Detecting Software Theft via Whole Program Path Birthmarks

Ginger Myles and Christian Collberg

Department of Computer Science, University of Arizona, Tucson, AZ, 85721, USA
{mylesg,collberg}@cs.arizona.edu

Abstract. A software birthmark is a unique characteristic of a program that can be used as a software theft detection technique. In this paper we present and empirically evaluate a novel birthmarking technique — *Whole Program Path Birthmarking* — which uniquely identifies a program based on a complete control flow trace of its execution. To evaluate the strength of the proposed technique we examine two important properties: credibility and tolerance against program transformations such as optimization and obfuscation. Our evaluation demonstrates that, for the detection of theft of an entire program, Whole Program Path birthmarks are more resilient to attack than previously proposed techniques. In addition, we illustrate several instances where a birthmark can be used to identify program theft even when an embedded watermark was destroyed by program transformation.

Keywords: software piracy, copyright protection, software birthmark.

1 Introduction

Suppose Alice creates a program $\mathcal{A}$ which she sells to Bob. Subsequently, Alice discovers Bob is selling a program $\mathcal{B}$ which is remarkably similar to $\mathcal{A}$. Alice suspects Bob copied $\mathcal{A}$ and is reselling it under the new name. In order to take legal action, Alice needs to be able to prove that $\mathcal{B}$ is indeed a copy of $\mathcal{A}$. In this paper we will describe a technique known as *software birthmarking* which can be used to provide such proof.

A software birthmark is a unique characteristic, or set of characteristics, that a program possesses and which can be used to identify the program. The general idea is that if two programs p and q both have the same birthmark then it is highly likely that one is a copy of the other. There are two important properties of a birthmarking technique that must be considered: the detector should not produce false positives (i.e. it should not say that p and q originate from the same source, if, in fact, they do not), and it should be resilient to semantics preserving transformations (such as optimization and obfuscation) that an attacker may launch in order to defeat the detector. In this paper we propose and evaluate a new software birthmarking technique we call *Whole Program Path Birthmarks* (WPPB). WPPB is a *dynamic* technique, relying on the *execution pattern* of the program to detect the birthmark. This is in contrast to previously

K. Zhang and Y. Zheng (Eds.): ISC 2004, LNCS 3225, pp. 404–415, 2004.

proposed techniques which are *static*, i.e. they compute the birthmark based on the characteristics of the program source or binary code. We will show that the WPPB technique is more resilient to attacks by semantic-preserving transformations than published static techniques.

This paper makes the following contributions:

1. We introduce a new category of software birthmarks which we call *dynamic birthmarks*.
2. We propose and evaluate a new dynamic birthmarking technique based on *Whole Program Paths* [16].
3. We evaluate the four static birthmarking techniques proposed by Tamada, et al. [23,24] and show that they are easily defeated by current code obfuscation tools.
4. We provide an empirical evaluation between our WPPB technique and Tamada's birthmarks, and demonstrate that WPPBs are less vulnerable to attacks by semantics-preserving transformations.
5. Finally, we show that birthmarks can be used to identify program theft even when an embedded watermark has been destroyed by a program transformation.

2 Related Work

There are three major threats recognized against the intellectual property contained in software. *Software piracy* is the illegal reselling of legally obtained copies of a program. *Software tampering* is the illegal modification of a program to circumvent license checks, to obtain access to digital media protected by the software, etc. *Malicious reverse engineering* is the extracting of a piece of a program in order to reuse it in ones own.

A variety of techniques have been proposed to address these attacks. Each technique targets a different attack and can often be combined to produce a stronger defense. *Code obfuscation* [12] is a technique developed to aid in the prevention of reverse engineering. An obfuscation is a semantics-preserving transformation which makes the program more difficult to understand and reverse engineer. Probably the most well-known technique for detecting software piracy is *software watermarking* [7,9,11,14,17,20,22,25]. The basic idea is to embed a unique identifier in the program. Piracy is confirmed by proving the program contains the watermark.

A lesser known technique for the detection of theft is *software birthmarks*. Software birthmarks differ from software watermarks in two important ways. First, it is often necessary to add code to the application in order to embed a watermark. In the case of a birthmark additional code is never needed. Instead a birthmark relies on an inherent characteristic of the application to show that one program is a copy of another. Secondly, a birthmark cannot prove authorship or be used to identify the source of an illegal redistribution. Rather, a birthmark can only confirm that one program is a copy of another. A strong birthmark will

be able to provide such confirmation even when code transformations have been applied to the code by the adversary in order to hide the theft.

One of the first occurrences of the use of the term birthmark was by Grover [15] where the term was used to mean characteristics occurring in the program by chance which could be used to aid in program identification. This term was distinguished from a fingerprint in that the characteristics used to embed the fingerprint are intentionally placed in the code. The general idea of a software birthmark is similar to that of a computer virus signature. An early example of the use of birthmarks was in an IBM court case [6]. In this case IBM used the order in which the registers were pushed and popped to prove that their PC-AT ROM had been illegally copied.

Tamada, et al. [23,24] have proposed four birthmarks that are specific to Java class files: constant values in field variables (CVFV), sequence of method calls (SMC), inheritance structure (IS), and used classes (UC). The CVFV birthmark extracts information about the variables declared in the class. For each variable the type t_i is extracted along with the initial value a_i. The birthmark is then the sequence $((t_1, a_1), (t_2, a_2), ..., (t_n, a_n))$. SMC examines the sequence of method calls as they appear in the class, but not necessarily in execution order. Because it is easy to change the names of the methods within the application only those method calls which are in a set of well-known classes are considered in the sequence. IS extracts the inheritance structure of the class. The birthmark is constructed by traversing the superclasses of the class back to `java.lang.Object`. All classes which are in the set of well-known classes are included in the sequence. The UC birthmark examines all classes which are used by a given class, i.e. they appear as a superclass of the given class, the return or argument types of a method, the types of fields, etc. All classes in the set of well-known classes are included in the sequence which is then arranged in alphabetical order. As we will see in Sect. 5 Tamada's birthmarks are easily defeated by applying simple code obfuscating transformations to the program.

Plagiarism detection is another area which is very similar to software birthmarking. A variety of plagiarism detection techniques have been proposed (e.g. Moss [5,21], Plaque [26], and YAP [27]) which have been quite successful at detecting plagiarism within student programs. Unfortunately, these systems compute similarity at the source code level. In many instances source code is unavailable. In addition, these systems do not consider semantics-preserving transformations and the effects of decompilation on the formatting of the source code. For example, it was shown by Collberg, et al. [10] that given the source code of a Java application, simply compiling then decompiling will cause Moss to indicate 0% similarity between the original and the decompiled source code.

3 Software Birthmarks

Before we can precisely define the idea of a birthmark we must define what it means for a program q to be a *copy* of another program p. The most obvious definition is where q is an exact duplicate of p. However, in order to hide the

fact that copying has taken place an attacker might apply semantics-preserving transformations to q. For example, all of the identifiers in q might have been renamed or an optimizing register allocator might have been applied to q so that q and p now have different register assignments. In this case we would still like to be able to say that q is a copy of p. In addition, it is important that our definition reflects that if q is a copy of p then p and q should exhibit the same external behavior. (Note that the reverse of this property does not necessary hold. It is possible to find two programs which exhibit the same external behavior but are not copies. An example is iterative and recursive versions of the same function.)

The following definition of a software birthmark is a restatement of the definition given by Tamada, et al. [23,24].

Definition 1 (Birthmark). *Let p, q be programs. Let f be a method for extracting a set of characteristics from a program. Then $f(p)$ is a birthmark of p iff:*

1. *$f(p)$ is obtained only from p itself (without any extra information), and*
2. *q is a copy of $p \Rightarrow f(p) = f(q)$.*

As with software watermarking we can characterize a birthmark as either static or dynamic. A static birthmark extracts the set of characteristics from the statically available information in a program such as information about the types or initial values of the fields. A dynamic birthmark relies on information gathered from the execution of the application. A dynamic algorithm typically works at the program level whereas a static algorithm targets an entire program or individual modules within the program. The same distinction is true with static and dynamic watermarking algorithms. A dynamic algorithm can provide evidence if an entire program is stolen and a static algorithm may be able to detect the theft of a single module. The four birthmark techniques proposed by Tamada, et al. are characterized as static and target class-level theft. Definition 1 above defines a static birthmark.

Definition 2 (Dynamic Birthmark). *Let p, q be programs and i an input to these programs. Let f be a method for extracting a set of characteristics from a program. Then $f(p, i)$ is a dynamic birthmark of p iff:*

1. *$f(p, i)$ is obtained only from p itself by executing p with the given input i, and*
2. *q is a copy of $p \Rightarrow f(p, i) = f(q, i)$.*

The Whole Program Path Birthmark proposed in this paper computes the birthmark from the execution trace of the program. It is therefore, a dynamic birthmark designed to detect program level theft.

3.1 Evaluating Software Birthmarks

We would like a birthmark technique to satisfy the following two properties.

Property 1 (Credibility). Let p and q be independently written programs which accomplish the same task. Then we say f is a credible measure if $f(p) \neq f(q)$.

Property 2 (Resistance to Transformation). Let p' be a program obtained from p by applying semantics-preserving transformations $\mathcal{T}$. Then we say f is resilient to $\mathcal{T}$ if $f(p) = f(p')$.

Property 1 is concerned with the possibility of the birthmark falsely indicating that q is a copy of p. This could occur with independently implemented programs which perform the same task. It is highly unlikely that two independently implemented algorithms will contain all of the same details so the birthmark should be designed to extract those details which are likely to differ.

Property 2 addresses the issue of identifying a copy in the presence of a transformation. With the proliferation of tools for code optimization and obfuscation, for example [1,2,3,4], it is highly probable that an attacker will apply at least one transformation prior to distributing an illegally copied program. It is desirable that a birthmark be able to detect a copy even if a transformation has been applied to that program.

4 Whole Program Path Based Birthmarks

In the next section we present the first known dynamic birthmark technique. Through experiments we have performed on the four techniques proposed by Tamada, et al. we believe they are susceptible to a variety of simple program transformations. Thus, there are other characteristics of a program which could be used to construct a stronger birthmark technique.

4.1 Whole Program Paths

Whole Program Paths (WPP) is a technique presented in [16] to represent a program's dynamic control flow. The WPP is constructed by collecting a trace of the path executed by the program. The trace is then transformed into a more compact form by identifying its regularity, which is repeated code. To collect the trace the edges of the program's control flow graph are instrumented, by uniquely labeling each edge. As the program executes the edges are recorded, producing a trace. The trace is then run through the SEQUITUR algorithm which compresses it and reveals its inherent regularity [18,19]. The output of the SEQUITUR algorithm is a context-free grammar from which a directed acyclic graph (DAG) is produced. Each rule of the grammar is composed of a non-terminal and a sequence of symbols which the non-terminal represents. To construct the DAG representation of the grammar a node is added for each unique symbol. For each rule an edge is added from the non-terminal to each of the symbols it represents. The final DAG is the WPP.

The construction of the WPP is illustrated in Fig. 1. At ⓐ a control flow graph with 6 basic blocks and 8 edges is constructed from the input program. The control flow graph is instrumented so that each edge is labeled. At ⓑ the program is executed producing an edge trace. The trace is run through the SEQUITUR algorithm at ⓒ to produce the given context-free grammar. This

grammar contains 3 unique non-terminals and 8 unique terminals. At ⓓ a DAG with 3 internal nodes, 8 leaf nodes, and 14 directed edges is constructed which represents the grammar.

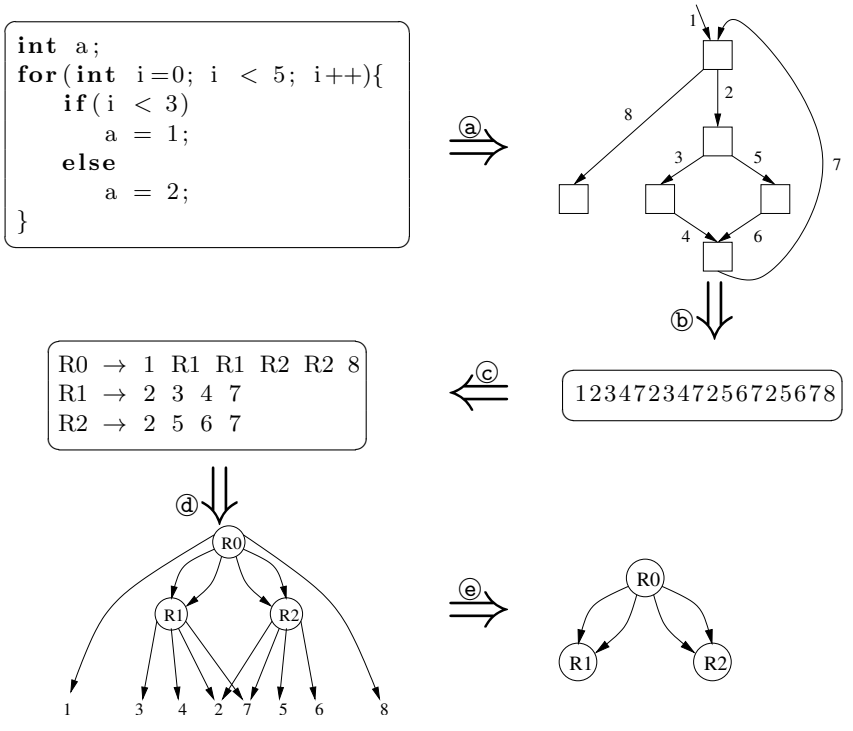

Fig. 1. An illustration of the stages involved in constructing a Whole Program Path (WPP). The construction begins with a program. A program control flow graph is constructed and instrumented. By executing the program on a given input an edge trace is constructed. This trace is run through the SEQUITUR algorithm to produce a context-free grammar. The grammar is then used to construct a directed acyclic graph which represents the WPP. All terminal nodes and corresponding edges are removed from the WPP to construct the WPP birthmark.

Our WPP birthmark is constructed in an identical manner as the WPP with the exception of the DAG in the final stage. An essential property of a birthmark is that it captures an inherent characteristic about the program which is difficult to modify through semantics-preserving transformations. The WPP birthmark captures the inherent regularity in the dynamic behavior of a program. Since we are only interested in the regularity we eliminate all terminal nodes in the DAG. It is the internal nodes which will be more difficult to modify through program transformations. Thus, the DAG in Fig. 1 is transformed into the birthmark of the example program at ⓔ.

4.2 Similarity of WPP Birthmarks

The WPP birthmark is in the form of a DAG. Suppose we have the birthmarks $f(p) = G_1$ and $f(q) = G_2$ for programs p and q. $f(p)$ and $f(q)$ are the same iff G_1 and G_2 are isomorphic. Since it is unlikely that q is an identical copy of p we would like to be able to say something about the similarity between $f(p)$ and $f(q)$. In other words, we would like to be able to conclude that q is a copy of p even in the presence of semantics-preserving transformations.

To compute similarity we use a slightly modified version of the graph distance metric in [8]. The similarity is based on finding a maximal common subgraph, G_3, between G_1 and G_2. The percentage of G_1 that we are able to identify in G_2 by finding the maximal common subgraph G_3 indicates the similarity between the two programs. The reason we are comparing the size of G_3 and G_1 instead of the maximum of G_1 and G_2 is that we are trying to identify a copy of p in q. We therefore want to know how much of G_1 is contained in G_2.

Definition 3 (Graph Distance). *The distance of two non-empty graphs* $G_1 = (V_1, E_1)$ *and* $G_2 = (V_2, E_2)$ *is defined as*

$$d(G_1, G_2) = \frac{|mcs(G_1, G_2)|}{|G_1|}$$

where $mcs(G_1, G_2)$ *is the maximum common subgraph of* G_1, G_2 *and* $|G| = |V| + |E|$.

Definition 4 (Similarity). *Let* $f(p) = G_1$ *and* $f(q) = G_2$ *be birthmarks extracted from programs* p *and* q. *The similarity between* $f(p)$ *and* $f(q)$ *is defined by:* $d(G_1, G_2) \times 100$.

5 Evaluation

To evaluate the effectiveness of the WPP birthmarking technique we examined its ability to satisfy the two properties from Sect. 3. We look at whether WPP birthmarks will produce false positives given two independently written applications which accomplish the same task and the tolerance of the birthmark against program transformations. As an additional evaluation we demonstrate how birthmarks can be used in conjunction with watermarking.

5.1 Credibility

To evaluate the credibility of WPP birthmarks we examined the ability to distinguish between two independently written applications which performed the same task. We looked at two problems: calculating a factorial and generating Fibonacci numbers. Each of these problems can be solved recursively and iteratively. The WPP birthmark found the factorial programs to be 50% similar and the Fibonacci programs 7% similar. From these results we are able to conclude

that the recursive and iterative forms of the programs were probably written independently.

Tamada, et al. [23,24] state that their birthmark techniques are unable to distinguish between independently written applications which are small. This is true given the Factorial and Fibonacci programs. Using the four birthmarks proposed by Tamada, et al. the recursive and iterative versions are found to be 100% similar. The only exception was SMC on factorial which had a similarity of 16%. Thus, with respect to small applications and credibility the WPP birthmarks provide stronger results.

5.2 Resistance to Transformation

To evaluate the WPP birthmark's resistance to transformation we applied various obfuscations and optimizations to automatically transform our test program into an equivalent, but not identical program. To perform the transformations we used Zelix Klassmaster (ZKM) [4], Smokescreen [3], Codeshield [1], and SandMark [2]. ZKM, Smokescreen, and Codeshield all include name obfuscation, the elimination of debugging information, and some type of control flow obfucations. Additionally, Smokescreen supports dead code elimination and ZKM includes string encryption. Our test program was a Java program `wc.jar` that works like the UNIX wc program.

For each of the tools except SandMark we applied the tool with the strongest level of obfuscation. The SandMark tool permitted us to pick and choose which obfuscations were applied to the program. SandMark includes 31 obfuscation algorithms which we applied individually to `wc.jar` obtaining 31 obfuscated programs. In addition, we applied multiple obfuscations in succession to `wc.jar`.

We computed the WPP birthmark for each of the transformed applications, the 31 from SandMark plus the three additional, as well as the original `wc.jar`. In every case the similarity between the original and the obfuscated applications was found to be 100%.

We performed the same evaluation of the four techniques proposed by Tamada, et al. Table 1 shows a comparison of the results with our WPP birthmark using ZKM, Smokescreen, and Codeshield. The table shows that only WPP and IS compute 100% for each of the three obfuscated programs. Even though IS computes 100% similarity we believe the technique is not strong enough to be used on its own. The reason for this is that the technique could produce many false positives for independently implemented programs which both do and do not perform the same task.

We also tested the four static birthmarks against each of the 31 obfuscations included in the SandMark tool. For CVFV, SMC, and UC we were able to find obfuscations which cast doubt on the similarity between the original and obfuscated version. Using the CVFV birthmark a less than 100% similarity was detected for the obfuscations Bogus Fields (75%), Node Splitter (0%), Objectify (66%), Opaque Branch Insertion (75%), and Transparent Branch Insertion (75%). When all five of these obfuscations were applied in conjunction to wc.jar CVFV detected a 0% similarity. The SMC birthmark detected a less than

100% similarity on four obfuscations: Buggy Code (69%), Primitive Promoter (5%), Static Method Bodies (82%), and Transparent Branch Insertion (83%). A similarity of 1% was detected when all four obfuscations were applied. Four obfuscations also caused the UC birthmark to detect a less than 100% similarity: Objectify (92%), Opaque Branch Insertion (92%), Primitive Promoter (56%), and Transparent Branch Insertion (92%). The combination of the obfuscations yielded a 52% similarity. These initial results indicate that the WPP birthmark is stronger then the four techniques proposed in [23,24] when theft of an entire application is in question.

Table 1. Similarity percentage found using each birthmark technique on an original and obfuscated version of `wc.jar`.

	ZKM	Smokescreen	Codeshield
WPP	100%	100%	100%
CVFV	66.7%	83.3%	83.3%
SMC	25.0%	15.9%	100%
IS	100%	100%	100%
UC	100%	100%	45.0%

We do know of two attacks that the WPP birthmark is currently vulnerable to. The first is any loop transformation that alters the loop in ways similar to loop unrolling or loop splitting. Executing the loop backwards, however, will not effect the WPP birthmark. WPP birthmarks are also vulnerable to method inlining in certain instances. If the method call occurs inside of a loop then inlining will not alter the birthmark. On the other hand, if the method is a helper method which is called from various locations throughout the program, inlining the method call will have an effect on the birthmark similarity.

5.3 Birthmarks and Watermarks

One limitation of software birthmarks is that they provide weaker evidence than software watermarks. They are only able to say that one program is likely to be a copy of another not who the original author is or who is guilty of piracy. However, birthmarks can be used in instances where watermarking is not feasible such as applications where code size is a concern and the watermark would insert additional code. Birthmarks can also be used in conjunction with watermarking to provide stronger evidence of theft. One such example is the watermarking algorithm proposed by Stern, et al. [22] which provides a probability that a specific watermark is contained in the program. If the watermarking algorithm does not 100% guarantee that the watermark is contained in the program then a birthmark could be used as additional evidence of theft. There are also instances where watermarks fail, e.g. an attacker is able to apply an obfuscation which destroys the watermark. In these instances a birthmark may still be able to

provide proof of program theft since the birthmark may be more resilient to transformations.

We were able to very easily construct three instances using the `wc.jar` program where a watermark is destroyed by an obfuscation, but WPP birthmarks still detect 100% similarity between the programs. In the first instance we used a very simple static watermarking algorithm which embeds the watermark by splitting it in half and using the first half to name a new field and the second in a name of a new method. We then applied an obfuscation which adds additional fields to the program. In the second instance the same watermarking algorithm is used but this time the obfuscation renames all of the identifiers in the program. In the third instance we watermarked the program using the algorithm proposed by Arboit [7] which encoded the watermark in opaque predicates [13] that are appended to various branches throughout the program. We then applied an obfuscation which adds opaque predicates to every boolean expression throughout the application. In each of these instance the watermark is destroyed which would have prevented piracy detection, but the WPP birthmark was able to detect 100% similarity.

6 Future Work

The most pressing future work is to conduct a more extensive evaluation of the WPP birthmark technique. The evaluation conducted in this paper was only preliminary and thus we would like to study the effectiveness on a larger set of test applications as well as more combinations of obfuscations.

As was discussed in Sect. 5.2 WPP birthmarks are susceptible to various loop transformations. To address this problem we want to evaluate the effectiveness of incorporating transformations, such as loop rerolling, in a preprocessing stage that would reverse the transformation. In addition, we would like to add functionality to the technique which would make it possible to target module level as well as program level theft. Once this functionality has been added we would like to evalute the effectiveness of WPP birthmarks in the detection of plagiarism within student programs.

Another interesting area of software birthmarks that should be explored is the combination of static and dynamic birthmarks. Unlike watermarks, where it is possible to destroy one watermark with another, two or more birthmarks can always be used in conjunction to provide stronger evidence of theft.

7 Summary

In this paper we expanded on the idea of software birthmarking by introducing dynamic birthmarks and in particular a specific dynamic birthmark called Whole Program Paths. We evaluated the technique with respect to two properties: credibility and resistance to transformation. In both evaluations the technique demonstrated promising results. WPP birthmarks did not falsely identify two independently written programs as being copies even though they perform the

same task. Based on the test program, `wc.jar`, and the available obfuscations WPP birthmarks calculated a similarity of 100% between the original and the transformed program. We also demonstrated how birthmarks can be used in conjunction with watermarks and in some instances are able to detect piracy even when the watermark has been destroyed.

References

1. Codeshield java bytecode obfuscator. http://www.codingart.com/codeshield.html.
2. Sandmark. http://www.cs.arizona.edu/sandmark/.
3. Smokescreen java obfuscator. http://leesw.com.
4. Zelix klassmaster. http://www.zelix.com/klassmaster/index.html.
5. Alex Aiken. Moss – a system for detecting software plagiarism. http://www.cs.berkeley.edu/ aiken/moss.html.
6. Ross J. Anderson and Fabien A. P. Petitcolas. On the limits of steganography. *IEEE Journal of Selected Areas in Communications*, 16(4):474–481, May 1998. Special issue on copyright & privacy protection.
7. Geneviève Arboit. A method for watermarking java programs via opaque predicates. In *The Fifth International Conference on Electronic Commerce Research (ICECR-5)*, 2002.
8. H. Bunke and K. Shearer. A graph distance metric based on the maximal common subgraph, 1998.
9. C. Collberg, E. Carter, S. Debray, A. Huntwork, C. Linn, and M. Stepp. Dynamic path-based software watermarking. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 04)*, 2004.
10. Christian Collberg, Ginger Myles, and Mike Stepp. Cheating cheating detectors. Technical Report TR04-05, University of Arizona, 2004.
11. Christian Collberg and Clark Thomborson. Software watermarking: Models and dynamic embeddings. In *In Conference Record of POPL '99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Jan. 1999)*, 1999. `http://citeseer.nj.nec.com/collberg99software.html`.
12. Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, University of Auckland, July 1997.
13. Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Principles of Programming Languages 1998, POPL'98*, San Diego, CA, January 1998.
14. R.L. Davidson and N. Myhrvold. Method and system for generating and auditing a signature for a computer program. US Patent 5,559,884, Assignee: Microsoft Corporation, 1996. `http://www.delphion.com/details?pn=US05559884__`.
15. Derrick Grover. Program identification. In Derrick Grover, editor, *The Protection of Computer Software – Its Technology and Applications*, pages 122–154. Cambridge University Press, 1989.
16. James R. Larus. Whole program paths. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 99)*, 1999.
17. A. Monden, H. Iida, K. Matsumoto, Katsuro Inoue, and Koji Torii. A practical method for watermarking java programs. In *compsac2000, 24th Computer Software and Applications Conference*, 2000.

18. C. G. Nevill-Manning and I. H. Witten. Compression and explanation using hierarchical grammars. *The Computer Journal*, 40(2/3), 1997.
19. C.G. Nevill-Manning and I.H. Witten. Linear-time, incremental hierarchy inference for compression. In *Proceedings of the Data Compression Conference (DCC '97)*, 1997.
20. Gang Qu and Miodrag Potkonjak. Hiding signatures in graph coloring solutions. In *Information Hiding*, pages 348–367, 1999.
21. Saul Schleimer, Daniel Wilkerson, and Alex Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the 2003 SIGMOD Conference*, 2003.
22. Julien P. Stern, Gael Hachez, Francois Koeune, and Jean-Jacques Quisquater. Robust object watermarking: Application to code. In *Information Hiding*, pages 368–378, 1999. `http://citeseer.nj.nec.com/stern00robust.html`.
23. Haruaki Tamada, Masahide Nakamura, Akito Monden, and Kenichi Matsumoto. Detecting the theft of programs using birthmarks. Information Science Technical Report NAIST-IS-TR2003014 ISSN 0919-9527, Graduate School of Information Science, Nara Institute of Science and Technology, Nov 2003.
24. Haruaki Tamada, Masahide Nakamura, Akito Monden, and Kenichi 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, Feb 2004.
25. Ramarathnam Venkatesan, Vijay Vazirani, and Saurabh Sinha. A graph theoretic approach to software watermarking. In *4th International Information Hiding Workshop*, Pittsburgh, PA, April 2001.
26. Geoff Whale. Identification of program similarity in large populations. *Computer Journal*, 33:140–146, 1990.
27. Micheal J. Wise. Detection of similarities in student programs: Yap'ing may be preferable to plague'ing. In *23rd SIGCSE Technical Symposium*, pages 268–271, 1992.