A Low-Cost Attack on Branch-Based Software Watermarking Schemes

Gaurav Gupta and Josef Pieprzyk

Centre for Advanced Computing - Algorithms and Cryptography,
Department of Computing, Division of Information and Communication Sciences,
Macquarie University, Sydney, NSW - 2109,
Australia
{ggupta, josef}@ics.mq.edu.au
http://www.comp.mq.edu.au/∼ggupta

Abstract. In 2005, Ginger Myles and Hongxia Jin proposed a software watermarking scheme based on converting *jump instructions* or *unconditional branch statements* (UBSs) by calls to a *fingerprint branch function* (FBF) that computes the correct target address of the UBS as a function of the generated fingerprint and integrity check. If the program is tampered with, the fingerprint and integrity checks change and the target address will not be computed correctly. In this paper, we present an attack based on tracking stack pointer modifications to break the scheme and provide implementation details. The key element of the attack is to remove the fingerprint and integrity check generating code from the program after disassociating the target address from the fingerprint and integrity value. Using the debugging tools that give vast control to the attacker to track stack pointer operations, we perform both subtractive and watermark replacement attacks. The major steps in the attack are automated resulting in a fast and low-cost attack.

Keywords: software, watermark, unconditional branch, breakpoint.

1 Introduction

In recent years, watermarking and fingerprinting have gathered significant attention due to the growing concerns over digital piracy and forgery of multimedia documents including software. *Fingerprinting* and *Software Authentication* are two major security aspects that have emerged. While the former is related to preventing illegal distribution, the latter tries to ensure that the software has not been tampered with. Numerous models have been proposed with these objectives, embedding watermarks, fingerprints, and integrity checks in the source codes and/or executable codes. Software watermarking schemes can be classified as follows:

– *Graph-based/branch-based software watermarking*: The software is treated as a graph G_s with sequential blocks of code as nodes and transfer instructions such as function calls and branch statements as edges connecting the nodes.

Y.Q. Shi and B. Jeon (Eds.): IWDW 2006, LNCS 4283, pp. 282–293, 2006.

The watermark is a separate code and realized as a graph G_w. The two graphs G_s and G_w are connected by inserting additional edges (implemented as branch statements). The resulting watermarked graph is $G_{s'} = G_s + G_w$ and source code s' is decoded from $G_{s'}$.

Venkatesan et al. [14] proposed the first graph-based software watermarking scheme. The central idea is to convert the software and the watermark code into digraphs and add new edges between the two graphs implemented by adding function calls between the software and watermark code. This scheme lacks error-correcting capabilities and is susceptible to re-ordering of instructions and addition of new function calls. Another problem in the scheme is that the random walk mentioned in the paper (refers to the next node to be added in the watermarked software graph being selected randomly from the software graph and the watermark graph) is not actually *random*. The node visited next is based on the number of remaining nodes belonging to software graph N_s and the number of remaining nodes belonging to watermark graph N_w. The next node is chosen from the watermark nodes with a probability of $\frac{N_w}{N_w+N_s}$ and from the software nodes with a probability of $\frac{N_s}{N_w+N_s}$. In a typical scenario, $N_s \gg N_w$, hence the watermark is skewed towards the tail of the watermarked program. This information is useful for probabilistic attacks. Alternatively, a pseudo-random permutation of the nodes to be visited can be generated. For further literature in graph-based software watermarking, the reader is referred to [1,2,3,4,13]. None of these schemes are completely secure against instruction and block re-ordering attacks.

- *Register-based software watermarking*: Registers used to store variables are changed depending on the watermark bit to be embedded by replacing higher level language code with an inline assembly code. The attacker intends to re-allocate variables in registers if the watermark has to be removed. Though, no such attack has yet been proposed.

  Register-based software watermarking based on the QP algorithm (named after authors Qu and Potkonjak) [10,11] is presented in [7]. It modifies registers used to store variables depending on which variables are required at the same time. The scheme is susceptible to register re-allocation attacks. A secondary watermark destroys the old watermark and inserting bogus methods renders the original watermark useless by changing the interference graph.
- *Thread-based software watermarking*: Nagra et al. [9] propose encoding the watermark in the sequence of the threads that are executed. For example, there are 3 threads; T_1, T_2, T_3, $T_1 \rightarrow T_2 \rightarrow T_3$ encodes watermark $(000)_2$ and $T_1 \rightarrow T_3 \rightarrow T_2$ encodes watermark $(001)_2$ and so on. However, without any additional error-control mechanism, changing threads that execute piece of a code would destroy the watermark. Again, there has been no scheme claiming to break the watermark using suggested approach.
- *Obfuscation-based software watermarking*: This class of watermarking is applicable to object-oriented softwares. Class C with functions $\{f_1, f_2, \ldots, f_n\}$ is partitioned into k subclasses $\{C_1, C_2, \ldots, C_k\}$ and the watermark is

encoded in the allocation of the functionalities Examples of such proposed schemes are [5,6,12].

This paper is organized as follows. Section 2 addresses related work in branch-based software watermarking and Section 3 describes watermarking scheme of Myles and Jin that we propose to attack. This is followed by a description of our attack in Section 4. Section 5 provides implementation details and results. We conclude our paper with a note on future enhancements in Section 6.

2 Related Work

There have been several research projects dealing with *branch-based* software watermarking. These schemes exploit possibilities to modify the program's execution path by altering branch statements. In this section, we discuss two papers closely related to our attack. The first by Collberg et al. [1] that introduces *Branch Functions*. The second paper is by Myles and Jin [8] and describes the watermarking scheme that we attack in this paper.

Collberg et al. introduce the notion of *Branch Function* [1]. *Jump instructions* or *unconditional branch statements* (UBSs) are replaced by calls to the *branch function* (for the sake of consistency, by *branch*, we mean an unconditional branch statement from now on) and modifies its own return address in order to return the control to the target of the branch statement. Figure 1 illustrates this process. If the program contains a jump instruction from l_{begin} to l_{end}, several intermediate *pit stops* are inserted so that the control-flow graph becomes $l_{begin} \rightarrow a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow l_{end}$ such that l_{begin} has a jump instruction to a_1 which has a jump instruction to a_2 and so on. The pit stops are inserted using the rule:

$address(a_i) < address(a_{i+1})$, if watermark bit $w_i = 1$
$address(a_i) > address(a_{i+1})$, if watermark bit $w_i = 0$

Finally all the jump instructions are replaced by call to the branch function that determines the correct target address based on the calling address and returns the control to it.

Obvious attacks on such a scheme are adding an additional pit stop to the chain $l_{begin} \rightarrow a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow l_{end}$ such that it becomes $l_{begin} \rightarrow a_1 \rightarrow a_{extra} \rightarrow a_2 \rightarrow \ldots \rightarrow l_{end}$ or deleting an existing pit stop such that it becomes $l_{begin} \rightarrow a_2 \rightarrow \ldots \rightarrow l_{end}$. The goal is to disturb the chain (thereby modify the watermark) yet keep the origin and target the same (hence keeping the execution path intact). Making similar changes, inserting secondary watermark is a trivial.

Myles and Jin propose an alternative fingerprinting model in [8]. The underlying concept remains the same, that is, a *branch function* transferring control to the target of the UBS, but in this case, the branch function contains the fingerprint-generating code, hence the name *Fingerprint Branch Function* (FBF). FBF also computes an integrity check on the source code to ensure that

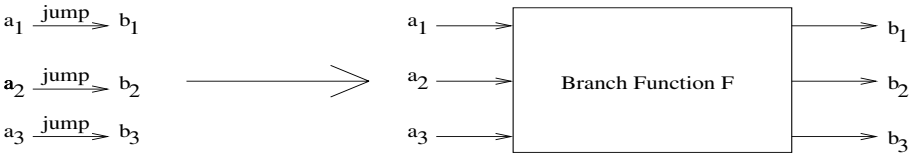

Fig. 1. Insertion of branch function F that changes the return address according to the calling address and transfers control to target of the UBS. The *jump instructions* are now replaced by calls to F.

it is not modified. In the following section, we discuss this scheme in detail and analyze its flaws and weaknesses.

3 Discussion on Watermarking Scheme

The watermarking scheme is applied to software containing *branch statements*. These statements are then replaced by calls to FBF which returns control to the target address. The target address is generated from a recursive process of deriving new keys from old keys and checking the program for integrity. Additionally, an integrity check branch function (ICBF) is inserted in the program that verifies the integrity of FBF. If the user manipulates the program, the keys derived and integrity check value would change and hence the target address will change. The modified target address can be valid (belonging to code section of the program) which will result in incorrect execution of the program. Alternatively the target address can be invalid (lying outside the code section) resulting in runtime error. We will now discuss the two algorithms in the scheme, *"embed"* that inserts the watermark in the software and *"recognize"* that extracts the watermark from the watermarked software. They are defined as:

1. embed$(P, AM, key_{AM}, key_{FM}) \rightarrow P', FM$
2. recognize$(P', key_{AM}, key_{FM}) \rightarrow AM, FM$

where

- P is the original software,
- AM is the authorship mark,
- key_{AM} is the secret input sequence to generate a trace of the program used to embed the watermark - the same for all copies of watermarked software,
- key_{FM} is an initial secret key for deriving further keys - different for each copy of the watermarked program,
- FM is the fingerprint mark
- P' is the watermarked software

3.1 Watermark Embedding

The steps involved in the ***embed*** algorithm are:

1. Let α be the set of all functions in P. Run the program with a secret input sequence key_{AM}.

2. Obtain set F of functions that lie in the execution path when the program is run with input sequence key_{AM}, let $\beta = \alpha - F$.
3. The number of UBSs in functions that belong to F is n and the number of UBSs in functions from β is m.
4. Insert the two integer arrays; T of size n and R of size m in the data section of the program.

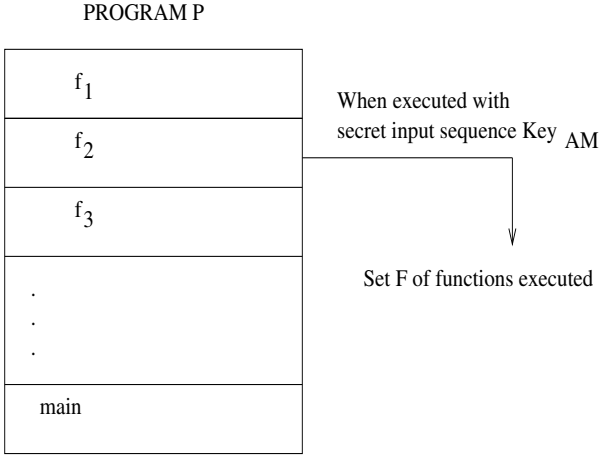

Fig. 2. The set of functions F is executed when the program P is run with the secret input key_{AM}

5. Compute displacement d_i between source s_i and target t_i of UBSs in functions that belong to F, so for instructions of the form s_i : $jmp\ t_i$, the displacement $d_i = t_i - s_i$
6. In the program P, insert FBF ξ that performs the following tasks:
   (a) Initializes $k_0 = key_{FM}$.
   (b) For $1 \leq i \leq n$,
      i. Computes integrity check value v_i.
      ii. Computes key k_i from (k_{i-1}, v_i, AM) by applying a one-way hash function SHA_1.

$$k_i = SHA_1[(k_{i-1} \bigoplus AM) \| v_i] \tag{1}$$

   (c) Stores d_i at $h(k_i)^{th}$ location in array T ($T[h(k_i)] = d_i$) where h is a hash function, $h : \{k_1, k_2, \ldots, k_n\} \rightarrow \{1, 2, \ldots, m\} (n \leq m)$.

7. Compute displacements e_i between source s_i and target t_i of UBSs in functions that belong to β.
8. Insert ICBF ϕ in the program that:
   (a) Computes integrity check value v_i. This value confirms the integrity of code section of the program containing ξ.

(b) Stores displacement e_i in array R at index computed as a one-way hash function of v_i ($R[h(u_i)] = e_i$). The hash function h is the same that was used in Step 6.(c).

9. Replace all UBSs in F by calls to ξ and UBSs in β by calls to ϕ.

The fingerprint is generated as the embedding process executes. The final fingerprint is combination of all derived keys - $FM = k_1 \| k_2 \| \dots k_n$. Users u_i, u_j have distinct initializing keys key_{FM_i}, key_{FM_j}, hence final fingerprints FM_i, FM_j are different.

3.2 Watermark Recognition

The *recognize* algorithm is run with the inputs P', key_{AM}, key_{FM} and outputs the authorship mark AM and fingerprint mark FM. When the program is run with the secret input key_{AM}, the function set F is executed which generates the fingerprint mark $FM = k_1 \| k_2 \| \dots k_n$ by initializing $k_0 = key_{FM}$ and deriving successive keys using Equation (1). The authorship mark AM can be extracted by isolating the one-way hash function $k_i = SHA_1[(k_{i-1} \bigoplus AM) \| v_i]$.

4 Proposed Attack

Objective of the attacker is to convert the fingerprinted program P' to the original program P. Since the displacements in T are permuted, determining the correct target address of UBSs is computationally infeasible. Even if the size of T is small, the program can have error-guards that intentionally corrupt the program after a specific number of run-time errors, making hit-and-trial attack impossible. The function ϕ checks the integrity of ξ, adding to the security of the scheme and thereby making the attack more difficult.

In ξ, the integrity check is done and a key is generated. The key is then mapped to the index in the displacement array where the correct displacement is stored. Security of the scheme depends on the correct execution path being a function of keys and integrity checks. If the key generated or the integrity value is incorrect, the displacement is wrong, and therefore the execution path is wrong. We concentrate our attack on this dependence. As soon as we can *disassociate* the correct execution path from the keys and integrity check, the code generating keys and integrity check can be deleted. The authors of [8] claim that the attacker needs to analyze the data section of the program to notice any changes and read the displacement array. This claim is fallacious as an attacker can track register values, including the stack pointer (SP) at:

1. Entry point of ξ: SP = sp_{i_1}
2. Exit/ Return instruction of ξ: SP = sp_{i_2}

The difference $sp_{i_2} - sp_{i_1}$ gives the displacement value d_i. Identification of the instructions participating in fingerprint generation is also achievable. According to [8], *"In the second phase of the algorithm, the branches in each function f*

that belongs to F are replaced by calls to the FBF". We can create a mapping of functions being called by other functions and thereby create sets of functions which all point to one particular function. ξ can be identified by the stack-pointer modifying statements and the set F can be identified as the set of functions calling ξ. Therefore, key_{AM} is no longer required to identify the set of functions participating in watermarking. Within the set F, each instruction calling ξ and having memory address sp_1 can now be replaced by an unconditional branch to the instruction at sp_2. This can be achieved using inline assembly programming. For example, in C++, a user can make use of *_asm* blocks. As a result, the displacement and hence the correct target address is no longer a function of the key and integrity check.

An example of such a block modifying the stack pointer is given below,

```
_asm {
1:      pop ECX;
2:      add ECX,dis;
3:      push ECX;
}
```

In the above code, statement 1 extracts the current value of Stack Pointer into register ECX. Statement 2 adds the intended displacement *dis* to the popped value and statement 3 pushes back the modified value onto the Stack. The Stack Pointer now contains a modified return address. If *dis* is positive, the new address a_t is greater than the original return address a_r ($a_t > a_r$) and the control is transferred "forward". If it is negative ($a_t < a_r$), control is transferred "backward". Observe that ϕ calls can similarly be replaced by the original UBSs.

After changing calls to ξ and ϕ by UBSs, the two functions (ξ, ϕ) can be deleted. When the *recognize* algorithm is run with input key_{AM}, key_{FM}, the inputs are unused dead variables, the algorithm doesn't output the fingerprint mark FM and the recognition algorithm fails. The resulting software is equivalent to an un-watermarked software.

Summarizing our described process, the steps performed by the attacker are:

1. *Identify* ξ: This task is accomplished by locating stack-pointer modifying statements. For example, in C/C++, searching for *_asm* blocks. If a program contains multiple *_asm* blocks, the ones with modification operation on ESP (Stack Pointer) requires to be targeted.
2. *Identify* F: After identifying ξ, the fact that only the functions that belong to F call ξ can be utilized to identify F.
3. *Displacement computation*: Stack pointer values are recorded at the entry and exit points of ξ (sp_{i_1} and sp_{i_2} respectively) and displacement d_i is equal to $sp_{i_2} - sp_{i_1}$. Target instructions are determined from calling instruction and displacement. In our implementation, we use breakpoints to track the register values.

4. *Replacement of ξ calls to UBSs*: If the purpose of the attack is to remove the watermark, the function calls to ξ are replaced by UBSs to obtain the original watermarked code.
5. *Creating a modified watermarked program*: The attacker can embed his/her own authorship mark AM' after removing the original authorship mark AM. For a successful attack, (AM', FM') should be recognized on running *recognize* algorithm with parameters P', key_{FM}, key_{AM} where $FM' \neq FM$.
   (a) For all f that belong to F, compute the displacement between the calling address and the target address and store in an array along with the calling address.
   (b) Replace the UBSs by call to a new Fingerprint Branch Function, $\widetilde{\xi}$.
   (c) $\widetilde{\xi}$ **need not** compute integrity check but simple calculates a new key based on the old key and attacker's authorship mark AM'.

$$k_i = SHA1[k_{i-1} \bigoplus AM']. \tag{2}$$

   Comparing (1) and (2), $k'_i \neq k_i$, $1 \leq i \leq n$.
   (d) Map the keys to correct displacement using hash,

$$h : \{k'_1, k'_2, \ldots, k'_n\} \rightarrow \{1, 2, \ldots, m\} (n \leq m)$$

$$T[h(k'_i)] = d_i$$

The key sequence FM' generated is different from the original key sequence FM as the individual keys are different. More formally,

$$k'_1 \neq k_1, k'_2 \neq k_2, \ldots, k'_n \neq k_n$$

$$\Rightarrow \{k'_1, k'_2, \ldots, k'_n\} \neq \{k_1, k_2, \ldots, k_n\}$$

$$\Rightarrow \{k'_1, k'_2, \ldots, k'_n\} \neq FM$$

$$\Rightarrow FM' \neq FM$$

The recognition algorithm now outputs FM', AM' when executed with the inputs P', key_{AM}, key_{FM}.

In terms of efficiency, the overall complexity of attack depends on complexities of steps 3 and 4 as others are one-off steps. Steps 3 and 4 have linear complexity and hence the attack has $O(n)$ complexity. Steps 1 and 2 are automated and no human inspection is required to identify ξ and F.

5 Implementation Details and Results

We have implemented the watermarking scheme in Visual C++ and carried out the attack using the same. The features useful in doing so are the debug lookup windows - disassembly and register. The stack pointer value can then

be tracked by using breakpoints under debugging mode and there is minimal manual intervention or inspection required. The following is disassembled code of the watermarked program used to compute displacement values.

Function f_i that belongs to F calling FBF ξ in statement 94:

```
0041198C  rep stos dword ptr es:[edi]
0041198E  mov         eax,dword ptr [a]
00411991  cmp eax,dword ptr [b]
00411994  jle         greater+2Bh (41199Bh)
00411996  call        fingerprint (411271h)
0041199B  push        offset string " is greater \n" (4177A8h)
004119A0  mov         esi,esp
004119A2  mov         eax,dword ptr [b]
004119A5  push        eax
004119A6  mov         ecx,dword ptr [__imp_std::cout (41A350h)]
004119AC  call        dword ptr
          [__imp_std::basic_ostream<char,
          std::char_traits<char>>::operator<< (41A354h)]
004119B2  cmp         esi,esp
004119B4 call @ILT+425(__RTC_CheckEsp) (4111AEh)
004119B9  push eax
004119BA  call std::operator<<<std::char_traits<char> > (411168h)
004119BF  add esp,8
004119C2  jmp         l1+27h (4119EBh)
004119C4  push        offset string " is greater \n" (4177A8h)
004119C9  mov         esi,esp
004119CB  mov         eax,dword ptr [a]
-------------------------------------------------------------------

Fingerprint branch function code modifying return address:

00414AF2  mov         eax,ebp
00414AF4  add         eax,4
00414AF7  mov         ebx,esp
00414AF9  mov         esp,eax
00414AFB pop ecx
00414AFC  sub eax,eax
00414AFE  add         eax,0Ah
00414B01  add ecx,dword ptr [dis (419334h)]
00414B07  push ecx
00414B08  mov esp,ebx
-------------------------------------------------------------------
```

Register values are tracked while the program is executed and the following results are obtained:
Statement 00414AF2: EIP stores calling address, EIP=00411996.

Statement 00414AFB: Return address, stored in the stack pointer, is popped into ECX, ECX = 0041199B.
Statement 00414B01: ECX adds displacement value to calling address, ECX = 004119C4.
Statement 00414B07: ECX value is pushed onto stack pointer. *fingerprint();* returns control to this address.

In a nutshell, instruction 94 calls *fingerprint();* which returns control to instruction 98 (the target of the original UBS) based on the value of *dis* looked up from array T. The attacker can thus compute the difference between ECX value at statement 80 (ECX_{80}) and ECX value at statement 83 (ECX_{83}) to find the value of displacement, then replace *fingerprint();* call at statement 94 by UBS transferring control to $\Psi(\Phi(94) + ECX_{83} - ECX_{80})$ (where $\Phi(x)$ denotes address of instruction x and $\Psi(y)$ represents instruction at address y).

6 Conclusion and Future Work

In this paper, we present a successful low-cost attack on the branch-based watermarking scheme proposed in [8]. The cost of the attack is low in terms of hardware resources required since the only resources required are a functional computer with sufficient memory, storage and speed. The attack is efficient as manual inspection is required only during the step in which displacement values are noted from the disassembly register window. Even this is a debugger-specific constraint and in theory, it can be automated, however, we are unaware of an existing debugger that can perform this task. We provided an implementation of our scheme and some practical examples. The work lays a strong foundation for attacking similar software watermarking models [1,2,3,4,13] that depend on branching and inserting bogus functions in the program in order to embed a watermark. This paper also shows that tracking registers and branches is a trivial task using debugging tools and hence opens up a very interesting question of how can the watermarking schemes survive attacks with such advanced capabilities? Our future work is concerned with the following:

– We have shown that the attack is correct in theory and implemented a semi-automated version of the attack. We will work towards enhancing the implementation such that a fingerprinted program written in any language can be attacked. Practically, this is feasible since the attack operates on the disassembled code which, irrespective of the programming language in which it is written, is similar. However, the challenge would be to make the register tracking process compiler-independent. We also intend to design attacks for other branch-based watermarking schemes. Since the central security guard in such schemes is the dependency of the target address on integrity check and watermark values, they can be attacked in a manner similar to our attack.

– Modifying scheme proposed in [8] so that the attack described in this paper is rendered ineffective by creating more complex dependency of inherent functionality of the program on the keys generated so that the attacker cannot remove fingerprint code without affecting the correct execution of the program. This can be done by introducing parameters other than displacement to bind the program's execution to the keys generated. As a simple example, a program may modify itself choosing from a set of modifications based on the key generated.

Acknowledgements

We would like to extend our appreciation for the contributions made by Saurabh Singh towards this research. The second author is supported by Australian Research Council grants DP0345366 and DF0451484.

References

1. Christian Collberg, Edward Carter, Saumya Debray, Andrew Huntwork, Cullen Linn, and Mike Stepp. Dynamic path-based software watermarking. In *Proceedings of Conference on Programming Language Design and Implementation*, volume 39, pages 107–118, June 2004.
2. Christian Collberg, Andrew Huntwork, Edward Carter, and Gregg Townsend. Graph theoretic software watermarks: Implementation, analysis, and attacks. In *Proceedings of 6th Information Hiding Workshop, LNCS*, volume 3200, pages 192–207, 2004.
3. Christian Collberg, Stephen Kobourov, Edward Carter, and Clark Thomborson. Error-correcting graphs for software watermarking. In *Proceedings of 29th Workshop on Graph Theoretic Concepts in Computer Science*, pages 156–167, 2003.
4. Christian Collberg and Clark Thomborson. Software watermarking: Models and dynamic embeddings. In *Proceedings of Principles of Programming Languages 1999, POPL'99*, pages 311–324, 1999.
5. Christian S. Collberg and Clark Thomborson. Watermarking, tamper-proofing, and obfuscation - tools for software protection. In *IEEE Transactions on Software Engineering*, volume 28, pages 735–746, August 2002.
6. Kazuhide Fukushima and Kouichi Sakurai. A software fingerprinting scheme for java using classfiles obfuscation. In *Proceedings of Information Security Applications, LNCS*, volume 2908, pages 303–316, 2004.
7. Ginger Myles and Christian Collberg. Software watermarking through register allocation: Implementation, analysis, and attacks. In *Proceedings of International Conference on Information Security and Cryptology, LNCS*, volume 2971, pages 274–293, 2003.
8. Ginger Myles and Hongxia Jin. Self-validating branch-based software watermarking. In *Proceedings of 7th Information Hiding Workshop, LNCS*, volume 3727, pages 342–356, 2005.
9. Jasvir Nagra and Clark Thomborson. Threading software watermarks. In *Proceedings of 6th Information Hiding Workshop, LNCS*, volume 3200, pages 208–223, 2004.

10. Gang Qu and Miodrag Potkonjak. Analysis of watermarking techniques for graph coloring problem. In *Proceedings of International Conference on Computer Aided Design*, pages 190–193, 1998.
11. Gang Qu and Miodrag Potkonjak. Hiding signatures in graph coloring solutions. In *Proceedings of 3rd Information Hiding Workshop, LNCS*, volume 1768, pages 348–367, 1999.
12. Mikhail Sosonkin, Gleb Naumovich, and Nasir Memon. Obfuscation of design intent in object-oriented applications. In *Proceedings of 3rd ACM workshop on Digital Rights Management*, pages 142–153, 2003.
13. Clark Thomborson, Jasvir Nagra, Ram Somaraju, and Charles He. Tamper-proofing software watermarks. In *Proceedings of Australasian Information Security Workshop*, volume 32, pages 27–36, 2004.
14. Ramarathnam Venkatesan, Vijay Vazirani, and Saurabh Sinha. A graph theoretic approach to software watermarking. In *Proceedings of 4th Information Hiding Workshop, LNCS*, volume 2137, pages 157–168, 2001.