A Semi-Dynamic Multiple Watermarking Scheme for Java Applications

Changjiang Zhang
School of Software
Tsinghua University
Beijing, China
zhang-cj06@mails.tsinghua.edu.cn

Jianmin Wang
School of Software
Tsinghua University
Beijing, China
jimwang@tsinghua.edu.cn

Clark Thomborson
Department of Computer Science
the University of Auckland
Auckland, New Zealand
cthombor@cs.auckland.ac.nz

Chaokun Wang
School of Software
Tsinghua University
Beijing, China
chaokun@tsinghua.edu.cn

Christian Collberg
Department of Computer Science
University of Arizona
Tucson, AZ 85721, USA
collberg@cs.arizona.edu

ABSTRACT

Software protection and security has been a more and more important issue. In order to prevent software from unauthorized use and modification, a great many techniques have been proposed and developed. In this paper, we address this issue through a prevention technique called *software watermarking*, and we propose a novel software watermarking scheme, which can embed multiple non-interfering watermarks into the same program. Unlike published schemes, this scheme encodes the watermark into mapping functions and then embeds the mapping codes, which are generated from these functions, into the program at the articulation points of its control flow graph. The extraction in this scheme, which is based on dynamically loading a reconstructed program to recover the watermark, is also a novel approach to the software watermarking field. Experimental results indicate that the size and performance overheads caused by this scheme can keep steady.

Categories and Subject Descriptors

K.4.1 [**Computers and Society**]: Public Policy Issues – *Intellectual property rights.* K.4.4 [**Computers and Society**]: Electronic Commerce – *Intellectual property, security.* K.5.1 [**Legal Aspects of Computing**]: Hardware/Software Protection – *Proprietary rights.*

General Terms: Security, Languages, Legal Aspects

Keywords: Software Watermarking, Software Piracy, Software Protection, Software Security, Semi-Dynamic, Multiple Watermarking

1. INTRODUCTION

Software piracy is the unauthorized copying or distribution of copyrighted software. This can be done by copying, downloading, sharing, selling, or installing multiple copies onto personal or work computers. It is estimated that 38% of the world's PC software is pirated and that the losses from this piracy may be as high as $48 billion in 2007 [4]. Moreover, half of the 108 individual countries studied have a piracy rate of 61% or higher. Software piracy negatively affects much more than just the industry. It also harms local resellers and services firms, lowers government tax revenues and increases the risk of cyber crime and security problem. It is therefore extremely important to combat software piracy and protect software intellectual property rights.

Software watermarking is a technique investigated to discourage piracy. In essence, it is a technique of embedding a piece of identifying information into the software, where the embedded identifying information is called a software watermark.

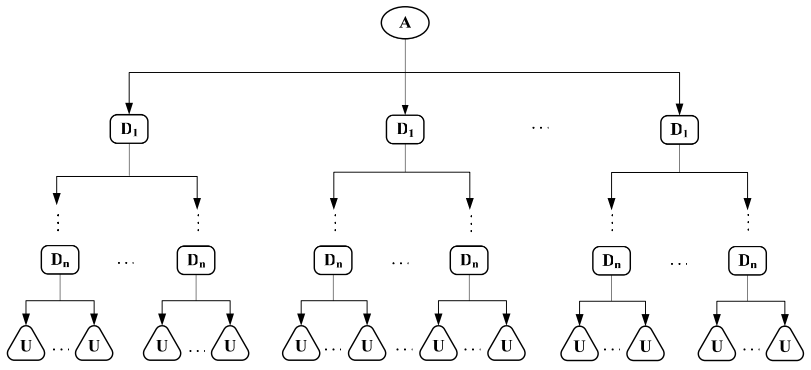


Figure 1: Software distribution model

The watermark can be used to demonstrate ownership of the software, where each legal copy of the software is watermarked with the same identifier, *i.e.* the authorship mark. Also, the watermark can be used to trace the source of illegal distribution, where each legal copy of the software is respectively watermarked with a piece of unique identifier, *i.e.* the fingerprinting mark, which identifies the original purchaser.

In a simplified model of software distribution, there are three primary actors: the author, the distributor and the consumer [17].The author wishes to distribute his software to his consumers via his distributors. A more common scenario allows multilevel distribution, which means that the distributors of each level are able to do a successive delegation to the next level distributors. As shown in Figure 1, an "A" refers to the author of software, a "D_1" refers to a first-level distributor, a "D_n" refers to an n^{th}-level distributor and a "U" refers to an end-user or a consumer. When software is successfully sold to a consumer, a distribution chain is formed, where the author is at the source, the consumer is at the end and the distributors of different level are at the middle positions. At each link of the distribution chain, there would be a different interest to protect. At the author level, he wants to protect his intellectual properties. This can be met by using the authorship mark. In addition, the author and the distributors at each level need to record the legal transactions of the software, where in case of suspected piracy, it should be possible to trace back to the person who originally obtained the software prior to illegal distribution. This can be met by using the fingerprinting marks, which means the author should record his first-level distributors by embedding the fingerprinting marks of them into the software, the first-level distributors should also record their second-level distributors by embedding the fingerprinting marks of them into the software, and the rest may be deduced by analogy. In this model, what we want is to be able to embed a sequence of distinct marks along with the order in which they were embedded, and to extract these marks. Given this capability, as the program passes through a sequence of distributors, each can add their own mark without fearing this will interfere with anyone else's marks.

Nowadays, software watermarking has received more and more interest from the research community and several techniques have been proposed by the community. The main idea of these techniques can be listed as follows. The Davidson-Myhrvold algorithm [13] encodes the watermark by reordering the basic blocks of the program, and the Register Allocation algorithm [20] encodes the watermark by register reassigning. The Collberg-Thomborson algorithm [7] and the Graph-Theoretic algorithm [25] encode the watermark based on graph encoding, while the Spread Spectrum algorithms [12, 22] encode the watermark based on spectrum spreading. The Dummy Method algorithm [15] encodes the watermark by replacing the opcodes or overwriting the numerical operands in a dummy method. The Opaque Predicates algorithm [3] encodes the watermark by opaque predicates. The Abstract Interpretation algorithm [11] encodes the watermark in such a way that it can only be extracted by analyzing the concrete semantics of the program. The Path-Based algorithm [9] encodes the watermark by manipulating the runtime branch structure of the program. The Thread-Based algorithm [18] encodes the watermark by manipulating the threading behavior of the program.

All the previously proposed watermarking techniques can be classified as static and dynamic watermarking according to their extracting techniques [7]. Static watermarking embeds the watermark in the data area or code section, and they can be extracted from the program without any need for execution. Dynamic watermarking embeds the watermark in the program execution state, so the watermark extraction needs the watermarked program to be executed. Our multiple watermarking scheme is fully automatic and different from all the existing static and dynamic watermarking. It could be classified as a static watermarking since the watermark is hidden in the code section of the program and the extraction of the watermark requires no execution of the entire program. However, the watermark cannot be extracted by merely static analysis of the program code. Our scheme could also be classified as a dynamic watermarking since the watermark is encoded in the concrete semantic meaning of some pieces of the program code (which may be a non-standard description). The extraction of the watermark is accomplished without executing the entire program, but the extractor does have to run (to be precise, to dynamically load) a reconstructed program which mainly contains the pieces of the code segments extracted from the program. From this point of view, we will call our watermarking scheme as a *semi-dynamic* watermarking.

As far as we know, little work has been done for multiple watermarking and semi-dynamic extraction techniques in software watermarking field. The primary contribution of this paper is to propose a novel watermarking scheme, which can provide strong support for embedding multiple non-interfering watermarks into the same program. For embedding a watermark, we encode the watermark into mapping codes and then insert these codes into the program at articulation points of its control flow graph. In addition, for extracting the watermark, we dynamically load a reconstructed program to recover the watermark, where the reconstructed program is mainly composed of the mapping codes extracted from the watermarked program. Both the embedding technique and the extraction technique are novel to the software watermarking field.

The rest of the paper is organized as follows. In section 2, we provide formal definitions of multiple watermarking and related concepts. In section 3 and section 4, we give the embedding and the extraction procedures in our scheme. Section 5 discusses some criteria satisfied by our scheme. Section 6 gives our experimental evaluations of the scheme. Section 7 has the summary and the future work.

2. PRINCIPLES

Fundamental terms in the field of software watermarking are formally defined by [26, 27]. However, in order to give a precise expression of our watermarking scheme, we need to redefine some fundamental terms as follows:

Let **P** denote the set of programs that are accepted by our watermarking system, let **W** denote the set of watermarks for this system, and let **K** denote the set of completed keys which include a primary key and three additional parameters n, m and q whose purposes will be defined in Section 3. Henceforth, we use K_c to denote a completed key and we use K to denote a primary key. In addition, we use **Z** to denote the set of integer numbers, and we use ε to denote the empty watermark whose length is 0.

2.1 Embedding

We call a function Em: $\mathbf{P}\times\mathbf{K}\times\mathbf{W}\rightarrow\mathbf{P}$ an embedder, which can be used to insert a watermark into a program. For $P\in\mathbf{P}$, $K_c\in\mathbf{K}$ and $W\in\mathbf{W}$, $P' = Em(P, K_c, W)$ is called a watermarked program and the program P is called the original program corresponding to the watermarked program P'.

A tuple $T=(K_c, W)\in\mathbf{K}\times\mathbf{W}$ is called a watermarkable tuple with respect to a program P and an embedder Em if $P'=Em(P,T)= Em(P,K_c,W)\neq P$. We use $\mathbf{T_B}(P,Em)$ to denote the set of all the watermarkable tuples with respect to P and Em.

2.2 Extraction

In order to effectively prove that the program is protected by watermarking, the watermarking system should provide an extractor to extract a valid watermark from the program.

Let Em be an embedder, then we call the function Ex: $\mathbf{P}\times\mathbf{K}\rightarrow\mathbf{W}$ an extractor corresponding to the embedder Em if Ex has the following property: $\forall P, P'\in\mathbf{P}$,

$Ex(P', K_c)=W$, if $(K_c, W)\in\mathbf{T_B}(P,Em)$ and $P'=Em(P, K_c, W)$.

$Ex(P', K_c)= \varepsilon$, otherwise.

2.3 Detection

In order to support multiple watermarks, our watermarking system should report whether a program has a watermark or some watermarks, and also the embedding order of all the watermarks that have been embedded in the program. To be specific, for embedding a new watermark, our detector returns the maximal value in the order numbers of all the embedded watermarks, so that the new watermark can be given a larger order number than the maximal value.

Let *Em* be an embedder, then we call the function *Dec*: **P**→**Z** a detector corresponding to the embedder *Em*. For $\forall P, P' \in \mathbf{P}$, we will have:

$Dec(P')=Dec(P)+1$, if $T \in \mathbf{T_B}(P,Em)$ and $P'=Em(P,T)$.

2.4 Recognition

For some **P**, **W** and **K**, we may be unable to devise an embedder, extractor and detector for a watermarking system. Besides, the extractor may be unacceptably inefficient [27]. In such cases, it may be preferable to devise a watermarking system with an embedder, recognizer, and detector. A recognizer is similar to an extractor, but it must be provided with a watermark as well as a completed key. As defined formally below, our recognizer returns the order number of the watermark, if it is found.

Let *Em* be an embedder and *Dec* be a detector corresponding to *Em*, then we call the function *Reg*: **P**×**K**×**W**→**Z** a recognizer corresponding to *Em* if *Reg* has the following property: $\forall P, P' \in \mathbf{P}$,

$Reg(P', T)=Dec(P)+1$, if $T \in \mathbf{T_B}(P, Em)$ and $P'=Em(P, T)$;

$Reg(P', T)=0$, otherwise.

A tuple $T=(K_c, W) \in \mathbf{K}\times\mathbf{W}$ is called a watermarked tuple with respect to a program *P* and an recognizer *Reg*, if $Reg(P,T)>0$. We use $\mathbf{T_D}(P,Reg)$ to denote the set of all the watermarked tuples with respect to *P* and *Reg*. A watermarked tuple is a tuple that exists in the program.

For $\forall P_1, P_2 \in \mathbf{P}$, if $T_1 \in \mathbf{T_D}(P_1,Reg)$, $T_2 \in \mathbf{T_D}(P_2,Reg)$, and $P_2=Em(P_1,T_2)$, then we will have:

$Reg(P_2,T_2) > Reg(P_2,T_1)$ and $Reg(P_2,T_1) = Reg(P_1,T_1)$.

3. EMBEDDING THE WATERMARK

In this scheme, both the watermark *W* and the primary key *K* are treated as an integer. The completed key K_c contains three additional integers, *n, m* and *q*, where *n* is the length of integer sequences that *W* and *K* are split into, *m* is the number of mapping functions constructed for each pair of corresponding elements in the two sequences, and *q* is a parameter of the splitting method which will be discussed below.

3.1 Splitting the Watermark and the Key

To make the watermark be encoded in a secret way, we split the watermark into a number of small integers, and then embed them into the program separately. In addition, we need to ensure that the watermark *W* and the key *K* are separately split into two integer sequences with the same length, and then we construct some mapping functions between each pair of elements in the two sequences. Collberg [10] proposed such a method that can split an integer *N* into a sequence of integers, where the length of the sequence can be appointed, so it is very suitable for our splitting the watermark and the key.

The sequence gained from Collberg's method is a monotonically increasing sequence. The correctness of our watermarking scheme requires that the integers in the sequence are different from each other. To achieve this goal, we therefore improve Collberg's splitting method as follows:

1. Compute the minimum exponent *l* such that *N* can be represented using *n*-1 digits of base q^l;
2. Split *N* into digits $v_0, v_1, .., v_{n-2}$ such that $0 \le v_j < q^l$ and $N=\sum_{j=0}^{n-2} v_j q^{jl}$;
3. Encode the digits in the sequence $S=\{s_0, s_1, ..., s_{n-1}\}$,where $s_0=l$ and $s_i= s_{i-1}+ v_{i-1}+1$.

In the improved splitting method above, the values of *n* and *q* are given by the copyright owner or the user of our scheme. Without knowing the exact values of *n* and *q*, the attacker cannot get the original watermark. So, keeping the two values secret can make the embedded watermark more secret.

For a concrete example, we let the watermark W=31415916, the length of the sequence is appointed as n=10 and the splitting parameter q=2. Then the watermark *W* will be split into a sequence with 10 integers: $\boldsymbol{w}=\{w_1, w_2, w_3, w_4, w_5, w_6, w_7, w_8, w_9, w_{10}\}=\{3, 10, 17, 19, 27, 33, 40, 48, 55, 57\}$. Using the same method, if we let K=12345, the sequence gained from splitting *K* will be: $\boldsymbol{k}=\{k_1, k_2, k_3, k_4, k_5, k_6, k_7, k_8, k_9, k_{10}\}=\{2, 4, 7, 11, 12, 13, 14, 18, 19, 20\}$.

3.2 Detecting the Embedding Order Number

For embedding a new watermark, we should ensure that the new watermark and the previous ones have no impact on each other. Besides, we should rule out the accidental impacts caused by the subject program (e.g. some programs may contain code that can lead to a failure in extraction). In our scheme, we associate an order number of embedding with every tuple to be embedded, which can greatly decrease the probability of watermarking error.

To determine the order number of current embedding, we need to determine the maximal value in the embedding order of all the watermarked tuples. We firstly extract the mapping functions from the program, which will be discussed in Section 4. Then we use a fixed value q_0, which is intrinsic to the watermarking system, to check the order number of each extracted mapping function f. For each function f, we will get a return value and the functions that return the same value will be drawn into the same group. In other words, the functions in the group have an intersection point at q_0. The same return value will be regarded as a valid order number, if the group has no less than 3 functions. We set 3 as the lower bound of judging, because it is the minimum value that can provide a successful detection according to our experiment results. After all the order numbers of previous embedding are determined, we return the maximal value, that is $Dec(P)$, to the embedder and the order number of current embedding will assigned as $Dec(P)+1$.

3.3 Constructing Mapping Functions

In our scheme, we will construct two kinds of mapping which are contained in only one function. Firstly, we need to construct a sequence $\boldsymbol{e}=\{e_1, e_2,\ldots, e_n\}$ from n and the fixed value q_0 using the following method: $e_i=q_0+i-1$, for $1\leq i\leq n$. In this way, no matter what value of n is given while embedding different watermarks, the first integer in the sequence is always the fixed value q_0, which is very useful to the detector. Secondly, we introduce a variable r, which is an offset, to avoid the collision caused by the sequence $\boldsymbol{e}$ and the sequence $\boldsymbol{k}$. The value of r is determined by $r=q_0+n$, where it satisfies that $r>e_n= q_0+n-1$. Next, we detect the program to get the value of $Dec(P)$, which has been discussed in Section 3.2. After the two steps, we construct the functions by the following method: for $1\leq i\leq n$, $1\leq j\leq m$, □P□$\mathbf{P}$, each function f_{ij} is subject to the following two constraints:

(1) $f_{ij}(e_i) = Dec(P)+1$;

(2) $f_{ij}(k_i+r) = w_i$;

The first constraint indicates a mapping that is constructed from an element in the sequence $\boldsymbol{e}$ to the order number of current embedding. The second constraint indicates another mapping that is constructed from an element in the sequence $\boldsymbol{k}$ with an offset r to the corresponding element (with the same index) in the sequence $\boldsymbol{w}$.

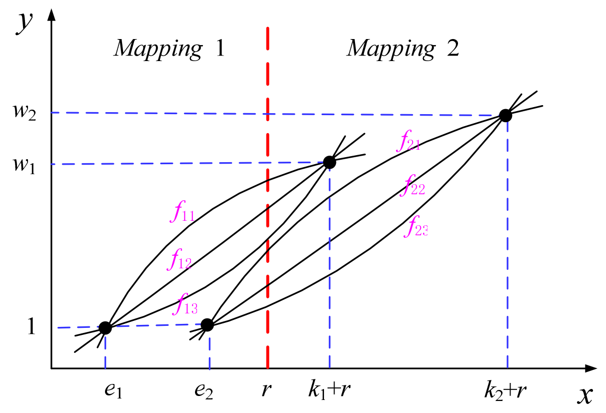


Figure 2: Example curves of constructed functions

The mathematical expression of function f_{ij} is determined by the equation $f_{ij}(x)=a_tx^t+a_{t-1}x^{t-1}+\ldots+a_1x+a_0$, where t satisfies that $2^t\geq m$. We substitute $(e_i, Dec(P)+1)$ and (k_i+r, w_i) into the equation and then calculate the value of each coefficient a_k $(1\leq k\leq t)$, which is a fraction number. From the graphical point of view, for each i in $[1, n]$, the curves of m functions $f_{i1}, f_{i2},\ldots, f_{im}$ will intersect at e_i and k_i+r. For example, we let $n=2$, $m=3$, $Dec(P)=0$, $t=2$, and we let the key and the watermark be separately split into: $\boldsymbol{k}=\{k_1, k_2\}$, $\boldsymbol{w}=\{w_1, w_2\}$, six (calculated from n multiplying m) curves will be constructed, as shown in Figure 2.

3.4 Finding Articulation Points

A control flow graph (CFG) is a graphical representation of the possible execution flow of a program[2]. The graph is composed of basic blocks and directed edges. Each basic block is a sequence of statements in which the flow of control starts at the beginning of the block and does not leave until the end of the block. Within a basic block a control transfer statement can only occur as the last statement in the block. Thus, all branch instructions as well as those instructions that either explicitly or implicitly throw an exception can occur as the last instructions of a basic block. Formally, we can use $G=(V,E)$ to represent a CFG, where V represents the set of basic block and E□$V\times V$ represents the set of directed edges.

Articulation point is also called separation point. For a connected, undirected graph $G= (V, E)$, if there is a set of different vertices v, a, w, and all the paths from v to w go through a, then the vertex a is called an articulation point of graph G [14]. In other words, if vertex a is an articulation point, then the removal of *a* and edges incident upon *a* splits the graph into two or more parts.

***Lemma* 1** [1]: Let $G = (V,E)$ be connected, undirected graph, and let $T_{DFS} = (V,T)$ be a depth-first spanning tree for *G*. Vertex *a* is an articulation point of *G* if and only if either:

1. *a* is the root and *a* has more than one son, or

2. *a* is not the root, and for some son *s* of *a* there is no back edge between any descendant of *s* (including *s* itself) and a proper ancestor of *a*.

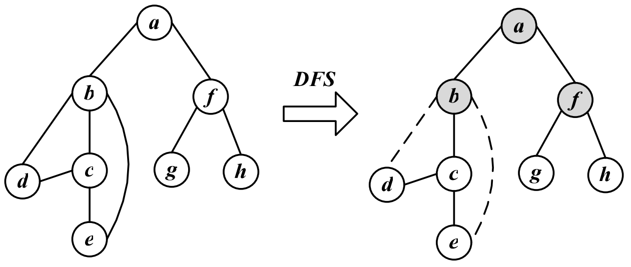


Figure 3: Searching the articulation points

By this lemma, we can determine whether vertex *a* is an articulation point or not in O(|*V*|+|*E*|) time. To cite a case, as shown in Figure 3, on the left is a connected undirected graph and on the right is a depth-first spanning tree of it, where the solid edges are tree edges and the dash edges are back edges. By *Lemma* 1, we can infer that gray vertices *a*, *b* and *f* are articulation points of the graph, and vertex *c* is not an articulation point because both of its two sons has a back edge connected to its ancestor.

In this paper, articulation points of a *CFG* refer to articulation points of its basic graph, which neglects the direction of the directed edges in the graph.

3.5 Generating the Mapping Code

To generate the code for each mapping function f_{ij}, we firstly select a Java method that has at least one articulation point in its *CFG*. Here, we just call the code generated from the mapping function as mapping code. At the worst, there is no such a method in the whole program. In this case, we need to produce an articulation point by introducing some new branches. Secondly, we assign a local variable for each coefficient a_k, dependent variable $f_{ij}(x)$ and independent variable *x*. The initial value of the local variables referring to each coefficient a_k can be assigned as the value obtained in Section 3.3. The initial value of the local variables referring to the independent variable *x* and the dependent variable $f_{ij}(x)$, which are also fraction numbers, should be assigned as a marked value separately. For example, we can assign the denominator or the numerator of the fraction number as special negative numbers. Next, we create a sequence of instructions that carries all the operations (such as addition, multiplication and assignment) in the expression of f_{ij}.

Two things should be considered before generating the mapping code. First, the generated mapping code should be "secure" for embedding, where "secure" means that embedding the mapping code can preserve the functionality of the selected method. To be specific, it should maintain the method's behavioral semantics in addition to its syntactic correctness. Second, the generated mapping code should be as stealthy as possible.

Before presenting how to make the generated mapping code be "secure", it is necessary to present the Java Virtual Machine execution model. As we know, Java code is executed inside threads. Each thread has its own execution stack, which is made of frames. Each frame contains two parts: a local variable part and an operand stack part. In our scheme, we generate and embed such mapping code that can keep the state of the execution frame of the watermarked method at the end of mapping code is the same as the state of the execution frame of the original method at the embedding point.

To increase the stealthiness of the mapping code, a good measure is to disguise it as the host code or as the general application code [10]. In our scheme, we take the following measures to increase the stealthiness of the mapping code. Firstly, while selecting a Java method, we give preference to those methods that contain more arithmetic and branch instructions. For one thing, embedding the mapping code into those methods containing more arithmetic instructions can improve the similarity of the mapping code and the host code. For another, the *CFG* of the method that contains more branch instructions is relatively complicated in its structure, so inserting a new branch instruction in the method will be less perceptible.

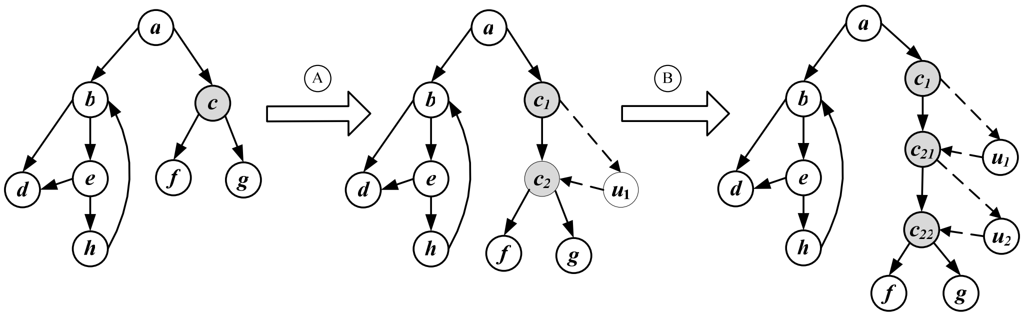


Figure 4: Watermarking and rewatermarking

Secondly, while generating the mapping code, we try to reuse some original local variables in the selected Java method if only the reusing can maintain the method's behavioral semantics and its syntactic correctness. Some optimizers, such as ProGuard [19], can also help to achieve it after the embedding. In other words, we can use these optimizers to increase the stealthiness of the mapping code. In addition, if there are few methods containing arithmetic instructions, we can replace the arithmetic instructions in the mapping code into a method call instructions (*i.e.* to replace the operation, such as addition and multiplication, into a Java method). By this way, the mapping code would be more similar to the general application code, so the stealthiness of the mapping code would be better locally.

3.6 Embedding the Mapping Code

In our scheme, we choose an articulation point of a *CFG* to insert the mapping code and insert some branch instruction before the mapping code. By contrast the original *CFG* and the watermarked *CFG*, the selected articulation point is split into two points and a new basic block carrying the mapping code is introduced between the two points, where one of them is a jumping target of the other. In this way, the basic block carrying the mapping code won't be an articulation point in the watermarked *CFG*, which ensures that it will never be selected as a position for embedding a new watermark. Also, it means that both the new watermark and the previous ones are extractable because they have no collision on their positions.

We give an example of watermarking and rewatermarking, as shown in Figure 4. In Ⓐ we split the articulation point c into c_1 and c_2, and then the new point u_1, which carries the first watermark, is introduced between c_1 and c_2. In Ⓑ we introduce another new point u_2, which carries the second watermark, into the firstly watermarked *CFG*. According to our embedding rules, the point u_1, which is not an articulation point in the watermarked *CFG*, won't be selected and split. So, introducing u_2 has no effect on u_1. Also, the existence of u_1 has no effect on u_2. This means that newly embedded watermarks and the previously embedded watermarks have no effect on each other.

4. EXTRACTING THE WATERMARK

The extraction of watermark contains two steps: extracting the mapping code from the watermarked program and recovering the watermark from the mapping code.

4.1 Extracting the Mapping Code

Before extracting the mapping code, the basic blocks carrying the mapping code should be determined. In other words, we should decide whether a basic block contains a piece of mapping code or not while searching in the *CFG* of each Java method. The following rules can help to make a decision:

(1) It should be incident upon at least one articulation point;

(2) The code carried in the basic block should contain some specified instructions, such as the instructions loading a local variable onto the stack and storing a value from the stack into a local variable;

(3) The code carried in the basic block should load some local variables and the number of local variables loaded should be no less than an appointed value. For example, the function $f(x)=a_2x^2+a_1x+a_0$ refers to at least five local variables, which refer to x, $f(x)$, a_2, a_1 and a_0;

(4) The code carried in the basic block should at least load one local variable more than an appointed times. For function $f(x)=a_2x^2+a_1x+a_0$, the local variable that refers to x should be loaded at least two times, where x^2 can be viewed as loading x only one time by calling a method that returns the result of x multiply itself;

The rules above can effectively help us to choose the basic blocks, which may carry the mapping code, from the exterior forms of the instructions contained. Then we extract the code contained in these chosen basic blocks and the code that related to the local variables such as the initialization instructions. Next, we construct a Java method for each piece of the code extracted. The input of this method is the local variable that is specified with a marked initial value and loaded at least one time. The output of this method is the local variable that is specified with another marked initial value and stored at least one time. When a Java method is constructed from the extracted code, we treat it as a mapping function is successfully extracted.

4.2 Recovering the Watermark

In order to recover the watermark, we need to reconstruct a new program (such as a Java class file) and then dynamically load it to get the running result. The reconstructed program contains all the methods extracted from the watermarked program. In addition, a "main" method is created to call all the extracted methods. While doing extraction, the extractor tries to run the reconstructed program by dynamical call to recover and return the watermark. This is a dynamic behavior of our extraction, but it needs no execution of the entire watermarked program, so we call our watermarking a semi-dynamic watermarking.

The mapping functions extracted may include some interferential functions. To rule out these interferential functions and recover the original watermark, we use the input key to check whether each function f can satisfy the following two constraints:

(1) $f(e_i) = t$, where $t > 0$;

(2) $f(k_i+r) = w_i$;

According to the first constraint, for each e_i in the sequence $\boldsymbol{e}$, the functions that are embedded at the same time should return the same value. According to the second constraint, for each element k_i in the sequence $\boldsymbol{k}$, the functions that are related to k_i should return the same value. The mapping functions that satisfy the two constraints will be drawn into a group. If the sequence $\boldsymbol{w}$, which is constructed from w_i, is strictly monotonic increasing, we consider that the watermark extracting is successful, and then we recover the watermark from the sequence $\boldsymbol{w}$.

5. SATISFIED CRITERIA

Our semi-dynamic multiple watermarking scheme satisfies a great deal of criteria discussed below which can be used to evaluate the quality of a software watermarking scheme.

Functionality Preservation: The watermarking should not only maintain a program's syntactic correctness but also preserve its behavioral semantics. In other words, the watermarked program should have the same output as its corresponding original program for the same input in addition to its correct running. For our scheme, we generate and embed such mapping code that can keep the state of the execution frame of the watermarked method at the end of mapping code is the same as the state of the execution frame of the original method at the embedding point, which leads to the functionality of the original program are exactly persevered.

Unbounded Watermark Size: The watermarking should put no bound to the size of the watermark (or it should be large enough in practice) thus it can embed arbitrarily unique watermarks (or provide a large enough space for selecting a watermark). There is no factor (as far as we know) that sets a limit on the size of the watermark and the key in our scheme. For this reason, our watermarking scheme allows fingerprinting to uniquely mark each program for every purchaser by a unique watermark (such as a serial number). For illegal redistribution by a purchaser, fingerprinting can be used as a valid proof.

Secret Embedding: In our scheme, the watermark and the key are split into small integers and encoded into some pieces of mapping code. The mapping code won't expose the watermark even if it arises some suspicion from the attacker, because the attacker will get little valuable information merely from the mapping code. If the copyright owner keeps the primary key and those three parameters n, m and q secret, the attackers cannot find any way to recover the watermark.

Redundancy Protection: To protect the watermark itself, the watermarking should spread it throughout the program. Our watermarking scheme splits the key and the watermark into pieces and distributes them all over the program. Besides, our scheme embeds redundant mapping code into the program, which can be accomplished by setting the supplementary parameter m>1. The larger m is, the better chance of survival the watermark will have. Even if some piece of mapping code is attacked to be inexactable, it is insufficient to lead to failure in our extraction. In other words, finding a subset of the mapping code will be enough to recover the watermark.

Multiple Watermarks: Our scheme can support several watermarks to be embedded in the same program. Since the watermarks embedded by our watermarking scheme have no collision on their positions. In addition, the watermarks are embedded with different order numbers carried in the mapping functions inherently, which can inherently help the extractor to exclude the functions that are not related to the input key during the extraction process. Formally, we let P' is the watermarked program and Reg is the recognizer, if (K_{c1},W_1), (K_{c2},W_2) □$\mathbf{T_D}(P', Reg)$, then we will have: $Reg(P', K_{c1},W_1) \neq Reg(P', K_{c2}, W_2)$, and $Ex(P', K_{c1}) = W_1$, $Ex(P', K_{c2}) = W_2$.

6. EXPERIMENTS

In this section we present the evaluations of our semi-dynamic multiple watermarking (SMW) algorithm which has been implemented in TRUP [23]. We have measured the code size and performance overheads incurred by the inserted mapping code and the resilience of the watermark to those known attacks.

We used the CaffeineMark benchmark suite [5] as our test benchmark. The CaffeineMark contains several microbenchmarks that can measure the performance of integer operations, loops, logical operations, method calls and floating point operations. In addition, the CaffeineMark scores roughly correlate with the number of Java instructions executed per second. So, if a high percentage of the instructions in CaffeineMark are executed frequently, it will lead to a low test score.

We empirically set the additional parameters as n=10 and m=3. We set the value of m as 3 because it is the minimum value that can provide a creditable extraction of the watermark. We set the value of n as 10 because it can keep most of the literal constants inserted by our embedder are in 4 digits in which case they can be considered to be not rare [11]. In addition, we set the primary key and the watermark with the same size.

All the tests were running on Sun's JVM (Version 1.6.0) and Windows XP Professional. Our hardware was a 3.06GHz Pentium 4 system with 1536MB RAM.

6.1 Overhead

We evaluated the code size and performance overheads of our SMW algorithm using 32, 64, 128, 256 and 512 bit watermarks separately. In addition, we compared the results with overheads caused by the Graph-Theoretic watermarking (GTW) algorithm, which may be the strongest static software watermarking algorithm [18].

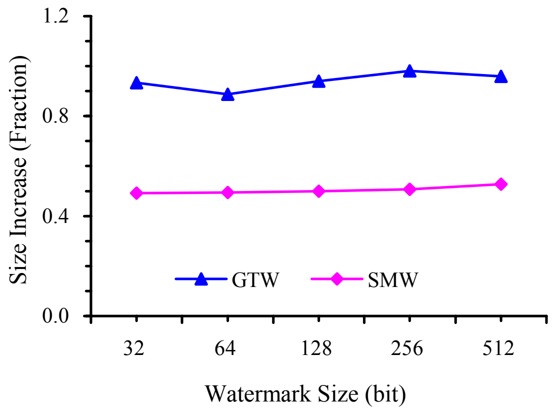


Figure 5: Evaluation result of size increases

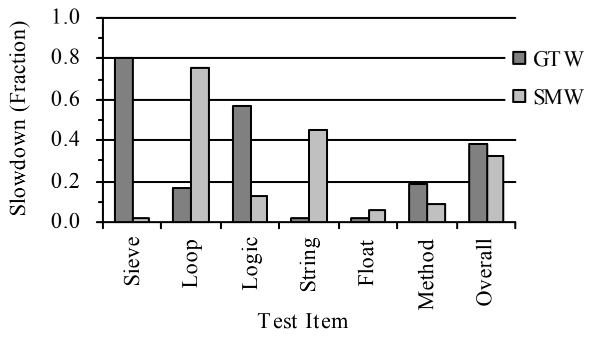
6.1.1 Size Overhead

As shown in Figure 5, the experimental results indicate that the size overhead of our SMW algorithm is obviously lower than the GTW algorithm. This is caused by different embedding techniques. The GTW algorithm encodes the watermark into a flow-graph and then adds it to the control flow graph of the original program, in which case the flow-graph may need lots of instructions to encode it. In contrast, our SMW algorithm encodes the watermark into mapping code which can reduce the size overhead sharply.

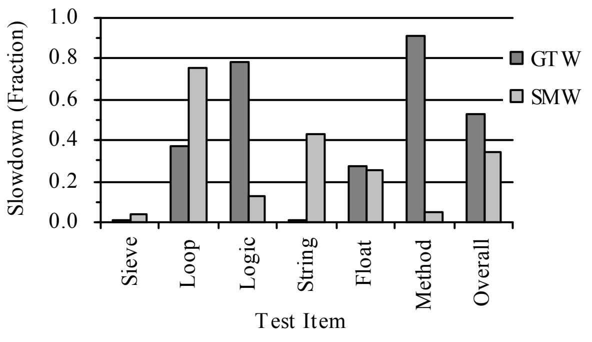
6.1.2 Performance Overhead

The CaffeineMark, which can perform no IO operations after it was started, provides 6 test items to measure the effect embedding a watermark has on the performance. In addition, an overall score is provided, which is the geometric mean of the 6 individual test scores.

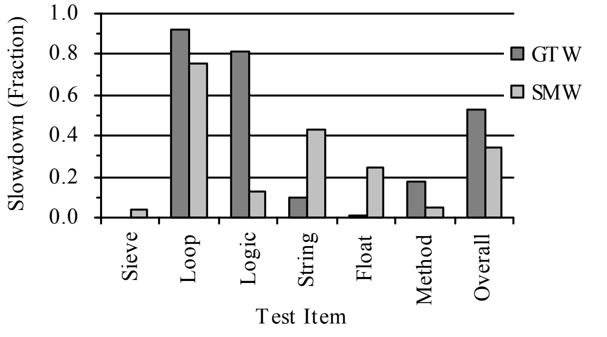
Experimental results are shown in Figure 6. The first five sub graphs show the performance slowdown incurred by the GTW and our SMW algorithm using 32, 64, 128, 256 and 512 bit watermarks separately. It is likely that the GTW algorithm brings more integer arithmetic operations, logical operations and method calls into the CaffeineMark benchmark application and sometimes the negative impacts are very serious, while our SMW algorithm brings more slowdown to the string and loop scores.



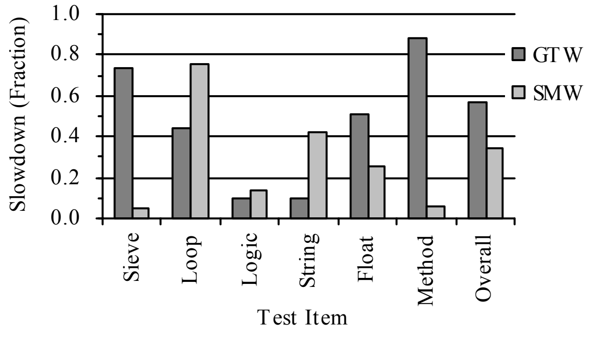
(a) CaffeineMark scores with a 32 bit watermark



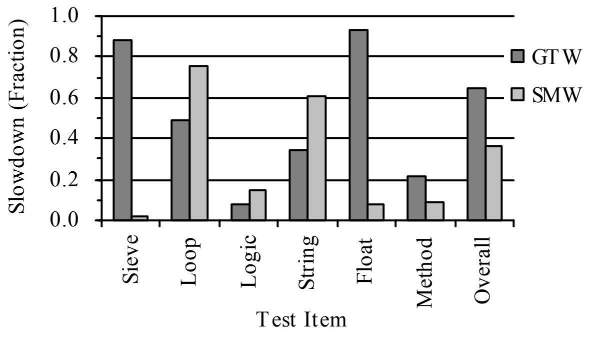
(b) CaffeineMark scores with a 64 bit watermark



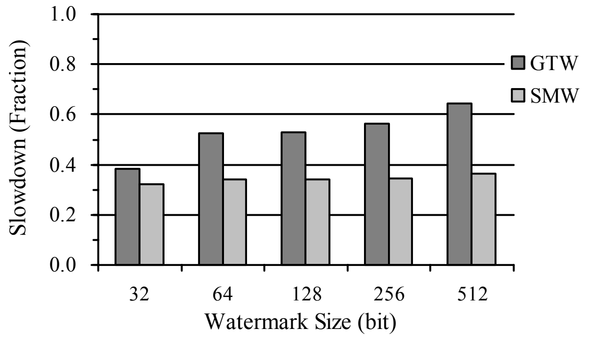
(c) CaffeineMark scores with a 128 bit watermark



(d) CaffeineMark scores with a 256 bit watermark



(e) CaffeineMark scores with a 512 bit watermark



(f) Overall scores of CaffeineMark

Figure 6: Evaluation results of performance slowdown on CaffeineMark

The overall scores reveal that, as shown in Figure 6(f), when more bits of watermark are inserted, the performance slowdown caused by our SMW algorithm can keep roughly steady while the slowdown caused by the GTW algorithm increases gradually. Furthermore, the minimal performance slowdown caused by the GTW algorithm is greater than the maximal slowdown caused by our SMW algorithm.

6.2 Resilience

It is generally agreed that it is impossible to devise a watermark that some sufficiently determined attacker would not be able to defeat. Therefore, the goal of the watermarking community is to design techniques that are "expensive enough" to break (in time, effort, or resources) for most attackers [9, 18].

6.2.1 Subtractive Attack

The first thing that an attacker may try to do is to detect the presence and approximate location of the watermark and then eliminate it from the watermarked program. So, the watermark with low stealthiness is easily to be detected and removed.

Manual subtractive attacks can hardly be avoided if enough time and manpower are available, so we concentrate on automatic subtractive attacks. Typical examples of automatic subtractive attacks include code shrinking and optimization, such as eliminating dead code, removing unused code (unused classes, fields and methods), removing debugging information, removing unnecessary comparisons and branches and so on. In our experiment, we use ProGuard as the subtractive attacker, which can remove unused code and can also perform optimizations inside and across methods at the bytecode level. The results reveal that both the GTW algorithm and the SMW algorithm can resist these automatic attacks.

To resist manual subtractive attacks, a good strategy is to improve the stealthiness of the watermark. Another good strategy is to tamper-proof (e.g. by installing some code-checks) the watermark or the whole watermarked program [8, 24], so that any small change (e.g. a single instruction) would be likely to result in a changed behavior but would be very unlikely to damage the watermark.

6.2.2 Additive Attack

Additive attacks are usually completed by watermarking the program with new watermarks. An effective additive attack is one in which the pirate watermark completely overrides the original watermark so that it can no longer be extracted, or where it is impossible to detect the original watermark temporally precedes the pirate one [6].

Table 1: Results of additive attacks

Watermarker	Resistance
Davidson-Myhrvold	**P**
Register Allocation	**P**
Dummy Method	**P**
Abstract Interpretation	**P**
Spread Spectrum	**N**
Graph-Theoretic	**N**
Opaque Predicates	**N**
Path-Based	**N**
Thread-Based	**N**
Semi-Dynamic Multiple	**Y**

Additive attacks include embedding a new watermark using the same algorithm and using a different algorithm. Additive attacks using a different watermarking algorithm are usually ineffective, so we concentrate on additive attacks from the same algorithm. Additive attacks from the same algorithm are very effective and fatal for some watermarking, such as the Opaque Predicates algorithm discussed in [16], however, our semi-dynamic SMW algorithm is quite resistant to additive attacks from itself since embedding a new watermark doesn't override the previous one and the new watermark will be recognized with a larger order number of embedding, which means that the new watermark is temporally posterior to the previous one. As shown in Table 1, a watermarker refers to a watermarking algorithm, an 'N' indicates that the watermarker cannot resist to additive attacks from itself, a 'Y' indicates that the watermarker can certainly resist to additive attacks from itself, and a 'P' indicates that the watermarker can partly resist to additive attacks from itself (to be precise), which means that the previous watermark may be found or it may be destroyed after embedding a new watermark.

6.2.3 Distortive Attack

Distortive attacks refer to semantics preserving transformations that can make the watermark inexactable or unrecoverable. Code obfuscation and some optimizations are such distortive attacks.

In the experiment of evaluating the resilience to the existing and known distortive attacks, we use ProGuard and SandMark as our distortive attackers. We select ProGuard as our distortive attacker, because it can not only perform lexical obfuscations but also can perform some optimizations inside and across methods at the bytecode level. We select SandMark as our another distortive attacker, because it provides 40 distortive attacks including Buggy Code, Class Splitter, Class Encrypt, irreducibility and so on.

For experiment using ProGuard, the experimental results indicate that both the obfuscations and optimizations performed by ProGuard were unable to destroy the watermark embedded by our watermarking scheme. However, the GTW algorithm was not able to resist the optimization attacks from ProGuard. For experiment using SandMark, the experimental results indicate that our SMW algorithm cannot resist 4 distortive attacks, including Class Encrypt, Opaque Branch Insertion, Interleave Methods and Dynamic Inline. The GTW algorithm cannot resist to 7 distortive attacks. Besides the 4 attacks that SMW algorithm cannot resist to, it also cannot resist to attacks from Buggy Code, Irreducibility and Random Dead Code. These distortive attack experiments indicate that our SMW algorithm is more robust than the GTW algorithm while countering against the existing and known distortive attacks.

6.2.4 Collusive Attack

Collusive attacks can remove the fingerprints by comparison of differently fingerprinted programs, because the programs only differ in their fingerprints. The attacker can perform both the manual attacks and the automatic attacks on the fingerprinted programs, or he can combine the two attacks on the programs. So, it is the case that if enough time and manpower are available for the attacker, there may be no effective resistance to collusive attacks.

However, we can make it "expensive enough" for the attackers to perform collusive attacks. To achieve this goal, a good way is to combine some transformations on the differently fingerprinted programs using different obfuscation and optimization methods. Another effective way is to tamperproof (e.g. by installing some code-checks) the fingerprinted programs, so that any small change (e.g. a single instruction) would be likely to result in a changed behavior but would be very unlikely to damage the fingerprints.

7. CONCLUSIONS

This paper presents a novel software watermarking scheme including a novel embedding technique and a novel extraction technique. While embedding a watermark, it encodes the watermark into mapping codes and then embeds these codes into the program at the articulation points of its control flow graph. The extraction technique is novel, because it is semi-dynamic, which means that the extraction doesn't need to execute the whole program but it needs to dynamically load a reconstructed program (extracted from the watermarked program) to recover the original watermark. With the help of articulation points of the control flow graph, our watermarking can provide strong support for multiple watermarks. In addition, experimental results reveal that the size and performance overheads caused by this scheme can keep steady.

In the future work, we will introduce tamper-proofing technique into our scheme to improve the robustness of our watermarking and the attack cost of the attackers.

8. ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China under Grant No. 90718010. We also acknowledge China Telecom Corporation Limited Shanghai Branch for their strong support on our work. Lastly, we should thank Zhiwei Yu and Junning Fu for their insightful comments on early draft of this paper.

9. REFERENCES

[1] A.Aho and J.Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1974, pages 182-187.

[2] A.Aho, R.Sethi and J.Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley Publishing Company, 2002, pages 532-533.

[3] G.Arboit. A method for watermarking java programs via opaque predicates. In *The 5th International Conference on Electronic Commerce Research* (*ICECR-5*), 2002.

[4] Business Software Alliance. Fifth Annual BSA and IDC Global Software Piracy Study: 2007 Global Software Piracy Study.http://global.bsa.org/idcglobalstudy2007/studies/2007_global_piracy_study.pdf.

[5] CaffeineMark.http://www.benchmarkhq.ru/cm30/.

[6] C. Collberg, C.Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Principles of Programming Languages 1998 (POPL'98)*, San Diego, CA, January, 1998.

[7] C.Collberg and C.Thomborson. Software watermarking: Models and Dynamic Embeddings. *In 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99)*, 1999.

[8] C.Collberg and C.Thomborson. Watermarking, tamper-proofing, and obfuscation-tools for software protection, *IEEE Transactions on Software Engineering*, vol.28, 735-746, 2002.

[9] C.Collberg, E.Carter, S.Debray, A.Huntwork, J.Kececioglu, C.Linn, M.Stepp. Dynamic path-based software watermarking. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'04)*, June, 2004.

[10] C.Collberg, A.Huntwork, E.Carter, G.Townsend. Graph theoretic software watermarks: Implementation, analysis and attacks. In *6th International Information Hiding Workshop*, 2004.

[11] P.Cousot and R.Cousot. An abstract interpretation-based framework for software watermarking. In *POPL'04*, Venice, Italy, 2004.

[12] D.Curran, N.Hurley and M.Cinneide. Securing Java through software watermarking. In *2nd International Conference on the Principles and Practice of Programming in Java (PPPJ 03)*, Kilkenny City, Ireland, 2003.

[13] R.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.

[14] J.Hopcroft and R.Tarjan. Dividing a graph into triconnected components. *SIAM J. Comput.* Volume 2, Issue 3, pages 135-158 (1973).

[15] A.Monden, H.Iida, K.Matsumoto, K.Inoue, and K.Torii. A practical method for watermarking Java programs. In *24th IEEE Computer Software and Application Conference*, *Compsac2000*, Taipei, Taiwan, Oct. 2000.

[16] G.Myles and C.Collberg. Software watermarking via opaque predicates: Implementation, analysis, and attacks. In *7th International Conference on Electronic Commerce Research*, 2004.

[17] J.Nagra, C.Thomborson and C.Collberg. Software watermarking: Protective terminology. In *Australian Computer Science Conference (ACSC2002)*. August, 2002.

[18] J.Nagra and C.Thomborson. Threading software watermarks. In *6th International Information Hiding Workshop*, 2004.

[19] ProGuard: http://proguard.sourceforge.net

[20] G.Qu and M.Potkonjak. Hiding signatures in graph coloring solutions. *In 3rd Information Hiding Workshop*, 1999.

[21] SandMark. http://sandmark.cs.arizona.edu

[22] J.Stern, G.Hachez, F.Koeune, and J.Quisquater. Robust object watermarking: Application to code. *In 3rd Information Hiding Workshop*, 1999.

[23] TRUP. http://ise.thss.tsinghua.edu.cn/trup/index.jsp.

[24] C.Thomborson, J.Nagra, R.Somaraju and C.He. Tamper-proofing software watermarks. In *Australasian Information Security Workshop 2004 (AISW2004)*, 2004.

[25] R.Venkatesan, V.Vazirani and S.Sinha. A graph theoretic approach to software watermarking. *In 4th International Information Hiding Workshop*, 2001.

[26] W.Zhu and C.Thomborson. Extraction in software watermarking. In *ACM Multimedia and Security Workshop,* Geneva*,* Switzerland, 2006.

[27] W.Zhu and C.Thomborson. Recognition in software watermarking. In *1st ACM Workshop on Contents Protection and Security,* Santa Barbara, CA, USA, October, 2006.