Dynamic Software Birthmarks to Detect the Theft of Windows Applications

Haruaki Tamada **Keiji Okamoto** **Masahide Nakamura** **Akito Monden**
Ken-ichi Matsumoto
Graduate School of Information Science,
Nara Institute of Science and Technology,
8916-5, Takayama, Ikoma, Nara 630-0101, Japan,
Email: {harua-t, keiji-o, masa-n, akito-m, matumoto}@is.naist.jp

ABSTRACT
This paper proposes dynamic software birthmarks which can be extracted during execution of Windows applications. Birthmarks are unique and native characteristics of software. For a pair of software p and q, if q has the same birthmarks as p's, q is suspected as a copy of p. Our security analysis showed that the proposed birthmark has good tolerance against various kinds of program transformation attacks.

Keywords
Copyright Protection, Birthmark, Dynamic Analysis

1 INTRODUCTION

Software theft is a growing concern in today's highly competitive world of computer software. Software theft includes not only duplicating a whole software product and selling the copies (called *software piracy*) but also stealing a part of a product (*e.g.* modules and code fragments) and reusing it in other product without permission. Indeed, many incidents have been reported, for example, distribution of WAREZ (pirated software) whose license checks were defeated [17], SCO Group's lawsuit against IBM for ownership violation [19], copyright infringement of freeware/shareware [23], and GPL infringement [7]. Software theft can cause severe damage to the software industries; hence, companies must protect their own intellectual properties from theft.

However, protecting software from such theft is not easy. Since enormous amounts of software products are distributed today, even detecting "suspected copies" is quite difficult, unless the product is well-known to the public. Moreover, each product generally consists of many modules and data files. Suppose that an adversary steals only some modules, builds them into his or her own code, and distributes this "new" code without the source code. Detection of the theft becomes much more difficult because finding the evidence of copying by the manual binary analysis generally requires a large amount of cost and extra-high skill.

This paper presents an easy-to-use method to provide reasonable evidence for the theft of programs. Specifically, we propose *birthmarks* to support the efficient detection of programs that are quite similar to (or exactly the same as) each other. Intuitively, a birthmark of a programs is the set of unique characteristics that the program *originally* possessed. If program p and q have the same birthmark, q is very likely to be a stolen copy of p (and vice versa).

In our previous work, we proposed four types of *static birthmarks* of Java class files [21]: (1) constant values in field variables, (2) a sequence of method calls, (3) an inheritance structure, and (4) used classes. However, these birthmarks are vulnerable against manual (hand) modifications. This paper proposes *dynamic birthmarks* using the history of API (Application Program Interface) function calls of a running program. Since it is difficult to change the dynamic behavior of a program without changing its functionality, the proposed birthmark is more resilient to program transformation attacks. Through security analyzes, this paper shows that our birthmark is reasonably resilient against various kinds of imaginable attacks.

2 PROPOSED METHOD

2.1 Key Idea

Applications running on the OS can use many features called API function calls, provided by the OS. The typical API function calls are file input/output, synchronized objects such as semaphore, mutex and critical section, user interface (Window system) and graphics.

Since the most of API function calls cannot be replaced by other instructions without affecting the program behavior, history of their executions can be used as robust birthmarks For example, the high level OS does not allow direct operations to the file system from user applications, and it only allows file input/output via API function calls. Also, the operations to GUIs are allowed only via API function calls. It indicates that the birthmark using API function calls has good tolerance against program transformation attacks.

This paper proposes a method for collecting history of API function calls during execution of software from which we extract birthmarks. As the target software, we limit the type of software to those using API function calls of Microsoft Windows.

In this paper, we proposed two types of birthmarks. First birthmark is a sequence of API function calls. The other is a frequency of API function calls. Both birthmarks are obtained by logging the API function calls of target software. We describe the detail of proposed birthmarks in Section 2.3.

2.2 Definition

To make our discussion clearer, this subsection formulates a notion of a birthmark. We start with formulation of the *copy relation* of programs.

Definition 1 (Copy Relation) Let $Prog$ be a set of given programs. Let $\equiv_{cp}$ denote an equivalent relation over $Prog$ such that: for $p, q \in Prog$, $p \equiv_{cp} q$ holds iff q is a *copy* of p (vice versa). The relation $\equiv_{cp}$ is called a *copy relation*.

The criteria for whether or not q is a *copy* of p can vary depending on the context. For example, each of the following criterion is relatively reasonable for general computer programs:

(a) q is an exact duplication of p,

(b) q is obtained from p by renaming all identifiers in the source code of p, or

(c) q is obtained from p by eliminating all the comment lines in the source code of p.

To avoid confusion, we suppose that $\equiv_{cp}$ is *originally given* by the user. Since $\equiv_{cp}$ is an equivalent relation, the following proposition holds.

Proposition 1 *For $p, q \in Prog$, the following properties hold. (Reflexive) $p \equiv_{cp} p$, (Symmetric) $p \equiv_{cp} q \Rightarrow q \equiv_{cp} p$, (Transitive) $p \equiv_{cp} q \wedge q \equiv_{cp} r \Rightarrow p \equiv_{cp} r$.*

All the above properties meet well the intuition of copy. Next, if q is a copy of p, the external behavior of q should be identical to p's.

Proposition 2 *Let $Spec(p)$ be a (external) specification conformed by p. Then, the following property holds: $p \equiv_{cp} q \Rightarrow Spec(p) = Spec(q)$.*

Note that the inverse of this proposition does not necessarily hold since we can see, in general, different program implementations conforming the same specification. Now we are ready to define a *dynamic birthmark* of a program. Note that we have defined a static birthmark of program in [21].

Definition 2 (Birthmark) Let p, q be programs, I be a given input and $\equiv_{cp}$ be a given copy relation. Let $f(p, I)$ be a set of characteristics extracted from p with input I by a certain method f. Then $f(p, I)$ is called a *dynamic birthmark* of p under $\equiv_{cp}$ iff both of the following conditions are satisfied.

Condition 1 *$f(p, I)$ is obtained from p itself and given input I without any extra information.*

Condition 2 $p \equiv_{cp} q \Rightarrow f(p, I) = f(q, I)$

Condition 1 means that the birthmark is not extra information and is required for p to run. Hence, extracting a birthmark does not require extra code as watermarking does. Condition 2 states that the same birthmark has to be obtained from copied programs. By contraposition, if birthmarks $f(p, I)$ and $f(q, I)$ are different, then $p \not\equiv_{cp} q$ holds. That is, we can guarantee that q is not a copy of p.

Hopefully, a birthmark will satisfy the following properties.

Property 1 (Preservation) *For p' obtained from p by any program transformation, $f(p, I) = f(p', I)$ holds.*

Property 2 (Distinction) *For p and q such that $Spec(p) = Spec(q)$, if p and q are written independently, then $f(p, I) \neq f(q, I)$.*

These properties strengthen Condition 2 of Definition 2. Property 1 specifies the *preservation property* of the birthmark against program transformation. We consider that clever crackers may try to modify birthmarks by transforming the original program into an equivalent one to hide the fact of theft. There are several automated tools to perform the transformation, involving program *obfuscators* and *optimizers*. These tools can be used as a means of attack against the birthmarks. Property 1 specifies that the same birthmark must be obtained from p and converted p'. However, there exist many ways to transform a program into an equivalent one. Hence, in reality, it is difficult to extract strong enough birthmarks to perfectly satisfy Property 1.

Property 2 specifies the distinction property of the birthmark, stating that: even though the specification of p and q is the same, if implemented separately, different birthmarks should be extracted. In general, the detail of two independent programs is almost never completely the same. However, in the case that p and q are both *tiny* programs, extracted birthmarks could become the same, even if p and q are written independently. Those properties should be tuned within an allowable range at the user's discretion.

The question is how to develop an effective method f for a set $Prog$ of programs and the copy relation $\equiv_{cp}$.

2.3 Proposed Birthmarks

2.3.1 Sequence of API Function Calls Birthmark

The order of API function calls during execution of software is considered unique characteristics of software. Changing the order of API function calls causes introduction of bugs or changing of the specification of software. Moreover, it is quite a rare case that two different software have the same order of API function calls even if they are using the same set

of API functions. Hence, it is convenient to use these characteristics as birthmarks. Below we formulate the sequence of API function calls birthmark.

Definition 3 (EXESEQ) Let p be a given program, I be a given input to p and W be a given set of API function names. Let $w_1, w_2, ..., w_n$ be a sequence of function calls called by executing p (execution order). If $w_i (1 \leq i \leq n)$ does not belong to W, we eliminate it from the sequence. Then, the resultant sequence $(w_1, w_2, ..., w_m)$ is called an *EXESEQ birthmark* of p, denoted by *EXESEQ(p, I)*.

2.3.2 Frequency of API Function Calls Birthmark

The number of calls each API function per unit time will not change even if an adversary could change the order of API function calls by a semantic analysis hacking. Therefore, the frequency of API function calls can be used as a good signature to characterize the software.

Definition 4 (EXEFREQ) Let p be a given program, I be a given input to p and $(w_1, w_2, ..., w_n)$ be a sequence of API function call birthmark of p (*EXESEQ(p, I)*). Let $(w'_1, w'_2, ..., w'_m)$ be a sequence of function names which obtained by eliminating duplicated function names from *EXESEQ(p, I)*. Also, let $k_i (1 \leq i \leq m)$ be a function name of w'_i and a_i be the number of appearance of k_i in *EXESEQ(p, I)*. Then, the sequence $((k_1, a_1), (k_2, a_2), ..., (k_m, a_m))$ is called an *EXEFREQ birthmark* of p, denoted by *EXEFREQ(p, I)*.

3 IMPLEMENTATION

3.1 Outline

In the proposed method, we must observe the API function calls during execution of programs. Unfortunately, in general use, we cannot observe the API function calls in Microsoft Windows platform. Windows platform has a function called "Windows hook" which enables us to observe message passings of GUI objects, keyboard and mouse. However, by using Windows hook, we cannot observe enough information to extract proposed birthmarks. Therefore, we propose a way to implement the birthmark extraction system as below.

Step 1 Installing (inserting) a routine for observing API function calls into target software.

Step 2 Changing "function call pointer table" in order to fetch API function calls.

Step 3 Intercepting and recording API function calls in the installed routine.

Step 4 Calling original API function calls by the installed routine.

Step 5 Extracting birthmarks.

We explain details of each step from next section.

3.2 Details of Implementation

3.2.1 Step 1: Installing the Observation Routine of API Function Calls

In order to record the API function calls, we must put routine for observing API function calls into target software. In this step, we build an observation routine as a DLL (dynamic link library); then, we exploit Windows hook mechanism to get this DLL forcibly loaded into target software. We call this DLL *parasite DLL* because the DLL becomes parasitic on the target software. We also call the routine in the parasite DLL *parasite routine*.

Figure 1 shows the mechanism of Windows hook. In the Windows platform, messages (e.g., mouse click and key input) are generally passed from the operating system to window objects (GUI widget) (Fig.1: dashed line arrow). We can take over the message using Windows hook (Fig.1: solid line arrow). Windows hook has a functionality to get any DLL loaded into the target software so that the DLL can intercept the messages in the target software during its execution. In this step, we do not use the mechanism of message interception of Windows hook. We only exploit the feature for loading the DLL to get our parasite DLL loaded into the target software.

3.2.2 Step 2: Changing the API Function Pointer Table

In this step, we highlight on the API function pointer table called *import section* whom every Windows software has. The import section is a pointer table comprising function pointers to the DLL implementing an API function. In figure 2, we show a generic API function call as a dashed line arrow. When Windows software calls an API function, the software obtains a function pointer of the corresponding DLL from the import section. Then, the DLL is executed through the pointer. Therefore, by changing the function pointer of the import section, we can change the to-be-executed DLL for an arbitrary API function call.

When our parasite DLL is loaded into the target software in Step 1, DLL-initialization code in this DLL is immediately executed (just as common DLLs). By exploiting this

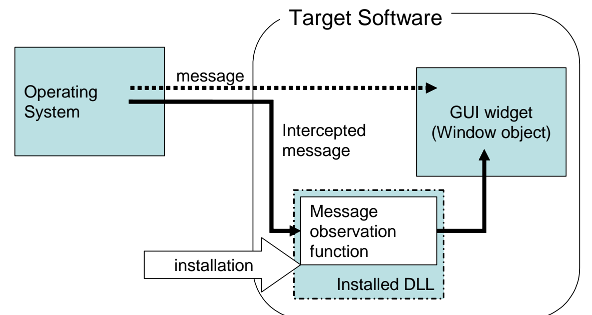

Figure 1: Installation of the DLL into target software during execution

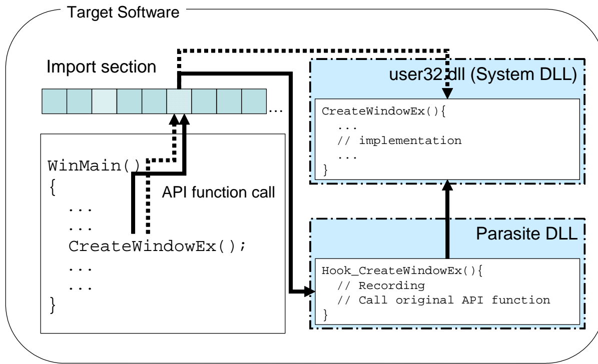

Figure 2: Mechanism of API function call observation

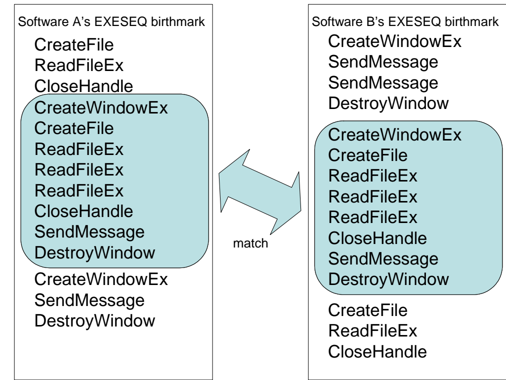

Figure 3: Example of EXESEQ birthmarks

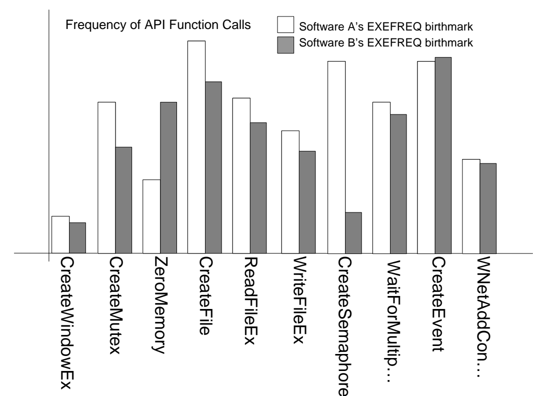

Figure 4: Example of EXEFREQ birthmark

mechanism, our DLL-initialization code overwrites a function pointer entry of an API function which we want to observe, by an alternative pointer of wrapper function we prepared in parasite DLL.

3.2.3 Step 3: Recording the History of API Function Calls

When software calls the API function, the wrapper function in the parasite DLL is called instead of original API. As Step 3, in the wrapper function, we record the following three types of information: the name of called API function, the time when the function is called, and the current thread id.

3.2.4 Step 4: Calling the Original Function

Software containing parasite DLL must keep the original software specification. Hence, in Step 4, our wrapper function calls an original API function after executing Step 3, and returns its output value. We show this behavior with solid line arrow in Figure 2. By this Step 4, the specification of original software is preserved.

3.2.5 Step 5: Extracting Birthmarks

By Step 1...4, we can collect the information of API function calls by running the software. If we found software having a suspicious function that seems to be implemented by a stolen software module or code, we can execute Step 1,..4 and run the software, then operate the suspicious function and recored API function call information from it. From the recorded API function call information, we can extract two birthmarks, EXESEQ and EXEFREQ birthmarks.

3.3 Comparing Birthmarks

3.3.1 Sequence of API Function Calls Birthmark

When we extract the EXESEQ birthmark from the recorded the sequence of API function calls, we must not use API function calls whose execution timing are unstable, e.g. ones related to the timer or exceptions. We must use stable API function calls to guarantee that the same birthmark is always extracted from a given particular program p and an input I.

To compare the EXESEQ birthmarks extracted from different software p and q, we can use string matching tools such as *diff* and *CCFinder* [9]. These tools can detect exact or nearly exact duplicated part of text sequences. If a part of p's EXESEQ birthmark are the same as q's, we can suspect that q contains p's code.

In figure 3, we show an example of comparing two EXESEQ birthmarks. In this example, gray parts indicate detected match between two birthmarks.

3.3.2 Frequency of API Function Calls Birthmark

We construct a vector to compare the EXEFREQ birthmarks, whose elements are the number of each API function calls. We define that the *similarity* of two EXEFREQ birthmarks is the angle of two vectors. Figure 4 shows an example of two EXEFREQ birthmarks. If a part of software, e.g. a DLL file, was suspected as a stolen code, we can extract an EXEFREQ birthmark of this suspected DLL and compare it with an original DLL's birthmark.

4 SECURITY ANALYSIS

In this section, we discuss the tolerance of the proposed birthmarks against program modification attacks.

4.1 Tolerance against Program Transformation Tools

Software obfuscation is a method for protecting intellectual property of software; however, this can also be used as an attacking tool to modify the birthmarks. Obfuscation translates a program so that it becomes more difficult to understand, yet is functionally equivalent to the original program. There are several obfuscation techniques for obfuscating binary programs [11, 12].

Here we discuss whether or not the proposed birthmarks have the tolerance against program obfuscators. The obfuscator statically analyzes an binary program or its source code before translating it. Based on the analysis, the obfuscator may translate jumps and calls for internal functions and may change the control flow of the program [11]. Some of our previously-proposed static birthmarks can be tampered with this attack [21]. However, the obfuscator cannot translate calls for external libraries because substitution of an external library call with another one is generally quite difficult.

Therefore, the proposed birthmarks are robust against program transformation attacks using obfuscators.

4.2 Tolerance against Manual Modification

An adversary would try to modify the birthmarks by *manual hacking*. In this case, the adversary analyzes the program and understands its specification; then, he or she is ready to replace API function calls into other equivalent ones.

First, we discuss whether or not the adversary can delete an API function call from the program. If there exists a set of functions that is functionally equivalent to the API function, then, the adversary is able to delete the API function call by replacing it to this function set. However, there are several Windows API function calls that are not substitutable, for example, an API function for creating GUI objects can not be implemented with other functions since it is a fundamental part of Microsoft Windows. Therefore, by using such "fundamental" API functions in our birthmark, the proposed birthmark becomes robust against manual modification attacks.

Next, we discuss whether or not the adversary is able to change the execution order of API functions. For some sequences of API function calls, it seems possible for the adversary to change its order without affecting the program specification. If the adversary succeeded in this attack, the EXESEQ birthmark of the attached program is changed. However, the EXEFREQ birthmark tolerates against this attack. Anyway, since manual binary analysis and modification is a quite time consuming task, and it requires extra-high skill, we believe there will be few adversaries that can tamper with our birthmark.

5 RELATED WORK

Software watermarking (often called *software fingerprinting*) is a well-known technique used to provide a way to prove ownership of stolen software. Therefore, it may be used for our objective. Watermarking is basically used to embed stealthy information in a piece of software, such as a software developer's copyright notation or a unique identifier of software in a static manner [6, 13, 14], or in a dynamic manner [4, 5, 22]. Unfortunately, watermarking is not always feasible because it requires software developers to embed a watermark *before* releasing the software. Thus, proofs cannot be given for already-released software without watermarks. In addition, strictly speaking, to protect all the modules in a software package, we need to embed watermarks into all of these modules. This is generally difficult to meet when the number of modules is large. Our birthmark approach provides a way to detect stolen software without embedding any additional information beforehand.

The most commonly used technique to detect a suspected copy is software similarity computation [1, 3, 18, 24], which is generally used for *plagiarism detection* in programming classes. A plagiarized program is defined as a program that has been reproduced from another program with only a small number of editions, and with no detailed understanding of the program required [16]. In order to detect plagiarized programs, various methods for similarity computation have been proposed based on attribute counting [15], structure metrics [1, 18, 24], and Kolmogorov complexity [3]. Unfortunately, since these methods require the source code of software to compute the similarity, they are not applicable in our problem setting where software products are usually distributed without a source code. Moreover, these techniques did not consider attacks by practical code transformation tools, such as software optimizers and obfuscators.

Another way to detect software theft is to find code clone pairs between two software products[2, 9]. Code clones are exact or nearly exact duplicate lines of code within the source code. In [9], by using a code clone detection tool called *CCFinder*, Kamiya et al. found that `sys/net/zlib.c` of FreeBSD and `drivers/net/zlib.c` of Linux are almost identical. Such code clone techniques are useful for detecting suspected copies; however, they are also susceptible to code transformation attacks.

Finally, we consider authorship analysis methods [10, 20]. In [10], programming style metrics and programming layout metrics are used to identify the author of a source code. In [20], Spafford and Weeber suggest that it might be feasible to analyze code remnants in executable code, such as data structures and algorithms and choice of system and library calls made by the programmer, which are typically the remains of a virus or Trojan horse, and identify its author. Such identifying information are called a "software birthmark" by Grover [8], although we proposed its formal definition in Section 2.2.

6 CONCLUSION

In this paper we proposed dynamic birthmarks to provide a reasonable evidence of theft of software. First we gave a

formal definition of a dynamic birthmark, then proposed two types of birthmarks, EXESEQ and EXEFREQ birthmark. Our preliminary security analysis showed that our birthmarks are reasonably robust against obfuscator attacks and manual hacking attacks.

In the future, we are planning to conduct experiments for evaluating proposed birthmarks. Investigation of other types of birthmarks is also an interesting future work.

REFERENCES

1. Alex Aiken. MOSS: A system for detecting software plagiarism, Jun 2004. http://www.cs.berkeley.edu/ aiken/moss.html.
2. Ira D. Baxter, Andrew Yahin, Leonardo M. De Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *ICSM: the International Conference on Software Maintenance*, pages 368–377, 1998.
3. Xin Chen, Brent Francia, Ming Li, Brian Mckinnon, and Amit Seker. SID plagiarism detection, Jun 2004. http://genome.math.uwaterloo.ca/SID/.
4. Christian Collberg. Sandmark: A tool for the study of software protection algorithms, 2000. http://www.cs.arizona.edu/sandmark/.
5. Christian Collberg and Clark Thomborson. Software watermarking: Models and dynamic embeddings. In *Principles of Programming Languages 1999, POPL'99*, pages 311–324, San Antonio, TX, Jan 1999.
6. Robert L. Davidson and Nathan Myhrvold. Method and system for generating and auditing a signature for a computer program, 1996. US Patent 5,559,884, Assignee: Microsoft Corporation.
7. Epson pulls linux software following gpl violations (slashdot.org), Sep 2002. http://slashdot.org/article.pl?sid=02/09/11/2225212.
8. Derrick Grover, editor. *The protection of computer software – its technology and applications Second edition*. The British Computer Society Monographs in Informatics Cambridge University Press, May 1992.
9. Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Trans. on Software Engineering*, 28(7):654–670, 2002.
10. Ivan Krsul and Eugene H. Spafford. Authorship analysis: identifying the author of a program. *Computers and Security*, 16(3):233–257, 1997.
11. Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proc. of the 10th ACM conference on Computer and communication security*, pages 290–299. ACM Press, 2003.
12. Masahiro Mambo, Takanori Murayama, and Eiji Okamoto. A tentative approach to constructing tamper-resistant software. In *Proc. of the 1997 workshop on New security paradigms*, pages 23–33. ACM Press, 1997.
13. Akito Monden. jmark: A lightweight tool for watermarking java class files, 2002. http://se.aist-nara.ac.jp/jmark/.
14. Akito Monden, Hajimu Iida, Kenichi Matsumoto, Katsuro Inoue, and Koji Torii. A practical method for watermarking java programs. In *Proc. COMPSAC 2000, 24th Computer Software and Applications Conference*, pages 191–197, 2000.
15. Karl J. Ottenstein. An algorithmic approach to the detection and prevention of plagiarism. *SIGCSE Bulletin*, 8(4):30–41, 1976.
16. Alan Parker and Hamblen O. James. Computer algorithms for plagiarism detection. *IEEE Transactions on Education*, 32(2):94–99, May 1989.
17. Andy Patrizio. Pirates experience Office XP (wired news), Mar 2001. http://www.wired.com/news/business/0,1367,42402,00.html.
18. Lutz Prechelt, Guido Malpohl, and Michael Philippsen. Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science*, 8(11):1038–1116, 2002.
19. Eric Raymond and Rob Landley. OSI position paper on the sco-vs.-ibm complaint, May 2004. http://www.opensource.org/sco-vs-ibm.html.
20. Eugene H. Spafford and Stephen A. Weeber. Software forensics: Can we track code to its authors? *Computers and Security*, 12(6):585–595, 1993.
21. Haruaki Tamada, Masahide Nakamura, Akito Monden, and Ken'ichi Matsumoto. Design and evaluation of birthmarks for detecting theft of java programs. In *Proc. IASTED International Conference on Software Engineering (IASTED SE 2004)*, pages 569–575, Innsbruck, Feb 2004.
22. Clark Thomborson, Jasvir Nagra, Ram Somaraju, and Charles He. Tamper-proofing software watermarks. In *Proc. of the second workshop on Australasian information security, Data Mining and Web Intelligence, and Software Internationalisation*, volume 32, pages 27–36. Australian Computer Society, Inc., 2004.
23. Tomohiro Ueno. The protest page to pocketmascot, Sep 2001. http://members.jcom.home.ne.jp/tomohiro-ueno/About_PocketMascot/About_PocketMascot_e.html.
24. Michael J. Wise. YAP3: Improved detection of similarities in computer program and other texts. In *Proc. of 27th SCGCSE technical Symposium*, pages 130–134, 1996.