Full citation: Gray, A.R., Sallis, P.J., & MacDonell, S.G. (1998) IDENTIFIED (Integrated Dictionary-based Extraction of Non-language-dependent Token Information for Forensic Identification, Examination, and Discrimination): a dictionary-based system for extracting source code metrics for software forensics, in Proceedings of Software Engineering: Education & Practice (SE:E&P'98). Dunedin, New Zealand, IEEE Computer Society Press, pp.252-259.
doi: 10.1109/SEEP.1998.707658

IDENTIFIED (Integrated Dictionary-based Extraction of Non-language-dependent Token Information for Forensic Identification, Examination, and Discrimination): A Dictionary-based System for Extracting Source Code Metrics for Software Forensics

Andrew Gray, Philip Sallis and Stephen G. MacDonell

Department of Information Science
University of Otago, PO Box 56
Dunedin, New Zealand
agray@commerce.otago.ac.nz

Abstract

The frequency and severity of computer-based attacks such as viruses and worms, logic bombs, trojan horses, computer fraud, and plagiarism of software code have all become of increasing concern to many of those involved with information systems. Part of the difficulty experienced in collecting evidence regarding the attack or theft in such situations has been the definition and collection of appropriate measurements to use in models of authorship. With this purpose in mind a system called IDENTIFIED is being developed to assist with the task of software forensics which is the use of software code authorship analysis for legal or official purposes. IDENTIFIED uses combinations of wildcards and special characters to define count-based metrics, allows for hierarchical meta-metric definitions, automates much of the file handling task, extracts metric values from source code, and assists with the analysis and modelling processes. It is hoped that the availability of such tools will encourage more detailed research into this area of ever-increasing importance.

1. SOFTWARE FORENSICS

1.1. Introduction

Source code is the textual form of a computer program that is written by a computer programmer in a computer programming language. These programming languages can in some respects be treated as a form of language from a linguistic perspective, or more precisely as a series of languages of particular types, but within some common family. In the same manner as written text can be analysed for evidence of authorship, as in [10], computer programs can also be examined from a forensics or linguistics viewpoint [11] for information regarding the program's authorship. The goals of computer program authorship are also often similar to, or even identical to, those encountered in forensic linguistics and computational linguistics.

Figure 1 (from [3]) shows two small code fragments that were written in C++ by two separate programmers. Both programs provide the same functionality (calculating the mathematical function *factorial(n)*, normally written as *n!*) from the users' perspective. That is to say, the same inputs will generate the same outputs for each of these programs.

```
// Factorial takes an integer as an input and returns
// the factorial of the input.
// This routine does not deal with negative values!

int Factorial (int Input)
{
       int Counter;
       int Fact;
       Fact=1;  // Initalises Fact to 1 since factorial 0 is 1
       for (Counter=Input; Counter>1; Counter=Counter-1)
       {
                    Fact=Fact*Counter;
       }
       return Fact;
}
```

```
int f(int x){
int a, y=1;
if (!x) return 1; else return   x*f(x-1);}
```

***Figure 1**. Program segments in C++*

As should be apparent, each programmer has solved the same problem, that of calculating the factorial of an input, in both a different manner (algorithm) and with a different style exhibited in his or her code. These stylistic differences include the use of comments, variable names, use of white space, indentation, and the levels of readability in each function.

These fragments are obviously far too short to make any substantial claims about the feasibility of using source code characteristics to make statements regarding the author(s). However, they do illustrate the fact that programmers writing programs will often do so in a significantly different manner to another programmer, without any instruction to do so. Both of these functions were written in the natural styles of their respective authors.

1.2. Flexibility in writing source code

While source code is certainly much more formal and restrictive than spoken or written languages in terms of acceptable grammar, computer programmers still have a large degree of flexibility when writing a program to achieve a particular purpose. This flexibility includes:

- the manner in which the task is achieved (the algorithm used to solve the problem),
- the way that the source code is presented in terms of layout (spacing, indentation, bordering characters used to set off sections of code, standard headings, etc.), and
- the stylistic manner in which the algorithm is implemented (the particular choice of program statements used where there is a choice, variable names, etc.).

Other options may also be available to the programmer, such as selecting the computer platform, programming language, compiler, and text editor to be used. These additional decisions may allow the programmer some further degrees of freedom, and thus expressiveness.

Many of these features of a computer program (algorithm, layout, style, and environment) can be quite specific to certain programmers or types of programmer. Ideally, such aspects in order to be useful for software authorship analysis have low within-programmer variability, and high between-programmer variability. This is especially likely for particular combinations of features and unusual programming idioms that generally make up a programmer's problem-solving vocabulary. Therefore, it seems that computer programs can contain some degree of information that provides evidence of the author's identity and characteristics [11].

Once the classification is made that program source code is in fact a type of language that is suitable for authorship analysis, a number of applications and techniques emerge. In fact, as [11] note, a reasonable proportion of the work already carried out in computational linguistics for text corpus authorship analysis has parallels for source code. Similarly, techniques used in forensics for handwriting and linguistic analysis can also, in some cases at least, be transferred in some respect to what is referred to here as *software forensics*.

Here it is assumed that the term software forensics refers to the use of measurements from software source code, or object code, for some *legal* or *official* purpose [3]. This is similar to, but in some respects also distinct from, the use of the term in some literature where the focus tends to be very much on malicious code analysis. The legal or official nature of software forensics requires a high level of objectivity, as well as methods for calculating the degrees of evidence provided and combining that evidence with other sources. Four broad areas of application emerge in software forensics and are discussed next.

1.3. Applications

1.3.1. Author identification. The goal here is to determine the likelihood of a particular author having written some piece(s) of code, usually based on other code samples from that programmer. This can also involve having samples of code for several programmers and determining the likelihood of a new piece of code having been written by each programmer. This application area is very similar to, for example, the attempts to determine the authorship of the Shakespearean plays or certain biblical passages. An example of this applied to source code would be ascribing authorship of a new piece of code, such as a computer virus, to an author where the code matches the profile of other pieces of code written by this author.

1.3.2 Authorship discrimination. This is the task of deciding whether some pieces of code were written by a single author or by (some number of) different authors. This can possibly also include an estimate of the number of distinct authors involved in writing a single piece or all pieces of code. It is obviously necessary to distinguish between identifying multiple authors for a series of programs and co-authorship on a single program. This task involves the calculation of similarity between the two or more pieces of code and possibly some estimate of between- and within-subject variability. An example of this would be showing that different authors, without actually identifying the authors in question, probably wrote the two or more pieces of code.

1.3.3. Author characterisation. This is based on determining some characteristics of the programmer of a code fragment, such as personality and educational background, based on their programming style. An example of this would be determining that a piece of code was most likely to have been written by someone with a particular educational background due to the programming style and techniques used.

1.3.4. Author intent determination. It may be possible to determine, in some cases, whether code that has had an undesired effect was written with deliberate malice, or was the result of an accidental error. Since the software development process is never error free and some errors can have catastrophic consequences, such questions can arise reasonably frequently. This can also be extended to check for negligence, where erroneous code is perhaps suspected to be much less rigorous than a programmer's usual code. This is a much-neglected aspect of source code authorship analysis [3] with no other literature

found that mentions its use. While this could be seen as the most difficult, and certainly the most subjective, of the applications it may also be one of the most crucial in practice.

1.4. Settings

1.4.1. Educational. The educational setting of software forensics is generally concerned with plagiarism detection [14]. A significant amount of literature has been produced detailing various schemes for detecting cases where programming assignments have been plagiarised, with or without the original author's consent. Generally plagiarism detection is a combination of author identification (who really wrote the code), and author discrimination (did the same person write both pieces of code). One significant problem that emerges when using plagiarism detection is the effect of discouraging collaboration between students. Other issues such as student's adopting tutors, lecturers, and/or textbook author's styles are also problematic.

1.4.2. Legal. The use of software forensics for tracking down the authors of malicious code has been the second most emphasised application after plagiarism detection [7, 12, 13]. Other issues such as the intent analysis of malicious code also appear under this heading.

1.4.3. Industrial. Within an industrial context there are fewer applications of software forensics, but cases would include identifying authors of code that needs to be maintained where this information is not otherwise recorded or may be incorrect, and checking for negligent programming.

1.4.4. Psychological. While the above areas are mostly practical, there are also several uses for software authorship analysis from a theoretical perspective. It is possible to use such metrics to examine the developmental process of programming skills, and to correlate individual characteristics to programming ones.

1.5. Using software forensics

If software forensics, the authorship analysis of software source code, is now accepted as possible, it remains to justify the usefulness of the field in a practical sense. As the incidence of computer related crime increases it will become increasingly important to have techniques that can be applied in a legal setting to assist the court in making judgements. In addition it becomes more important for academic institutions and commercial organisations to provide sufficient justification for their official decisions. For example, a university accusing a student of plagiarism would be well advised to have sufficient evidence to back up that claim should the student take the matter higher.

Some types of these undesirable activities include attacks from malicious code (such as viruses, worms, trojan horses, and logic bombs), plagiarism (theft of code), and computer fraud. It is to be expected that the frequency of these crimes will continue to rise as increasing numbers of people gain the requisite technical skills and as the incentives rise. In the case of academic plagiarism the incidence is likely to increase as more varied types of students take degrees with some component of programming, with some of these more likely to struggle with programming.

Some of these problems are already faced with a variety of techniques. What is proposed here is that a complete and well-defined field is required, with its own techniques and tools. Without the creation of the field of software forensics, such issues as were just mentioned will continue to be tackled in an *ad hoc* manner. As the importance and frequency of such incidences increase, such a strategy will not be adequate or acceptable to participants in the process.

2. CONCEPTUAL METRICS FOR SOFTWARE FORENSICS

2.1. Source code metrics

Expert opinion can, potentially, be given on the degrees of similarity and difference between code fragments. Psychological analysis of code can also be performed, even as a simple matter of opinion. However, a more scientific approach may also be taken (and should be taken) since both quantitative and qualitative measurements can be made on computer program source code and object code. These measurements can be either automatically extracted by analysis tools, calculated by an expert, or arrived at by using some combination of these two methods. Some metrics can obviously only be calculated by an expert, such as the degree to which the comments in code match the actual behaviour of that code.

Here these measurements are referred to as *metrics* for reasons of tradition and include some borrowed and adapted from conventional software metrics and linguistics. A vast number of different metrics can be extracted from source code. Some examples of the types of metrics that can be extracted and that may be useful for authorship analysis purposes include, but are not limited to, the following list.

- The number of each type of data structure used can be indicative of the background and sophistication of a program author. A preference for certain data structures can also indicate a certain mental model that they operate within.
- The cyclomatic complexity of the control flow of the program can show the characteristic style of a programmer and may suggest the manner in which the code was written. For example, code tends to appear quite different when written all at once or over time, especially if significant new functionality has been added to the original program.
- The quantity and quality of comments in the code can provide evidence of linguistic characteristics

such as writing style, errors in spelling and grammar, etc.

- The types of variable names used within the program (capitalisation, corrupted forms, etc.) can provide clues as to background and personality.
- The use of layout conventions such as indentation and borders around sections of code tends to depend on background and the programmer's cognitive style.

These metrics, which obviously require more formal definition to be useful, could all be expected to exhibit larger between-subject variation than within-subject variation. In other words, it could be expected that a given set of programs from one author would be more similar in terms of these measurements than a set of programs from a variety of authors. Many other such metrics can also be extracted from code but this short list hopefully provides some of the flavour of candidate metrics.

Metrics such as these can be expressed as interval/ratio scale variables (such as code length in terms of lines of code or the number of uses of `while` statements). Nominal variables can also be used (for example to describe the different patterns of indentation) with binary variables a special case (use of pointers would be an example). Finally, it is also possible to use fuzzy variables to describe certain aspects of code, such as how well the comments match the behaviour of the code [5].

Many of the structural type metrics can be obtained, perhaps with modifications to definitions, from the software metrics literature. Software metric definitions, and also extraction tools, are available for such aspects of computer programs as complexity, comprehensibility, the degree of reuse made from other code, and various measures of size. The customary uses of these metrics are in managing the software development process, but many are transferable to authorship analysis.

In any case, the fundamental concepts that have emerged within the field of software metrics are very useful as starting points for defining authorship metrics. In addition, the metrics extracted from source code can often be similar, or even identical, to stylistic tests used in computational linguistics, especially where sufficient quantities of comments are available.

2.2. Object code metrics

While not part of source code analysis itself, some environmental measurements can sometimes also be extracted from executable code such as the hardware platform and the compiler employed for its production. Executable code can also be *decompiled*; a process where a source program that could then be compiled into the executable is created by reversing the compiling process. Since many source programs can be written to create the same executable there is considerable information loss, but some of the source code metrics can still be applicable.

2.3. Metric models of authorship

Once these metrics have been extracted, a number of different modelling techniques, such as cluster analysis, logistic regression, and discriminant analysis, can be used to derive models. The form of the model, the technique used, and the metrics of use all depend greatly on the purpose of the analysis and on the information available. In most respects the particular technique used for the modelling process is less important than the variables selected and their coding.

3. THE FEASIBILITY OF SOFTWARE FORENSICS FOR PRACTICAL USE

The fundamental assumption of software forensics is that programmers tend to have coding styles that are distinct, at least to some degree. As such these styles and features are often recognisable to their colleagues, or to experts in source code analysis who are provided with samples of their code [11].

However, as [11] note, the issue of how well this individuality can be *hidden*, or *mimicked*, is also of obvious importance when ascribing authorship to an individual. In [13] it is commented that, in their opinion, there might still be evidence of identity remaining after the author's attempts to disguise their identity. In other words, some aspects of a programmer's style cannot be changed if they are to program in an effective manner. Another important question is whether or not authorship can be *sufficiently accurately* recognised in itself, even without masking attempts.

These points lead to the fundamental question of whether or not there is in fact sufficient information available using these techniques to provide adequate authorship evidence for use within a *legal context*. In other words, the question is whether authorship identification or characterisation can be performed at levels of sufficient certainty for these results to then be presented as legal argument. Such evidence could be statistical or expert-opinion based.

If the argument, as presented here, that there is such information is accepted then certain requirements from a legal perspective need to be met before such evidence is admissible. In addition, a means of quantifying the strength of the evidence is necessary, as is a method for presenting such evidence to laypersons.

The focus in this paper is on software forensics, which has already been defined as *the general field of analysing computer program authorship for legal reasons*. However, in order to indicate the place of this area within the entire range of authorship analysis activities for source code Figure 2 shows the relationship between some of these areas.

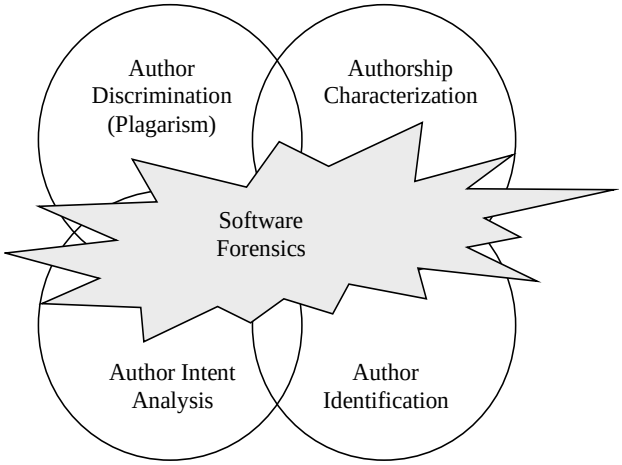

Figure 2. *Software forensics*

4. MALICIOUS CODE ANALYSIS

4.1. Introduction

This section looks at malicious code analysis since this is the area that best fits the label software forensics. In order to ascertain the circumstances that lead to a defect in code or a malicious application, a series of questions need to be answered:

1. What does the code do? While this may appear trivial, in complex real-world systems determining the effect of a piece of code can involve considerable effort, or may even be impractical. This is especially likely for legacy systems where the original programmers have since left the organisation. This question is not an authorship question *per se*, and should be left to software engineers.
2. Who wrote the code? This is the authorship question that is the focus of this section. As noted in [13], the anonymous nature of computer crimes such as viruses, worms, and logic bombs makes the attack more attractive. Identifying the author of the malicious code is not necessarily the same as identifying the author of the system. Since many systems involve a large number of developers the identification of the most likely author can be difficult, even more so if the code could have been written by non-members of the programming team. In the case of standalone systems such a viruses, code may be matched to viruses already attributed to a certain author.
3. When was the code written? Since programmers' styles change over time it may be possible to identify roughly when the malicious code was written. At the very least, for malicious code contained in a larger system it may be possible to determine whether or not the code was part of the original system or added at a later date.
4. What is the intent of the code? In many cases this will be obvious, but in others it may be the case that the code could be an error or deliberate.

An application for authorship analysis that has not been found in any literature other than that by the authors is the answering of the fourth question above: determination of intent, malice or otherwise, once code has been found that could have been maliciously programmed. Certain cases, such as salami attacks and logic bombs that are triggered by the removal of an employee from the organisation's payroll, are *prima facie* malicious. However, there may also exist cases where undesirable behaviour in an application could be either maliciously programmed, or could simply be the inevitability of defects in the code.

4.2. Cases of malicious code analysis

The two main cases where malicious source code has been examined in detail are the WANK and OILS worms [7] and the Internet Worm [12]. In [12] the Internet Worm, written by Robert Morris and released onto the Internet on November 1988 is discussed from the perspective of authorship analysis and technical analysis. In [7] the WANK and OILZ worms were studied. These were released in 1989 attacking NASA and DOE systems. The worms were both written in DCL, with the WANK worm proceeding OILZ by about two weeks.

4.3. Metrics for malicious code analysis

In [13] the authors suggest a number of features that can be used to analyse source code for malicious programs and the following list of features is a subset of these, as well as containing some additional features.

- 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 [13], 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.
- Also not mentioned by [13], but nonetheless important is reuse of code. If code from a previously identified author has been reused then this could indicate authorship or association.
- Data structure 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.
- 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.

5. SPECIFIC METRICS

Specific metrics generally match one-to-many with the conceptual metrics discussed above. While many metrics could be listed, the purpose of this section is simply to provide some of the flavour of such metrics. Listing all of the possible metrics would require a substantial volume in itself. The difficulty is not with formulating such metrics, but rather with selecting those necessary. Software forensics is still an empirically young discipline and there has been only limited work towards identifying a collection of metrics that would provide all necessary and sufficient aspects of the programs.

The following metrics provide simple examples of authorship related metrics, most of which can be automatically collected using IDENTIFIED as will be described in the next section. These are merely a small number of the possible measurements and are intended to provide some indication of the flavour of such metrics.

Metric 1: Mean length of source code lines in terms of the number of characters.

Metric 2: Mean variable name length in terms of characters.

Metric 3: Variable names are meaningful or not.

Metric 4: Pointers are used or not used.

Metric 5: Mean length of a function in lines of code.

Metric 6: Ratio of comment lines to non-comment lines of code.

Metric 7: Ratio of blank lines to non-blank lines.

Metric 8: Ratios of use of `for`/`repeat`/`while` type constructs.

Metric 9: Most commonly used indentation style (number of characters indented by and when used).

Metric 10: Use of global variables.

As a further example, the specific metrics suggested by [11] for plagiarism detection are given below. As well as illustrating some other specific metrics, these also show how tradition software metrics can be used for authorship analysis.

- Volume measured as Halstead's n, N, and V [4].
- Control flow measured by McCabe's V(G) [8].
- Structure measured by Leach's coupling assessment [6].
- Data dependency measured by Bieman and Debnath's GPG assessment [1].
- Nesting depth measured by program nesting depth and average nesting depth [2].
- Control structure measured by Nejmeh's (1988) NPATH [9].

6. IDENTIFIED

6.1. Introduction

IDENTIFIED (Integrated Dictionary-based Extraction of Non-language-dependent Token Information for Forensic Identification, Examination, and Discrimination) is a prototype implementation of a dictionary-based metric extraction tool with modules for analysing the resultant metric data. The main module is the Scan program as shown in Figure 3. The overall structure of IDENTIFIED is as shown in Figure 4.

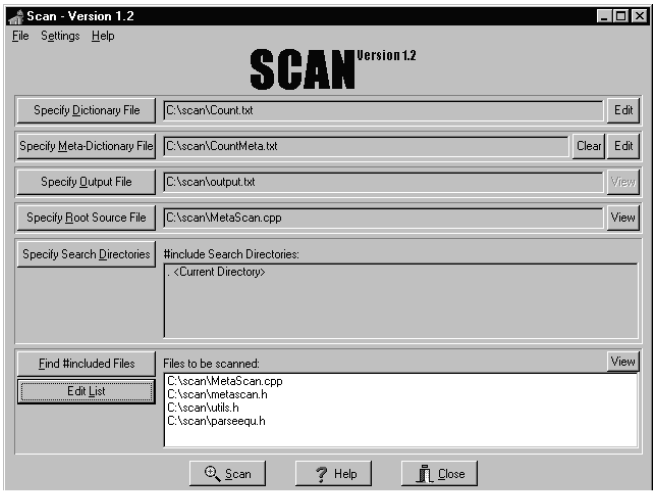

Figure 3. *The Scan Module of IDENTIFIED*

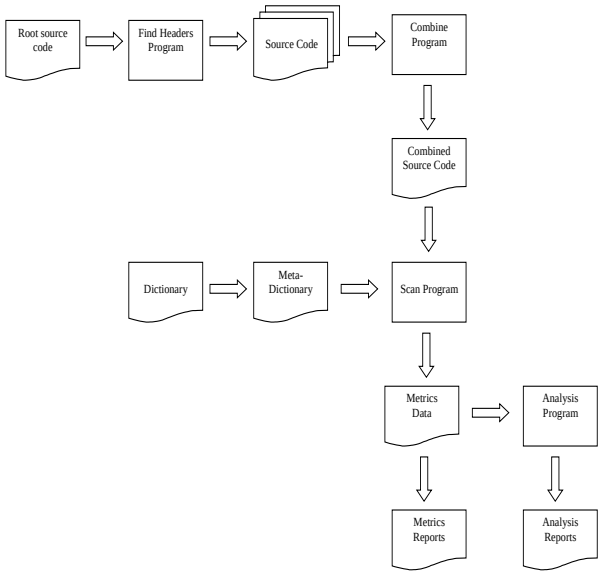

Figure 4. *Structure of IDENTIFIED*

In order to illustrate the functionality of the system, a trivial analysis of source code will be described. The general order of processing is to first select a root source code file along with directories from where files included or called can be obtained. The Find program then constructs a list of all programs that the root source program depends on that can be found in the specified directories. This allows for easily omitting standard libraries.

These files can then be merged to produce a single source code file. Subsequent analysis can be performed on both the separate files and the combined file.

The source code can then be analysed using a metrics definition file which using a series of special wildcards, special code characters, and options allows for the creation of most metrics of interest (Figure 5 shows a trivial example for line comment characters and lines of code in C++). The Scan routine uses these entries to count the number of occurrences for each metrics. This file can then be used to extract metrics as shown in Figure 6, or meta-metrics can be defined as shown in Figure 7. Definitions of meta-metrics include the standard arithmetical operations and can refer to lower level metrics as well as other meta-metrics. Metrics can be defined with as much hierarchy as is needed and different meta-metric files can refer back to the same base metrics dictionary file.

The system is not language dependent in any important way, since new dictionary files and meta-dictionaries can be easily created using a wizard system. The header scanning system is however limited to supported languages, although source code in other languages can be combined by manually specifying the filenames.

```
_"_Comments
//

_"_LOC
_C!__&0__~_//_~__*_
```

Figure 5. *Metrics Definition File*

```
Scan results for file:
C:\scan\temp\parseequ.h

Comments
11 : //

LOC
34 : _C!__&0__~_//_~__*_
```

Figure 6. *Metrics Count File*

```
META-DICTIONARY

SquaredLOC
//e
+o
_C!__&0__~_//_~__*_e
*o
_C!__&0__~_//_~__*_e
+o
//e
```

Figure 7. *Meta-Dictionary File*

The results from the metric extraction can then be passed to modules for displaying the data and carrying out analysis such as cluster analysis, cased-based reasoning, and discriminant analysis. In addition the results can be exported to a spreadsheet or statistical package such as SPSS for further analysis.

7. CONCLUSIONS

It appears that software forensics has the potential to become both an important area of practice in computer security, computer law, and academia as well as an exciting new area of research. As part of this development in the field there is the necessity for more formally defined methods and metrics. It is hoped that the IDENTIFIED system will provide one of these steps towards the creation of a new scientific discipline.

More work is continuing on the analysis routines for IDENTIFIED, along with more powerful pattern matching options for the Scan program. Once this is complete then a large scale empirical study of source code will begin with the goal of identifying useful models of authorship.

ACKNOWLEDGEMENTS

The authors would like to gratefully acknowledge the assistance of Mr. Grant MacLennan with the designing and programming of the IDENTIFIED system.

REFERENCES

[1] Bieman, J.M., and Debnath, N.C. (1985). An Analysis of Software Structure Using a Generalized Program Graph. Proceedings of COMPSAC'85 254-259.

[2] Dunsmore, H.E. (1984). Software Metrics: An Overview of an Evolving Methodology. Information Processing & Management 20:183-192.

[3] Gray, A.R., Sallis, P.J., and MacDonell, S.G. (1997) Software Forensics: Extending Authorship Analysis Techniques to Computer Programs. Presented at The Third Biannual Conference of the International Association of Forensic Linguists, 4-7 September 1997, at Duke University, Durham, North Carolina, USA.

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

[5] Kilgour, R.I., Gray, A.R., Sallis, P.J., and MacDonell, S.G. (1997) A Fuzzy Logic Approach to Computer Software Source Code Authorship Analysis. Accepted in The Fourth International Conference on Neural Information Processing - The Annual Conference of the Asian Pacific Neural Network Assembly (ICONIP'97), 24-28 November 1997, at the University of Otago, Dunedin, New Zealand.

[6] Leach, R.J. (1995). Using Metrics to Evaluate Student Programs. ACM SIGCSE Bulletin 27:41-43,48.

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

[8] McCabe, T.J. (1976). A Complexity Measure. IEEE Transactions on Software Engineering 2(4):308-320.

[9] Nejmeh, B.A. (1988). NPATH: A Measure of Execution Path Complexity and its Applications. Communications of the ACM 31:188-200.

[10] Sallis, P. (1994). Contemporary Computing Methods for the Authorship Characterisation Problem in Computational Linguistics, New Zealand Journal of Computing, 5, 85-95.

[11] Sallis P., Aakjaer, A., and MacDonell, S. (1996). Software Forensics: Old Methods for a New Science. Proceedings of SE:E&P'96 (Software Engineering: Education and Practice). Dunedin, New Zealand, IEEE Computer Society Press, 367-371.

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

[13] Spafford, E.H., and Weeber, S.A. (1993). Software Forensics: Can we track Code to its Authors? Computers & Security. 12:585-595.

[14] Whale, G. (1990). Software Metrics and Plagiarism Detection. Journal of Systems and Software. 13:131-138.