Software Forensics: Can We Track Code to its Authors?

Purdue Technical Report CSD–TR 92–010

SERC Technical Report SERC–TR 110–P

Eugene H. Spafford   Stephen A. Weeber

Department of Computer Sciences
1398 Computer Science Building
Purdue University
West Lafayette, IN 47907–1398

19 February 1992

Abstract

Viruses, worms, trojan horses, and crackers all exist and threaten the security of our computer systems. Often, we are aware of an intrusion only after it has occurred. On some occasions, we may have a fragment of code left behind — used by an adversary to gain access or damage the system. A natural question to ask is "Can we use this remnant of code to positively identify the culprit?"

In this paper, we detail some of the features of code remnants that might be analyzed and then used to identify their authors. We further outline some of the difficulties involved in tracing an intruder by analyzing code. We conclude by discussing some future work that needs to be done before this approach can be properly evaluated. We refer to our process as ***software forensics***, similar to medical forensics: we are examining the remains to obtain evidence about the factors involved.

1 Introduction

An aspect of both computer crime and computer vandalism that makes them more attractive activities is their anonymity. Whether the method of attack is virus, worm, logic bomb, or account breaking, tracing the actions back to an individual is generally an extremely difficult task. In one well-known case, Cliff Stoll's German hacker did not fear discipline even after being detected, trusting in the inability of anyone to trace his many network hops.[4] Authors of viruses distribute them without worry of being identified as the source. Participants in discussions in electronic forums tend to become more hostile than they ever would in face-to-face conversations; the anonymity of the transaction lowers their inhibitions.

Taking steps to remove the anonymity in computer use, such as more complete session logging and improved network protocols that include authentication information, can only help to discourage an attacker. However, there are limits to the strength of methods that can be economically employed. Furthermore, no method is 100% effective under all circumstances.

Often, the evidence remaining after a computer attack has occurred includes the instructions introduced into the system to cause the damage. Viruses, for example, usually leave their code in the infected programs. These remnants of an attack may take many forms, including programming language source files, object files, executable code, shell scripts, changes made to existing programs, or even a text file written by the attacker. It would be useful if these pieces of information could be utilized in a way that could help identify or confirm the source of the attack. This would be similar to the use of handwriting analysis by law enforcement officials to identify the authors of documents involved in crimes, or to provide confirmation of the role of a suspect.

Handwriting analysis involves identifying features of the writing in question. A feature of the writing can be anything identifiable about the writing, such as the way the i's are dotted or average height of the characters. The features useful in handwriting analysis are the writer-specific features. A feature is said to be writer-specific if it shows only small variations in the writings of an individual and large variations over the writings of different authors.

Features considered in handwriting analysis today include shape of dots, proportions of lengths, shape of loops, horizontal and vertical expansion of writing, slant, regularity, and fluency.[10] The majority of features in most handwriting are ordinary. However, most writing will also contain features that set it apart from the samples of other authors, features that to some degree are unusual.[7] A sample that contains i's dotted with a single dot probably will not yield much information from that feature. However, if all of the o's in the sample have their centers filled in, that feature may identify the author.

Identification of writer-specific features generally requires many samples. A person's handwriting is expected to change both as time passes and under different writing conditions. Too few samples can lead to misidentification of writer-specific features. Additionally, care must be taken in selecting samples that show the natural writing of an individual. Individuals often attempt to hide their identity by disguising their writing.

Identification of computer code by matching identifying features should likewise be possible. Programming, especially in a language rich in data types and control structures, has considerable room for variation and innovation. Even if coding is from detailed specifications, room exists for personalization. Programmers generally acknowledge that they each have a unique coding style. Using appropriate stylistic elements may help in the production, reuse, and debugging of code. Many texts recommend elements of style to use when programming, and often programmers integrate selected elements of others' styles into their own repertoire as they gain experience programming (cf. [1, 9, 11, 6, 17]).

The keys to identifying the author of suspect code are selection of an appropriate body of code and identification of appropriate features for comparison. This may not be easy to do if the programmer has attempted to hide his authorship, or if appropriate sample code is not available. Nonetheless, our personal experience is such that we believe important features might still be present for analysis, in some cases. At the least, analysis of the characteristics of the code might well lead to the identification of suspects to examine further.

Additionally, if sufficient background research is done to establish a good statistical base, and if large enough samples of code are present, known statistical methods currently applied to determine authorship of prose may also be applied to code.[12] These methods, although perhaps not certain, may possibly be combined with the analysis of stylistic features to provide clues to the authorship of a piece of code.

In the following section, we have detailed some of the features that we believe to be the most useful in such a comparison. We believe that an in-depth study of these in features in the code of many programmers may result in some useful forensic information.

2 Analysis of Unauthorized Code

We will consider two different cases where code remnants might be analyzed. These differ in the nature of the code that was left for analysis.

2.1 Analysis of Executable Code

Often, the remnant of an attack is a piece of executable code, such as a virus or worm program. Unfortunately, many of the features of the code that could have been used in analysis have been stripped away during compilation. Comments and indentation have been removed, and identifiers have been replaced by memory addresses or register names. Additionally, optimizations may have been performed on the code, possibly giving the executable code a very different structure than the original program source.

For example, an optimizing compiler might generate the same executable code for each of the following C language program fragments:

```
for (x = 0; x < 10; x++) {
        func(x);
}

x = 0;
while (x < 10) {
        func(x);
        ++x;
}

x = 0;
while (TRUE) {
        func(x);
        if (x++ == 10) break;
}

x = 1;
do {
        func(x-1);
        x++;
} while (x <= 10);
```

The original source code might have actually been in Fortran:

```
      DO 15 X=0, 9, 1
         CALL FUNC(X+0)
 15   CONTINUE
```

or in Pascal:

```
    for x := 0 to 9 do
        func(x);
```

Each of these different source code segments exhibits features that could possibly be used in identifying the style of programming of an individual. These features may be lost to the examiner of the resultant executable code.

For example, during the analysis of the Internet Worm program ([5, 15]), that remnant was reverse-engineered to C programs that compiled to identical binary versions. In many cases, the analysts chose arbitrary names for variables and local subroutines — the compiler would not save the values, so the choices did not matter. When the disassembled code was later matched against a copy of the "real" source code, many small differences with the reverse-engineered copies were observed that compiled to the same binary.

Executable code, even if optimized, still contains many features that may be considered in the analysis:

Data Structures and Algorithms Competence with, and preference for, certain data structures may be extracted from executable code. This may provide a clue to the background of the code author. For example, it is unreasonable to suspect a beginning programmer of authoring code that made extensive use of a B-tree for data storage. Similarly, the choice of algorithms used in a program may be a feature worthy of analysis. It seems likely to conclude that a programmer will continue to use algorithms with which they are particularly comfortable.[1]

As an example, consider the Internet Worm mentioned earlier. The code used linked lists as the primary data structure for building long lists that were repeatedly searched. This was certainly a

[1]Our experience with both undergraduate and graduate student programmers supports this supposition.

poor approach, as the repeated searches of long lists dramatically reduced the efficiency of the program. This was noted in [15], and a correspondent later related that the Worm's author, Robert T. Morris, had been instructed in the Lisp programming language in his first undergraduate data structures and algorithms course. Although a coincidence such as this is certainly not sufficient upon which to base any specific action, it may help reinforce other evidence, obtained through other means.

Related to this is the manner in which data structures are accessed. In languages with both pointers and arrays, the choice of which is used is often very programmer-specific. Likewise, using overlapped structures (the `EQUIVALENCE` statement in Fortran, and the `union` statement in C, for instance) provide an indicator. Some programmers use these structures, while others use coercion and bitwise operations.

Compiler and System Information Executable code may contain telltale signs of its origin. A unique ordering of the instructions may point to a specific compiler as the source of the code. The code may contain invocations of system calls found only in certain operating systems. These bits of information may rule out or support individuals as the author of the code.

In the case of many viruses, analysis of the binary code may reveal that it was written in C or Pascal from a certain vendor. This can be determined because support routines (sometimes known as "thunks") and library calls unique to that vendor are present in the binary.

Programming Skill and System Knowledge The level of expertise of the author of the program, with both the operating system in question and computer programming in general, may be estimated from the executable code. For example, programming that duplicates functionality already provided by standard system calls, makes use of recursion, or makes proper calls to advanced system functions could indicate different levels of knowledge and skill.

Additionally, the inclusion or omission of error-checking code is also quite telling. Some programmers seldom (or never) include exception handling code in their programs. Others always include such code. In instances where the code is sometimes included, this may provide an identifiable set of routines that the author always checks (perhaps because of past program failures with those routines). This set could then be compared with the set from other, known programs as a metric of similarity.

Choice of System Calls The support functions used in the code may also indicate something about the background of the programmer. For instance, in the UNIX system, there are sometimes two difference calls to locate the first instance of a particular character in a string. The `index` routine is derived from the Berkeley (BSD) version of UNIX, and the `strchr` function is derived from the System V version of UNIX. Users will usually exhibit a distinct preference for one call or the other when programming in an environment that provides both functions. Experience with reading and porting code has convinced us there are many such observable preferences.

Errors Programmers will usually make errors in all but the simplest or most carefully coded programs. Some programmers will consistently make the same types of errors, such as off-by-one errors in loops processing arrays.[2] Cataloging and comparing these faults may provide yet another metric for determining authorship of suspect code.

It is possible that the symbol table may still be present in the executable, as is often the case when the compiler is told to generate debugging information. In this case, several of the features normally associated with program source code may also be examined in the executable code.

2.2 Analysis of Source Files

Program source code provides a far richer base for writer-specific programming features.

Language Perhaps the most immediate feature of the code is the programming language chosen by the author. The reasons behind the choice may not be obvious, but could include availability and

[2]This same tendency can be used in other contexts, to direct software testing to likely faults.[2, 16]

knowledge. It would be unreasonable to suspect an individual of being the author of a program written in a programming language that he does not know.

Formatting The formatting of source code often exhibits a very personal style. Format also tends to be consistent between programs, making it easier for an author to read what she has written. These factors indicate that the formatting style of code should yield writer-specific features. Placement of compound statement delimiters, multiple statements per line, format of type declarations, formatting of function arguments, and many other characteristics may be identified in the code in question. This assumes that the programming environment in question does not have a widely-used, rigid code formatter ("pretty-printer") that may have produced the observed style.

Another bit of information that could become available in this analysis is editor choice. For example, it may be possible to recognize the formatting styles produced by and editor such as Emacs, or to detect embedded mode-setting commands. Syntax-directed program editors may also provide a distinct and unusual style, should they become somewhat more common.

Special features Some compilers support ***pragmas*** or special macros that are not present on every system. The presence of any of these special features may provide clues as to the software development environment of the author. Inclusion of conditional compilation constructs, especially those involving initialization and declaration files, may also provide similar information about environment.

Comment Styles Users often tend to have a distinctive style of commenting their programs. Some use lines of a graphic character to set off comments from code. Others place comment headers above each function, describing it. Still others avoid comments at all costs.

The frequency and detail of the comments present may also be distinctive. Some programmers comment with short tags, and others write whole paragraphs. This may result in a measurable pattern.

Variable Names Choice of variable names is another aspect of programming that often indicates something about the author. Some programmers prefer to connect words in identifiers with an underscore, others take the SmallTalk approach and capitalize the first letter of each word with no separator. Ardent software engineers may use a naming scheme, such as Meta-Programming, that includes type information in the variable name.[13] Still others would never dream of using more than one or two characters in a variable name. A useful metric for identifier analysis might be something such as the distribution of Hamming distances between names.

Most experienced programmers have a set of "utility" variable names they use for local variables when coding small segments. Common examples include `junk`, `ii`, and `indx`. An analysis of these names may be useful in matching against other code by the same author.

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

For example, a former colleague of one of us would consistently misspell forms of the word "separate." Thus, seeing a prompt in a program that read

```
Enter 3 values, seperated by a blank:
```

was a fairly certain indicator that he had written the code.

Use of Language Features The way in which authors make use of a programming language may also differentiate them. Some authors may consistently use a subset of the features available, while others may make more complete use of all features. For example, an author may consistently use a `while` loop, even when a `for/do` or `repeat..until` loop would be more appropriate. Similarly, the use of nested `if` statements in place of `case` statements, or the (lack of) specification of default options in `case` statements could be differentiating features of code.

Other examples that fall into this category include returning values in procedure parameters versus function return values, use of enumerated data types, use of subrange types, use of bitwise boolean operations, use of constant data types, and use of structures and pointers. The average size of routines may also be used as an identifying feature: some programmers will code 300 line modules, and others will never have a module larger than will fit on the screen all at once.

One aspect of use of language features relates to computer languages that a programmer may know best or learned first. For instance, programmers who spend most of their time using procedural languages seem to seldom use recursion. Learning programming in a language such as Basic or Fortran is also likely to lead to reduced use of `while` and `do ...until` structures. Further study of such influences may yield a discernable tendency to use or avoid particular language features.

Scoping The ratio of global to local identifiers may be an author-specific trait. Additionally, declaring helper functions as accessible only in a limited scope may also contribute to identification of the programmer.

Execution paths A common factor found when analyzing student programs and also when analyzing some malicious code[3] is the presence of code that cannot be executed. The code is present either as a feature that was never fully enabled, or is present as code that was present for debugging and not removed. This is different from code that is present but not executed because of an error in a logic condition — it is code that is fully functional, but never referenced by any execution path.

As an example, consider the following section of code in the C language:

```
#define DEBUG 0
main() {
```

[3]Including [15].

```
    /* some amount of code here */

if (DEBUG) {
    printf ... many debugging values here ...
}
```

In this example the code will never be executed. The manner in which it is elided leaves the code intact, and may provide some clue to the manner in which the program was developed. Furthermore, it may contain references to variables and code that was not included in working parts of the final program — possibly providing clues to the author and to other sources of code used in this program.

Bugs Some authors consistently make the same mistakes in their coding. Often, these are faults that only rarely cause problems, and then only with extremal values or when ported to other hardware. It is precisely because these bugs seldom cause problems that users tend to continue to introduce them into their code. The presence of identifying bugs should provide very strong evidence of similarity between two pieces of code.

As examples, we have noted the following in code by both students and colleagues:

- Failure to code bitwise operations to reflect different byte ordering on the target machine — the so-called "little-endian" vs. "big-endian" problem.
- Failure to check for numeric overflow or underflow, or assuming that the internal numeric representation was of a certain (different) form (cf. [16]).
- Assuming that uninitialized pointers can be dereferenced without generating a fault.
- Assuming the stack can hold very large value-copy parameter structures when doing subroutine calls.
- Failure to check error returns from some system calls that can (rarely) fail.

Metrics Software metrics might be employed to identify an individual's average traits. Some applicable metrics could include number of lines of code per function, comment-to-code ratio, function complexity measures, Halstead measures, and McCabe metrics.[3]

3 Application and Difficulties

It seems clear that there are many potential factors that could be examined to determine authorship of a piece of software. Ideally, this analysis would be used to identify a suspect, and then a search would be made of storage and archival media to locate incriminating sources. However, a more likely scenario would see a set of metrics and characteristics derived from the code remnant and then compared with representative samples written by the suspects. This comparison must be made with considerable care, however, to prevent complicating factors from producing either false positive or false negative indications.

One such complication, for instance, is the amount of code compared. A small amount of suspect code (e.g., a computer virus) might not be sufficient to make a reasoned comparison unless very unusual indicators are present.

Another complication is the reuse of code. If the author has reused code from her earlier work, or code written by others, the effect may be to skew any metrics derived from the suspect code. It might be enough to correctly indicate *original* authorship, but that might not identify the actual culprit. In some cases, code reuse may be obvious and it may be omitted from the comparison. However, there may be cases where that is not possible. Likewise, if the suspect code was written as part of a collaboration, the characteristics of the individual authors may be subsumed or eliminated entirely.

A clever programmer, aware of this method, might disguise his code. This would probably involve using different algorithms and data structures than what he would normally use. Although this might eliminate the possibility of a match based on internal characteristics, it might also make the code more likely to fail in use. This should also make the programmer use more testing, and keep intermediate versions of the program that could later be matched against the suspect code.

There is also the potential that the underlying application may have a strong influence on the overall style and nature of the code. For instance, if we are attempting to match characteristics of a small MS-DOS boot record virus, and the code we compare against is for a UNIX-based screen editor, it is unlikely that we would find much correspondence between the two, even if they were written by the same author. Therefore, we must be certain that we compare similar bodies of code.

4 Concluding Remarks

There are many differences between handwritten prose and computer programs. Handwriting samples are usually fixed in an instant, and prose is usually not incrementally developed, while a program evolves over time. Multiple changes to a section of code as a program is developed can lead to a structure that the author would have been unlikely to create under other circumstances.

Coding is also different in that code written by others is often incorporated into a program. Often, a program is not the result of the influence of only one author. We suspect that this would severely impair the selection of writer-specific code features without knowledge of the development of the program.

Nonetheless, if there is a sufficiently large sample of code and sufficient suspect code, if there are unusual features present, and if we have correctly chosen our points of comparison, this method may prove to be quite valuable. Currently, similar *ad hoc* methods are used by instructors when they compare student assignments for unauthorized collaboration (cheating). The samples are usually not big, but the characteristics are often distinctive enough to make valid conclusions about authorship. Developing and applying more formal methods should only improve the accuracy of such methods, and make them available for more in-depth investigations.

Not only would a formal method of *software forensics* aid in the determination of malicious code authorship, it would have other uses as well. For instance, determining authorship of code is often central to many lawsuits involving trade secret and patent claims. The characteristics we have outlined in this paper might be used to determine if code is, in fact, original with an author or derived from other code. However, a rigorous mathematical approach is needed if any of these kinds of results are to be applied in a court of law (cf. [14]).

We believe that if this approach is developed, it may also prove useful in applications of reverse-engineering for reuse and debugging. The analysis of code to determine characteristics is, at the heart, a form of reverse-engineering. Existing techniques, however, have focused more on how to recover specifications and programmer decisions rather than to determine programmer-specific characteristics (cf., [8]).

Further research into this technique, based on examination of large amounts of code, should provide further insight into the utility of what we have proposed. In particular, studies are needed to determine which characteristics of code are most significant, how they vary from programmer to programmer, and how best to measure similarities. Different programming languages and systems should be studied, to determine environment-specific factors that may influence comparisons. And most importantly, studies should be conducted to determine the accuracy of this method; false negatives can be tolerated, but false positives would indicate that the method is not useful for any but the most obvious of cases.

Acknowledgments

Our thanks to Richard DeMillo for suggesting some related references of interest. Thanks to both Ronnie Martin and Tom Longstaff for their comments. We are grateful to Gene Schultz for his comments, and for encouraging us to commit our long-standing interest in this area to paper.

References

[1] Louis J. Chmura and Henry F. Ledgard. *COBOL with Style: Programming Proverbs*. Hyden Book Company, Inc., Rochelle Park, NJ, 1976.

[2] B. J. Choi, R. A. DeMillo, E. W. Krauser, R. J. Martin, A. P. Mathur, A. J. Offutt, H. Pan, and E. H. Spafford. The Mothra tools set. In *Proceedings of the 22nd Hawaii International Conference on Systems and Software*, pages 275–284, Kona, HI, January 1989.

[3] S. D. Conte, H. E. Dunsmore, and V. Y. Shen. *Software Engineering Metrics and Models*. Benjamin/Cummings, 1986.

[4] The Cuckoo's Egg. *Clifford Stoll.* Doubleday, New York, NY, 1989.

[5] Mark W. Eichin and Jon A. Rochlis. With microscope and tweezers: an analysis of the Internet virus of November 1988. In *Proceedings of the Symposium on Research in Security and Privacy*, Oakland, CA, May 1989. IEEE-CS.

[6] L.W. Cannon et. al. *Recommended C Style and Coding Standards.* Pocket reference guide. Specialized Systems Consultants, 1991. Updated version of AT&T's Indian Hill coding guidelines.

[7] Joseph A. Fanciulli. The process of handwriting comparison. *FBI Law Enforcement Bulletin*, pages 5–8, October 1979.

[8] M. F. Interrante and Z. Basrawala. Reverse engineering annotated bibliography. Technical Report SERC-TR-12-F, Software Engineering Research Center, University of Florida, January 1988.

[9] Brian W. Kernighan and P. J. Plauger. *The Elements of Programming Style.* Mcgraw-Hill, second edition, 1978.

[10] V. Klement, R. Naske, and K. Steinke. The application of image processing and pattern recognition techniques to the forensic analysis of handwriting. In *1980 International Conference: Security Through Science and Engineering*, pages 5–11, 1980.

[11] Henry F. Ledgard, Paul A. Nagin, and John F. Hueras. *Pascal with Style.* Hayden, 1979.

[12] Frederick Mosteller and David L. Wallace. *Applied Bayesian and Classical Inference: The Case of the Federalist Papers.* Springer Series in Statistics. Springer-Verlag, 1964.

[13] Charles Simonyi. Meta-programming: A software production technique. *Byte*, pages 34–45, September 1991.

[14] Herbert Solomon. *Confidence Intervals in Legal Settings*, pages 455–473. John Wiley & Sons, 1986.

[15] Eugene H. Spafford. The Internet worm program: an analysis. *Computer Communication Review*, 19(1), January 1989. Also issued as Purdue CS technical report TR-CSD-823.

[16] Eugene H. Spafford. Extending mutation testing to find environmental bugs. *Software Practice and Experience*, 20(2):181–189, February 1990.

[17] Dennie Van Tassel. *Program Style, Design, Efficiency, Debugging, and Testing*. Prentice-Hall, Englewood Cliffs, NJ, second edition, 1978.