrics were calculated, and then the database of developer profiles and the calculated metrics of the unidentified piece of code were presented to two different classification tools, the Bayes and the Voting Feature Interval (VFI). The approach was demonstrated on two different data sets achieving greater than 70% accuracy in choosing the single nearest match and greater than 90% accuracy in choosing the top three ordered nearest matches. Another conclusion of this study was that the 4-grams based metrics significantly outperformed the style based metrics. This conclusion supports our approach which is entirely based on the n-grams.

2.3.1. Features for Source Code Authorship Identification

Cook and Oman (1989) describe the use of markers to represent 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:

- Inline comments on the same line as source code.
- Blocked comments (two or more comments occurring together).
- Bordered comments (set of by repetitive characters).
- Keywords followed by comments.
- One or two space indentation occurred more frequently.
- Three or four space indentation occurred more frequently.
- Five spaces or greater indentation occurred more frequently.
- Lower case characters only (all source code).
- Upper case characters only (all source code).
- Case used to distinguish between keywords and identifiers.
- Underscore used in identifiers.
- BEGIN followed by a statement on the same line.
- THEN followed by a statement on the same line.
- Multiple statements per line.
- Blank lines in program body

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 is 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.

Oman and Cook (1991) collected a list of 236 style rules that could be used as a base for extracting metrics dealing with programming style. The programming style taxonomy they suggested includes the following basic categories.

General programming practices: Rules and guidelines pertaining to the programming process that directly affect the style of the product.

Typographic style: Style characteristics affecting only the typographic layout and commenting of code with no affect on program execution.

Control structure style: Style characteristics pertaining to the choice and use of control flow constructs, the manner in which the program or system is decomposed into algorithms, and the method in which those algorithms are implemented.

Information structure style: Style characteristics pertaining to the choice and use of data structure and data flow techniques.

Spafford and Weber (1993) suggested that a technique they called *software forensics* could be used to examine and analyze software in any form, be it source code for any language or executable images, to identify the author. In their study describe a set of high level features that could be considered as author-specific programming features. These features include:

Programming language. The language choice can indicate a number of features about the author. This can include their background (since they would be unlikely to use a language that they were not already familiar with). Not noted by Spafford and Weber (1993), but important nonetheless, are the psychological preferences that some programmers may feel for certain languages.

Formatting of code. The manner in which the source code is formatted can indicate both author features and some psychological information about the author. Pretty-printers are commonly used to automatically format source code and while this removes author-specific features it introduces information about what pretty-printer may have been used.

Special features such as macros may be used that indicate to some degree which compiler or library was used.

Commenting style. This can be a very distinctive aspect of a programmer's style. If comments are sufficiently large then traditional textual linguistic analysis may be appropriate.

Variable naming conventions are another distinctive aspect of an author's style. The use of meaningful versus non-meaningful names, the use of standards (such as Hungarian notation), and the capitalisation of variable names are all features that programmers can adopt.

Spelling and grammar. Where comments are available an examination of their spelling and grammar can be a useful indication of authorship. Spelling errors may also be present in function and variable names.

Use of language features. Some programmers prefer to use certain aspects of a language than others.

Size. The size of routines can indicate the degree of cognitive chunking used by the programmer.

Errors. As noted in the section above on executable code, programmers often consistently make the same or similar errors.

The list of measurements suggested by Spafford and Weber is comprehensive, but the derivation of some of these is 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.”

MacDonell et al (2001) used a set of 26 metrics automatically extracted from a set of 351 C++ programs. These metrics were extracted using as a tool IDENTIFIED (Gray 1998) designed to assist with the extraction of count based metrics. The metrics considered were:

WHITE	Proportion of lines that are blank
SPACE-1	Proportion of operators with whitespace on both sides
SPACE-2	Proportion of operators with whitespace on left side
SPACE-3	Proportion of operators with whitespace on right side
SPACE-4	Proportion of operators with whitespace on neither side
LOCCHARS	Mean number of characters per line
CAPS	Proportion of letters that are upper case
LOC	Non-whitespace lines of code
DEBUGSYM	Debug variables per line of code (LOC)
DEBUGPRN	Commented out debug print statements per LOC
COM	Proportion of LOC that are purely comment

INLCOM	Proportion of LOC that have inline comments
ENDCOM	Proportion of end-of-block braces labeled with comments
GOTO	Gotos per non-comment LOC (NCLOC)
COND-1	Number of #if per NCLOC
COND-2	Number of #elif per NCLOC
COND-3	Number of #ifdef per NCLOC
COND-4	Number of #ifndef per NCLOC
COND-5	Number of #else per NCLOC
COND-6	Number of #endif per NCLOC
COND	Conditional compilation keywords per NCLOC
CCN	McCabe's cyclomatic complexity number
DEC-IF	if statements per NCLOC
DEC-SWITCH	switch statements per NCLOC
DEC-WHILE	while statements per NCLOC
DEC	Decision statements per NCLOC

Finally, Krsul and Spafford (1995) and Kilgour et al. (1998) used the following set of quantitative metrics in order to classify C programs. Similar metrics have been used by Ding (2004) in order to identify the author of programs written in Java.

Programming layout metrics include those metrics that deal with the layout of the program. For example metrics that measure indentation, placement of comments, placement of braces etc. These metrics are fragile because the information required can be easily changed using code formatters. Also many programmers learn programming in university courses that impose a specific set of style rules regarding indentations, placement of comments etc.

Programming style metrics are those features that are difficult to change automatically by code formatters and are also related to the layout of the code. For example such metrics include character preferences, construct

preferences, statistical distribution of variable lengths and function name lengths etc.

Programming structure metrics include metrics that we hypothesize are dependent on the programming experience and ability of the programmer. For example such metrics include the statistical distribution of lines of code per function, ratio of keywords per lines of code etc.

Measurements in these categories are automatically extracted from the source code using pattern matching algorithms. These metrics are primarily used in managing the software development process, but many are transferable to authorship analysis.

2.3.2. Features for Executable Code Authorship Identification

It is possible to perform authorship analysis on the executable code, which is the usual form of an attack in the form of viruses, trojan horses, worms etc. In order to perform such analysis executable code is decompiled (Gray et al., 1997), a process where a source program is created by reversing the compiling process. Although there is a considerable information loss during this process there are many code metrics still applicable. The most common types of executable code that may attack a system are:

- Viruses. A virus can be defined as a program that attaches itself to other programs in order to replicate.
- Worms. A worm is a standalone program that propagates through making copies of itself, similar to a virus but without a host program.
- Trojan horse. A trojan horse is a program that carries out undesirable behaviour while masquerading as a useful program. This can either be a program written as a trojan, or may be the result of modifications made to an existing program.
- Logic bomb. A logic bomb is a part of a program that is written to cause undesirable actions when a certain event triggers its execution.

As Spafford and Weber (1993) note, viruses usually leave their code in infected programs, and code remaining after a variety of attack methods may

include source code, object code, executables, scripts, etc. However, for compiled code much evidence is lost, including variable names, layout, and comments. Compilers may also perform optimisations that lead to the executable code having a significantly different structure to the original source code. Irrespective of the loss of some information, Spafford and Weber are still able to point out some features that will remain. These include:

Data structures and algorithms. This can be a useful indication of the programmer's background since they are more likely to use certain algorithms that they have been taught or had exposure to, and are therefore more comfortable with. Non-optimal choices may indicate a lack of knowledge or even that the programmer uses another language's programming style, perhaps indicating their preferred or first programming language.

Compiler and system information. Executable code contains a number of signs that may indicate the compiler used.

Level of programming skill and areas of knowledge. The degree of sophistication and optimisation can provide useful indications of the author. Differences in sophistication within a program may indicate a mixture of authors or an author who specialises in a particular area.

Use of system and library calls. These may provide some information regarding the author's background.

Errors present in the code. Almost all code contains errors, and any complex system will almost certainly have defects. Programmers are often consistent in terms of the errors that they make.

Symbol table. If an executable is produced using a *debug mode*, rather than a *release mode*, then much information that is part of the source code will still remain. A debug version of a program contains much extra information in the object code that the compiler uses to give feedback while the program is

executing. Surprisingly often, programmers release programs that contain this additional data. A release version simply lacks this superfluous information, making it smaller and faster to execute.

2.3.3 Analytical techniques used on programming authorship identification

Once the programmer –related metrics have been extracted, a number of different modelling techniques, such as neural networks, discriminant analysis, case based reasoning can be used to develop models that are capable of discriminating between several authors. These analytical techniques belong to the following categories:

Manual Approach.

This approach involves examination and analysis of a piece of code by an expert. The objective is to draw conclusions about the authors' characteristics such as educational background, and technical skill. This method has been used in early studies (Spafford 89 and Longstaff and Shultz (1993) to analyze worm programs that infected computer systems. This technique can also be used also in combination with an automated approach (Kilgour et al., 1998), in order to derive linguistic variables to capture more subjective elements of authorship, such as the degree to which comments match the actual source code's behaviour etc.

Statistical Methods

The most widely used technique in source code authorship analysis is discriminant analysis (Krsul and Spafford, 1995; Kilgour et al., 1998; Ding and Samadzadeh, 2004). It uses continuous variable measurements on different groups of items to highlight aspects that distinguish the groups and to use these measurements to classify new items.

An important advantage of the technique (MacDonell et al., 2001) is the availability of stepwise procedures for controlling the entry and removal of

variables. By working with only those necessary variables we increase the chance of the model being able to generalize to new sets of data.

Machine Learning Techniques.

As discussed above the first step of program authorship identification process makes measures of the discriminatory features proposed for authorship attribution. This reduces the style of a particular author's profile to a pattern. Machine learning is particularly suited to pattern matching problems and was used as a tool in this research for classification of authorship patterns. Machine learning techniques have the ability to predict a classification for an unseen test point, i.e. to generalize about unseen data.

A machine learning algorithm attempts to learn from a set of example data in order to generalise about unseen data. We can train the algorithm by optimizing the learning process via manipulation of the variables of the algorithm itself and of the problem domain. The algorithm must produce some type of model representing the knowledge it has learned, and we must measure its performance or its ability to classify unknown examples to determine how good the model is. The machine learning algorithms used in program authorship attribution are:

- *Neural Networks* Neural networks are examples of nonparametric methods, meaning that they can construct a representation of a problem from data where an explicit model of the problem domain is difficult to calculate or is unknown. Values for data features are fed to the input nodes of the neural network and are manipulated by transfer functions at each node. The input data is passed through one or more hidden layers of nodes and finally on to a set of output nodes. The input nodes are fully connected to each node in the hidden layer and the hidden layer is similarly connected to each node in the set of output nodes. The transfer functions may be non-

linear in nature. The neural network must be trained by adjusting the weights of the connections between the nodes to minimise the error rate of the output nodes with the training data. Unseen test data can be fed into the trained neural network and the output class will be predicted.

Feed-Forward Neural Networks (FFNNs) are the most commonly used form of NNs and have been used in source code authorship analysis (MacDonell et al., 2001). Krsul and Spafford, (1995) have also used Multi-Layer Perceptron (MLP) neural network to classify the programmer in the test data with error rates as low as 2%.

- *Case Base Reasoning* is a machine learning method originating in analogical reasoning, and dynamic memory and the role of previous situations in learning and problem solving (Schank, 1982). Cases are abstractions of events (solved or unsolved problems), limited in time and space.

Aarmodt and Plaza (1994) describe CBR as being cyclic and composed of four stages, the *retrieval* of similar cases, the *reuse* of the retrieved cases to find a solution to the problem, the *revision* of the proposed solution if necessary and the *retention* of the solution to form a new case.

When a new problem arises, a possible solution can be found by retrieving similar cases from the case repository. The solution may be revised based upon experience of reusing previous cases and the outcome retained to supplement the case repository. One particular case-based reasoning system that has been previously used for software metric research and in source code authorship analysis is the ANGEL system (Shepperd and Schofield, 1997). MacDonell et al

(2001) used this technique and its performance reached accuracy of 88%, the highest of all methods used.

- *Rule Based Learners* Rule based learners attempt to make rules from the feature values in the training data. For each feature in the data, the algorithm determines the frequency of the feature values or discretised bands of feature values and determines the class of instances to which the most common value belongs. A rule is created for each feature that assigns the class from the feature value or range of values and each rule is then tested using each feature. The rules with the lowest error rates are then chosen to classify unseen data. The Binary tree classifier used by Krsul and Spafford, (1995) belongs in this category. However, its performance was less than optimal, with an error rate of 30%.
- *Instance Based Learners* An instance based learning algorithm uses a distance function to determine which member of the training set an unknown test instance is closest to. This method is particularly suitable for numeric data as the distance function is easily calculated in these cases. The k-nearest neighbour classifier uses the Euclidean distance function and this method has become widely used for pattern recognition problems. Keslej's (2003) text authorship classification method used this technique achieving very good accuracy. Additionally our proposed SCAP (Frantzeskou et al 2007a) source code authorship identification method uses a technique that belongs to this category with surprisingly accurate results

2.4. Summary

In this chapter we reviewed the literature related to natural text and computational authorship. The conclusions we reached are:

The general methodology of authorship attribution applies to texts in both natural and computing languages. This authorship attribution methodology requires two main steps. The first step is the extraction of data for selected features that are said to represent each author's style. The second step normally involves the application of a statistical or machine learning algorithm to these variables in order to develop models that are capable of discriminating between potentially several authors.

In general, when authorship attribution methods have been developed for programming languages, the software features used are language-dependent and require computational cost and/or human effort in their derivation and calculation. The main focus of the early approaches was on the definition of the most appropriate features in representing the style of an author.

While the metric extraction approach to software forensics has been dominant for the last decade it is not without its limitations. The first is that at least some of the software metrics collected are programming-language dependent. For example, metrics specifically appropriate to Java programs are not inherently useful for examining C or Pascal programs – some may simply not be available from programs written in a different language. The second limitation is that the selection of useful metrics is not a trivial process and usually involves setting (possibly arbitrary) thresholds to eliminate those metrics that contribute little to a classification or prediction model. Third, some of the metrics are not readily extracted automatically because they involve judgments, adding both effort overhead and subjectivity to the process.

In sum, the previous work in author identification of programming code has exhibited varying degrees of language-dependence and has achieved a range of levels of effectiveness. In this context, our goal is to provide a fully-automated, language-independent method with high reliability for distinguishing authors and assigning programs to programmers. Furthermore, we aim to identify the language features that contribute to authorship identification and measure the significance of their contribution.

Chapter 3. The SCAP Approach

Our approach to source code authorship attribution, named the Source Code Author Profiles (SCAP) approach, is an extension of a method that has been successfully applied to text authorship identification by Keselj et al. (2003). It is based on the extraction and analysis of byte-level n-grams. An n-gram is an n-contiguous sequence and can be defined at the byte, character, or word level. Character level n-grams are defined as sequences of letters where all other characters are replaced by a space, and letters are turned uppercase. Byte level n-grams are defined as raw character n-grams, without any pre-processing. For example the word sequence “In the” would be composed of the following byte-level N-grams (the character “_” stands for space) :

bi-grams: In, n_, _t, th, he
tri-grams: In_, n_t, _th, the
4-grams: In_t, n_th, _the
5-grams In_th, n_the
6-grams In_the

N-grams have been successfully used for a long time in a wide variety of problems and domains, including information retrieval (Heer, 1974), detection of typographical errors (Morris and Cherry, 1975), language identification (Schmitt, 1991), automatic text categorization (Cavnar and Trenkle, 1994), music representation (Downie, 1999), computational immunology (Marceau, 2000), analysis of whole genome protein sequences (Ganapathiraju et al., 2002), authorship attribution (Keselj et al., 2003), optical character recognition (Adnan et al., 2003), protein classification (Solovyev and Makarova, 1993), protein classification (Cheng et al., 2005) and phylogenetic tree reconstruction (Qi et al., 2004).

We have chosen to use n-grams in this method as they are more flexible and expressive in comparison with fixed lists of tokens, they are language independent, and they can be extracted without the need to construct special mining tools (Juola, 2006). In addition, their use has shown good results in the natural language authorship identification field. Furthermore, the frequency-based analysis method of the SCAP approach is preferred over machine learning techniques since it is simpler to use and interpret, and we have achieved better preliminary results in classification accuracy with this method in comparison with a selection of machine learning techniques.

Programming languages resemble natural languages. Both 'ordinary' texts written in natural language and computer programs can be represented as strings of symbols (words, characters, sentences, etc.) (Miller, 1991; Kokol et al., 1999; Schenkel et al., 1993).

While both rely on the application of rules regarding the structure and formation of artifacts, programming languages are more restricted and formal than (many) natural languages and have much more limited vocabularies. This has been demonstrated by an experiment counting the number of character n-grams (i.e. bigrams, 3-grams, 4-grams and so on) extracted from three files equal in size (0.5 MB). One file contained Java source code text, the second Common Lisp code and the third English text. Figure 3.1 shows the results of a comparison of n-gram 'density', illustrating that the number of n-grams is much larger in the natural language text for all but the smallest n-gram size.

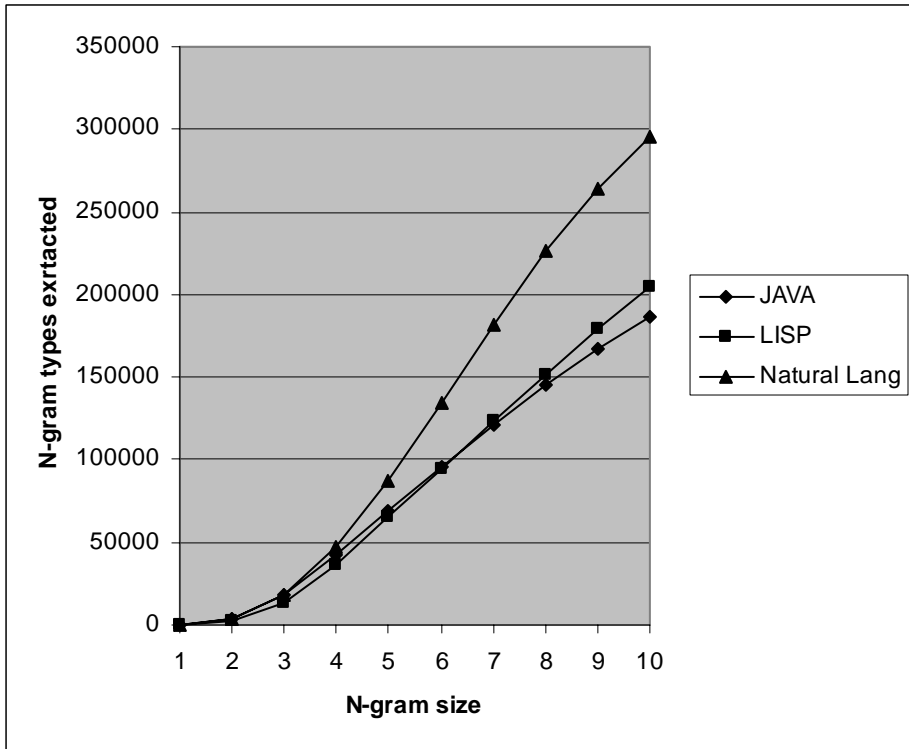


Figure 3.1 Total number of n-gram types extracted from three files equal in size (1 Java file, 1 Common Lisp, 1 Natural Language), for different sizes of n-gram.

3.1. Description of the SCAP method

The SCAP approach is based on the extraction and analysis of byte-level n-grams. An n-gram is an n-contiguous sequence and can be defined at the byte, character, or word level. For example, the byte level 3-grams extracted from 'The first' are (the character _ indicates the space character): The, he_, e_f, _fi, fir, irs, rst. Byte, character and word n-grams have been used in a variety of applications such as text authorship attribution, speech recognition, language modelling, context sensitive spelling correction, and optical character recognition.

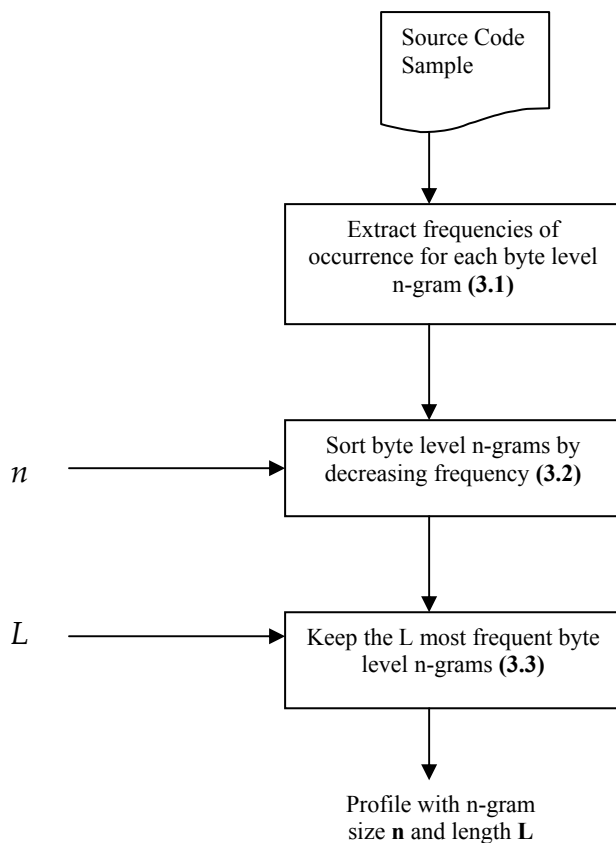


Figure 3.2 Extraction of source code profiles for a given n (n-gram length) and L (profile size).

The SCAP procedure is explained in the following steps and is illustrated in Figures 3.2 and 3.3. Figure 3.2 shows step 3 of the procedure (dealing with profile creation) in detail. The bolded numbers shown in the figures indicate the corresponding step in the description that follows. The SCAP method, as it is described below, calculates the most likely author of a given file for different values of n-gram size n and profile length L . Figure 3.3 illustrates the SCAP procedure for specific values of n-gram size n and profile size L . (For this reason Steps 4.1 and 4.2 are omitted from the diagram.)

1. Divide the known source code programs for each author into training and testing data.
2. Concatenate all the programs in each author's training set into one file. Leave the testing data programs in their own files.

3. For each author training and testing file, get the corresponding profile:
 - 3.1. Extract the n-grams at the byte-level, including all non-printing characters. That is, all characters, including spaces, tabs, and new line characters are included in the extraction of the n-grams. In our analyses, Keselj's (2003) Perl package Text::N-grams has been used to produce n-gram tables for each file or set of files that is required.
 - 3.2. Sort the n-grams by frequency, in descending order, so that the most frequently-occurring n-grams are listed first. The n-grams extracted from the training file correspond to the author profile, which will have varying lengths depending on the length (in terms of characters) of the programming data and the value of n (n-gram length). The profile created for each author will be called the Simplified Profile (SP) since it is simpler than the profile used by Keselj (2003), which uses the n-grams together with their normalized frequencies.
 - 3.3. Keep the L most frequent n-grams $\{x_1, x_2, \dots, x_L\}$. The actual frequency is not used except for ranking the n-grams.
4. For each test file, compare its profile to each author using the Simplified Profile Intersection (SPI) measure:
 - 4.1. Select a specific n-gram length, such as trigram (For the experiments in this paper, we used a range of lengths, 3-grams up to 10-grams).
 - 4.2. Select a specific profile length L, at which to cut off the author profile, smaller than the maximum author profile length.
 - 4.3. For each pair of test and known author profiles, create the SPI measure. Letting SP_A and SP_T be the simplified profiles of one known author and the test or disputed program, respectively,

then the distance measure is given by the size of the intersection of the two profiles:

$$|SP_A \cap SP_T|$$

In other words, the distance measure we propose is the amount of common n-grams in the profiles of the test case T and the author A. The SPI measure it is a similarity (rather than dissimilarity) measure, that is the higher the $|SP_A \cap SP_T|$ the more likely for the test program T to be assigned to author A. In addition this measure does not make use of the frequency information for each n-gram.

- 4.4. Classify the test program to the author whose profile at the specified length has the highest number of common n-grams with the test program profile at the specified length. In other words, the test program is classified to the author with whom we achieved the largest amount of intersection. We have developed a number of Perl scripts in order to create the sets of n-gram tables for the different values of n (n-gram length), L (profile length) and for the classification of the program file to the author with the smallest distance (i.e., greatest overlap). By shifting the n-gram length n and the profile length L (or cut-off, or number of n-gram types included in the SPI), we can test how accurate the method is under different n, L combinations.

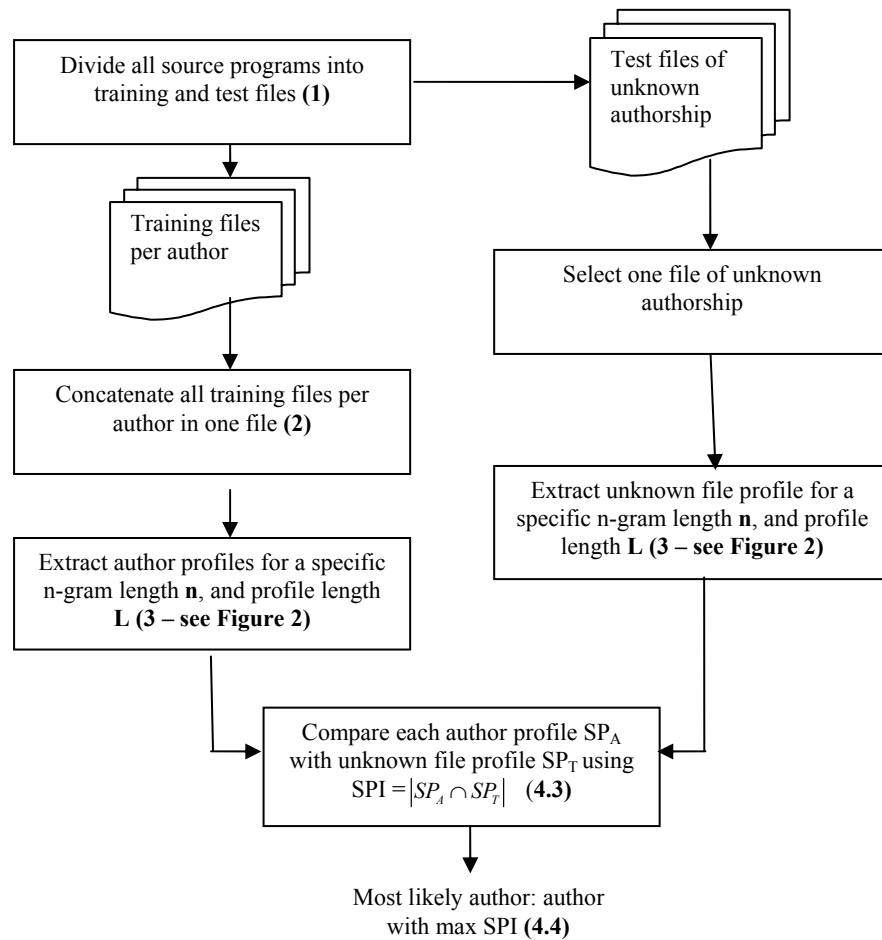


Figure 3.3 Estimation of most likely author of an unknown source code sample using the SCAP approach

3.2. Empirical Study – Hypotheses and Method

The aim of the empirical study conducted was to check the following:

H1 The SCAP method is an effective approach for identifying the author of a source code program given a set of predefined authors.

Since the SCAP method is based on low level information the second hypotheses will be:

H2 The SCAP method is language independent.

Furthermore, the fact that the SCAP method is an extension of a method that has been successfully applied to text authorship identification (Keselj, 2003) leads us to the following hypotheses:

H3 The SCAP method can identify the most likely author of a piece of code by using only the comments present in the program.

The following two sections, 3.2 and 3.3 include all the experiments conducted in this empirical study. The first section includes experiments in order to evaluate the effectiveness of the SCAP method (H1). Data sets written in Java, C and C++ were used with different combinations of profile size L and n-gram size n . In one of the experiments we used the data set used by Mac Donell et al (2001) in order to compare the effectiveness of the SCAP approach against a different source code authorship identification method. In addition this section includes an experiment in order to check whether it is possible to identify the author of a piece of code by using only the comments of a program (H3.).

All the experiments in the first section were conducted using two similarity measures: RD used by Keselj (2003) and SPI used in the SCAP approach. The purpose of this was to check whether the similarity measure used by the SCAP method (SPI) is more effective in identifying the author of a program than the RD similarity measure (RD) used by Keselj in text authorship identification.

Although the experiments with programs written in Java and C in section 3.2 have indicated that the SCAP method is language independent, another set of experiments has been performed in section 3.3 with programs written in Java and Common Lisp in order to evaluate hypotheses H2. These two languages have been chosen as they represent different styles of programming – Java is highly object-oriented, while Common Lisp is multi-paradigm, supporting functional, imperative, and object-oriented

programming styles. Given language similarities it could be expected that programs written in C++ would have similar results to those achieved with Java code, and Prolog programs should behave similarly to Lisp programs.

3.2. Evaluating the effectiveness of the SCAP Method

Table 3.1. Data Sets

	MacDonnellC++	Student Java	OSJava1	NoCom Java	OnlyCom Java	OS Java2
No Authors	6	8	8	8	6	30
Samples per Author	5-114	5-8	4-29	4-29	9-25	4-29
Total Samples	268	54	107	107	92	333
Training Set Samples	134	26	56	56	46	170
Testing Set Samples	133	28	51	51	43	163
Size of smallest sample	19	36	23	10	6	20
Size of biggest sample	1449	258	760	639	332	980
Mean LOC in Training Set	206.4	131.67	155.48	122.28	64.58	170.84
Mean LOC in Test Set	213	127.19	134.17	95.92	56.48	173.03
Mean LOC/sample	210	129	145	109.1	60.53	172

3.2.1. Comparison of SCAP and Keselj's approach on MacDonell Data

Our purpose in this experiment was to check that the SCAP works at least equally as well as the previous methodologies for source code author identification. As mentioned in the previous chapter, MacDonell et al. 2001

reported the best result using the case-based reasoning (that is, a memory-based learning) algorithm for classification accuracy was 88%.

The MacDonell data set was split (as equally as possible) into the training set (134 programs) and the test set (133 programs). We ran the aforementioned perl programs to extract n-grams from two to eight consecutive byte-level characters, to calculate the frequencies and assign the authorship based on the greatest amount of overlapping or shared n-grams between the known author profile and the test source code profile. Table 3.2 presents the results, demonstrating clearly that the Relative Distance method and the SCAP method are both capable of highly reliable results, with most assignments being 100% accurate.

Table 3.2. Classification accuracy (%) on the MacDonell C++ data set using RD (Keselj’s approach) and SPI (SCAP approach).

Profile Size L	n-gram Size													
	2		3		4		5		6		7		8	
	RD	SPI	RD	SPI	RD	SPI	RD	SPI	RD	SPI	RD	SPI	RD	SPI
200	98	98	98	98	97	97	96	96	95	96	93	93	93	95
500	100	100	100	100	100	100	99	100	98	98	98	98	98	98
1000	51	99	100	100	100	100	100	100	100	100	100	100	99	99
1500	5	98	100	100	100	100	100	100	100	100	99	99	99	100
2000	2	98	98	100	100	100	100	100	100	100	100	100	100	100
2500	2	96	99	100	100	100	100	100	100	100	100	100	100	100
3000	2	96	55	100	100	100	100	100	100	100	100	100	100	100

Table 3.2 also shows that the SCAP method outperforms the RD method especially with bi-grams and profile lengths of 1000 or more, although the RD and SPI results equalize with tri-grams and larger n-grams at the 1000 profile length. Consequently, bi-grams (n=2) seem insufficient to discriminate between the authors and will not be examined in the remaining experiments.

In addition to demonstrating the effectiveness of the SCAP methodology for authorship identification of source code, this experiment also allows us to compare the effectiveness of RD and SPI calculations. More importantly, RD performs much worse than SPI in all cases where at least one author profile is

shorter than the selected L profile length. Based on the definition of profile length discussed earlier, Table 3.3 shows the profile length for each of the six authors in this experiment

Table 3.3 Profile Lengths of Six Authors in MacDonell Dataset

Author	1	2	3	4	5	6
Profile Length for bigram	1949	2391	1580	2219	767	1522
Profile Length for trigram	8487	12687	5778	7815	1893	6060
Profile Length for 4-gram	20080	21224	10666	14353	2915	13543
Profile Length for 5-gram	34407	31732	15268	20533	3710	22492
Profile Length for 6-gram	48462	41733	19338	26304	4411	31757
Profile Length for 7-gram	61362	51561	22992	31697	5008	41190
Profile Length for 8-gram	72791	61050	26122	36776	5533	50471

For $L=1000$ and $n=2$, L is greater than the size of the profile of Author Number 5 (the maximum L of the profile of Author No 5 for $n=2$ is 769). The accuracy rate declines to 51% using the RD similarity measure. This occurs because the RD similarity measure (1) is affected by the size of the shortest author profile. When the size of an author profile is lower than L , some programs are wrongly classified to that author. This is because when the shortest profile of an author P_{A1} is compared with a test program P_T the proportion of common n -grams located is more likely to be bigger with P_{A1} than will with a bigger author profile P_{A2} . If you also take into account that the contribution to RD similarity measure of a n -gram found in the author profile but not found in the test program is 2^2 , then is more likely $RD(P_{A1}, P_T)$ to be smaller than $RD(P_{A2}, P_T)$. In summary, we can conclude that the RD similarity measure is not as accurate for those n, L combinations where L exceeds the size of even one author profile in the dataset. In all cases, the accuracy using the SPI similarity measure is better than (or equal to) that of RD. This indicates that this new and simpler similarity measure included in SCAP approach is not affected by cases where L is greater than the smaller author profile.

3.2.2. Performance of SCAP and Keselj’s approach on A Different Programming Language

In order to check that the SCAP method can work effectively independent of particular programming languages, the second experiment was performed on programs written in Java, labelled dataset OS Java in Table 2. Source code samples by 8 different authors were downloaded from the website freshmeat.net. The amount of programs per programmer is highly unbalanced, ranging from 4 to 29 programs per author. The source code sample size was between 23-760 lines of code. In many cases, source code samples by the same programmer share common comment lines at the beginning of the program. Such “common comment” lines were manually removed since they could (positively) influence the classification accuracy. The total number of programs was 107 and they were split into equally-sized training and test sets. This data set provides a more realistic case of source code author identification than typical student programs. Open source code is similar to commercial programs which usually contain comment, are well-structured, and are longer than typical student programs.

Table 3.4. Classification accuracy (%) on the OSJava1 data set for different values of n-gram size, profile length and similarity measure (Relative Distance or Simplified Profile Intersection)

Profile Size L	n-gram Size											
	3		4		5		6		7		8	
	RD	SPI	RD	SPI	RD	SPI	RD	SPI	RD	SPI	RD	SPI
1500	88	100	100	100	100	100	100	100	100	100	100	100
2000	35	100	80	100	100	100	100	100	100	100	100	100

Table 3.4 shows the classification accuracy for tri-grams and longer strings with profile lengths of at least 1500 n-grams. The classification results for the OS Java data set are perfect (100%) for any n-gram size with profile size at least 1500 using the SPI similarity measure. The SPI similarity measure

outperforms the RD measure with trigrams and 4-grams, but then the two measures equalize.

These very high accuracy results mainly are due to the fact that the source code samples of this data set are relatively long, enabling us to set the length of the compared profiles at 1500 and greater. Moreover, for many candidate authors there is a sufficient amount of training samples so that the author profiles are at least as long as the selected profile lengths. Finally, this experiment demonstrates that the SCAP approach works very reliably independently of which programming language (C++ or Java) is analyzed.

3.2.3. Performance of SCAP and Keselj’s approach on Comment-free Source Code

Since the source code used in malicious cyber attacks typically do not contain comments, the third experiment reported here examines the performance of SCAP on comment-free code. We first filtered out any comments from the OS Java data set resulting on a new data set identified as NoComJava in Table 3.2.

Table 3.5 shows that the SPI metric consistently outperforms the RD metric when the n-grams are less than seven characters long and the selected profile lengths are 500 n-grams or greater. Further, the best accuracy rates for SPI occur when the profile length is set at 2000.

Table 3.5 Classification accuracy (%) on the NoComJava data set for different n-gram size, profile size and two similarity measures (Relative Distance or Simplified Profile Intersection)

Profile Size	n-gram Size											
	3		4		5		6		7		8	
	RD	SPI	RD	SPI	RD	SPI	RD	SPI	RD	SPI	RD	SPI
500	94	94	94	94	94	94	94	94	92	94	92	92
1500	35	98	47	90	80	98	96	98	98	98	98	98
2000	33	92	14	98	20	100	31	100	61	100	78	100

In more detail, for $L=500$, when the n -gram ranges from three to eight consecutive characters, RD and SPI have (almost) identical performance. When L increases to 1500, the accuracy of RD drops for shorter n -grams, i.e., low values of n ($n < 6$). When L increases to 2000, the accuracy of RD drops for all values of n . This happens because at least one author has an author profile shorter than the predefined value of L . Just as we saw in the first experiment, RD is not able to handle effectively cases in which an author's profile is shorter than the predefined length of the profile for comparison. Note that the accuracy of SPI increases with L . This is a strong indication that the SPI similarity measure in SCAP suits the source code author identification problem well.

3.2.4. Performance of SCAP and Keselj's approach on Difficult Student Data

The fourth experiment reported here was performed on student programs written in Java, identified as the Student Java dataset (see Table 3.2). Several characteristics of this dataset make it particularly difficult. First, there are only 6-8 programs per author for the 8 different programmers in total. Second, the size of the programs was between only 36 and 258 lines of code, with the mean LOC per program 129 (see Table 3.2). In particular, the source code samples of this data set include assignments from an introductory programming course. The programs written by students usually have no comments while their programming style is influenced by the guidelines of the instructor. More significantly, the source code samples are plagiarised. All these facts introduce some extra difficulties in the source code authorship analysis. As a consequence, the classification results for the Student Java data set are expected to be lower than that of MacDonell C++ data set.

The data set was split into quasi equally-sized training and test sets. As shown in Table 3.6, the best result achieved for the similarity measure RD was

84.6% and the best result for the similarity measure SPI was 88.5% (which is at least equal to MacDonell’s best result with case-based reasoning). These results are quite satisfactory given the difficulties of this data set. This indicates that the SCAP method can reliably handle difficult cases. Again, SPI is more robust in comparison to RD, especially for high values of L when L is more likely to be shorter than an author’s profile.

Table 3.6 Classification accuracy (%) on the StudentJava data set for different values of n-gram size and profile size using two similarity measures: Relative Distance and Simplified Profile Intersection.

Profile Size L	n-gram Size											
	3		4		5		6		7		8	
	RD	SPI	RD	SPI	RD	SPI	RD	SPI	RD	SPI	RD	SPI
200	69.2	69.2	73.1	73.1	73.1	73.1	69	69	73.1	73.1	73.1	73.1
500	76.9	76.9	80.8	80.8	80.8	80.8	84.6	84.6	76.9	76.9	73.1	73.1
1000	80.8	80.8	84.6	84.6	84.6	84.6	80.8	80.8	80.8	80.8	84.6	84.6
1500	84.6	84.6	76.9	76.9	80.8	80.8	84.6	84.6	80.8	80.8	80.8	80.8
2000	53.8	80.8	65.4	80.8	76.9	80.8	84.6	88.5	84.6	84.6	84.6	84.6
2500	53.8	73.1	53.8	76.9	53.8	80.8	84.6	88.5	84.6	88.5	84.6	84.6
3000	53.8	73.1	53.8	80.8	50	76.9	53.8	84.6	69.2	84.6	84.6	84.6

3.2.5. Dealing with many authors using the SCAP and Keselj’s approach

The previous experiments have shown that our approach is quite reliable when dealing with a limited number of candidate authors (6 to 8). In this section we present an experiment that demonstrates the effectiveness of the proposed method when dealing with dozens of candidate authors. For that purpose a data set was created by downloading open-source code samples by 30 different authors from freshmeat.net. Hereafter, this data set will be called OSJava2. Details on this data set can be found in Table 3.2. Note that the available texts per author ranges from 4 to 29. Moreover, in average the samples of this data set are longer in comparison to the OSJava1. Again, all the introductory comments, usually common in the samples of each author, were manually removed. This data set includes programs on the same

application domain written by different authors. In addition the samples of many authors are written over a long time period and therefore there might be programming style changes of certain authors.

The samples were split into equally-sized training and test set. Note that the training set was highly unbalanced (as OSJava1). The best accuracy result was 96.9% and has been achieved using the SPI similarity measure as can be seen in Table 3.7. Again, RD fails to deal with cases where at least one author profile is shorter than L. In most cases, accuracy exceeds 95%, using the SPI similarity measure indicating that the SCAP approach can reliably identify the author of a source code sample even when there are multiple candidate authors. Again, the best result corresponds to profile size of 1500.

Table 3.7 Classification accuracy (%) on the OSJava2 data set for different values of n-gram size and profile size using the SPI similarity measure.

Profile Size L	n-gram Size											
	3		4		5		6		7		8	
	RD	SPI	RD	SPI	RD	SPI	RD	SPI	RD	SPI	RD	SPI
1000	92.9	92.9	92.9	93.9	95.1	95.1	93.9	93.9	95.7	95.1	93.9	94.5
1500	92	92	93.9	93.9	95.1	95.1	95.7	95.7	95.7	96.9	95.1	95.1
2000	30.7	92	71.8	93.9	95.1	95.1	95.1	94.5	95.1	95.7	95.7	95.7
2500	12.8	93.3	36.8	94.5	54.1	95.1	79.1	94.5	94.5	94.5	95.1	95.1
3000	14.1	89.0	13.4	94.5	24.5	95.1	38	95.1	57.7	95.1	75.5	95.1

3.2.6. Performance of SCAP and Keselj's approach on Comments

The aim of this experiment was to check the performance of SCAP method on Comments. Since the method is an extension of an approach applied to natural language, we run this experiment in order to measure the performance of the SCAP approach on text incorporated in source code programs as comments. For this reason we filtered all source code from the OSJava1 data set, leaving the comments only. The resulting data set, which will be called OnlyComJava, includes fewer programs than the original

because any source code files with no comments were removed. As can be seen in Table 3.2 the OnlyComJava data set includes samples by 6 different authors with 9 – 25 files per author.

The application of the SCAP methodology to OnlyComJava data set is described in Table 3.8. Notice that two different profile sizes are indicated (1500 and 2000) since they provide the best results (as can be seen from the previous experiments).

The results demonstrate the RD similarity measure is more competitive, which indicates that it better suits natural language. Again, the best results are obtained using the SPI measure. Probably, this is explained by the extremely short samples that constitute the OnlyComJava data set. Furthermore, the results show that in some cases the author of a program could be identified by examining the comments present in a source code file.

Table 3.8 Classification accuracy (%) on the OnlyComJava data set for different values of n-gram size and profile size using two similarity measures: Relative Distance and Simplified Profile Intersection.

Profile Size L	n-gram Size											
	3		4		5		6		7		8	
	RD	SPI	RD	SPI	RD	SPI	RD	SPI	RD	SPI	RD	SPI
1500	98	98	98	98	98	100	95	95	95	95	95	95
2000	23.2	91	98	100	98	100	95	100	95	98	95	98

3.2.7. The Significance of Training Set Size

The purpose of this experiment was to examine the degree in which the training set size affects the classification accuracy. For this reason we used the C++ data set for which we reached classification accuracy of 100% for many n, L combinations with both similarity measures. This result has been achieved by using a training set of 134 programs in total. For the purposes of this experiment we used the same test set as in the experiment of section 4.1 but now we used training sets of different, smaller size. The smallest training set

was comprised by only one program from each author and the biggest by 5 programs from each one (with the exception of one author for whom the available training programs were only 3). The presented source code author identification approach was applied to these new training sets using $n=6$ and $L=1500$ and similarity measure SPI. Note that the training size of authors was smaller than L in many of these experiments and as already explained, in such cases the classification accuracy decreases dramatically when using the similarity measure RD

The accuracy results achieved are shown in Table 3.9. As can be seen, even with just one program per author available in the training set, high classification accuracy was achieved. By adding a second program per author the accuracy increased significantly above 96%. Note that the second programs added in the training set were in average longer than the first programs (see second column in Table 3.9). We reached 100% of accuracy for training set based on five programs per author. This is a strong indication that our approach is quite effective even when very limited size of training set is available; a condition usually met in source code author identification problems.

Table 3.9 Classification Accuracy (%) on the C++ data set using different training set size (in programs per author).

Training Set Size	Mean LOC in Training Set	Accuracy (%)
1	52	63.9
2	212	96.2
3	171	97
4	170	99.2
5	197	100

3.3. Implementing SCAP to languages that represent different programming style

In order to check that the SCAP method works effectively independent of any particular programming language, a number of initial tests were performed on programs written in two programming languages – Java and Common Lisp. These languages were chosen because they foster very different styles of programming. Following this, a set of experiments were undertaken in order to assess the importance of the factors above that are asserted to contribute to authorship attribution (reported in the following section).

When using Java, the programmer must ‘create’ some words when writing a program, such as a class name or a method name (Lewis and Loftus, 1998). Other terms, such as `String`, `System.out.println`, are not created by the person who writes the piece of code but they are drawn from the author of the Java API and are simply selected for use by the programmer. Reserved words are terms that have special meaning in a programming language and can only be used in predefined ways. The Java language comprises 59 reserved words, including for example `class`, `public`, `static` and `void`.

The fundamental values manipulated by Common Lisp are called atoms (Lamkins, 2004). An atom is either a number (integer or real) or a symbol that looks like a typical identifier (such as `ABC` or `L10`). Their most common use is to assign a label to a value. This is the role played by variable and function names in other languages. Symbols can be defined by the person who writes the program (for example `open-joysticks`, `padding-x`) or by the author of the package (`sdl-data:data-file`, `sdl:init-video`) or can be one of the built-in symbols found in the Common Lisp package (for example `array`, `gensym`). The Common Lisp package contains the primitives of the Common Lisp system as defined by the language specification (The

Harlequin Group Ltd, 2007). It contains 978 symbols. All programs could use any of these symbols as they are all defined by Common Lisp specification.

Table 3.10 Data Sets.

	Common Lisp	Java
No Authors	8	8
Samples per Author	2-5	4-5
Total Samples	35	35
Training Set Samples	16	18
Testing Set Samples	19	17
Size of smallest sample (LOC)	49	52
Size of biggest sample (LOC)	906	519
Mean LOC in Training Set	309	200
Mean LOC in Test Set	171	167
Mean LOC/sample	240	184

3.3.1. The Common Lisp data set

Common Lisp source code samples written by eight different authors were downloaded from the website freshmeat.net. The authors were from four different projects. Two were from project1, three from project2, two from project3 and one from project4. This distribution therefore presented an additional challenge in terms of authorship attribution, as we had programs on the same subject (project) written by different authors. The total number of programs was 35. In order to ensure adequate splits of the sample for each author, sixteen (16) programs were assigned to the training set and nineteen (19) to the test set (see Table 3.10). The data set is from this point referred to as the CLisp dataset. Table 3.11 shows the classification accuracy results achieved on the test data set using various combinations of profile parameter values. The highest level of accuracy achieved on this dataset was 89.5%, shown in bold in the table. The best results were achieved in three instances, all where $n > 6$ and $L > 3000$.

Table 3.11. Accuracy of classification for the CLisp data set

Profile Size (L)	n-gram size							
	3	4	5	6	7	8	9	10
2000	63.2	63.2	68.4	68.4	68.4	68.4	73.7	73.7
3000	73.7	73.7	68.4	68.4	73.7	73.7	78.9	84.2
4000	68.4	84.2	73.7	78.9	89.5	84.2	84.2	89.5
5000	68.4	84.2	78.9	78.9	84.2	78.9	84.2	89.5
6000	68.4	84.2	78.9	78.9	84.2	78.9	78.9	84.2
7000	68.4	84.2	78.9	78.9	84.2	78.9	78.9	78.9
8000	68.4	84.2	84.2	78.9	84.2	78.9	78.9	78.9
9000	68.4	84.2	84.2	78.9	84.2	84.2	78.9	78.9
10000	68.4	84.2	84.2	84.2	78.9	84.2	78.9	78.9

3.3.2. The Java data set

The Java data set included programs by eight different authors. The programs were open source and were found in the freshmeat.net web site. The programs were split into equally sized training and test sets. In order to make the classification 'subject independent' all programs from each author that were placed in the training set were from a different project than the programs placed in the test set. Hence, we had programs from sixteen different projects, two projects for each author. Consequently, the programs in each set did not share common characteristics because they were from different projects. The total number of programs was 35. Eighteen (18) programs were allocated to the training set and seventeen (17) to the test set. This data set is from this point referred to as the Java dataset (see Table 3.10). The results achieved in the Java data set experiment are given in Table 3.12. As can be seen, accuracy reaches 100% in several cases, many of them for $L > 4000$ and $n = 6, 7$ and 8 .

Table 3.12. Accuracy of classification for the Java data set.

Profile Size (L)	n-gram size							
	3	4	5	6	7	8	9	10
2000	58.8	88.2	94.1	94.1	94.1	82.4	88.2	88.2
3000	35.3	82.4	94.1	94.1	100	88.2	88.2	88.2
4000	35.3	70.6	82.4	100	94.1	88.2	88.2	88.2
5000	35.3	47.1	88.2	100	100	100	88.2	88.2
6000	35.3	41.2	76.5	94.1	100	100	88.2	94.1
7000	35.3	41.2	70.6	88.2	100	100	94.1	82.4
8000	35.3	41.2	70.6	82.4	94.1	100	94.1	100
9000	35.3	41.2	70.6	76.5	94.1	94.1	94.1	94.1
10000	35.3	41.2	70.6	76.5	94.1	94.1	94.1	94.1

3.4. Summary

In this chapter we presented a new approach to source code authorship identification, called the SCAP (Source Code Author Profiles) approach. It is based on byte-level n-gram profiles in order to represent a source code author's style. Experiments on stratified data sets of different programming-language (Java, C++ and Common Lisp) and varying difficulty (6 to 30 candidate authors) demonstrate the effectiveness of the proposed approach.

The conclusions reached in relation to the SCAP method are as follows:

- A comparison with a previous source code authorship identification study based on more complicated information shows that the n-gram author profiles are better able to capture the idiosyncrasies of the source code authors.
- One of the inherent advantages of this approach over others is that it is language independent since it is based on low-level non-metric information. Also, the experiments performed with languages that represent different programming styles have demonstrated that the SCAP method can reliably identify the most likely author.

- Experiments with data sets in Java and C++ and Common Lisp have shown that it is highly effective in terms of classification accuracy.
- Comments alone can be used to identify the most likely author in open-source code samples, where there are detailed comments in each program sample. Furthermore, the SCAP method can also reliably identify the most likely author even when there are no comments in the available source code samples.
- The SCAP approach can deal with cases where very limited training data per author is available or there are multiple candidate authors, with no significant compromise in performance.
- The SCAP approach outperforms Keselj's method in cases where the data set consists of a few training programs for an author and many programs available for other authors.
- Many experiments are required in order to identify the most appropriate combination of n-gram size n and profile size L .

Chapter 4. Significance of high-level programming features

As described in the previous chapter, the use of Source Code Author Profiles (SCAP) represents a new, highly accurate approach to source code authorship identification that is, unlike previous methods, language independent. While accuracy is clearly a crucial requirement of any author identification method, in cases of litigation regarding authorship, plagiarism, and so on, there is also a need to know why it is claimed that a piece of code is written by a particular author. What is it about that piece of code that suggests a particular author? What features in the code make one author more likely than another? The provision of evidence to support or refute claims of authorship depends on our ability to answer this question (MacDonell et al., 2002). In this chapter, we describe a means of identifying the high level features that contribute to source code authorship identification using as a tool the SCAP method.

4.1. Program features and source code authorship identification

Computer programs are written according to strict grammatical rules (context free and regular grammars) (Floyd and Beigl, 1994). Programming languages have vocabularies of keywords, reserved words and operators, from which programmers select appropriate terms during the programming process (Kokol and Kokol, 1996). In addition, programs have vocabularies of numbers and vocabularies of identifiers (names of variables, procedures, functions, modules, labels and the like) created by programmers. These are, in general, not language dependent.

Based on previous research efforts (Ding and Samadzadeh, 2004; Krsul and Spafford, 1995) and the broad language characteristics just described, the

features that could influence source code authorship attribution can be considered in the following categorisation:

- *Comments*: Comments are the natural language text statements created by the programmer that generally explain the functionality of the program, possibly including further information regarding the history of the program's development. The programmer is free to use any words he or she prefers. Recently, a number of authorship attribution approaches have been presented (Stamatatos et al., 2000; Peng et al., 2004; Chaski, 2005) proving that the author of a natural language (i.e. free-form) text can be reliably identified. Thus, we assert that comments could contribute to source code authorship identification.
- *Programming layout features*: This category includes those features that deal with the layout of the program and could reflect a programmer's style. Such features include indentation, placement of comments, placement of braces and placement of tabs spaces.
- *Identifiers*: Each programmer is free to create his or her own variable names, function names and similar labels. Also, within a program there are commonly names not created by the programmer but by the author of a package which is imported.
- *Programming structure features*: In previous research efforts, this term has been used to describe certain language-dependent features that might reflect source authorship identification. For example, the "ratio of keyword while to lines of non-comment code", or "ratio of keyword private to lines of non-comment code". The way this term is used in this paper is to describe the keywords that are "built-in" to the language. In Java, this maps to 59 reserved words and in Common Lisp to 978 symbols that are used in the Common Lisp package.

The following subsections describe the abovementioned program features in more detail.

4.1.1. Comments

Source code can be divided into program code (which consists of machine-translatable instructions); and comments which include human-readable notes and other kinds of annotations in support of the program code (Grubb, 2003).

Some contend that comments are unnecessary because well-written source should be self explanatory; others contend that source code should be extensively commented (it is not uncommon for over 50% of the non-whitespace characters in source code to be contained within comments).

In between these views is the philosophy that comments are neither beneficial nor harmful by themselves, and what matters is that they are correct and kept in synch with the source code, and omitted if they are superfluous, excessive, difficult to maintain or otherwise unhelpful (Dewhurst, 2002).

4.1.1.1 Styles of Comments

Comments are generally formatted as *block comments*, also called *prologue comments* or *line comments* also called *inline comments* (Dixit, 2003). Block comments are delimited by a sequence of characters that mark the start of the comment and continue until the sequence of characters that mark the end of the comment. Block comments are allowed to span multiple lines of the source code. Typically, block comments do not nest, so any comment start delimiter encountered within a comment body is ignored. Some languages allow nested block comments to facilitate using comments to comment-out blocks of code that may itself contain block comments. Line comments start with a comment delimiter and continue until the end of the line, or in some

cases, start at a specific column (character line offset) in the source code and continue until the end of the line.

Some programming languages employ both block and line comments with different comment delimiters. For example, C++ has block comments delimited by `/*` and `*/` that can span multiple lines and line comments delimited by `//`.

For example, C++-style comments could look like this

```
// this is a line comment

/*
    This is the comment body.
*/
```

or maybe this

```
/******\
*                                           *
* This is the comment body. *
*                                           *
\*****/
```

Different styles can be chosen for different areas of code, from individual lines to paragraphs, routines, files, and programs. If the syntax supports both line comments and block comments, one approach is to use line comments only for minor comments (declarations, blocks and edits) and to use block comments to describe higher-level abstractions (functions, classes, files and modules).

4.1.1.2 Uses of Comments

How best to make use of comments is subject to dispute, different commentators have offered varied and sometimes opposing viewpoints (Dietrich, 2003; Keyes, 2003):

- *Code description* Comments can be used to summarise code or to explain the programmer's intent. This is called the *why rather than how* approach.

According to this school of thought, restating the code in plain English is a waste of time; the need to explain the code may be a sign that it is too complex and should be rewritten. "Good comments don't repeat the code or explain it. They clarify its intent. Comments should explain, at a higher level of abstraction than the code, what you're trying to do." (Mc Connell, 1993).

- *Algorithmic description* Sometimes source code contains a novel or noteworthy solution to a specific problem. In such cases, comments may contain an explanation of the methodology. Such explanations may include diagrams and formal mathematical proofs. This might especially be true in the case of highly-specialized problem domains; or rarely-used optimizations, constructs or function-calls (Spinellis 2003). For example, a programmer may add a comment to explain why an *insertion sort* was chosen instead of a *quicksort*, as the former is, in theory, slower than the latter.
- *Resource inclusion* Logos, diagrams, and flowcharts consisting of ASCII art constructions can be inserted into source code formatted as a comment. Additionally, copyright notices can be embedded within source code as comments. Binary data may also be encoded in comments through a process known as binary to text encoding, although such practice is uncommon and typically relegated to external resource files.
- *Debugging* A common developer practice is to comment out a code snippet as a comment, such that it will not be executed in the final program.
- *Automatic documentation generation* Some programming tools read structured information in comments and automatically generate documentation. Automatic documentation generation from the comments in the source code provides a means of maintaining consistency between the documentation of the interface and use of the code. Keeping the

documentation within the code makes it easier, and thus more likely, to keep the documentation up to date with changes in the code. Examples of documentation generators include the *javadoc* program, designed to be used with the Java programming language, *Ddoc* for the D programming language and *doxygen*, to be used with C++, C, Java, IDL and others.

- *Metadata and annotations* Developer tools sometimes store metadata in comments, such as insert positions for automatic header file inclusion, commands to set the file's syntax highlighting mode, or the file's revision number. These functional control comments are commonly referred to as annotations. Comments are often employed for these and related methods because they allow the use of syntax and lexical conventions that might otherwise conflict with those of the enclosing programming language. This is another sense in which it is helpful that compilers and interpreters "ignore" comments.
- *Warnings* Some comments are intended as warnings to other programmers that code may not be complete or may have known limitations which should be addressed. One convention for such comments is to prefix them with XXX in order to allow ease of later identifications of potential problems (Arensburger, 2001).

4.1.2. Programming Layout features

Programming layout features refer to a set of rules or guidelines used when writing the source code for a computer program. It is often claimed that following a particular programming style will help programmers quickly read and understand source code conforming to the style as well as helping to avoid introducing faults (McConnell 1993).

The programming layout features used in a particular program may be derived from the *coding standards* or *code conventions* of a company, a project or

other computing organization (Mozilla.org 2007), as well as the preferences of the author of the code. Furthermore, many programming languages (Sun Developer Network, 1999; Norvig and Pitman 1993; Cargill, 1992) have their own of “how to write a program” guidelines. The issues considered as part of the of the programming layout features include:

Indenting Indent styles assist in identifying control flow and blocks of code. In programming languages that use indentation to delimit logical blocks of code, indentation style directly affects the behaviour of the resulting program. In other languages, such as those that use brackets to delimit code blocks, the indentation style does not directly affect the product. Instead, using a logical and consistent indentation style makes code more readable. An example with code written in C++ is given below. The second example is more readable than the first because proper indentation has been used.

```
int main(){
char apples[];
if (apples == "green"){
cout << "Apples are green" << endl;
}}
```

```
int main()
{
char apples[];

    if (apples == "green")
    {
        cout << "Apples are green" << endl;
    }
}
```

Vertical Alignment. It is often helpful to align similar elements vertically, to make typo-generated bugs more obvious. Compare:

```
$search = array('a', 'b', 'c', 'd', 'e');
```

```
$replacement = array('foo', 'bar', 'baz', 'quux');
```

with:

```
$search      = array('a', 'b', 'c', 'd', 'e');  
$replacement = array('foo', 'bar', 'baz', 'quux');
```

Whitespace The term *white space* refers to how text displays in contrast to the surrounding white space. For example, in reading this book, the text displays in a readable manner due to the use of indents, spaces, and paragraphs. Using white space in programming provides the same results as it does for this book: It provides easier readability. Consider the following two Java code samples.

```
//Prompt the user to enter the commission rate  
SimpleIO.prompt("Enter commission rate: ");  
userInput=SimpleIO.readLine();  
double commissionRate=Double.parseDouble(userInput)/100;  
//Compute and display the commission  
double commission=totalValue*commissionRate;  
commission=Math.round(commission*100)/100.0;  
System.out.println("Commission: $" + commission);
```

```
// Prompt the user to enter the commission rate  
SimpleIO.prompt("Enter commission rate: ");  
userInput = SimpleIO.readLine();  
double commissionRate = Double.parseDouble(userInput) / 100;  
  
// Compute and display the commission, roundin  
double commission = totalValue * commissionRate;  
commission = Math.round(commission * 100) / 100.0;  
System.out.println("Commission: $" + commission);
```

The first example does not use white space, but the second example does. . Although not required, it's usually good practice to incorporate white space into your code. Readable code not only aids you in reducing and debugging errors quickly, but it also aids others who might need to read and understand your application's code at a later date.

4.1.3. Identifiers

In computer languages, identifiers are textual tokens (also called symbols) which name language entities. Some of the kinds of entities an identifier might denote include variables, data types, subroutines, and packages.

Variables In computer source code, a variable name is one way to bind a variable to a memory location; the corresponding value is stored as a data object in that location so that the object can be accessed and manipulated later via the variable's name. In statically-typed languages such as Java or ML, a variable also has a "type", meaning that only values of a given class (or set of classes) can be stored in it. In dynamically-typed languages such as Python, it is values, not variables, which carry type. In Common Lisp, both situations exist simultaneously: a variable is given a type (if undeclared, it is assumed to be "T", the universal supertype) which exists at compile time. Values also have types, which can be checked and queried at runtime.

A *data type* is a constraint placed upon the interpretation of data in computer programming. Common types of data in programming languages include

- Primitive types (such as integers, floating point numbers or characters)
- Composite types are datatypes which can be constructed in a programming language out of that language's primitive types and other composite types.
- Class type is a programming language construct that is used to group related instance variables and methods. A method, called a function in some languages, is a set of instructions that are specific to a class. Depending on the language, classes may support multiple inheritances or may require the use of interfaces to extend other classes.

A *subroutine* (function, method, procedure, or subprogram) is a portion of code within a larger program, which performs a specific task and is relatively independent of the remaining code. The syntax of many programming languages includes support for creating self contained subroutines, and for *calling* and returning from them.

Packages are components used in a modular program that can be integrated into the main program through a well-defined interface at source code level. In object-oriented programming packages are used to name a group of related classes of a program. In this meaning, packages are especially useful to measure and control the inherent coupling of a program. Identifiers that belong on “Package” category are usually not written by the programmer who writes the program file we examine. They can be either standard packages, relevant to the programming language or they could be project specific. Consequently these packages could be used by all programmers on a certain project.

4.1.3.1 Naming Convention

Computer languages usually place restrictions on what characters may appear in an identifier. For example, in early versions the C and C++ language, identifiers are restricted to being a sequence of one or more ASCII letters, digits (these may not appear as the first character), and underscores. Later versions of these languages, along with many other modern languages support almost all Unicode characters in an identifier (a common restriction is not to permit white space characters and language operators). In Lisp, these are called *symbols*.

Although in most programming languages there is no character restriction there is a naming convention to be followed (IRT Group and CERN, 2000; Sun Microsystems, 1997). That is a set of rules for choosing the character sequence to be used for identifiers in source code and documentation.

Reasons for using a naming convention as opposed to allowing programmers to choose any character sequence include the following are:

- to make source code easier to read and understand with less effort
- to enhance source code appearance (for example, by disallowing overly long names or abbreviations);

The exact rules of a naming convention depend on the context in which they are employed. Nevertheless, there are several common elements that influence most if not all naming conventions in common use today:

Length of Identifiers A fundamental element of all naming conventions are the rules related to identifier length (i.e., the finite number of individual characters allowed in an identifier). Some rules dictate a fixed numerical bound, while others specify less precise heuristics or guidelines.

Identifier length rules are routinely contested in practice and subject to much debate academically.

It is an open research issue whether programmers prefer shorter identifiers because they are easier to type, or think up, than longer identifiers, or because in many situations a longer identifier simply clutters the visible code and provides no perceived additional benefit.

Brevity in programming could be in part attributed to early linkers which required variable names to be restricted to 6 characters in order to save memory.

Letter case and numerals Some naming conventions limit whether letters may appear in uppercase or lowercase. Other conventions do not restrict letter case, but attach a well-defined interpretation based on letter case. Some naming conventions specify whether alphabetic, numeric, or alphanumeric characters may be used, and if so, in what sequence.

Multiple word identifiers A common recommendation is "Use meaningful identifiers." A single word may not be as meaningful, or specific, as multiple words. Consequently, some naming conventions specify rules for the treatment of "compound" identifiers containing more than one word.

Word boundaries As most programming languages do not allow whitespace in identifiers, a method of delimiting each word is needed (to make it easier for subsequent readers to interpret which characters belong to which word). One approach is to delimit separate words with a non-alphanumeric character. The two characters commonly used for this purpose are the hyphen ('-') and the underscore ('_'), e.g., the two-word name *two words* would be represented as *two-words* or *two_words*. An alternate approach is to indicate word boundaries using capitalization, thus rendering *two words* as either *twoWords* or *TwoWords*.

4.1.4. Programming Structure features

In previous research efforts in source code authorship identification, this term has been used to describe certain language-dependent features that might reflect source authorship identification. For example, the "ratio of keyword while to lines of non-comment code", or "ratio of keyword private to lines of non-comment code". The way this term is used in this paper is to describe the keywords that are "built-in" to the language. For example in C and its kin, this maps to a reserved word which identifies a syntactic form (Kernighan and Ritchie, 1988; Sun Microsystems, 2007). Words used in control flow constructs, such as *if*, *then*, and *else* are programming structure features. In these languages, these keywords cannot also be used as the names of variables or functions. In Common Lisp this maps to 978 symbols that are used in the Common Lisp package which contains the primitives of the

Common Lisp system as defined by this specification (The Harlequin Group Ltd, 2007).

4.2. Datasets and initial empirical analysis – Hypotheses and Method

As it is stated above, the questions we have tried to answer in this empirical study are: What is it about that piece of code that suggests a particular author? What features in the code make one author more likely than another? Since all features described in section 4.1 could influence source code authorship identification, the hypotheses of this empirical study is:

H1 Comments influence source code authorship identification.

H2 Layout features influence source code authorship identification

H3 Identifiers influence source code authorship identification.

Note, that this empirical study did not examine the influence that programming structure features had on authorship identification. As these features are influenced heavily by the program topic, it would be necessary to create a special data set in order to check their contribution. This data set should contain sufficient programs from each author (8-10 programs) where each program has been written by all the authors of interest. Thus, by examining the contribution of each language keyword, the result will be related to each author's choice and not to the underlying program algorithm.

The approach we have followed in order to measure the contribution of each feature to authorship identification is to run a sequence of experiments, each time removing (or disguising) a certain feature. We are then able to measure the effect that each feature removal has on the authorship classification accuracy. This measure effectively indicates the relative significance of this feature's contribution to authorship identification. If a high

level programming feature is positively influential in reflecting program authorship then we would expect that classification performance would deteriorate if this feature was 'hidden' or 'removed' from the analysis (shown as 'Worse' column in Symmary Tables, Tables 4.11 and 4.12 and with the - sign after the accuracy value in the detailed tables.). On the other hand, if a certain feature was not influenced by their authors then we would likely see the same levels of performance achieved in the allocation of test programs to authors as that achieved in the benchmark tests (i.e. 'Same' column in Symmary Tables Tables 4.11 and 4.12, and with no sign in the detailed tables). The two examples below demonstrate this approach by showing a piece of Java code before and after the removal of the layout features.

```
public void add(PlaylistItem[] items, int pos) {
    int start=playlist.size();
    for (int i=0; i<items.length; i++) {
        playlist.add(start + i, items[i]);
    }
    fireItemsAdded(pos, pos + (items.length - 1));
    fixCursor();
    for (int i=0; i<items.length; i++) {
        if (items[i].supportsTags()) {
            tagThread.add(items[i]);
        }
    }
}
```

```
public void add(PlaylistItem[]items, int pos)
{
    int start = playlist.size();
    for (int i = 0; i < items.length; i++)
    {
        playlist.add(start + i, items[i]);
    }
    fireItemsAdded(pos, pos + (items.length - 1));
    fixCursor();
    for (int i = 0; i < items.length; i++)
    {
        if (items[i].supportsTags())
        {
            tagThread.add(items[i]);
        }
    }
}
```


The experiments have been performed using programs written in Java and Common Lisp. These two languages have been chosen as they represent different styles of programming – Java is highly object-oriented, while Common Lisp is multi-paradigm, supporting functional, imperative, and object-oriented programming styles. We acknowledge that this is just one set of experiments, and that further work could be done with larger samples, other languages and so on. Having said that, given language similarities it could be expected that programs written in C++ would have similar results to those achieved with Java code, and Prolog programs should behave similarly to Lisp programs.

4.2.1. The Common Lisp data set

The Common Lisp data set used for these experiments is the same as the one described in subsection 3.3 and is presented in detail in Table 3.10.

We ran a first experiment on this data set, with all features intact, to establish benchmark classification accuracy figures (referred to as the “CLisp benchmark”) against which we could compare performance after the removal of comments. Table 3.11 shows the classification accuracy results achieved on the test data set using various combinations of profile parameter values. (Note that in all other experiments performance is compared to that achieved with the non-commented version of the data set, referred to as the “CLisp[or Java]NoCom benchmark”, because our aim in those tests was to consider the features of the source code that contributed to authorship identification without the ‘influence’ of comments).

4.2.1. The Java data set

The Java data set used for these experiments is the same as the one described in subsection 3.3 and is presented in detail in Table 3.10. The results

achieved in the Java all-features benchmark experiment (referred to as the “Java benchmark”) with this data set are given in Table 3.12.

4.3. Significant features for the Common Lisp data set

Focusing on the features said to influence authorship identification, as described in subsection 4.1 a set of experiments was performed on both the Common Lisp and Java program sets in order to measure each feature’s contribution to accurate classification. To aid understandability of the following results, we have augmented the entries in each table with an indication of comparative performance. In all subsequent tables, the sign ‘-’ to the right side of a value indicates a drop in accuracy in comparison with the associated benchmark data (either all features or NoCom), the sign ‘+’ indicates increased accuracy, whereas no sign alongside the value indicates the same level of performance.

Our first set of experiments was conducted with the Common Lisp programs. We retained the same split of programs across training and test sets as used in the initial empirical analysis reported above.

4.3.1. Contribution of Comments

In order to assess the level of influence that comments have on authorship attribution, all comments were removed from the programs, including the documentation part of Common Lisp statements such as `defun`, `defvar`, `defparameter`. The accuracy achieved on the test data set dropped from that reported previously in the CLisp benchmark. Comparing the results obtained in this experiment (on the CLispNoCom data set) with those obtained from the analysis of the original Common Lisp data set (i.e. comparing the results presented in Tables 3.11 and 4.1) we can see that

accuracy dropped in 61 of the 72 cases, by 10.5% on average. In the remaining 11 cases, accuracy remained the same. The conclusion we reach from this experiment is that comments do appear to influence authorship attribution in Common Lisp programs.

Table 4.1 Accuracy of classification for the CLispNoCom data set.

Profile Size (L)	n-gram size							
	3	4	5	6	7	8	9	10
2000	63.2	63.2	63.2-	57.9-	63.2-	68.4	63.2-	63.2-
3000	68.4-	63.2-	63.2-	68.4	57.9-	68.4-	73.7-	68.4-
4000	68.4	68.4-	68.4-	68.4-	68.4-	68.4-	73.7-	68.4-
5000	68.4	68.4-	63.2-	68.4-	63.2-	73.7-	68.4-	73.7-
6000	68.4	68.4-	68.4-	68.4-	63.2-	73.7-	68.4-	73.7-
7000	68.4	68.4-	68.4-	73.7-	73.7-	73.7-	68.4-	73.7-
8000	68.4	68.4-	68.4-	73.7-	73.7-	73.7-	68.4-	68.4-
9000	68.4	68.4-	68.4-	73.7-	73.7-	73.7-	73.7-	73.7-
10000	68.4	68.4-	68.4-	73.7-	73.7-	73.7-	73.7-	73.7-

4.3.2. Contribution of Layout

Does the layout of Common Lisp programs contribute to authorship classification accuracy, and if yes, to what degree? Note that in general, Common Lisp programs do not differ a lot in terms of layout (Lamkins, 2004). The reason for this is that Common Lisp's simple, consistent syntax eliminates the need for the rules of style that characterize more complicated languages. The most important prerequisite, in terms of legible Common Lisp code, is a simple consistent style of indentation (Seibel 2005).

The objective of this particular experiment was to assess the contribution of program layout to authorship attribution. This was made possible by the removal of the layout features of all programs in the CLispNoCom data set followed by measurement of the effect that this had on classification accuracy. All programs were transformed to a unified-layout data set by removing all indentation and by placing in the previous line of source code all parentheses that were on a separate line. The resulting dataset is referred to as the

CLispLayout data set. The accuracy results achieved with this dataset are given in the Table 4.2.

Comparing the results obtained from this analysis with the CLispNoCom benchmark results (comparing Tables 4.1 and 4.2), we found that the classification accuracy was unaffected in 15 of the 72 cases, in 3 cases we attained better results (by 5.3% on average) and in 54 cases classification accuracy decreased (typically by about 5.5%). The conclusion drawn from this experiment is that, in this case, layout-related features have a consistent but relatively low level of influence in correctly assigning authorship.

Table 4.2 Accuracy of classification for the CLispLayout data set.

Profile Size (L)	n-gram size							
	3	4	5	6	7	8	9	10
2000	57.9-	57.9-	63.2	57.9	57.9-	63.2-	57.9-	63.2-
3000	63.2-	57.9-	63.2	63.2-	63.2+	57.9-	68.4-	57.9-
4000	63.2-	68.4	63.2-	73.7+	63.2-	68.4	68.4-	63.2-
5000	63.2-	63.2-	63.2	68.4	63.2	68.4-	68.4-	68.4-
6000	63.2-	63.2-	63.2-	68.4	68.4+	68.4-	68.4-	68.4-
7000	63.2-	63.2-	63.2-	68.4-	68.4-	68.4-	68.4-	68.4-
8000	63.2-	63.2-	63.2-	68.4-	68.4-	68.4-	68.4-	68.4-
9000	63.2-	63.2-	63.2-	68.4-	68.4-	68.4-	68.4-	68.4-
10000	63.2-	63.2-	63.2-	68.4-	68.4-	68.4-	68.4-	68.4-

4.3.3. Contribution of Identifiers

Another aspect of source code that is author-dependent is the naming convention used. As explained earlier, in Common Lisp the programmer creates his or her own symbols that are analogous to identifiers in other languages. In our experiments, we divided the symbols used in a Common Lisp program into two main categories and conducted a separate experiment for the set of Common Lisp programs, masking instances of symbols from each category in turn.

The first category comprises all symbols that are defined by the programmer who wrote the piece of code. This category is referred to as Symbol Name Identifiers. In the second category, we include all symbols that are package-related but do not belong to the Common Lisp package. Lisp uses packages in order to avoid namespace collisions in a group development environment. In some cases, these symbols are not defined by the user who wrote the piece of code but by the author of the package. These symbols can be distinguished because they include the character ":". Some examples of such symbols are `foo:bar`, `:bar`, and `cl::print-name`. This category is referred to as Package Name Identifiers.

4.3.3.1 Contribution of *Symbol Name Identifiers*

The first Identifiers experiment was conducted on the CLispNoCom data set, after changing all names that belonged to the Symbol Name Identifiers category to unique identifiers. This action neutralized the effect that these names might have had on authorship attribution. If the same identifier was used in two different files then it was changed to two different unique names. An example could be the symbol name 'action' that was used (perhaps by a certain programmer) in two different programs. It was changed to 'a123' in the first program and 'a234' in the second. The data set thus derived is referred to as CLispSymbol.

The accuracy results obtained in this experiment are shown in Table 4.3. Comparing these results with the CLispNoCom benchmark results, it can be seen that in 17 out of the 72 cases we had poorer attribution performance (by about 13.0% on average), in 33 cases the same level of accuracy was achieved, and in 22 cases we achieved improved results (by 6.7% on average). This is explained in part by the fact that the unique identifiers, that replaced the user-defined names, eliminated some of the common n-grams between programs from different authors, which were based on coincidentally common variable

names between different programmers. The conclusion drawn from these rather mixed results is that the names defined by the user in Common Lisp programs do not play a significant role in authorship attribution using the SCAP method.

Table 4.3. Accuracy of classification for the CLispSymbol data set.

Profile Size (L)	n-gram size							
	3	4	5	6	7	8	9	10
2000	73.7+	68.4+	73.7+	57.9	68.4+	63.2-	57.9-	63.2
3000	47.4-	68.4+	73.7+	73.7+	68.4+	68.4	68.4-	73.7+
4000	47.4-	68.4	73.7+	73.7+	73.7+	68.4	68.4-	73.7+
5000	47.4-	68.4	68.4+	73.7+	73.7+	73.7	73.7+	63.2-
6000	47.4-	68.4	68.4	73.7+	73.7+	73.7	73.7+	68.4-
7000	47.4-	68.4	68.4	73.7	73.7	73.7	73.7+	68.4-
8000	47.4-	68.4	68.4	73.7	73.7	73.7	73.7+	68.4
9000	47.4-	68.4	68.4	73.7	73.7	73.7	73.7	68.4-
10000	47.4-	68.4	68.4	73.7	73.7	73.7	73.7	68.4-

4.3.3.2 Contribution of Package Name Identifiers

Similarly, in the second Common Lisp Identifier experiment each name in the training and test program sets that pertained to the Package Naming category was changed to a unique identifier, affecting multiple instances as above. (All the names that belonged to the first category remained unchanged.) An example could be the name `cl:print-name` used in two different programs. It was changed to `b45:b671` in the first file and to `c56:k43` in the second. This action eliminated all common n-grams between the test and author profiles that were based on package-related names. The resulting data set is referred to as CLispPackNam.

The attribution accuracy results obtained from this experiment can be seen in Table 4.4. Comparing these results with the CLispNoCom benchmark results, it can be observed that in 34 of the 72 cases we achieved poorer accuracy outcomes (by approximately 9.1% on average), in 23 cases we achieved the same level of accuracy, and in 15 cases we achieved better results (typically by about 5.6%). Overall, we conclude that in this case Package

Naming does influence accuracy, albeit only slightly, and that it seems to have a greater impact than Symbol Naming.

Table 4.4 Accuracy of classification for the CLispPackNam data set.

Profile Size (L)	n-gram size							
	3	4	5	6	7	8	9	10
2000	63.2	57.9-	63.2	52.6-	63.2	57.9-	52.6-	52.6-
3000	63.2-	57.9-	68.4+	63.2-	63.2+	57.9-	63.2-	52.6-
4000	63.2-	68.4	68.4	68.4	68.4	68.4	57.9-	57.9-
5000	63.2-	68.4	73.7+	68.4	68.4+	68.4-	57.9-	63.2-
6000	63.2-	68.4	73.7+	68.4	68.4+	73.7	57.9-	57.9-
7000	63.2-	68.4	73.7+	73.7	68.4-	73.7	57.9-	57.9-
8000	63.2-	68.4	73.7+	79.0+	73.7	73.7	57.9-	57.9-
9000	63.2-	68.4	73.7+	79.0+	79.0+	73.7	63.2-	57.9-
10000	63.2-	68.4	73.7+	79.0+	79.0+	73.7	63.2-	63.2-

4.3.3.3 Contribution of All Identifiers

One further experiment was conducted to assess the impact of the neutralizing of all names, belonging to either the Symbol or Package Name category. The data set derived is referred to as CLispAllNames. This would show us the effect of naming as a whole on authorship classification accuracy. The results of this experiment are presented in Table 4.5. Comparing these results with those obtained for the CLispNoCom benchmark, accuracy decreased in 34 of the 72 cases (by 8.2% on average), it was improved in 27 cases (by around 6.4%) and in 11 cases it was the same as for the benchmark data. Again, the improvement in accuracy is explained by the fact that the unique identifiers, that replaced the user-defined names, eliminated some of the common n-grams between programs from different authors, which were based on coincidentally common variable names between different programmers.

4.4. Significant features for the Java data set

Our second set of experiments was conducted using the set of open source Java programs described previously. We retained the same split of programs across training and test sets as used in the initial empirical analysis reported above.

Table 4.5. Accuracy of classification for the CLispAllNames data set.

Profile Size(L)	n-gram size							
	3	4	5	6	7	8	9	10
2000	63.2	73.7+	79.0+	63.2+	63.2	63.2-	59.0-	68.4+
3000	57.9-	68.4+	79.0+	68.4	63.2+	63.2-	59.0-	68.4
4000	57.9-	73.7+	73.7+	73.7+	73.7+	63.2-	59.0-	68.4
5000	57.9-	73.7+	73.7+	73.7+	68.4+	73.7	63.2-	68.4-
6000	57.9-	73.7+	73.7+	73.7+	68.4+	73.7	63.2-	63.2-
7000	57.9-	73.7+	73.7+	73.7	68.4-	68.4-	63.2-	63.2-
8000	57.9-	73.7+	73.7+	73.7	68.4-	68.4-	63.2-	59.0-
9000	57.9-	73.7+	73.7+	73.7	68.4-	68.4-	63.2-	52.6-
10000	57.9-	73.7+	73.7+	73.7	68.4-	68.4-	63.2-	52.6-

4.4.1. Contribution of Comments

By removing the comments from the original data set we were able to evaluate their impact on classification accuracy. The results achieved for this new data set, referred to as JavaNoCom, are shown in Table 4.6. Comparing the results shown in this table with those obtained from the analysis of the original data set represented as the Java benchmark (i.e. comparing Tables 3.12 and 4.6) we can see that in 5 out of 72 cases we achieved the same levels of accuracy, in 17 cases the results were better (by 12.5% on average) and in 50 cases the results were worse (typically by 14.5%). The dominance of poorer results leads us to conclude from this experiment, that comments do play an important role in authorship attribution in Java programs.

Table 4.6 Accuracy of classification for the JavaNoCom data set.

Profile Size (L)	n-gram size							
	3	4	5	6	7	8	9	10
2000	52.9-	58.8-	70.6-	76.5-	70.6-	76.5-	82.4-	82.4-
3000	41.2+	70.6-	64.7-	76.5-	76.5-	82.4 -	76.5-	82.4-
4000	41.2+	64.7-	70.6-	82.4-	76.5-	82.4-	76.5-	76.5-
5000	41.2+	64.7+	70.6-	82.4-	82.4-	82.4-	82.4-	82.4-
6000	41.2+	64.7+	70.6-	88.2-	82.4-	82.4-	76.5-	88.2-
7000	41.2+	64.7+	70.6	88.2	82.4-	82.4-	70.6-	76.5-
8000	41.2+	64.7+	70.6	88.2+	82.4-	82.4-	70.6-	76.5-
9000	41.2+	64.7+	70.6	88.2+	82.4-	82.4-	70.6-	76.5-
10000	41.2+	64.7+	70.6	88.2+	82.4-	82.4 -	70.6-	76.5-

4.4.2 Contribution of Layout

In order to assess the contribution of program layout to authorship classification we needed to first create a data set with a unified layout style for all authors. To achieve this, we transformed the programs in the JavaNoCom data set with the use of the style formatter SourceFormatX. The coding style used by the formatter is based on the layout style devised by Sun Microsystems (1997).

The layout features that were unified were as follows:

- All braces were placed on a separate line.
- The indentation of braces was made uniform at two blank characters.
- A blank character was added after each conditional statement, a comma and a semicolon.
- Line length was set to a maximum of 80 characters.
- Long lines were split.
- A blank character was added on the right and left side of the following symbols: ; , if and ?.

- A blank character was added on the right and left side of all operators. Operators included were: ==, +,-, *, /,%, +=, -=, *=, /=,!=?, =>,<?=>, <=&&,| |,&,|^,<<,>>,>>>,&=,|=,^=,<<=,>>=,>>>=

The resulting dataset is referred to as JavaLayout. The accuracy levels achieved in author attribution with this data set are shown in Table 4.7. Comparing these results with the JavaNoCom benchmark results, we can see that performance was worse in all 72 cases, by about 38.6% on average. In other words, for this data set at least, the impact that the layout-related features have on authorship attribution is significant. In many cases the accuracy drops below 40%.

Table 4.7 Accuracy of classification for the JavaLayout data set.

Profile Size(L)	n-gram size							
	3	4	5	6	7	8	9	10
2000	41.2-	23.5-	47.1-	52.9-	47.1-	47.1-	41.2-	52.9-
3000	17.6-	29.4-	29.4-	29.4-	41.2-	52.9-	52.9-	52.9-
4000	17.6-	29.4-	29.4-	35.3-	29.4-	41.2-	41.2-	47.1-
5000	17.6-	29.4	35.3-	29.4-	35.3-	35.3-	41.2-	35.3-
6000	17.6-	29.4-	35.3-	17.6-	35.3-	35.3-	41.2-	35.3-
7000	17.6-	29.4-	35.3-	17.6-	35.3-	35.3-	41.2-	35.3-
8000	17.6-	29.4-	35.3-	17.6-	35.3-	35.3-	41.2-	35.3-
9000	17.6-	29.4-	35.3-	17.6-	35.3-	35.3-	41.2-	35.3-
10000	17.6-	29.4-	35.3-	17.6-	35.3-	35.3-	41.2-	35.3-

4.4.3 Contribution of Identifiers

As for the Common Lisp data set, we addressed the influence of Identifiers on Java program authorship attribution through three experiments, dealing with the effect of user defined identifiers, package name identifiers and then their combination in turn.

4.4.3.1 Contribution of User Defined Identifiers

The aim of the first experiment was to assess the degree to which the names defined by the programmer contributed to authorship identification.

This category included all simple variable names, method names, class names, class variable names and so on that were defined within the program by the programmer. All instances of these names were changed to a unique identifier comprised of a letter and a number. If the same name was used in more than one program, these were changed to different unique identifiers in order to eliminate the common byte level n-grams based on these variables (thus creating a conservative test). The only identifiers that were left unchanged for this experiment were those that were not user defined but were imported using the import statement at the beginning of the program.

The results achieved with this data set (referred to as JavaUserNam) are shown in Table 4.8. By comparing these results to those obtained from the analysis of the JavaNoCom benchmark, it can be observed that accuracy remained the same in 23 of the 72 cases and was in fact improved in the other 49 cases (by 9.0% on average). This indicates that, in this case, the names defined by the user did not contribute positively to authorship attribution. This apparent improvement in accuracy for many of the n, L combinations is explained by the fact that, as evident in the programs in this sample, many programmers use the same names for simple variables or class variable names or methods. Some examples of the commonly used names encountered across different programmers were `name`, `e`, `file`, `text`, and `x`. The byte-level n-grams derived from these commonly used names were responsible for the incorrect classification of some programs in the JavaNoCom dataset. By making each user-defined identifier unique in each program, we eliminated all these common n-grams across the different programmers, thus improving overall classification accuracy.

Table 4.8 Accuracy of classification for the JavaUserNam data set.

Profile Size(L)	n-gram size							
	3	4	5	6	7	8	9	10
2000	52.9	76.5+	76.5+	82.4+	82.4+	88.2+	82.4+	82.4+
3000	47.1+	76.5+	88.2+	88.2+	88.2+	82.4+	76.5+	88.2+
4000	47.1+	64.7	82.4+	88.2+	88.2+	88.2+	88.2+	88.2+
5000	47.1+	64.7	76.5+	88.2+	88.2+	82.4+	88.2+	88.2+
6000	47.1+	64.7	76.5+	88.2+	88.2+	88.2+	82.4+	88.2+
7000	47.1+	64.7	76.5+	88.2+	82.4+	88.2+	88.2+	88.2+
8000	47.1+	64.7	76.5+	88.2+	82.4+	88.2+	88.2+	88.2+
9000	47.1+	64.7	76.5+	88.2+	82.4+	88.2+	88.2+	88.2+
10000	47.1+	64.7	76.5+	88.2+	82.4+	88.2+	88.2+	88.2+

4.4.3.2 Contribution of Package-Related Name Identifiers

This experiment was performed to evaluate the degree to which package-related naming contributed to accurate authorship attribution. Any program written in Java can have a number of import package statements (with associated naming) at the beginning of the file. The import statements allow all classes and methods of the associated packages to be visible to the classes in the current program. These packages could be either project-related (an example could be the `org.alltimeflashdreamer.util.StringUtils` package) or one of the numerous standard packages defined in Java (for instance the `java.io.FileInputStream` package). The second case is the more commonly used. The project related packages in our sample were a very small percentage – less than 1% of all packages. Among the standard classes and their related methods defined by Java that were ‘neutralized’ were the class `String`, `File` and `IOException`, which are used heavily by all programmers. This experiment was performed in a similar way as the previous one, the only difference being that in this experiment we changed only names within the program that were related to all imported packages, leaving all user defined names unchanged. This data set is referred to as `JavaPackNam`. The authorship attribution results for this experiment are shown in Table 4.9. In general, they indicate that package-related naming

does reflect authorship identification. Comparing these outcomes with the JavaNoCom benchmark, the results are worse (by about 11% on average) in 55 of the 72 cases, in 7 cases performance was improved (by typically 5.9%) and in 10 cases the same levels of accuracy were achieved.

Table 4.9. Accuracy of classification for the JavaPackNam data set.

Profile Size(L)	n-gram size							
	3	4	5	6	7	8	9	10
2000	47.1-	52.9-	64.7-	52.9-	70.6	70.6-	70.6-	70.6-
3000	41.2	64.7-	64.7	64.7-	70.6-	70.6-	70.6-	70.6-
4000	41.2	70.6+	58.8-	70.6-	70.6-	70.6-	70.6-	70.6-
5000	41.2	70.6+	64.7-	70.6-	64.7-	64.7-	64.7-	70.6-
6000	41.2	70.6+	64.7-	70.6-	70.6-	64.7-	64.7-	70.6-
7000	41.2	70.6+	64.7-	70.6-	70.6-	64.7-	64.7-	70.6-
8000	41.2	70.6+	64.7-	70.6-	70.6-	64.7-	64.7-	70.6-
9000	41.2	70.6+	64.7-	70.6-	70.6-	64.7-	64.7-	70.6-
10000	41.2	70.6+	64.7-	70.6-	70.6-	64.7-	64.7-	70.6-

4.4.3.3 Contribution of All Identifiers

For this experiment, all names were changed. This included simple variables, class variables and methods defined by the programmer and all class names and method names imported with the import package statement(s) at the beginning of each program. The resulting data set is referred to as JavaAllNames. The purpose of this experiment was to assess the extent of influence that all names used within a program had on authorship attribution. The programs were changed so that all names were replaced by unique identifiers. If the same name appeared in more than one program, it was replaced by a different identifier in each case. The results achieved from the analysis of this data set are given in Table 4.10. In comparing these levels of accuracy against those obtained for the JavaNoCom benchmark, it appears that the likelihood of improvement and deterioration are roughly equivalent – in 9 cases the results were the same, in 34 they were better (by about 17.8%) and in 29 cases they were worse (by about 8.3% on average).

Table 4.10 Accuracy of classification for the JavaAllNames data set.

Profile Size(L)	n-gram size							
	3	4	5	6	7	8	9	10
2000	76.5+	82.4+	76.5+	70.6-	70.6	70.6-	70.6-	70.6-
3000	64.7+	88.2+	82.4+	82.4+	70.6-	70.6-	70.6-	70.6-
4000	64.7+	88.2+	88.2+	82.4	76.5	70.6-	70.6-	70.6-
5000	64.7+	88.2+	88.2+	88.2+	82.4	70.6-	64.7-	70.6-
6000	64.7+	88.2+	88.2+	88.2	88.2+	76.5-	64.7-	70.6-
7000	64.7+	88.2+	88.2+	88.2	88.2+	76.5-	64.7-	70.6-
8000	64.7+	88.2+	88.2+	88.2	88.2+	76.5-	64.7-	70.6-
9000	64.7+	88.+2	88.2+	88.2	88.2+	76.5-	64.7-	70.6-
10000	64.7+	88.2+	88.2+	88.2	88.2+	76.5-	64.7-	70.6-

4.5 Summary of performance

We here provide a set of summary tables that illustrate the various levels of accuracy achieved under the different experimental scenarios for each data set.

In Tables 4.11 and 4.12, the numbers shown in the second, third and fourth columns are out of the 72 considered in total in each experiment, with the number in parentheses indicating the mean deviation from the associated benchmark data set. For the ‘comments removed’ experiment (row 2 in each table), the benchmark data set is the original all-features set (referred to as ‘CLisp benchmark’ and ‘Java benchmark’ respectively). For the remaining experiments, the benchmark data set is the ‘NoCom’ version for each language.

Table 4.11 Summary of results for the set of CLisp programs

Dataset	Worse	Same	Better	Mean accuracy
CLisp				78.0%
CLispNoCom	61 (-10.5%)	11	0	69.0%
CLispLayout	54 (-5.5%)	15	3 (5.3%)	65.1%
CLispSymbols	17 (-13%)	33	22 (6.7%)	68.0%
CLispPackNam	34 (-9.1%)	23	15 (5.6%)	65.9%
CLispAllNam	34 (-8.2%)	11	27 (6.4%)	67.3%

Table 4.12 Summary of results for the set of Java programs

DataSet	Worse	Same	Better	Mean Accuracy
Java				79.3%
JavaNoCom	50 (-14.5%)	5	17 (12.5%)	72.2%
JavaLayout	72 (-38.6%)	0	0	33.7%
JavaUserNam	0	23	49 (9.0%)	78.3%
JavaPackNam	55 (-11%)	10	7 (5.9%)	64.5%
JavaAllNam	29 (-8.3%)	9	34 (17.8%)	77.3%

The relative contribution of the various high-level program features to authorship attribution are summarised in Tables 4.13 and 4.14 (for the Lisp and Java programs, respectively). Again, the values for the ‘Comments’ entries reflect the difference between the original all-features programs and those excluding comments. The remaining differences are between the ‘NoComments’ versions and those produced through manipulation of the other features. When a certain feature contributes positively to authorship attribution, it is indicated by a positive number, and when an other feature does not contribute positively to authorship attribution this is indicated by a negative number in parentheses.

Table 4.13 High-level features and mean accuracy deviation – Common Lisp programs

Common Lisp High-Level Feature Contributions	
Original/Comments	+9.0%
NoComments/Layout	+3.9%
NoComments/Identifiers:Symbols	+1.0%
NoComments/Identifiers:Package	+3.1%
NoComments/Identifiers	+1.7%

Table 4.14 High-level features and mean accuracy deviation – Java programs

Java High-Level Feature Contribution	
Original/Comments	+7.1%
NoComments/Layout	+38.5%
NoComments/Identifiers:User Defined	(-6.1%)
NoComments/Identifiers:Package	+7.7%
NoComments/Identifiers	(-5.1%)

In examining the results presented in Tables 4.13 and 4.14, it is evident that, as might be expected, comments play a significant role in authorship attribution, an outcome that holds across both the Common Lisp and Java experiments. Layout is the next most influential feature, but was much more significant in our Java analysis than in our experiments with Common Lisp code. The use of Identifiers produced mixed outcomes. In assessing its impact on Common Lisp authorship, naming had a small but evident impact on the ability to identify an author using the SCAP approach. While the same outcome was found in relation to Package Naming in Java code in fact the removal of user-defined names *enhanced* the levels of classification accuracy. While initially unexpected, an explanation for this was identified in terms of the incidence of coincidentally common names in programs written by different authors.

4.6 Summary

A number of experiments have been performed in order to empirically identify and assess the impact of high level program features that contribute to source code authorship attribution, using the Source Code Author Profile approach. In these experiments, programs written in two languages that represent two different programming styles were used: Java, which uses objects, and Common Lisp, which uses a functional/imperative programming style. We acknowledge that this is just one set of experiments, and that further work could be done with larger samples, other languages and so on. Having said that, we intentionally selected languages that represent two different programming styles, so that insights into a range of languages might be gained. Given language similarities it could be expected that programs written in C++ would have similar results to those achieved with Java code,

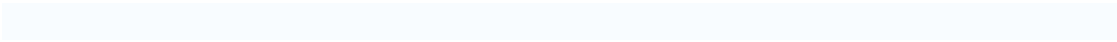
and Prolog programs should behave similarly to Lisp programs. The code used in the data sets was Open Source, which also implies that it follows (without being mandatory) the code conventions recommended by the Open Source Community (Spinellis, 2003), thus reducing the distinctions that might arise if programmers were allowed to use their 'natural' approach.

In each case one feature at a time was either removed or 'neutralised', in order to provide a means of measuring the difference between classification accuracy with and without the feature available. The results of these experiments (presented in summary form in Tables 4.11 through 4.14) have shown the following for the data sets assessed here:

- The accuracy of source code authorship attribution is improved by the existence of comments in the code.
- Layout-related features play a role in determining program authorship, but the extent to which this is an influential characteristic may vary from language to language. In our experiments, the level of impact for the programs written in Java was substantial, but this level was much lower for the programs written in Common Lisp. (The contribution of layout-related features in identifying the author of a Java program is also a conclusion reached by Ding and Samadzadeh (2004).)
- Variable and function names defined by the programmer do not seem to influence classification accuracy – and in fact in some cases accuracy might be improved by 'neutralizing' these names. This is due to the fact that programmers have been shown to use the same names for simple variables, class variable names, methods or functions. In our case, this conclusion certainly applied to the Java programs, and to those written in Common Lisp to a lesser extent.

- Package-related naming influences accuracy, an outcome evident for programs written in both languages.

Overall, the authorship of Java programs was generally more susceptible to influence, with particularly high influence from program layout. In comparison, the authorship of Common Lisp programs did not appear to be as strongly influenced by the features we considered. This could be explained by the fact that the programming structure features that remained unchanged, influence authorship identification more in Common Lisp than in Java, perhaps because Common Lisp has a richer vocabulary than Java.



Chapter 5. The significance of user defined identifiers in Java source code authorship identification

Although programming languages are generally more formal and restrictive in their form and composition than spoken or written languages, program authors are still afforded a large degree of flexibility when composing source code (Krsul, and Spafford, 1995]. Each programming language has a vocabulary of keywords, reserved words and operators, from which program authors select appropriate terms during the programming process (Kokol, and Kokol, 1996). In addition, source code contains vocabularies of identifiers (names of variables, procedures, functions, modules, labels and the like) created by programmers. Identifier naming can be influenced by many things – the application domain, agreed coding styles, organisational guidelines, or an educator’s advice in the case of code written by students. Although the importance of meaningful identifiers is stressed by educators, Sneed observes that “in many legacy systems, procedures and data are named arbitrarily.... programmers often choose to name procedures after their girlfriends [*sic*] or favourite sportsmen” (Sneed 1996). Thus while coding styles and the like exist, the degree to which these actually influence identifier naming – and the consequent impact on authorship analysis – is unknown. It may be that programmers create their identifiers in a systematic or consistent way so that any resulting program reflects its author. With this in mind, our aim is to assess the impact that identifiers have on the accuracy of Java source code authorship attribution, using two sets of Java programs. In other words, the questions we address here are: Do Java identifiers contribute to correct authorship identification? Is it possible to hide the provenance of some Java program by identifier renaming?

These questions are very important whenever a need for evidence arises in regard to source code authorship (Mac Donnell et al 2002), such as in formal dispute proceedings. For example, we may be able to assert *with evidence* that Java programmer A is the author of a disputed program because the class variables and/or method names used closely resemble those used elsewhere by programmer A – and because they do *not* resemble those used by one or more other Java programmers. While the work described in subsection 4.4.3 has indicated that Java identifiers defined by the programmer do not influence classification accuracy, and in fact in some cases accuracy might be improved by ‘neutralizing’ these names, that study examined all user-defined identifiers together. In this chapter we check whether this conclusion holds when we examine each type of programmer-defined identifier separately.

In conducting our analysis, we use the SCAP approach as a tool for assessing the significance of each type of Java identifier. Three different types of Java identifiers are considered here: class, method and simple identifiers. A number of experiments have been performed in order to assess the impact that each type of identifier has on source code authorship attribution. We first assess classification accuracy using the complete source programs, creating an initial performance benchmark. We then measure the contribution of each identifier type in authorship identification by running a sequence of experiments, each time disguising a certain type of identifier in the source code. We are therefore able to measure the effect that ‘neutralization’ of each type of identifier has on the accuracy of the classification – the difference in each case effectively indicates the relative significance of this type of identifier to authorship identification. The experiments have been performed using two different data sets for which the authors of the programs are known. One data set comprises open source programs and the second is made up of programs written by students during an introductory Java course.

5.1. Java Identifiers and Source Code Authorship Identification

Computer programs are written according to strict grammatical rules (context free and regular grammars) (Floyd and Beigl, 1994). Each language has a vocabulary of keywords, reserved words and operators, from which programmers select appropriate terms during the programming process (Kokol and Kokol, 1996). In addition, programs have vocabularies of numbers and vocabularies of identifiers (names of variables, procedures, functions, modules, labels and the like) created by programmers. These are, in general, not language dependent.

When using Java, the programmer must 'create' some words when writing a program, such as a class name or a method name (Lewis and Loftus, 1998). Our aim in this study is to measure the significance of Java programmer defined identifiers to authorship identification. We have considered these identifiers in terms of three main categories:

Simple Identifiers: This category includes all variables defined within the program with type, `int`, `char`, `byte`, `long`, `Boolean`, `int[]` and `long[]`. Some examples of such variables are `year` with type `int` or `flag` with type `boolean`.

Class Identifiers: This category includes all variables that are defined within the program and their type is a class. For example, `name` with type `String` or `ex` with type `Exception`.

Method Identifiers: This category includes all method names that are defined within the program. Some examples are `getInteger`, `drive` and `getYear`.

All Identifiers: All user-defined identifiers, effectively the aggregation of those elements in the three categories listed.

Other source code terms, such as `String` and `System.out.println`, are not created by the person who writes the piece of code but they are drawn from the author of the Java API and are simply selected for use by the programmer. These names are not part of our current study even though they might influence classification accuracy of Java programs as it has been shown in the previous chapter in subsection 4.4.3.

Table 5.1. Characteristics of the two Data Sets

	OSJava	Student Java
No Authors	8	8
Samples per Author	4-5	5-8
Total Samples	35	54
Training Set Samples	18	26
Testing Set Samples	17	28
Size of smallest sample (LOC)	52	36
Size of biggest sample (LOC)	519	258
Mean LOC in Training Set	200	131.67
Mean LOC in Test Set	167	127.19
Mean LOC/sample	184	129

5.2 Empirical Analysis - Our approach

The questions we address in this empirical study are: Do Java identifiers contribute to correct authorship identification? Is it possible to hide the provenance of some Java program by identifier renaming?

Focusing on each of the four categories of identifiers that might influence authorship identification in turn, as described in the previous section, a set of experiments was performed on two program sets, the OSJava and StudentJava that are described below. The aim of each experiment was to

measure the contribution of each type of identifier to accurate classification. Hence, for each category of identifier we report the results obtained from both the OSJava and the StudentJava data sets after disguising the relevant identifiers and then analyzing and classifying the programs using the SCAP approach. The approach we have followed in this empirical study is the same with the method used in the previous chapter. If a category of identifiers is positively influential in reflecting program authorship then we would expect that classification performance would deteriorate if those identifiers were 'hidden' from the analysis (shown as 'Worse than OSJava/StudentJava' in Summary Tables, Tables 5.12 and 5.13 and with the sign – after the accuracy value in the detailed tables.). On the other hand, if a certain group of identifiers was not influenced by their authors then we would likely see the same levels of performance achieved in the allocation of test programs to authors as that achieved in the benchmark tests (i.e. 'Same as OSJava/StudentJava', and with no sign in the detailed tables). The two examples below demonstrate this approach by showing a piece of Java code before and after disguising the simple identifiers.

```
int start=playlist.size();
for (int i=0; i<items.length; i++) {
    playlist.add(start + i, items[i]);
}
```

```
int a2b2g2 = playlist.size();
for (int a3b3g3 = 0; a3b3g3 < items.length; a3b3g3++) {
    playlist.add(a2b2g2 + a3b3g3, items[a3b3g3]);
}
```

5.3 Data Sets Analysed

In order to assess the contribution of user- (or programmer-) defined identifiers to authorship identification of Java programs, two different data

sets have been considered here. The first data set included open source (OS) programs written by eight different authors (see Table 5.1, column OSJava) as found in the freshmeat.net web site. The programs were allocated to approximately equally-sized training and test sets. In order to make the allocation and analysis 'domain independent' all programs from each author that were placed in the training set were from a different project from the programs placed in the test set. Hence, we had programs from sixteen different projects, two projects for each author. Consequently, since they were from different projects the programs in each set for each author did not share inherently common characteristics. The total number of programs in this data set was 35. Eighteen programs were allocated to the training set and 17 to the test set. This data set is from this point referred to as the OSJava data set.

The second data set comprised programs written during an introductory Java course. These programs were written by 8 different programmers, making up a sample of 54 programs in total. The data set was split into quasi equally-sized training and test sets. As these programs were student assignments from an introductory programming course there is a high degree of likelihood that the identifiers used will have been influenced by the guidance of the instructor, and perhaps by that found in course texts. More significantly, we know that some of the source code samples had been plagiarized. In addition, most of the program samples are from the same domain (i.e. sorting algorithms, binary search). All these facts imply that the programs in this data set potentially share several common characteristics. More details on this data set can be found in Table 5.1. This data set is from this point referred to as the StudentJava data set.

5.3.1 The Open Source Java data set

In order to establish our performance benchmark for this data set (referred to as OSJava benchmark), we removed all comments from the OSJava programs and ran a first experiment with all identifiers intact. The comments were removed from the data set because our aim in the tests was to consider the degree to which identifiers in the source code contributed to authorship identification without the ‘influence’ of comments. Table 5.2 shows the classification accuracy results achieved on the test data set using various combinations of profile parameter values. The highest level of accuracy achieved on this dataset was 88.2%, shown in bold in the table. The best results were achieved in four instances, all where $n > 5$ and $L > 5000$.

Table 5.2 Accuracy of classification for the OSJava data set.

Profile Size (L)	n-gram size							
	3	4	5	6	7	8	9	10
2000	52.9	58.8	70.6	76.5	70.6	76.5	82.4	82.4
3000	41.2	70.6	64.7	76.5	76.5	82.4	76.5	82.4
4000	41.2	64.7	70.6	82.4	76.5	82.4	76.5	76.5
5000	41.2	64.7	70.6	82.4	82.4	82.4	82.4	82.4
6000	41.2	64.7	70.6	88.2	82.4	82.4	76.5	88.2
7000	41.2	64.7	70.6	88.2	82.4	82.4	70.6	76.5
8000	41.2	64.7	70.6	88.2	82.4	82.4	70.6	76.5

5.3.2 The Student Java data set

In order to establish our second data set benchmark (referred to as StudentJava benchmark), as above we removed all comments from the StudentJava programs and ran a first experiment with all identifiers intact. Table 5.3 shows the classification accuracy results achieved on the test data set using various combinations of profile parameter values. The highest level of accuracy achieved on this dataset was 88.5%, shown in bold in the table. The best result was achieved in one instance, for $n=8$ and $L=3000$.

Table 5.3 Accuracy of classification for the StudentJava data set.

Profile Size (L)	n-gram size							
	3	4	5	6	7	8	9	10
2000	69.2	76.9	80.8	84.6	84.6	84.6	80.8	76.9
3000	69.2	76.9	76.9	80.8	80.8	88.5	84.6	84.6
4000	69.2	80.8	76.9	76.9	76.9	80.8	80.8	84.6
5000	69.2	80.8	76.9	76.9	76.9	80.8	80.8	80.8
6000	69.2	80.8	76.9	76.9	76.9	76.9	80.8	80.8
7000	69.2	80.8	76.9	76.9	76.9	76.9	80.8	76.9
8000	69.2	80.8	76.9	76.9	76.9	76.9	80.8	80.8

5.4 Significance of Identifiers

Focusing on the different types of identifiers that might influence authorship identification, as described in subsection 5.3, a set of experiments was performed on both the OSJava and StudentJava program sets in order to measure the contribution of each type of identifier to accurate classification. Hence, for each type of identifier we report the results obtained from both the OSJava and the StudentJava data sets, after disguising a certain type of identifier. To aid understandability of the results, we have augmented the entries in each table with an indication of comparative performance. In all subsequent tables, the sign ‘-’ to the right side of a value indicates a drop in accuracy after neutralization in comparison with the associated benchmark data, the sign ‘+’ indicates increased accuracy, whereas no sign alongside the value indicates the same level of performance.

In order to assess the statistical significance of the results we obtained, we performed one-tailed t-tests to compare the mean accuracy achieved in the benchmark OSJava and StudentJava tests and the mean accuracy obtained after disguising each identifier type. The significance level used was 5% for all experiments. (Note that we first checked whether our data were normally

distributed and it was found that they were approximately normal. In any case our results have been confirmed using the nonparametric Wilcoxon test.)

5.4.1 Contribution of Simple Identifiers

The aim of this set of experiments was to assess the degree to which the simple identifiers defined by the programmer contributed to authorship identification. This category included all simple variable names that were defined within the program by the programmer. For example, variable names of type `int`, `int[]`, `long`, `char`, `boolean` and so on.

All instances of these names were changed to a unique identifier comprised of a letter and a number followed by a different letter and the same number. This task was performed on both benchmark datasets, OSJava and StudentJava, resulting in two datasets named OSSimple and StudentSimple. An example of one such unique identifier after disguising would be `a15b15`. If the same variable name was used in more than one program, these were changed to different unique identifiers in order to eliminate the common byte level n-grams based on these variables (thus creating a conservative test). User defined Identifiers that remained unchanged for this experiment, were all class variables and method names.

The results achieved on the OSSimple data set are shown in Table 5.4. By comparing these results to those obtained from the analysis of the OSJava benchmark, it can be observed that accuracy remained the same in 17 of the 56 OSSimple n-L cases, it was improved in 23 cases and in the remaining 16 cases accuracy was worse. As illustrated in Table 5.14 the difference between the two means is negative, meaning that (on average) the ability to classify authorship correctly improved slightly with the disguising of simple identifiers. Using the one-tailed t-test to compare the mean levels of accuracy achieved with the OSJava and OSSimple data sets we found that the p-value

was 0.0989, greater than our threshold of 0.05. Therefore the difference between the accuracy levels across the two data sets (at mean -1.06) is not statistically significant.

The results achieved on the StudentSimple data set are shown in Table 5.5. Comparing the results obtained from this analysis with the StudentJava benchmark results (comparing Tables A2 and A4), we found that classification accuracy was unaffected in 25 of the 56 cases, in 26 cases we attained better results and in 5 cases classification accuracy decreased – thus again suggesting that classification accuracy improved with the disguising of simple identifiers. Using the one-tailed t-test of the difference between the StudentJava and StudentSimple mean accuracy (-1.72 as shown in Table 5.15), we found that the p-value was 0.0000. The difference between the two data sets is in this case statistically significant.

The conclusion drawn from these rather mixed results is that, when analyzed using the SCAP method, simple variables in these programs do not play a significantly positive role in authorship attribution. The apparent improvement in accuracy achieved for many of the n, L combinations across both data sets appears to be due to the fact that many programmers use the same (or similar) names for simple variables. Some examples of the commonly used names encountered across different programmers were `year`, `e`, `f`, `i`, `mid`. The byte-level n -grams derived from these commonly used names were responsible for the initially incorrect classification of some programs in the benchmark analyses, particularly in the StudentJava data set. By making each user-defined simple variable unique in each program, we eliminated all these common n -grams across the different programmers, leading to an apparent improvement in overall classification accuracy.

Table 5.4. Accuracy of classification for the OSSimple data set.

Profile Size (L)	n-gram Size							
	3	4	5	6	7	8	9	10
2000	47.1-	64.7+	76.5+	76.5	82.4+	82.4	70.6-	82.4
3000	47.1+	70.6-	76.5+	70.6-	82.4+	76.5-	76.5	76.5-
4000	47.1+	52.9-	76.5+	82.4	76.5	82.4	76.5	76.5
5000	47.1+	58.8-	76.5+	76.5-	82.4	76.5-	82.4	76.5-
6000	47.1+	58.8-	82.4+	82.4-	88.2+	82.4	82.4+	88.2
7000	47.1+	58.8-	82.4+	82.4-	88.2+	82.4	70.6	82.4+
8000	47.1+	58.8-	82.4+	82.4-	88.2+	82.4	70.6	82.4+

Table 5.5. Accuracy of classification for the StudentSimple data set.

Profile Size (L)	n-gram size							
	3	4	5	6	7	8	9	10
2000	76.9+	80.8+	84.6+	88.5+	88.5+	84.6	84.6+	80.8+
3000	73.1+	76.9	80.8+	84.6+	88.5+	88.5	84.6	84.6
4000	73.1+	76.9-	76.9	84.6+	80.8+	80.8	80.8	84.6
5000	73.1+	76.9-	76.9	76.9	84.6+	84.6+	84.6+	80.8
6000	73.1+	76.9-	76.9	76.9	76.9	80.8+	84.6+	80.8
7000	73.1+	76.9-	76.9	76.9	76.9	80.8+	80.8	80.8+
8000	73.1+	76.9-	76.9	76.9	76.9	80.8+	80.8	80.8

5.4.2 Contribution of Class Identifiers

In the second set of Identifier experiments each name in the training and test program sets from both data sets (OSJava and StudentJava) that pertained to the Class Identifiers category was changed to a unique identifier following the same pattern as above. Again, if the same class identifier was used in two different files, then it was changed to two different unique names. An example could be the class name 'owner' that was used (perhaps by a certain programmer) in two different programs. It was changed to 'a123b123' in the first program and 'a34b34' in the second. The data sets thus derived are referred to as OSClass and StudentClass. (All the names that are either simple variables or method names remained unchanged.).

The accuracy results obtained in this experiment from the OSClass data set are shown in Table 5.6. Comparing these results with the OSJava benchmark

results, it can be seen that in 33 of the 56 cases we had poorer author attribution performance, in 18 cases the same level of accuracy was achieved, and in 5 cases we achieved improved accuracy after identifier neutralization. The p-value obtained from the one-tailed t-test comparing the OSJava and OSClass mean accuracy values (being 4.42) was 0.0000, indicating that the disguising of class identifiers makes a statistically significant difference to classification accuracy (Table 5.14).

The results obtained in this experiment from the StudentClass data set are shown in Table 5.7. When these results are compared with those obtained for the StudentJava benchmark experiment, it can be seen that in 18 of the 56 cases we had poorer attribution performance, in 37 cases the same level of accuracy was achieved, and in 1 case we achieved an improved result. The p-value obtained from the one-tailed t-test comparing the mean accuracy levels achieved for the StudentJava and StudentClass data sets (at 1.31) was 0.0000 (Table 5.15), illustrating that once again the difference in accuracies between the two data sets is statistically significant.

The results from these two experiments consistently indicate that class variables do appear to reflect program authorship. While it is true that the mean difference between the StudentClass and StudentJava data sets is smaller than that found between the OSJava and OSClass sets, this is due in part to the fact that identifier naming in the StudentJava data set had been influenced to a degree by the instructor and the domain (being the same in most programs). In addition some of the programs had been plagiarised. Each of these factors would increase the likelihood of finding the same or similar class names across programs from different authors. Even when these inherently similar names were replaced by the unique identifiers (thus eliminating some of the common n-grams between programs from different authors), accuracy only improved in a single case. Thus, we can be reasonably

confident that class naming (i) is influential in reflecting authorship, and (ii) is reasonably robust to external influence and potentially to manipulation associated with the masking of plagiarism.

Table 5.6 Accuracy of classification for the OSClass data set.

Profile Size (L)	n-gram size							
	3	4	5	6	7	8	9	10
2000	58.8+	58.8	70.6	64.7-	76.5+	70.6-	82.4	82.4
3000	35.3-	58.8-	58.8-	64.7-	70.6-	76.5-	76.5	88.2+
4000	35.3-	58.8-	58.8-	70.6-	76.5	76.5-	76.5	76.5
5000	35.3-	58.8-	64.7-	70.6-	82.4	82.4	76.5-	82.4
6000	35.3-	58.8-	70.6	76.5-	76.5-	82.4	76.5	82.4-
7000	35.3-	58.8-	70.6	76.5-	70.6-	70.6-	70.6	82.4+
8000	35.3-	58.8-	70.6	76.5-	70.6-	64.7-	70.6	82.4+

Table 5.7. Accuracy of classification for the StudentClass data set.

Profile Size (L)	n-gram size s							
	3	4	5	6	7	8	9	10
2000	69.2	76.9	80.8	84.6	80.8-	80.8-	76.9-	76.9
3000	69.2	76.9	76.9	80.8	80.8	80.8-	80.8-	76.9-
4000	69.2	76.9-	76.9	73.1-	80.8+	80.8	80.8	80.8-
5000	69.2	76.9-	76.9	76.9	76.9	80.8	76.9-	80.8
6000	69.2	76.9-	76.9	76.9	76.9	76.9	76.9-	80.8
7000	69.2	76.9-	76.9	76.9	76.9	76.9	76.9-	76.9
8000	69.2	76.9-	76.9	76.9	76.9	76.9	76.9-	76.9-

5.4.3 Contribution of Method Identifiers

This set of experiments was performed to evaluate the degree to which method names contributed to accurate authorship attribution. All method names defined within the programs in both the OSJava and StudentJava data sets were changed to unique identifiers. If the same method name appeared in more than one program, it was replaced by a different identifier in each case. All identifiers that were either simple variables or class variables were left unaffected. The resulting data sets are referred to as OSMethod and StudentMethod respectively.

The authorship attribution results for the experiment on the OSMethod data set are shown in Table 5.8. Comparing these outcomes with the OSJava benchmark, the results are worse in 15 of the 56 cases, in 27 cases performance was improved and in 14 cases the same levels of accuracy were achieved. The next step was to perform a one-tailed t-test to evaluate the difference between the accuracies achieved using the OSJava and OSMethod programs (at a value of -1.06). The p-value of this test was found to be 0.0987, higher than our threshold for p of 0.05 (Table 5.14). This shows that the difference between the accuracies in these data sets is not significant.

The results achieved from the analysis of the StudentMethod data set are given in Table 5.9. Comparing these levels of accuracy against those obtained for the StudentJava benchmark, the results are worse in 5 of the 56 cases, in 37 cases performance was improved and in 14 cases the same levels of accuracy were achieved. The p-value obtained by comparing the StudentJava and StudentMethod mean accuracy values (at -2.62) was 0.0000 (see Table 5.15). This shows that the SCAP analysis of the StudentMethod data set, with its method names disguised, achieved statistically better classification accuracy than achieved through the analysis of the StudentJava data set.

The improvement in accuracy in both sets, in spite of the disguising of method names, is again explained in part by the fact that the unique identifiers that replaced the user-defined method names eliminated some of the common n-grams that were derived from coincidentally common or similar method names used by different programmers (and that negatively affected the level of correct classification achieved in the benchmark tests). Examples of such names included `getInteger`, `setString`, `init set`. The degree of improvement observed after identifier neutralization is greater in the instructor-influenced single domain StudentJava data set because the number of common method names used by different programmers is higher

in this set. The conclusion drawn from these results is that method names defined by the user in these Java programs do not play a significantly positive role in authorship attribution using the SCAP method.

Table 5.8 Accuracy of classification for the OSMethod data set.

Profile Size (L)	n-gram Size							
	3	4	5	6	7	8	9	10
2000	58.8+	64.7+	76.5+	76.5	76.5+	82.4+	82.4	82.4
3000	47.1+	64.7-	76.5+	76.5	82.4+	76.5-	76.5	82.4
4000	47.1+	58.8-	76.5+	88.2+	82.4+	82.4	82.4+	82.4+
5000	47.1+	58.8-	82.4+	88.2+	88.2+	82.4	82.4	82.4
6000	47.1+	58.8-	76.5+	82.4-	82.4	76.5-	70.6-	82.4-
7000	47.1+	58.8-	76.5+	76.5-	82.4	70.6+	70.6	82.4+
8000	47.1+	58.8-	76.5+	76.5-	82.4	70.6+	76.5+	82.4+

Table 5.9 Accuracy of classification for the StudentMethod data set.

Profile Size (L)	n-gram size							
	3	4	5	6	7	8	9	10
2000	73.1+	80.8+	80.8	84.6	84.6	84.6	84.6+	84.6+
3000	73.1+	76.9	80.8+	84.6+	84.6+	88.5	88.5+	88.5+
4000	73.1+	76.9-	76.9	88.5+	80.8+	84.6+	88.5+	88.5+
5000	73.1+	76.9-	76.9	80.8+	80.8+	84.6+	84.6+	88.5+
6000	73.1+	76.9-	76.9	80.8+	80.8+	80.8+	80.8	84.6+
7000	73.1+	76.9-	76.9	80.8+	80.8+	80.8+	84.6+	84.6+
8000	73.1+	76.9-	76.9	80.8+	80.8+	80.8+	80.8	80.8

5.4.4 Contribution of all User defined Identifiers

One further experiment was conducted to assess the impact of neutralizing all names, belonging to all three categories examined above. In this experiment all identifiers including simple variables, method names and class variables defined by the programmer within each program in the OSJava and StudentJava datasets have been replaced by unique identifiers. If the same name appeared in more than one program, it was replaced by a different identifier in each case. The purpose of this experiment was to assess the extent of influence that all names used within a program had on authorship

attribution. The data sets derived are referred to as OSAll and StudentAll correspondingly.

The results achieved with the OSAll data set are shown in Table 5.10. By comparing these results to those obtained from the analysis of the OSJava benchmark, it can be observed that accuracy remained the same in 17 of the 56 cases and was in fact improved in the other 39 cases. This indicates that, in this case, the names defined by the users did not contribute positively to authorship attribution. The p-value obtained in comparing the OSJava and OSAll mean accuracies using a paired-sample t-test was 0.0000 (Table 5.14). This shows that the levels of classification accuracy obtained from the two analyses are significantly different.

Similarly the results from the StudentAll data set are shown in Table 5.11. A comparison of the StudentJava and StudentAll classification performance reveals that accuracy was the same in 1 of the 56 cases and was improved in the other 55. This again provides evidence that the user-defined names in these programs did not contribute positively to authorship attribution. The t-test p-value obtained in comparing the two mean accuracies was 0.0000, suggesting that the levels of accuracy achieved in the analysis of the two data sets are significantly different (Table 5.15).

The improvement in classification accuracy obtained *after* the disguising of identifiers is highest in this last experiment, for both data sets – not unexpected given the results obtained in the three preceding tests. Examination of the analyses revealed that, as for the prior experiments, this improvement can be explained by the fact that programs written by different programmers contained the same or similar names (for example `value` and `val`, or `fragment`, `fragmentation`, `fragmentname` and `fragments`). The byte-level n-grams derived from these commonly used names were responsible for the originally incorrect classification of some programs in the

benchmark analyses. By making each user-defined identifier unique in each program we eliminated all these common n-grams across the different programmers, thus improving overall classification accuracy when compared to the two benchmark sets.

Table 5.10 Accuracy of classification for the OSAll data set.

Profile Size(L)	n-gram size							
	3	4	5	6	7	8	9	10
2000	52.9	76.5+	76.5+	82.4+	82.4+	88.2+	82.4+	82.4+
3000	47.1+	76.5 +	88.2+	88.2+	88.2+	82.4+	76.5+	88.2+
4000	47.1+	64.7	82.4+	88.2+	88.2+	88.2+	88.2+	88.2+
5000	47.1+	64.7	76.5+	88.2+	88.2+	82.4+	88.2+	88.2+
6000	47.1+	64.7	76.5+	88.2+	88.2+	88.2+	82.4+	88.2+
7000	47.1+	64.7	76.5+	88.2+	82.4+	88.2+	88.2+	88.2+
8000	47.1+	64.7	76.5+	88.2+	82.4+	88.2+	88.2+	88.2+

Table 5.11 Accuracy of classification for the StudentAll data set.

Profile Size(L)	n-gram size							
	3	4	5	6	7	8	9	10
2000	88.5+	84.6+	88.5+	92.3+	92.3+	88.5+	84.6+	80.8+
3000	84.6+	88.5+	88.5+	88.5+	92.3+	92.3+	84.6	88.5+
4000	84.6+	84.6+	88.5+	92.3+	88.5+	88.5+	88.5+	88.5+
5000	84.6+	84.6+	80.8+	88.5+	92.3+	92.3+	88.5+	88.5+
6000	84.6+	84.6+	80.8+	80.8+	88.5+	88.5+	92.3+	88.5+
7000	84.6+	84.6+	80.8+	80.8+	84.6+	88.5+	88.5+	88.5+
8000	84.6+	84.6+	80.8+	80.8+	84.6+	84.6+	88.5+	88.5+

5.5 Summary of Performance

We here provide a set of tables that show a summary of the classification accuracy results achieved for the four OS data set and the four StudentJava set experiments across the various combinations of SCAP profile parameter values. Tables 5.12 and 5.13 present in summary form the results obtained from all experiments described in this section. Tables 5.14 and 5.15 show the results of the one-tailed t-test paired difference between the OSJava/StudentJava and corresponding identifier type data sets.

Table 5.12 Performance summary of the OSJava and corresponding identifier type data sets

	OSJava	OSSimple	OSClass	OSMethod	OSAll
Mean classification accuracy	72.3%	73.3%	67.9%	73.3%	78.4%
Median classification accuracy	76.5%	76.5%	70.6%	76.5%	82.4%
Minimum classification accuracy	41.2%	47.1%	35.3%	47.1%	47.1%
Maximum classification accuracy	88.2%	88.2%	88.2%	88.2%	88.2%
Std. Deviation	13.3%	12.7%	13.9%	12.1%	13.6%
Worse than OSJava		16	33	15	0
Better than OSJava		23	5	27	39
Same as OSJava		17	18	14	17

Table 5.13 Performance summary of the StudentJava and corresponding identifier type data sets

	StudentJava	StudentSimple	StudentClass	StudentMethod	StudentAll
Mean classification accuracy	78.3%	80.0%	77.0%	80.9%	86.8%
Median classification accuracy	76.9%	80.8%	76.9%	80.8%	88.5%
Minimum classification accuracy	69.2%	73.1%	69.2%	73.1%	80.8%
Maximum classification accuracy	88.5%	88.5%	84.6%	88.5%	92.3%
Std. Deviation	4.4%	4.3%	3.6%	4.6%	3.5%
Worse than StudentJava		5	18	5	0
Better than StudentJava		26	1	37	55
Same as StudentJava		25	37	14	1

Table 5.14 One-tailed t-test paired difference between the OSJava and corresponding identifier type data sets

	Mean	Std. Deviation	p-value
OSJavaAccuracy - OSSimpleAccuracy	-1.06	6.06	0.0989
OSJavaAccuracy - OSClassAccuracy	4.42	5.64	0.0000
OSJavaAccuracy - OSMethodAccuracy	-1.06	6.05	0.0987
OSJavaAccuracy - OSAllAccuracy	-6.07	5.48	0.0000

Table 5.15 One-tailed t-test paired difference between StudentJava and corresponding identifier type data sets

	Mean	Std. Deviation	p-value
StudentJavaAccuracy - StudentSimpleAccuracy	-1.72	2.94	0.0000
StudentJavaAccuracy - StudentClassAccuracy	1.31	2.24	0.0000
StudentJavaAccuracy - StudentMethodAccuracy	-2.62	3.14	0.0000
StudentJavaAccuracy - StudentAllAccuracy	-8.47	4.43	0.0000

5.6 Summary

We have performed a number of experiments in order to assess the impact of different Java identifier types on source code authorship attribution, using the Source Code Author Profile approach. In these experiments, programs from two different Java data sets with different characteristics were used. The first data set contained open source code and programs that were ‘domain independent’ since all programs from each author that were placed in the training set were from a different project than the programs placed in the test set. Hence, the programs in this data set did not share common characteristics. In contrast, the second data set was formed by programs written during an introductory Java course, the consequence being that naming in these programs was influenced by the instructor and that some program samples had been plagiarized. In addition most programs in this data set belonged to the same application domain. As a result the programs in this data set shared several common characteristics and identifiers.

In each experiment one category of identifiers was neutralized, in order to provide a means of measuring the difference between classification accuracy with and without the certain type of identifier available. The results of these experiments (presented in summary form in Tables 5.12 to 5.15) have shown the following for the data sets assessed here:

- Simple variables and method names defined by the programmer do not seem to positively influence classification accuracy – and in fact in some cases accuracy could be improved if these names were neutralized before the SCAP analysis. This is due to the fact that programmers have been shown to use the same or similar names for simple variables and method names. This conclusion applied to both Java data sets considered here but to a lesser extent for the OSJava data set, where the

programs were from a different application domain for each programmer.

- Class naming does positively influence authorship classification accuracy, an outcome evident for programs in both data sets.
- Accuracy classification is improved by neutralizing all user-defined identifiers in both data sets.

At the outset of this study we asked the following questions: Do Java identifiers contribute to correct authorship identification? Is it possible to hide the provenance of some Java program by identifier renaming? The results of our analyses suggest that the answer to the first question is a partial 'Yes', Java *class* identifiers do contribute to correct authorship identification. The answer to the second question appears to be 'No' – it is not possible to hide the provenance of some Java program by identifier renaming. In fact, globally renaming all identifiers – neutralizing them – enabled us to actually improve our authorship classification accuracy over the benchmark levels achieved with identifiers intact.

Chapter 6 Conclusions

Nowadays, in a wide variety of cases source code authorship identification has become an issue of major concern. Such situations include authorship disputes, proof of authorship in court, cyber attacks in the form of viruses, trojan horses, logic bombs, fraud, and credit card cloning etc. Identifying the authorship of malicious or stolen source code in a reliable way has become a common goal for digital investigators. Zheng et al. (2003) proposed the adoption of an authorship analysis framework in the context of cybercrime investigation to help law enforcement agencies deal with the identity tracing problem.

In this context, the goals of this research work were:

- to describe and compare all the previous studies in the field of source code authorship identification.
- to develop a new approach to source code authorship identification that will eliminate some of the limitations of the previous methodologies.
- to identify the features of the source code that contributes to correct authorship identification.

The following sections contain conclusions reached on each of the above studies.

6.1. Description and comparison of previous studies

The most extensive and comprehensive application of authorship analysis is in literature. One famous authorship analysis study is related to Shakespeare's works and is dating back over several centuries.

The general methodology of authorship attribution applies to texts in both natural and computing languages. This authorship attribution methodology requires two main steps. The first step is the extraction of data for selected features that are said to represent each author's style. The second step normally involves the application of a statistical or machine learning algorithm to these variables in order to develop models that are capable of discriminating between potentially several authors.

In general, when authorship attribution methods have been developed for programming languages, the software features used are language-dependent and require computational cost and/or human effort in their derivation and calculation. The main focus of the early approaches was on the definition of the most appropriate features in representing the style of an author.

While the metric extraction approach to software forensics has been dominant for the last decade it is not without its limitations. The first is that at least some of the software metrics collected are programming-language dependent. For example, metrics specifically appropriate to Java programs are not inherently useful for examining C or Pascal programs – some may simply not be available from programs written in a different language. The second limitation is that the selection of useful metrics is not a trivial process and usually involves setting (possibly arbitrary) thresholds to eliminate those metrics that contribute little to a classification or prediction model. Third, some of the metrics are not readily extracted automatically because they

involve judgments, adding both effort overhead and subjectivity to the process.

In sum, the previous work in author identification of programming code has exhibited varying degrees of language-dependence and has achieved a range of levels of effectiveness.

6.2. Development of a new approach to source code authorship identification

We have developed the Source Code Author Profiles (SCAP) methodology that represents a new approach to source code authorship identification and classification that is both highly effective and language-independent, since it is based on low-level non-metric information. In this method, byte-level n-grams are utilised to establish and assess code against author profiles. Our method was applied to data sets of different programming languages (C++, Java and Common Lisp) and varying difficulty demonstrating surprising effectiveness.

The conclusions reached in relation to the SCAP method are as follows:

- A comparison with a previous source code authorship identification study based on more complicated information shows that the n-gram author profiles are better able to capture the idiosyncrasies of the source code authors.
- One of the inherent advantages of this approach over others is that it is language independent since it is based on low-level information.
- Experiments with data sets in Java and C++ and Common Lisp have shown that it is highly effective in terms of classification accuracy.
- Comments alone can be used to identify the most likely author in open-source code samples, where there are detailed comments in each

program sample. Furthermore, the SCAP method can also reliably identify the most likely author even when there are no comments in the available source code samples.

- The SCAP approach can deal with cases where very limited training data per author is available or there are multiple candidate authors, with no significant compromise in performance.
- Many experiments are required in order to identify the most appropriate combination of n-gram size n and profile size L .

6.3. The significance of high-level programming features in source code authorship identification

The question we addressed here is: which are the features of the source code that contribute to correct authorship identification? What is it about that piece of code that suggests a particular author? A number of experiments have been performed in order to answer the questions above using programs written in two languages that represent two different programming styles we: Java, which uses objects, and Common Lisp, which uses a functional/imperative programming style. We intentionally selected languages that represent two different programming styles, so that insights into a range of languages might be gained. Given language similarities it could be expected that programs written in C++ would have similar results to those achieved with Java code, and Prolog programs should behave similarly to Lisp programs.

In each case one feature at a time was either removed or 'neutralised', in order to provide a means of measuring the difference between classification accuracy with and without the feature available. The results of these experiments have shown the following for the data sets assessed here:

- The accuracy of source code authorship attribution is improved by the existence of comments in the code.
- Layout-related features play a role in determining program authorship but the extent to which this is an influential characteristic may vary from language to language. In our experiments, the level of impact for the programs written in Java was substantial, but this level was much lower for the programs written in Common Lisp. (The contribution of layout-related features in identifying the author of a Java program is also a conclusion reached by Ding and Samadzadeh (2004).)
- Variable and function names defined by the programmer do not seem to influence classification accuracy – and in fact in some cases accuracy might be improved by ‘neutralizing’ these names. This is due to the fact that programmers have been shown to use the same names for simple variables, class variable names, methods or functions. In our case, this conclusion certainly applied to the Java programs, and to those written in Common Lisp to a lesser extent.
- Package-related naming influences accuracy, an outcome evident for programs written in both languages.

One of the implications of our work is that future authorship identification systems, which are intended to explain ‘why’ it is claimed that a piece of code is written by a particular author, should concentrate on the features that are the most important in determining authorship based on the findings of this study.

On the other hand, systems that deal with plagiarism detection could use the findings of our work in order to locate the features of a piece of code that could be plagiarised. For example, when looking for plagiarism in a piece of code written in Java one should first concentrate on the comments and the

layout of the program and not on the user defined identifiers which might be otherwise one of the most obvious first choices.

6.3.1 The significance of user-defined identifiers in Java source code authorship identification

While the work described above has indicated that Java identifiers defined by the programmer do not influence classification accuracy, and in fact in some cases accuracy might be improved by ‘neutralizing’ these variable names, that study examined all user-defined identifiers together. Further experiments have been performed to check whether this conclusion holds when we examine each type of Java programmer-defined identifier separately. In these experiments, programs from two different Java data sets with different characteristics were used. The first data set contained open source code and programs that were ‘domain independent’ since all programs from each author that were placed in the training set were from a different project than the programs placed in the test set. Hence, the programs in this data set did not share common characteristics. In contrast, the second data set was formed by programs written during an introductory Java course, the consequence being that naming in these programs was influenced by the instructor and that some program samples had been plagiarized. In addition most programs in this data set belonged to the same application domain. As a result the programs in this data set shared several common characteristics and identifiers.

In each experiment one category of identifiers was neutralized, in order to provide a means of measuring the difference between classification accuracy with and without the certain type of identifier available. The results of these experiments have shown the following for the data sets assessed here:

- Simple variables and method names defined by the programmer do not seem to positively influence classification accuracy – and in fact in some cases accuracy could be improved if these names were neutralized before the SCAP analysis. This is due to the fact that programmers have been shown to use the same or similar names for simple variables and method names. This conclusion applied to both Java data sets considered here but to a lesser extent for the OSJava data set, where the programs were from a different application domain for each programmer.
- Class naming does positively influence authorship classification accuracy, an outcome evident for programs in both data sets.
- Accuracy classification is improved by neutralizing all user-defined identifiers in both data sets. This conclusion has also been reached when in the previous experiment where we examined all user-defined identifiers together.

One of the implications of our work is that future Java authorship identification systems that are intended to explain why it is claimed that a piece of code is written by a particular author should concentrate on the class identifiers in analysing and assigning authorship. More broadly, identifier neutralization could be used as a means of improving accuracy in Java authorship identification cases. In contexts in which identifiers might be named in ‘standard’ ways the masking of identifiers (perhaps apart from class names) should be performed before authorship analysis is undertaken.

6.4. Future Work

The research work described in this thesis has revealed a number of open issues that could be investigated in the future:

- All the experiments to classify a program to an author have been performed using a number of different combinations of n-gram size n and profile size L . Although these experiments have indicated some optimum combinations of the n-gram size n and profile size L , more experiments have to be performed on various data sets in order to be able to define the most appropriate combination of n-gram size and profile size for a given problem.
- Further work could be undertaken for the development of a statistical likelihood which we can attach to the yes/no classification results, since courts are not only interested in the accuracy rates of methods such as SCAP, but also the likelihood of a particular classification for a particular set of programs in a particular case.
- Another useful direction worthy of research investigation would be the discrimination of different programming styles – and authors – in collaborative and community-authored projects.
- Analysis of code written in other languages would add to our understanding of the influence of particular programming features – as the SCAP method is language-independent it is ideally suited to such work.
- Further research could include applying the SCAP approach to programs written by the same authors in different languages.
- We have performed a number of experiments in order to assess the impact of different Java identifier types on source code authorship attribution, using the Source Code Author Profile approach. Future work could include research on other specific programming languages, in order to check in detail whether our findings are language-or data set-specific.

References

Aarmodt, A., and Plaza, E., 1994. Case-Based Reasoning: Foundational issues, Methodical Variations and System Approaches. *AI Communications*, vol 7(1)

Abbasi, A., and Chen., H., 2005. Applying Authorship Analysis to Extremist-Group Web Forum Messages, *IEEE Intelligent Systems* 20(5): 67-75.

Abelson, H., and Sussman., G., J., 1996. Structure and interpretation of computer programs. MIT Press, Cambridge, Mass., second edition.

Adnan, El-N., Veermachaneni, S., Nagy, G., 2003. Handwriting recognition using position sensitive letter n -gram matching, *Proceedings of the Seventh International Conference on Document Analysis and Recognition (ICDAR 2003)*.

Arensburger, A., 2001 */* You Are Expected to Understand This */*.
<http://freshmeat.net/articles/view/238/>.

Baayen, R., H., Van Halteren, H., and Tweedie., F. J., 1996. Outside the cave of shadows: Using syntactic annotation to enhance authorship attribution. *Literary and Linguistic Computing*, 11(3):121–131.

Benedetto, B., Caglioti, E., Loreto, V., 2002. Language trees and zipping. *Physical Review Letters* 88(048702) (2002).

Burrows, J., F., 1989. "An ocean where each kind..." Statistical Analysis and some major determinants of literary style. *Computers and the Humanities* 23, 309-321

Burrows, J., F., 1987. Word patterns and story shapes: The statistical analysis of narrative style. *Literary and Linguistic Computing*, 61-67.

Cargill, T., 1992. *C++ Programming/Code Style*, Addison-Wesley.

Cavnar, W.,B., Trenkle, J.,M., 1994. N-Gram-based text categorization, *Proceedings of the 1994 Symposium on Document Analysis and Information Retrieval*.

Chaski, C., 1998. A Daubert-inspired assessment of current techniques for language-based author identification". Technical report, US National Institute of Justice. Available through www.ncjrs.org.

Chaski, C., 2001. Empirical evaluations of language-based author identification techniques". *Forensic Linguistics*.

Chaski, C.E., 2005. Who's At the Keyboard? Recent results in authorship attribution, *International Journal of Digital Evidence*, 4(1). Available at www.ijde.org

Cheng, B.,Y., Carbonell, J.,G., Klein-Seetharaman, J., 2005. Protein classification based on text document classification techniques, *Proteins* 58, 955–970.

Dewhurst, S., C., 2002. *C++ Gotchas: Avoiding Common Problems in Coding and Design*. Addison-Wesley Professional.

Diederich, J., Kindermann, J., Leopold, E., and Paass. G., 2000. Authorship attribution with Support Vector Machines". *Applied Intelligence*, 19, 109-123.

Dietrich, W., R., 2003. *Applied Pattern Recognition: Algorithms and Implementation in C++*. Springer.

Ding, H., Samadzadeh, M. H., 2004. Extraction of Java program fingerprints for software authorship identification, *The Journal of Systems and Software*, 72(1): 49-57.

Dixit, J.B. 2003. *Computer Fundamentals and Programming in C*. Laxmi Publications.

Downie, J.,S., 1999. Evaluating a simple approach to musical information retrieval: conceiving melodic *n*-grams as text, PhD thesis, University of Western Ontario.

Elliott, W., E., Y., and Valenza, R., J., 1991. Was the Earl of Oxford the true Shakespeare? A computer aided analysis. *Notes and Queries*, 236:501–506.

Farrington, J., M., Morton, A., Q., and Farrington, M., G., 1996. *Analysing for Authorship: A Guide to the Cusum Technique*, University of Wales Press, Cardiff.

Floyd, R. W., Beigl, R., 1994. *The language of Machines*, New York: Computer Science Press.

Frank, E., Chui, C., Witten, I.H., 2000. Text Categorization Using Compression Models. Proc. of DCC-00, IEEE Data Compression Conference (2000) 200–209.

Frantzeskou, G., Stamatatos, E., Gritzalis, S., Chaski, C.,E., and Howald B.,S., 2007a. Identifying Authorship by Byte- Byte-Level N-Grams: The Source Code Author Profile Method, *International Journal of Digital Evidence*, 6(1).

Frantzeskou, G., MacDonell, S.G., Stamatatos, E., and Gritzalis, S., 2007b. Examining the significance of high-level programming features in source code author classification, *Journal Systems and Software*, in press, doi:10.1016/j.jss.2007.03.004, Elsevier.

Frantzeskou, G., MacDonell, S.G., Stamatatos, E., and Gritzalis, S., 2007c. The significance of user-defined identifiers in Java source code authorship identification, *Journal of Information and Software Technology*, Elsevier, submitted for publication.

Frantzeskou, G., Stamatatos, E., Gritzalis, S., and Katsikas, S., 2006a. Effective Identification of Source Code Authors Using Byte-Level Information, in *Proceedings of the 28th International Conference on Software Engineering ICSE 2006 - Emerging Results Track*, B. Cheng, B. Shen (Eds.), Shanghai, China, ACM Press.

Frantzeskou, G., E. Stamatatos, S. Gritzalis, and S., Katsikas 2006b. Source Code Author Identification Based on N-gram Author Profiles In *Proceedings of 3rd IFIP Conference on Artificial Intelligence Applications & Innovations (AIAI'06)*, pp. 508-515, Springer.

Frantzeskou, G., Stamatatos, E., Gritzalis, S., 2005a. Supporting the Digital Crime Investigation Process: Effective Discrimination of Source Code Authors based on Byte-level Information, in *Proceedings of the ICETE'2005 International Conference on eBusiness and Telecommunication Networks – Security and Reliability in Information Systems and Networks Track*, UK, Springer.

Frantzeskou, G., Stamatatos, E., Gritzalis, S., 2005b. "Source Code Authorship Analysis using N-grams", in *Proceedings of the 7th Biennial Conference on Forensic Linguistics*, July 2005, Cardiff, UK

Frantzeskou, G., Gritzalis, S., MacDonell, S., 2004. Source Code Authorship Analysis for supporting the cybercrime investigation process, in *Proceedings of the ICETE'2004 International Conference on eBusiness and Telecommunication Networks – Security and Reliability in Information Systems and Networks Track*, Vol. 2, 85-92, Springer.

Ganapathiraju, M., Weisser, D., Rosenfeld, R., Carbonell, J., Reddy, R., Klein-Seetharaman, J., 2002. Comparative n-gram analysis of whole-genome protein sequences, HLT'02, *Proceedings of the Human Language Technologies Conference*, San Diego.

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

Gray, A., Sallis, P., MacDonell, S., 1997. Software forensics: Extending authorship analysis techniques to computer programs, in *Proc. 3rd Biannual Conf. Int. Assoc. of Forensic Linguists (IAFL'97)*, pages 1-8.

Gray, A., Sallis, P., MacDonell, S., 1998. Identified: A dictionary-based system for extracting source code metrics for software forensics, in *Proceedings of SE:E&P'98*, IEEE Computer Society Press, 252–259.

Grubb, P., A., Takang, A., 2003. *Software Maintenance: Concepts and Practice*, World Scientific.

Heer, T. De, 1974. Experiments with syntactic traces in information retrieval, *Inform. Storage Retrieval* 10, 133–144.

Holmes., D., I., 1998. The evolution of stylometry in humanities scholarship, *Literary and Linguistic Computing*, 13(3):111–117.

Holmes, D., I., and Forsyth, R., 1995. The Federalist revisited: New directions in authorship attribution. *Literary and Linguistic Computing*, 10(2):111–127.

Holmes, D.J., 1992. A stylometric analysis of Mormon scripture and related texts. *Journal of Royal Statistical Society*, 155, 91-120.

IRT Group, CERN, 2000. C++ Coding Standard Specification, Handbook/Programming/CodingStandard/c++standard.pdf

Juola, P., 2006. Authorship attribution for electronic documents, In Olivier and Sheno (Eds.), *Advances in Digital Forensics II*, pp. 119-130, Springer.

Kernighan, B., W., and Ritchie, D., M., 1978. *The C Programming Language*, Prentice-Hall, Inc.

Keselj, V., Peng, F., Cercone, N., Thomas, C., 2003. N-gram based author profiles for authorship attribution, in *Proceedings of Pacific Association for Computational Linguistics*.

Keyes, J., 2003. *Software Engineering Handbook*, Auerbach.

Khmelev, D., V., and Tweedie, F., J., 2002. Using Markov chains for identification of writers, *Literary and Linguistic Computing*, 16(4):299–307.

Khmelev, D., and Teahan, W., 2003. A Repetition Based Measure for Verification of Text Collections and for Text Categorization, in *Proc. of the 26th ACM SIGIR, 2003*, pp. 104-110.

Knuth, D., E., 1997. *The Art of Computer Programming, Volume 1, 3rd Edition*. Boston: Addison-Wesley.

Kokol, P., Podgorelec, V., Zorman, M., Kokol, T., Njivar, T., 1999. Computer and Natural Language Texts – A Comparison Based on Long-Range Correlations, *Journal of the American Society for Information Science*, John Wiley & Sons, 50(14), 1295-1301. Miller, G.A., 1991. *The Science of Words*, New York: Scientific American Library.

Kokol, P., Kokol, T., 1996. Linguistic laws and computer programs, *Journal of the American Society for Information Science*, 47(10), 781-785.

Kothari, J., Shevertalov, M., Stehle, E., and Mancoridis, S., 2007. A Probabilistic Approach to Source Code Authorship Identification, in *Proc. of Third International Conference on Information Technology New Generations (ITNG 2007)*.

Krsul, I., and Spafford, E. H, 1995. Authorship analysis: Identifying the author of a program, in *Proceedings of 8th National Information Systems Security Conference*, National Institute of Standards and Technology, 514-524.

Kilgour, R. I., Gray, A.R., Sallis, P. J., and MacDonell, S. G., 1998. A Fuzzy Logic Approach to Computer Software Source Code Authorship Analysis, in *Proceedings of ICONIP'97*, Springer-Verlag, 865-868.

Lamkins, D., 2004. Successful Lisp: How to Understand and Use Common Lisp, bookfix.com. Also available at <http://psg.com/~dlamkins/sl/>

Lange, R., S., Mancoridis, 2007. Using Code Metric Histograms and Genetic Algorithms to Perform Author Identification for Software Forensics, in Proc of Genetic and Evolutionary Computation Conference (GECCO 2007), Track Real – World Applications 5.

Ledger, G.R., & Merriam 1994. Shakespeare, Fletcher and the two noble Kingmen, *Literary and Linguistic Computing*, 9, 235-248

Longstaff, T. A., and Schultz, E. E., 1993. Beyond Preliminary Analysis of the WANK and OILZ Worms: A Case Study of Malicious Code, *Computers and Security*, 12(1), 61-77.

Lowe, D., and Matthews, R., 1995. Shakespeare vs. Fletcher: A stylometric analysis by Radial Basis Functions. *Computers and the Humanities*, 29:449–461.

MacDonell, S.G., Buckingham, D., Gray, A.R., and Sallis, P.J., 2002. Software forensics: extending authorship analysis techniques to computer programs, *Journal of Law and Information Science* 13(1), 34-69.

MacDonell, S.G, and Gray, A.R., 2001. Software forensics applied to the task of discriminating between program authors, *Journal of Systems Research and Information Systems* 10:113-127.

MacLennan, Bruce, J., 1987. Principles of Programming Languages, Oxford University Press.

Marceau, C., 2000. Characterizing the behaviour of a program using multiple-length *n*-grams, *Proceedings of the 2000 Workshop on New Security Paradigms*, pp. 101–110.

Mc Connell, S., 1993. Code Complete, Microsoft Press.

Meanland, D.,L., 1995. Correspondence analysis of Luke, *Literary and Linguistic Computing*, 10, 171-182

Mendenhall, T., C., 1887. The characteristic curves of composition. *Science*, 9:237–249.

Merriam, T., 1996. Marlowe’s hand in Edward III revisited. *Literary and Linguistic Computing*, 11(1):19–22.

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

Morris, A., and Cherry, L., 1975. Computer Detection of Typographical Errors, *IEEE Transactions on Professional Communication*, 18(1), 54-56.

Mosteller, F., and Wallace, D., L., 1964. Inference and Disputed Authorship: The Federalist, Addison-Wesley Publishing Company, Inc., Reading, MA.

Mozilla.org , 2007. [Mozilla Coding Style Guide](http://www.mozilla.org/hacking/mozilla-style-guide.html),
<http://www.mozilla.org/hacking/mozilla-style-guide.html>

Norvig P., Pitman, K., 1993. Tutorial on Good Lisp Programming Style, in Proc of Lisp users and Vendors conference

Oman, P., and Cook, C., 1989. Programming style authorship analysis, In Seventeenth Annual ACM Science Conference Proceedings, ACM.

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

Peng, F., Shuurmans, D., and Wang S., 2004. Augmenting naive bayes classifiers with statistical language models, Information Retrieval Journal, 7(1): 317-345.

Qi, J., Luo, H., Hao, B., 2004. CVTree: a phylogenetic tree reconstruction tool based on whole genomes, Nucleic Acids Res, 32, 45–47.

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

Schank, R., 1982. Dynamic Memory: A theory of reminding and learning in computers and people, Cambridge University Press.

Schenkel, A., Zhang, J., Zhang, Y., 1993. Long range correlations in human writings, Fractals, 1(1), 47-55.

Seibel, P., 2005. Practical Common Lisp, Apress. Also on line <http://www.gigamonkeys.com/book/>

Shepperd, M. J., and Schofield, C., 1997. Estimating software project effort using analogies, IEEE Transactions on Software Engineering, 23(11), 736-743

Schmitt, J., C., 1991. Trigram-based method of language identification, U.S. Patent 5,062,143.

Sneed, H., 1996. Object-oriented COBOL recycling. In Proc. of the 3rd Working Conference on Reverse Engineering, IEEE Computer Society, pp 169-178.

Solovyev, V.,V., and Makarova, K.,S., 1993. A novel method of protein sequence classification based on oligopeptide frequency analysis and its application to search for functional sites and to domain localization, Comput. Appl. Biosci., 9, 17–24.

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

Spafford, E. H., and Weber, S. A., 1993. Software forensics: tracking code to its authors, *Computers and Security*, 12(6), 585-595.

Spinellis, D., 2003. Code reading: The Open Source Perspective. Addison-Wesley.

Stamatatos, E., Fakotakis, N., Kokkinakis, G., 2001. Computer based authorship attribution without lexical measures. *Computers and the Humanities*, 35(2), 193-214.

Stamatatos, E., Fakotakis N., and Kokkinakis G., 2000. Automatic text categorization in terms of genre and author, *Computational Linguistics*, 26(4), 471-495.

Sun Developer Network, 1999. Code Conventions for the Java Programming Language, <http://java.sun.com/docs/codeconv/>

Sun Microsystems 1997, Code Conventions
java.sun.com/docs/codeconv/CodeConventions.pdf

Sun Microsystems 2007, Java Language Keywords
<http://java.sun.com/docs/books/tutorial/java/nutsandbolts/keywords.html>

The Harlequin Group Ltd 2007, Common Lisp Specification, Available at
<http://www.lisp.org/HyperSpec/FrontMatter/index.html>

Tweedie F., J., Baayen R., H., 1998. How variable may a constant be? Measures of lexical richness in perspective. *Computers and the Humanities*, 32(5):323–352.

Tweedie, F., J., Singh, S., and Holmes, D., I., 1996. Neural network applications in stylometry: The Federalist papers. *Computers and the Humanities*, 30(1):1–10.

Yule G., U., 1938. On sentence-length as a statistical characteristic of style in prose, with applications to two cases of disputed authorship, *Biometrika*, 30:363–390.

Yule, G., U., 1944. The Statistical Study of Literary Vocabulary, Cambridge University Press.

Vel, O., Anderson, A., Corney, M., and Mohay, G., 2001. “Mining E-mail Content for Author Identification Forensics”, *SIGMOD Record Web Edition*, 30(4).

Zipf, G., K., 1932. Selected Studies of the Principle of Relative Frequency in Language, Harvard University Press, Cambridge, MA.

Zheng, R., Qin, Y., Huang, Z., Chen, H., 2003. Authorship Analysis in Cybercrime Investigation, NSF/NIJ Symposium on Intelligence and Security Informatics (ISI'03), Tucson, Arizona, Springer-Verlag Berlin Heidelberg.