Available online at www.sciencedirect.com
ScienceDirect
The Journal of Systems and Software 81 (2008) 447–460
The Journal of Systems and Software
www.elsevier.com/locate/jss

Examining the significance of high-level programming features in source code author classification

Georgia Frantzeskou [a,*], Stephen MacDonell [b], Efstathios Stamatatos [a], Stefanos Gritzalis [a]

[a] Department of Information and Communication Systems Engineering, University of the Aegean, Samos 83200, Greece
[b] School of Computing and Mathematical Sciences, Auckland University of Technology, Private Bag 92006, Auckland 1020, New Zealand

Received 7 October 2006; received in revised form 4 March 2007; accepted 7 March 2007
Available online 14 March 2007

Abstract

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? In this study, we describe a means of 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. The results show that, for these programs, comments, layout features and package-related naming influence classification accuracy whereas user-defined naming, an obvious programmer related feature, does not appear to influence accuracy. A comparison is also made between the relative feature contributions in programs written in the two languages.

Keywords: Authorship; Source code; Program features; Fraud

1. Introduction

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. 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 and Gray, 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.

* Corresponding author. Tel.: +30 693 2337537; fax: +30 227 3024082.
E-mail addresses: gfran@aegean.gr (G. Frantzeskou), smacdone@aut.ac.nz (S. MacDonell), stamatatos@aegean.gr (E. Stamatatos), sgritz@aegean.gr (S. Gritzalis).

0164-1212/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2007.03.004

The use of Source Code Author Profiles (SCAP) represents a new approach to source code authorship identification and classification that is both highly effective (Frantzeskou et al., 2005, 2006) 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. The aim of this paper is to demonstrate the SCAP approach in action with two different styles of programming language and to assess the impact of particular features on the accuracy of authorship attribution. In other words, the question we address here is: which are the features of the source code that contribute to correct authorship identification? This question is very important whenever a need for evidence arises (MacDonell et al., 2002). Cases of litigation arising from disputes over authorship clearly have such a need. When submitting evidence to court, it is vital to be able to say with conviction and proof which features of a program are those that identify an author. For example, we may be able to assert with evidence that programmer A is the author of a disputed program because the layout of that program is highly similar to others written by programmer A, or the variable and function names used closely resemble those used elsewhere by programmer A – and that they do *not* resemble those used by other programmers. In plagiarism cases, we need to know which features of the program prove that code has been plagiarized, and by whom. Is it, for instance, the function names used, the style of comments, or the names of the variables?

A number of high-level features including those just mentioned are considered here. The SCAP approach, based on byte-level features, is used as a tool for assessing the significance of high-level programming features. 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. 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.

The remainder of this paper is organized as follows. Section 2 contains a review of past research efforts in the area of source code authorship analysis, a description of the SCAP approach and a discussion of the high-level features that might influence source code authorship identification. Section 3 describes the two data sets used: the Java data set and the Common Lisp data set. Section 4 details all the experiments performed on the two data sets in order to examine which high-level programming features contribute to authorship identification, and to what degree. Finally, Section 5 summarizes the conclusions drawn by this study and proposes future work directions.

2. Related work on code authorship

The first part of this section describes previous studies in the area of source code authorship analysis and identification. This is followed by a detailed description of the SCAP approach, including conclusions of the method as described in previous work (Frantzeskou et al., 2005, 2006). Also, this subsection includes a discussion about the high-level features that might influence authorship identification.

2.1. Previous studies

Although the programming languages used to produce source code are much more syntactically restrictive than natural languages, there is still a large degree of flexibility available to the programmer when writing code (Krsul and Spafford, 1995). As stated above, the conventional authorship attribution methodology for programming languages requires two main steps (Krsul and Spafford, 1995; MacDonell and Gray, 2001; Ding and Samadzadeh, 2004). 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 (Frantzeskou et al., 2004).

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. For instance, Oman and Cook (1989) focused initially on typographic features of (Pascal) programs and then collected a list of 236 style rules that could be used as a base for extracting metrics dealing with programming style (Oman and Cook, 1991). Spafford (1989) conducted an analysis of the worm program, which infected the Internet on the evening of 2 November 1988, 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. Benander and Benander (1989) built a Cobol style analyzer in order to demonstrate that good programming style improves programmer productivity. Longstaff and Schultz (1993) focused their attention on code structures in their manual analysis of the WANK and OILZ worms which in 1989 attacked NASA and DOE systems. They concluded that three distinct authors worked on the worms and inferred educational backgrounds and programming levels for the alleged authors. More recent work has considered the effectiveness of such features across authorship identification, classification, discrimination

and the like. Further work has addressed the data analysis itself – which statistical or machine learning methods appeared to be the most effective in answering questions over authorship (Krsul and Spafford, 1995; MacDonell and Gray, 2001; Ding and Samadzadeh, 2004).

Ideally, the selected features should have low within-author variability, and high between-author variability (Krsul and Spafford, 1995; Chaski, 1997). For programming languages, previously proposed features (commonly referred to as metrics given their emergence from token-based code analysis) include (Krsul and Spafford, 1995; Kilgour et al., 1998; Ding and Samadzadeh, 2004):

- *Programming layout metrics*: These are metrics that characterise the form and the shape of the code. For example, metrics that measure indentation, placement of comments, and the use of white space.
- *Programming style metrics*: Such metrics include character preferences, construct preferences, statistical distribution of variable lengths, and capitalisation.
- *Programming structure metrics*: These are metrics assumed to be related to the programming experience and ability of the author. For example, such metrics include the statistical distribution of lines of code per function, the ratio of keywords per line of code, the relative frequency of use of complex branching constructs and so on.
- *Linguistic metrics*: Metrics in this category enable the capture of concepts that might be said to reflect the maturity or capability of an author, such as deliberate versus non-deliberate spelling errors, the degree to which code and comments match, and whether identifiers used are meaningful.

The earliest work in software forensics (Spafford and Weber, 1993) focused on a combination of features reflecting data structures and algorithms, compiler and system information, programming skill and system knowledge, choice of system calls, errors, choice of programming language, use of language features, comment style, variable names, spelling and grammar. Sallis et al. (1996) extended 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. This general approach – the extraction of metric features said to reflect an author's profile – has dominated work to date. Its application has been described in a small number of empirical studies.

Krsul and Spafford (1995) automated the process just described in order to identify the author of a program written in C. The study relied on the use of three categories of software metrics, reflecting layout, style and structure. These features were extracted using a software analyzer program from 88 programs written by 29 authors. A tool was developed to visualize the metrics collected and help select those metrics that exhibited little within-author variation but large between-author variation. Discriminant function analysis was applied on the chosen subset of metrics to classify the programs by author. The experiment achieved 73% overall accuracy in classification using leave-one-out validation.

Kilgour et al. (1998) and MacDonell and Gray (2001) examined the authorship of computer programs written in C++. In related work, Gray et al. (1998) developed a dictionary-based system called IDENTIFIED (Integrated Dictionary-based Extraction of Non-language-dependent Token Information for Forensic Identification, Examination, and Discrimination) to enable the extraction, visualisation and analysis of source code metrics for authorship classification and prediction. In MacDonell and Gray's (2001) work, satisfactory results were obtained for C++ programs using case-based reasoning, a feed-forward neural network, and multiple discriminant analysis. The best prediction accuracy – at 88% for 7 different authors – was achieved using case-based reasoning.

In more recently reported work focused on Java source code, Ding and Samadzadeh (2004) investigated the extraction of a set of software metrics that could be used as a so-called 'fingerprint' to identify the author of a program. The contributions of the selected metrics to authorship identification were measured by canonical discriminant analysis. A set of 56 metrics of Java programs was proposed for authorship analysis. Forty-six groups of programs were diversely collected. Classification accuracies were up to 67.2% when the metrics were selected manually, while they were up to 87.0% with the use of canonical variates.

While this 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.

2.2. The SCAP approach

Our approach to source code authorship attribution is called the Source Code Author Profiles (SCAP) approach

and is an extension of a method that has been successfully applied to text authorship identification by Keselj et al. (2003). 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. Fig. 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.

2.2.1. 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.

The SCAP procedure is explained in the following steps and is illustrated in Figs. 2 and 3 (Frantzeskou et al., 2005, 2006). Fig. 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. Fig. 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.)

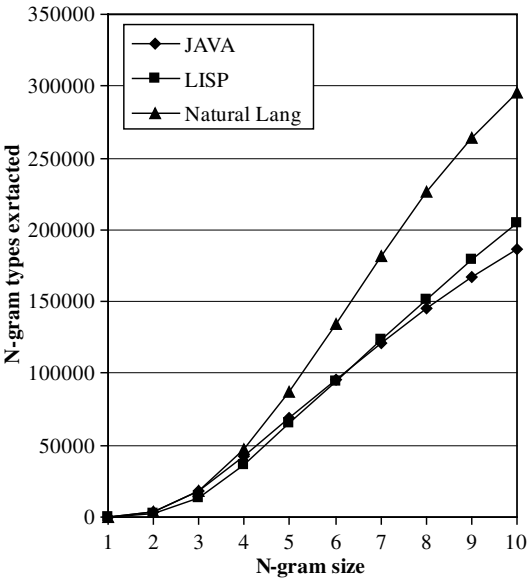

Fig. 1. Total number of *n*-gram types extracted from three files equal in size (1 Java file, 1 Common Lisp, 1 Natural Language), for different sizes of *n*-gram.

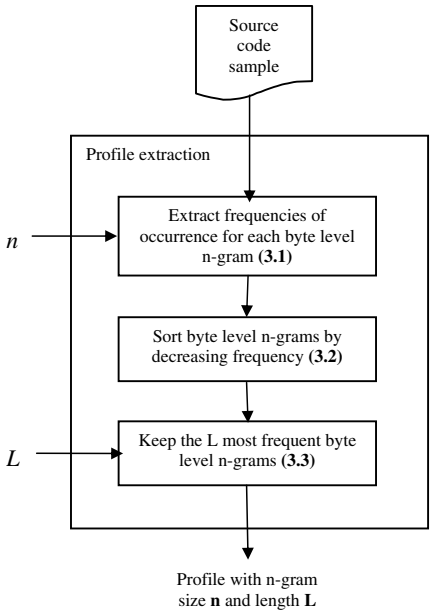

Fig. 2. Extraction of source code profiles for a given *n* (*n*-gram length) and *L* (profile size).

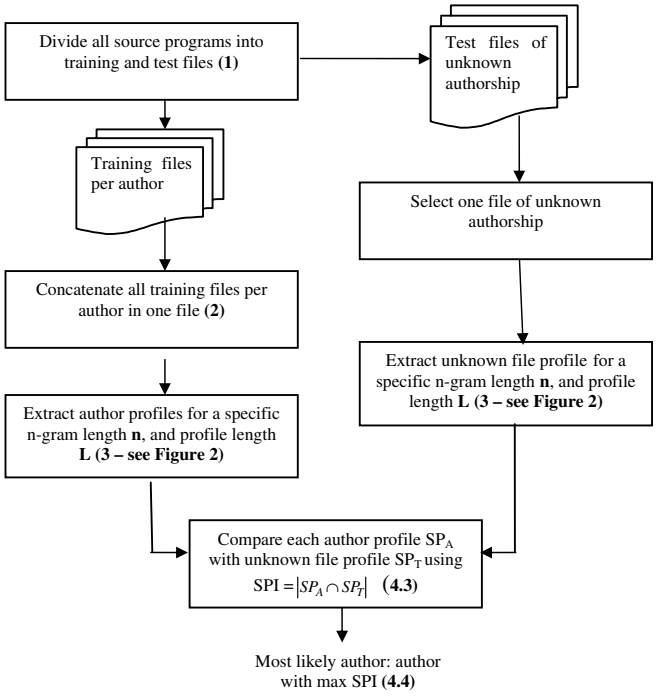

Fig. 3. Estimation of most likely author of an unknown source code sample using the SCAP approach.

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).
   3.3 Keep the L most frequent *n*-grams $\{x_1, x_2, \ldots, x_L\}$. The actual frequency is not used mathematically 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 similarity distance is given by the size of the intersection of the two profiles:

   $$|SP_A \cap SP_T|$$

   In other words, the similarity measure we propose is the amount of common *n*-grams in the profiles of the test case T and the author A.
   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.

2.2.2. Previous outcomes and our current goal

In summary, the conclusions reached in previous research efforts in relation to the SCAP method are as follows (see Frantzeskou et al., 2005, 2006):

- 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.
- Experiments with data sets in Java and C++ 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.

The principal research question addressed in this paper is a follow-on from these outcomes: which are the high-level programming language features that contribute most to correct authorship classification? The provision of evidence to support or refute claims of authorship depends on our ability to answer this question (MacDonell et al., 2002). Additionally, the work in this paper is a further evaluation of the effectiveness of SCAP approach, this time using programs written in languages that represent two different programming styles: Java, which uses objects heavily, and Common Lisp, which uses a functional/imperative programming style.

2.2.3. 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.

3. Datasets and initial empirical analysis

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, 1996). It contains 978 symbols. All programs could use any of these symbols as they are all defined by Common Lisp specification.

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 34. In order to ensure adequate splits of the sample for each author, 16 programs were assigned to the training set and 19 to the test set. The data set is from this point referred to as the CLisp dataset. 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 1 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 1
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
10,000	68.4	84.2	84.2	84.2	78.9	84.2	78.9	78.9

Table 2
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
10,000	35.3	41.2	70.6	76.5	94.1	94.1	94.1	94.1

table. The best results were achieved in three instances, all where $n > 6$ and $L > 3000$. (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).

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 16 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 programs were allocated to the training set and 17 to the test set. This data set is from this point referred to as the Java dataset. The results achieved in the Java all-features benchmark experiment (referred to as the "Java benchmark") with this data set are given in Table 2. As can be seen, accuracy reaches 100% in several cases, many of them for $L > 4000$ and $n = 6$, 7 and 8.

4. Significance of high-level programming features

Focusing on the features said to influence authorship identification, as described in Section 2.2.3, 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.

4.1. Significant features for the Common Lisp data set

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.

4.1.1. 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 `deffun`, `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 1 and 3) 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.

4.1.2. 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

Table 3
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**–
10,000	68.4	68.4–	68.4–	**73.7**–	**73.7**–	**73.7**–	**73.7**–	**73.7**–

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 Table 4.

Comparing the results obtained from this analysis with the CLispNoCom benchmark results (comparing Tables 3 and 4), 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.

4.1.3. 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.

4.1.3.1. 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 5. 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.

4.1.3.2. 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 6. Comparing these results with the CLispNoCom benchmark results, it can be observed that in 34 of the 72 cases we achieved poorer

Table 4
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−
10,000	63.2−	63.2−	63.2−	68.4−	68.4−	68.4−	68.4−	68.4−

Table 5
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−
10,000	47.4−	68.4	68.4	**73.7**	**73.7**	**73.7**	**73.7**	68.4−

Table 6
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−
10,000	63.2−	68.4	73.7+	**79.0+**	**79.0+**	73.7	63.2−	63.2−

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.

4.1.3.3. 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 7. 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.

4.2. 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.

4.2.1. 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 8. 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 2 and 8) 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.

4.2.2. 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 (1999).

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: `==`, `+`,`-`, `*`, `/`,`%`, `+=`, `-=`, `*=`, `/=`,`!=`,`%=`, `=`,`>`,`<`,`>=`, `<=`,`& &` ,`||`,`& `,`|`,`^`,`≪`,`≫`,`≫>`,`& =`,`|=`, `^=`,`≪=`,`≫=`,`≫>=`

Table 7
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−
10,000	57.9−	73.7 +	73.7+	73.7	68.4−	68.4−	63.2−	52.6−

Table 8
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−
10,000	41.2+	64.7+	70.6	**88.2+**	82.4−	82.4 −	70.6−	76.5−

Table 9
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–
10,000	17.6–	29.4–	35.3–	17.6–	35.3–	35.3–	41.2–	35.3–

Table 10
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+**
10,000	47.1+	64.7	76.5+	**88.2**	82.4	**88.2+**	**88.2+**	**88.2+**

The resulting dataset is referred to as JavaLayout. The accuracy levels achieved in author attribution with this data set are shown in Table 9. 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%.

4.2.3. 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.

4.2.3.1. 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 10. 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.

4.2.3.2. 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 11. 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 11
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**−
10,000	41.2	**70.6+**	64.7−	**70.6**−	**70.6**−	64.7−	64.7−	**70.6**−

4.2.3.3. 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 12. 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).

4.3. 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 13 and 14, 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 12
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−
10,000	64.7+	**88.2+**	**88.2+**	**88.2**	**88.2+**	76.5−	64.7−	70.6−

Table 13
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 14
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 15 and 16 (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.

In examining the results presented in Tables 15 and 16, it is evident that, as might be expected, comments play a significant role in authorship attribution, an outcome that

Table 15
High-level features and mean accuracy deviation – Common Lisp programs

Original/Comments	9.0%
NoComments/Layout	3.9%
NoComments/Identifiers:Symbols	1.0%
NoComments/Identifiers:Package	3.1%
NoComments/Identifiers	1.7%

Table 16
High-level features and mean accuracy deviation – Java programs

Original/Comments	7.1%
NoComments/Layout	38.5%
NoComments/Identifiers:User Defined	(6.1%)
NoComments/Identifiers:Package	7.7%
NoComments/Identifiers	(5.1%)

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.

5. Conclusions

A number of experiments have been performed in order to 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, 2006), 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 13–16) 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.

This 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.

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.

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. Finally, another useful direction worthy of research investigation would be the discrimination of different programming styles – and authors – in collaborative and community-authored projects.

References

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

Chaski, C.E., 1997. Who wrote it? Steps toward a Science of Authorship Identification. National Institute of Justice Journal, 15–22. Available from: <www.ncjrs.org> .

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

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.

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

Frantzeskou, G., Gritzalis, S., MacDonell, S., 2004. Source code authorship analysis for supporting the cybercrime investigation process, in: ICETE04, vol. 2, pp. 85–92.

Frantzeskou, G., Stamatatos, E., Gritzalis, S., 2005. Supporting the digital crime investigation process: effective discrimination of source code authors based on byte-level information. Proceedings of the ICETE'2005 International Conference on eBusiness and Telecommunication Networks – Security and Reliability in Information Systems and Networks Track. Springer, UK.

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

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

Keselj, V., 2003. Perl package Text::N-grams. <http://www.cs.dal.ca/~vlado/srcperl/N-grams> or <http://search.cpan.org/author/VLADO/Text-N-grams-0.03/N-grams.pm>.

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.

Kilgour, R.I., Gray, A.R., Sallis, P.J., MacDonell, S.G., 1998. A fuzzy logic approach to computer software source code authorship analysis. Proceedings of ICONIP'97. Springer-Verlag, pp. 865–868.

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

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 50 (14), 1295–1301.

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

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

Lewis, J., Loftus, W., 1998. Java Software Solutions: Foundations of Program Design. Addison-Wesley Longman Inc.

Longstaff, T.A., 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.

MacDonell, S.G., 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.

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

Miller, G.A., 1991. The Science of Words. Scientific American Library, New York.

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

Oman, P., Cook, C., 1991. A Programming style taxonomy. The Journal of Systems and Software 15 (3), 287–301.

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

Sallis, P., Aakjaer, A., MacDonell, S., 1996. Software forensics: old methods for a new science. Proceedings of SE:E&P'96. IEEE Computer Society Press, Dunedin, New Zealand, pp. 367–371.

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/>.

Spafford, E.H., 1989. The Internet worm program: an analysis. Computer Communications Review 19 (1), 17–49.

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

Spinellis, D., 2006. Code Quality: The Open Source Perspective. Addison-Wesley.

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

Sun Microsystems, 1999. Code Conventions for the Java Programming Language. <http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html>.

The Harlequin Group Ltd, 1996. Common Lisp Specification. Available from: <http://www.lisp.org/HyperSpec/FrontMatter/index.html>.

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.

Georgia Frantzeskou is currently pursuing a Ph.D in Software Forensics at the Department of Information and Communication Systems Engineering, University of the Aegean, Greece. She holds a B.Sc. in Mathematics from the University of Athens and a M.Sc. in Computer Science from Aston University, Birmingham UK. During her 10 year long career in the IT industry in Greece and UK, she has been involved in a number of different roles and projects. Some of the projects she has worked on include, the London AirTraffic Control System, Office Automation Systems, Customer Care Systems etc. Her research interests are in the fields of Software Forensics, Software Metrics, and Machine Learning Techniques.

Stephen MacDonell is Professor of Software Engineering and Director of the Software Engineering Research Lab at the Auckland University of Technology (AUT) New Zealand. Stephen teaches mainly in the areas of information systems development, project management, software engineering and software measurement, and information technology research methods. He undertakes research in software metrics and measurement, project planning, estimation and management, software forensics, and the application of statistical, machine learning and knowledge-based analysis methods to complex data sets, particularly those collected in relation to software engineering.

Efstathios Stamatatos is a Lecturer of the Department of Information and Communication Systems Engineering and a member of the Artificial Intelligence lab., at the University of the Aegean, Greece. He received the Diploma degree in Electrical Engineering and the doctoral degree in Electrical and Computer Engineering, both from the University of Patras, Greece. He joined the Polytechnic University of Madrid as a Visiting Researcher and the Austrian Research Institute for Artificial Intelligence as a Post-doc researcher. His research interests include text categorization, natural language processing, and intelligent music processing.

Stefanos Gritzalis is an Associate Professor, the Head of the Department of Information and Communication Systems Engineering, at the University of the Aegean, Greece, and the Director of the Laboratory of Information and Communication Systems Security. He has been involved in several national and EU funded R&D projects in the areas of Information and Communication Systems Security. His published scientific work includes several books on Information and Communication Technologies topics, and more than 130 journal and national and international conference papers. The focus of these publications is on Information and Communication Systems Security. He is a member of the ACM and the IEEE. Since 2006 he is a member of the "IEEE Communications and Information Security Technical Committee" of the IEEE Communications Society, and of the "IFIP WG 11.6 Identity Management".