Software Forensics: Extending Authorship Analysis Techniques to Computer Programs

Andrew Gray Philip Sallis Stephen MacDonell

Software Metrics Research Laboratory
Department of Information Science
University of Otago
PO Box 56, Dunedin, New Zealand
+64 3 479 5282
agray@commerce.otago.ac.nz

ABSTRACT
The number of occurrences and severity of computer-based attacks such as viruses and worms, logic bombs, trojan horses, computer fraud, and plagiarism of code have become of increasing concern. In an attempt to better deal with these problems it is proposed that methods for examining the authorship of computer programs are necessary. This field is referred to here as *software forensics*. This involves the areas of author discrimination, identification, and characterisation, as well as intent analysis. Borrowing extensively from the existing fields of linguistics and software metrics, this can be seen as a new and exciting area for forensics to extend into.

Keywords
Authorship analysis, computer programming, malicious programs, software forensics, software metrics, source code

1. INTRODUCTION

Computer programs are generally written in what is referred to as *source code*. Source code is the textual form of a computer program that is written by a computer programmer. In some cases source code is produced by another program and this generated source code is based on higher-level instructions (written in more abstract source code or using a visual design environment) created by a programmer. In any case (disregarding visual programming), source code of all levels is written 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. Languages for writing computer programs differ in terms of their so-called generation (roughly the time that they were devised and reflecting their level of abstraction) and type (such as procedural, declarative, object-oriented, and functional). In the same manner as written text, such programs can also be examined from a forensics viewpoint.

Figure 1 shows two small code fragments that were written in a popular programming language called 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 in both a different manner (algorithm) and with a different style exhibited in his or her code. The first algorithm is a simple loop through the values from 1 through to the input into the function (in reverse), while the second employs a more sophisticated (but also worse performing) recursive definition. The 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. However, they do illustrate the ability for programmers to write programs 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 and so should reflect the types of differences that would be evident in general between their programs.

Source code for most programming languages[1] is, as shown in this figure, a series of statements that provide instructions to the computer for the purpose of achieving some task. In this case the statements define very small and simple functions that can be called from other parts of the program. Such programs are usually structured with distinct sections and a general ordering, or sequence, of execution. This sequencing may include conditional ordering (i.e. logical conditions may, when met, cause the repetition, addition, or omission of certain instructions contained within the code).

While source code is certainly much more formal and restrictive than spoken or written languages, computer programmers still have a large degree of flexibility when writing a program to achieve a particular function. 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, etc.), and the stylistic manner in which the algorithm is implemented (the particular choice of program statements used, variable names, etc.).

Other choices 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[2]. This is especially true for particular combinations of features and programming idioms that 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 (Sallis et al., 1996).

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 Sallis et al. (1996) note, a reasonable proportion of the work already carried out in computational linguistics for text corpus authorship analysis (such as Sallis, 1994) 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 as will be explained below, for some *legal* or *official* purpose. 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.

2. APPLICATIONS OF SOFTWARE FORENSICS

The four principal aspects of authorship analysis that can be applied to software source code, and that are of interest to the discipline of software forensics, are as follows.

1. *Author 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 include an estimate of the number of distinct authors involved in writing a single piece or all pieces of code[3]. This involves the calculation of some similarity between the two or more pieces of code and possibility some estimate of between- and within-subject variability. An example of this would be showing that two pieces of code were probably written by different authors[4], without actually identifying the authors in question.
2. *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 historical 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.
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] Here the focus is on what are referred to as third-generation languages which include most of the popular and common programming languages, although the principles can in some manner be extended to higher and lower level languages.

[2] Ideally, such aspects have low within-programmer variability, and high between-programmer variability.

[3] It is obviously necessary to distinguish between identifying multiple authors for a series of programs and co-authorship on a single program.

[4] The converse that the code was written by the same author can be used as a test for plagiarism from that particular author. See Whale (1990) for more on plagiarism detection.

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[5] 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.

Such forensic techniques are not limited to analysing just the source code files written by a programmer and used to create a program. Source code is, usually, compiled into *object code*[6] that is then combined (linked) into the executable as shown in Figure 2. This executable is the program in its usable form that the end-users see and use for whatever purpose.

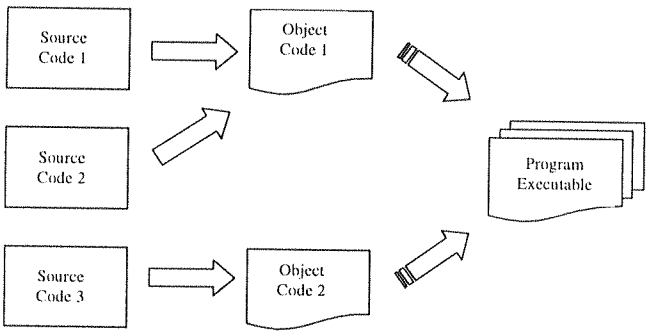

Figure 2 Program source code being compiled and linked into an executable

Information can also be extracted from the object/executable code, by decompiling it into source code with information loss[7] or from certain features contained in the executable that suggest the compiler and/or platform used.

Some programming languages, instead of using the compiling process described above, work on an interpreter system. In this case, the executable is the source code itself, and the program is executed within an environment that translates the code into machine-understandable instructions as the program operates. This is less common for programming languages as such, but still popular for scripting languages. In general, the term executable refers to a compiled program.

Therefore, irrespective of whatever form the program being examined takes (source code or compiled executable) measurements can be taken and used for a number of authorship analysis purposes. The types of information available will obviously depend on the program's form (source code or object code), and for different cases one form may be superior to the other. In general, source code will provide the greatest amount of information. However, some information is always available irrespective of the form of the program, and may be useful when combined with other available knowledge that does not concern the program itself.

3. MOTIVATION FOR SOFTWARE FORENSICS

If software forensics, the authorship analysis of software source code, is then 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. Some types of these crimes 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.

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 cases increases, such a strategy will not be adequate or acceptable to participants in the process.

As is hopefully apparent from the above discussion, software forensics can be seen as a new and exciting area of both forensics and linguistics. There is the potential, and it is claimed here also the necessity, for building on current results and techniques from the associated fields as well as developing new and specialised methods of analysing the authorship of computer programs. The next question that needs to be asked is how exactly can such a discipline be performed in the real world?

4. THE PRACTICE OF SOFTWARE FORENSICS

Expert opinion can, potentially, be given on the degrees of similarity and difference between code fragments. This could be based on general appearance or the use of programming idioms. Psychological analysis of code can also be performed, even as a simple matter of opinion.

However, a more scientific approach may also be taken[8] since both quantitative and qualitative measurements can be made on computer program source code and object code. Qualitative measurements can be usefully expressed using fuzzy logic techniques as shown in Kilgour, et al. (1997). These measurements can be either

---

[5] Errors in software programs are often referred to as *bugs*.

[6] This is the program *translated* into the computer's internal language.

[7] Generally, compilers optimize the code, losing some structure, and use symbols in place of names.

[8] It would be strongly argued here by the authors that a more scientific approach *must* be taken if the field is to accomplish anything of use.

automatically extracted by analysis tools, calculated by an expert[9], or arrived at by using some combination of these two methods. Here these measurements are referred to as *metrics* for reasons of tradition[10] 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.
- 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 personality.

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.

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.

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[11] there is considerable information loss, but some of the source code metrics can still be applicable. In the case of source code some information about the editor and supporting tools used may also be available from layout of the code and reused code.

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.

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 (Sallis et al., 1996). However, as Sallis et al. note, the issue of how well this individuality can be *hidden*, or *mimicked*, is also of obvious importance when ascribing authorship to an individual. Spafford and Weber (1993) comment that, in their opinion, there might still be evidence of identity remaining after the author's attempts to disguise their identity. In other words, that 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

---

[9] Some metrics can only be calculated by an expert, such as the degree to which comments in source code match the behavior of the code itself. Other metrics, such as the ratio of statement lines to blank lines, are obviously best automated for large systems.

[10] Measurements made on software programs (and the development process) for the purposes of managing software development are traditionally referred to as *software metrics*. The term is also used for measurements of software for educational reasons (assessment, style, etc.), user satisfaction, and many other purposes. Thus, software metrics can be regarded as a generic label for any measurements made on a computer program.

[11] Theoretically, an infinite number of programs could be written to simulate the executable.

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.

As was mentioned above, software code authorship analysis could be, and in some cases has been, used for a number of diverse tasks. These include the principal areas of malicious code analysis and the detection of plagiarism. Each of these application areas requires a specialised approach, although there is also some degree of overlap. Some other less common, and also in general less forensically applicable, reasons for authorship analysis include psychological studies of programming, assessing source code for quality, and identifying authors of code for maintenance purposes.

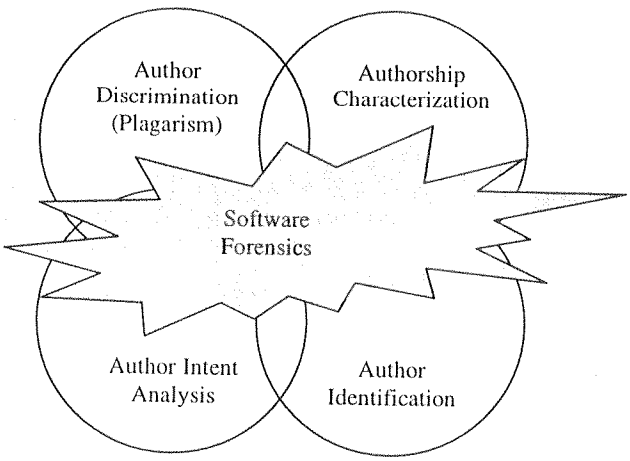

Figure 3 Software forensics

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 3 shows the relationship between some of these areas.

5. DETERMINING AUTHORSHIP AND INTENT OF MALICIOUS CODE

This aspect of software forensics assumes that some undesirable behaviour, such as deleting records, incorrect calculations, or suchlike has been observed in a system. Two possibilities exist here, that the malicious code is part of another system (such as a logic bomb) or that the code is an application itself (such as a virus).

5.1. Types of code available for analysis

As has already been discussed above, the code that an analyst has available will generally be either source code (text) or the executable. Many of the identifying features in executable code can also be found in source code as was discussed above. Each type of code however requires a different approach to its analysis.

5.1.1. Executable code

The most common types of executable code that may attack a system are:

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

As Spafford and Weber (1993) note, viruses usually leave their code in infected programs, and code remaining after a variety of attack methods may include source code, object code, executables, scripts, etc. However, for compiled code much evidence is lost, including variable names, layout, and comments. Compilers may also perform optimisations that lead to the executable code having a significantly different structure to the original source code. Irrespective of the loss of some information, Spafford and Weber are still able to point out some features that will remain. These include (with some additional points not made by Spafford and Weber):

- Data structures and algorithms. This can be a useful indication of the programmer's background since they are more likely to use certain algorithms that they have been taught or had exposure to, and are therefore more comfortable with. Non-optimal choices may indicate a lack of knowledge or even that the programmer uses another language's programming style, perhaps indicating their preferred or first programming language.
- Compiler and system information. Executable code contains a number of signs that may indicate the compiler used.
- Level of programming skill and areas of knowledge. The degree of sophistication and optimisation can provide useful indications of the author. Differences in sophistication within a program may indicate a

mixture of authors or an author who specialises in a particular area.

- Use of system and library calls. These may provide some information regarding the author's background.
- Errors present in the code. Almost all code contains errors, and any complex system will almost certainly have defects. Programmers are often consistent in terms of the errors that they make.
- Symbol table. If an executable is produced using a *debug mode*[12], rather then a *release mode*, then much information that is part of the source code will still remain.

5.1.2. Source code

When undesirable modifications are made to software developed within an organisation the source code will generally be available for examination[13]. Source code may also be available for some of the other types of attack programs mentioned above where the program was written using an interpreted language or (more commonly) a scripting language.

Spafford and Weber (1993) 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 Spafford and Weber (1993), but important nonetheless, are the psychological preferences that some programmers may feel for certain languages.
- Formatting of code. The manner in which the source code is formatted can indicate both author features and some psychological information about the author. Pretty-printers are commonly used to automatically format source code and while this removes author-specific features it introduces information about what pretty-printer may have been used.
- Special features such as macros may be used that indicate to some degree which compiler or library was used.
- Commenting style. This can be a very distinctive aspect of a programmer's style. If comments are sufficiently large then traditional textual linguistic analysis may be appropriate.
- Variable naming conventions are another distinctive aspect of an author's style. The use of meaningful versus non-meaningful names, the use of standards (such as Hungarian notation), and the capitalisation of variable names are all features that programmers can adopt.
- Spelling and grammar. Where comments are available an examination of their spelling and grammar can be a useful indication of authorship. Spelling errors may also be present in function and variable names.
- Use of language features. Some programmers prefer to use certain aspects of a language than others.
- Size. The size of routines can indicate the degree of cognitive chunking used by the programmer.
- Errors. As noted in the section above on executable code, programmers often consistently make the same or similar errors.
- Also not mentioned by Spafford and Weber (1993), but nonetheless important is reuse of code. If code from a previously identified author has been reused then this could indicate authorship or association.

5.2. Analysis of malicious code

In order to ascertain the circumstances that led to such 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 main focus of this section. As noted in Spafford and Weber (1993), the anonymous nature of computer crimes such as viruses, worms, and logic bombs makes the attacks even 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 as 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

---

[12] A debug version of a program contains much extra information in the object code that the compiler uses to give feedback while the program is executing. Surprisingly often, programmers release programs that contain this additional data. A release version simply lacks this superfluous information, making it smaller and faster to execute.

[13] An obvious exception to this is where the code, or part of the code, self-destructs upon activation or a certain event. Even in this case however, backups should enable analysts to obtain copies of the full code.

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. Also the structure of programs tends to be quite different depending on the order in which the code was written.

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 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.

5.3. Case studies

The two most discussed cases where malicious source code has been examined are the WANK and OILZ worms (Longstaff and Schultz, 1993) and the Internet Worm (Spafford, 1989).

5.3.1. The Internet Worm

In Spafford (1989) 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. Here the focus is only on the former aspect.

Spafford's (1989) analysis of the Internet Worm is based on three separately reversed-engineered versions of the worm. The three versions were created independently and agree in almost all details suggesting that they do indeed reflect the original code. Some of the conclusions that Spafford was able to make are:

- The code is not well written and contains many errors and inefficiencies.
- The code is not portable.
- The code was probably not checked using lint[14]
- The code contains little error-handling behaviour, suggesting that the author was sloppy and performed little testing. Another possibility is that the worm's release was premature.
- The code indicates that the final version would have been much more comprehensive.
- The structures used are all linked lists that were inefficient and indicated a lack of advanced programming ability and/or tuition.
- The code contains redundancy of processing.
- A section that performs cryptographic functions is exceptionally efficient and provides functionality not used by the worm. According to Spafford (1989) this does not appear to be written by the author of the rest of the worm.
- The code seemed to have been written over a long period of time.

This list of observations made by Spafford (1989) is impressive and indicates the amount of knowledge that can be extracted from such source code. Especially important are the observations of the lack of ability of the author, the suggestion that the release may have been premature, the dual authorship, and the suggestion that the code was written over a considerable period of time.

By comparing the code to Robert Morris' (and also to other potential suspects), while this was not necessary, it seems likely that the above list of features as well as more detailed metrics would have led to Morris' identification as the author. It is unfortunate that this was not performed, since there is a lack of real-world case studies of such events.

5.3.2. The WANK and OILZ worms

In Longstaff and Schultz (1993) 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. The focus of this paper is on analysing the response to the threat, predicting the response to similar threats, exploring the evolution of the code, and examining the code's authorship. Only the later two are of interest here.

The fact that the worms were written in DCL, a scripting language, i.e. not compiled, provides much more information than a compiled version. The WANK worm is 785 lines long and exhibits structural coding. Longstaff and Schultz (1993) suggest the following:

- Three distinct authors worked on the system.
- Author one:
  - Academic style of programming
  - Descriptive and lower case variable names
  - Flow based on variables, gotos, and subroutines and is complex
  - High level of understanding
  - Experimentation rather than malicious intent
- Author two:
  - Malicious code with hostile intent
  - Use of profanities
  - Capitalisation
  - Simple programming style

[14] A very popular programming utility.

- Author three:
  - Combined the others' code
  - Mixed case
  - Non-descriptive variable names
  - Simple coding that resembles BASIC
- Attempts to correct bugs in the code - the OILZ worm corrects some bugs in WANK.

These pieces of evidence are of considerable value in an investigation of the attack. The main points that emerged are the multiple authors and some of the differences between them. While such information is unlikely to lead directly to identifying those involved, it does provide additional evidence that may be sufficient to reach a satisfactory level of certainty.

6. CONCLUSIONS

In this paper the concept of authorship analysis for software source code has been briefly introduced, and a subset of the field has been identified that is called software forensics. This is a field that it is believed will become increasingly important as part of criminal and civil proceedings, as well as in other official matters such as academic plagiarism detection.

The authors are currently developing a toolkit called IDENTIFIED (Integrated Dictionary-based Extraction of Non-language-dependent Token Information for Forensic Identification, Examination, and Discrimination) (Gray, et al., 1998). This assists with the automatic extraction of a wide variety of metrics (some of which have been mentioned in this paper) that can be used for software forensics. The package also contains modules for case-based reasoning, discriminant analysis, and other analysis techniques.

The work described here is also being extended by the collection of a number of formally defined metrics that can be used for software forensics; with the eventual goal of producing a list that is for all intensive purposes complete for certain subsets of the task. These metrics are included as a dictionary file in IDENTIFIED.

Other issues such as statistical models of certainty and combining evidence for source code authorship analysis are also being investigated. The next stage for the work described here is to determine the legal issues that would be involved in using such evidence.

REFERENCES

Gray, A.R., Sallis, P.J., and 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. Submitted to *SE:E&P'98 Software Engineering: Education & Practice*. Dunedin. New Zealand.

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 for *The Fourth International Conference on Neural Information Processing -- The Annual Conference of the Asian Pacific Neural Network Assembly (ICONIP'97)*. Dunedin. New Zealand.

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.

Sallis, P.J. (1994). Contemporary Computing Methods for the Authorship Characterisation Problem in Computational Linguistics. *New Zealand Journal of Computing*. 5(1):85-95.

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.

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

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

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