Software Watermarking Through Obfuscated Interpretation: Implementation and Analysis

Ying Zeng
Zhengzhou Information Science and Technology Institute, Zhengzhou, China
Email: zengying510@yahoo.com.cn

Fenlin Liu and Xiangyang Luo and Chunfang Yang
Zhengzhou Information Science and Technology Institute, Zhengzhou, China
Email: liufenlin@vip.sina.com, {xiangyangluo, chunfangyang}@126.com

***Abstract*—A robust software watermarking scheme under which the watermark can resist against various attacks including collusion attacks is proposed based on obfuscated interpretation. The idea is to spread the watermark over the entire program by modifying instruction frequencies. The obfuscated interpretation technique is introduced into the scheme to not only hide the functionality of a given program but also provide an alternative simple way to manipulate the instruction frequencies so that embeds the watermark. A series of experimental results show that the proposed watermark can resist against various semantics-preserving transformations and collusion attacks as well.**

***Index Terms*—software watermarking, obfuscated interpretation, robustness, collusion attack**

I. Introduction

Along with the rapid development of software industry and Internet, software piracy has become a major concern for many software companies and IT sectors. The losses caused by software piracy are increasing every year, and exceeded 51 billion dollars in commercial value in 2009 [1]. Software watermarking, a technique used to dissuade illegal copying and resale of programs, has been widely applied to software protection. The idea is to embed a watermark or a fingerprint, such as a copyright notice or a customer identification number, into the code or data segments of an application. The watermark asserts the ownership of the program, while the fingerprint allows the tracking of intellectual property violators. Technically, most of the software watermarking techniques fall under two categories: static and dynamic [2].

Static watermarking techniques embed the watermark in the target application executable, such as the initialized data section, the text section, and the symbol information.

Moskowitz et al. [3] proposed to watermark a program by watermarking a media object and storing it in the code of the program. Unfortunately, there are many ways to disturb the mark in a media object.

Davidson et al. [4] proposed to embed a watermark by reordering the basic block placement. Hattanda et al. [5] and Myles et al. [6] later presented more detailed implementations. Qu et al. [7] proposed to select the register assignment in a way that embeds a watermark. However, since these two algorithms are both based on reordering, they are easy to defeat: attackers simply have to reorder every “reorderable” list of items in the program themselves.

Venkatesan et al. [8] described a graph theoretic approach to software watermarking, which marks a program by the addition of code for which the topology of the control-flow graph encodes a watermark. Then, Collberg et al. [9, 10] implemented and evaluated the algorithm proposed by [8], and showed that its fundamental dependence on static block marking leaves watermarked programs vulnerable to distortive attacks.

Stern et al. [11] proposed a robust spread spectrum based software watermarking algorithm, which embeds a watermark by changing the statistical properties of a program. The algorithm extracts a frequency vector of instruction groups from the original program and then modifies the vector to embed the watermark. However, the transformations used to modify the frequency vector in the three known implementations of this algorithm (Stern et al. [11], Collberg et al. [12], and Hachez [13]) all seem to be undoable by trivial obfuscations or by a competent optimizer. The result is that any inherent robustness of the algorithm is canceled out by the lack of robustness of the transformations [14].

Dynamic watermarking techniques store the watermark in the execution state of the program. The first such technique was proposed by Collberg et al. [15], which embeds the watermark in a graph structure constructed at execution time. Later, they further extended this watermarking algorithm to a dynamic software fingerprinting algorithm [16], which embeds a fingerprint in the topology of a dummy graph that is built on the heap at runtime. And since pointer alias is hard to analysis, the fingerprint can resist automatic attacks.

Nagra et al. [17, 18] proposed to embed new thread into single-threaded portions of a program so that when given the right input, the dynamic behavior of the thread is distinctive and encodes the watermark. This algorithm is secure against static analysis, but for an attacker who has the ability to execute the program, it may fail very easily. Moreover, it is not suitable for programs where speed is critical.

Collberg et al. [19] proposed to encode the watermark in forward branches. It has two ways of encoding the watermark bits, as a sequence of if-statements and as a loop. And it uses the error-correcting code to ensure that only a subset of the watermark pieces is necessary to recover the mark.

Cousot et al. [20] presented a watermarking scheme which embeds the watermark in the values of some designated local variables during the program execution. The advantage of the scheme is that the watermark can be recovered even if only small part of the code is available.

Liu et al. [21] gave a robust software watermarking algorithm based on chaos. The algorithm combines the anti-reverse engineering technique, chaotic system and the idea of Easter Egg software watermarks. Analysis indicates that the algorithm resist against various semantics-preserving transformation and has a good tolerance for reverse engineering.

Kamel et al. [22] gave a method for watermarking R-tree data structure and its variants used by program execution, and showed the robustness of the embedded watermarks by a detailed security analysis and performance evaluation.

From the existing literatures on software watermarking, we know that although dynamic watermarking schemes are more robust than static ones, they have one main drawback, namely, the risk that the recognizer will be affected by small changes to the execution environment such as packet delivery times in the network and the initialization of random number generators [14]. Moreover, it is more difficult to implement the dynamic watermarking schemes, and both the watermark embedding and extraction need to execute the program. Hence despite the fact that current researches on software watermarking have made a great progress, the robustness of watermarks and the efficiency of watermarking schemes still need to be improved.

In this paper, a robust static software watermarking scheme, which can not only resist against semantics-preserving transformations but also collusion attacks, is proposed based on obfuscated interpretation. A frequency vector of instruction groups extracted from the original program is modified to embed the watermark by translation and code insertion. Translation rules defined in obfuscated interpretation provide an alternative simple way to find enough substitution patterns for vector instruction groups, and thus enhance the efficiency of the scheme. Since the software program is protected by obfuscated interpretation, the embedded watermark is robust to various semantics-preserving transformation attacks that modify the program code. Furthermore, as instructions are translated using randomly selected translation rules, attackers can not recognize the location of a watermark by comparing differently watermarked versions of the same program. And this makes the proposed scheme have the ability of resisting against collusive attacks as well.

II. Watermarking Scheme Based on Obufscated Interpretation

Software watermarking is a technique for embedding a unique identifier into the executable of a program. Depending on what aspect of rights we're trying to protect, we can classify a watermark as being robust or fragile. A robust watermark retains its legibility (can be successfully extracted) after undergoing moderate distortive attacks. A fragile watermark is easily destroyed by transformations to the cover object [14]. In this paper, we focus on robust software watermarking.

As we know, code is so fluid and so susceptible to transformation that there seems to be an infinite number of ways of moving code around, splitting and merging pieces of code, adding redundant code, and so on. Thus, improving the robustness of a watermark turns out to be difficult. Like other security concepts, robustness is a relative measure. There is no system that is 100% secure or a watermark is 100% robust [22]. In practice, a system is consider secure or robust enough if

- destroying or removing the watermark would significantly affect the functionality or the performance of the underlying code;
- the cost of breaking into the system exceeds the cost of the system or benefit from breaking into it, or;
- the time required to break into the system exceeds the life time of the data.

Stern et al. made an attempt at improving the robustness of software watermarks. And we refer to their algorithm as the SHKQ algorithm. The SHKQ algorithm watermarks in the frequency domain. Code is not considered as a linear succession of instructions, but rather a whole statistical object. The basic idea is to extract a vector of instruction-group frequencies from the original program and then iteratively modify the program until this vector is "different enough" from the original to embed the watermark. It took the advantage of the secure spread spectrum watermarking algorithm for images presented by Cox et al. [23], which is that if an attacker cannot perform "stronger" modifications than the original one, then the watermark will be recovered with very high probability.

However, in the three known implementation of the SHKQ algorithm, the transformations that are used to modify the code all seem to be undoable by trivial obfuscations or by a competent optimizer. The result is that any inherent robustness of the algorithm is canceled out by the lack of robustness of the transformations.

To solve the above problem and further improve the robustness of the SHKQ algorithm, we proposed a robust software watermarking scheme based on obfuscated interpretation in this paper. Our watermarking scheme is based on obfuscated interpretation for three reasons:

1) As a software protection technique, which combines obfuscation, diversity and tamper-proofing, obfuscated interpretation enables us to hide the functionality of a given program so that improves the robustness of the watermark embedded in the program;

2) The translation procedure of obfuscated interpretation provides us an alternatively new transformation to modify code;
3) Combining the translation procedure of obfuscated interpretation with watermark embedding makes it difficult for attackers to recognize whether a difference between two differently watermarked versions of the same program is caused by watermark embedding or translation. Hence, it makes our scheme robust to collusion attacks.

In the following, we will first give a brief introduction on obfuscated interpretation, and analyze how it can be used to embed watermarks, and then we will present the framework of our watermarking scheme.

A. *Brief Introduction on Obfuscated Interpretation*

Obfuscated interpretation is a software protection technique, which combines obfuscation, diversity and tamper-proofing. Instead of obfuscating a program itself, it obfuscates the interpretation process of the program. There is no static relationship between the instructions and their semantics. The interpretation for a given instruction is not fixed; specifically, it is determined not only by the instruction itself but also by some other auxiliary input. Fig. 1 shows this non-static relationship between instructions and their semantics.

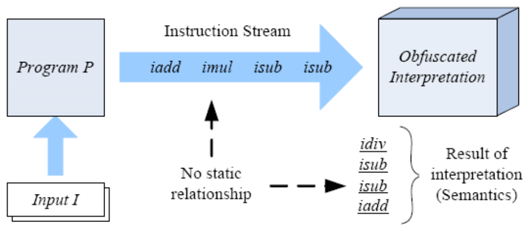


Figure 1. Obfuscated interpretation concept [24]

Monden et al. [24] gave an idea for obfuscating the program interpretation based on FSM (Finite State Machine). However, the FSM interpretation unit is hardware imple-mented, and its state transition rules can not be changed any more once embedded. Zhang et al. [25] proposed a similar framework for obfuscating the interpretation process of Java programs. It is achieved by a custom interpreter which allows the mapping rules between instructions and semantics to be easily changed at will. In our watermarking scheme, we use the obfuscated interpretation framework proposed by Zhang et al. to embed watermarks and protect programs.

The process of obfuscated interpretation is illustrated in Fig. 2. For a given program P, which is intended to be hidden from hostile users, it is first translated by a program translator T. The program translator T automatically translates P into P' by randomly selecting translation rules from the one-to-many translation mapping for all possible instructions. P' is the "translated program" containing translated instructions whose semantics are determined during execution according to the auxiliary input A_u. A_u records the indexes of interpretation rules corresponding to each translated instruction in P'. When P' is executed, the permutation-based interpreter W_p evaluates the translated instructions of P' according to the auxiliary input A_u. Finally it is ensured that the obfuscated interpretation results of P' are the same as the conventional interpretation results of P.

Since programs need to be translated before obfuscated interpretation, the translation procedure provides us a new transformation to modify the code and embed watermarks. Next, let us discuss the translation procedure and see how it can be used to embed watermarks.

Let $\Sigma=\{c_0,c_1,\ldots,c_l\}$ be a set of instructions, $\Psi=\{\underline{c_0},\underline{c_1},\ldots,\underline{c_l}\}$ be a set of instruction semantics, $\Pi=\{\pi_0,\pi_1,\ldots,\pi_r\}$ be a set of indexes of interpretation rules, and $Id:\Sigma\to\Psi$ be the regular mapping between instructions and semantics, i.e. $Id(c_i)=\underline{c_i}$, $0\le i\le l$. For an interpretation mapping $\lambda:\Sigma\times\Pi\to\Psi$, a translation mapping $\pi:\Sigma\to\Sigma\times\Pi$ can be derived by Id and λ, i.e. $\pi=\lambda^{-1}\circ Id$. π is a one-to-many mapping, thus an instruction can be translated by randomly selecting a translation rule from this one-to-many mapping.

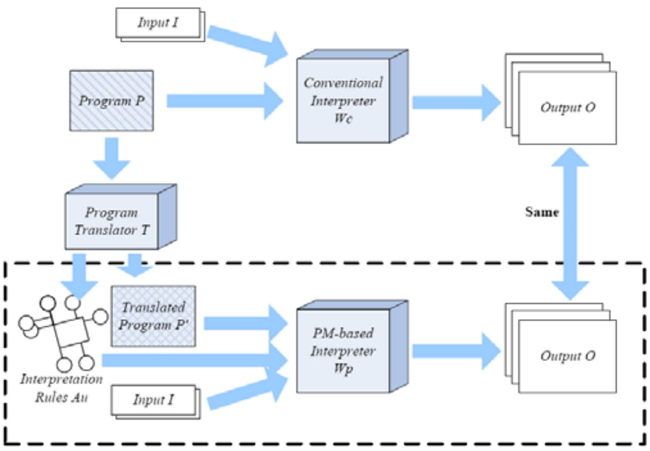


Figure 2. Obfuscated interpretation framework [25]

For instance, in the example showed in Fig. 3, there are four translation rules for *iadd*, which are $\pi(iadd)=(iadd,2)$, $\pi(iadd)=(isub,2)$, $\pi(iadd)=(imul,0)$, $\pi(iadd)=(idiv,3)$. When *iadd* is translated, a random selection can be made from these rules. In the example $\pi(iadd)=(isub,2)$ is chosen. Thus, *iadd* is translated into *isub*, and the corresponding interpretation rule index "2" is recorded. Then during the interpretation process, *isub* is interpreted as $\underline{iadd}$ using the interpretation rule $\lambda(isub,2)=\underline{iadd}$ indexed by "2".

The translation of an instruction is random. Hence we can take advantage of this translation procedure to modify the frequencies of instruction groups. To illustrate this point, let us consider a simple example. Assume the existence of the vector instruction group [$bipush, isub$], where $bipush\ X$ pushes a literal integer X onto the evaluation stack, $isub\ X\ Y$ pops two integers from the top of the stack, subtract X from Y, and pushes the result. If an instruction group [*bipush*, *iadd*] is found in the code, then we can choose to translated the instruction *iadd* into *isub* , so that increases the frequency of the vector instruction group [*bipush*, *isub*] by one.

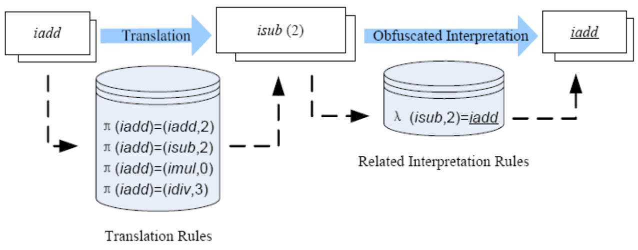


Figure 3. An example of translation of instruction *iadd*

B. The Framework of the Obfuscated Interpretation-based Watermarking Scheme

The framework of our proposed watermarking scheme is illustrated in Fig. 4. It includes four phases: code book construction, watermark embedding, obfuscated interpretation, and watermark extraction.

In the phase of code book construction, we first profile a set of benchmark programs to extract a table of frequently occurring code patterns. Then we construct a code book from these patterns manually. A set of vector instruction groups as well as a set of substitution patterns for these groups are defined in the code book. We select code patterns that include instructions needed to be translated as vector instruction groups, so that we can use the translation rules of the instructions needed to be translated to design the substitution patterns for vector instruction groups.

In the phase of watermark embedding, we extract a vector v of instruction group frequencies from a program P that needs to be watermarked. We then embed a watermark vector w into the program by manipulating instruction group frequencies using the substitution patterns in the code book. Finally, by modifying the code, the new frequency vector v' extracted from the watermarked program P' should be equal to $v+w$. Meanwhile, we randomly translate the program P, and record the indexes of the interpretation rules corresponding to each translated instruction in P. This auxiliary information A_u is encrypted and stored in the watermarked program P'. The following algorithm formalizes the above steps in the phase of watermark embedding.

Algorithm 1 (Watermark Embedding)

Input: program P needs to be watermarked, a vector $S=(s_1,\ldots,s_n)$ of instruction groups.

Output: watermarked program P', auxiliary information A_u that are the indexes of the interpretation rules corresponding to each translated instruction.

Steps:

1) For each instruction group s_i in S, count the frequency v_i of the group in the program P, and form the frequency vector $v=(v_1,\ldots,v_n)$.
2) Choose an n-coordinate vector $w=(w_1,\ldots,w_n)$ whose coefficients are randomly distributed following a normal law with standard deviation α.
3) Modify the program P in such a way that the new extracted frequency vector v' is $v+w$.
4) Randomly translate the watermarked program, and obtain the program P'.
5) Record the indexes of the interpretation rules corresponding to each translated instruction. Encrypt and store this information in P'.
6) Return P' and A_u.

In the phase of obfuscated interpretation, we execute the watermarked program P' by the permutation-based interpreter W_p. W_p first decrypt the auxiliary information A_u stored in P', and then it evaluates each instruction in P' and decides whether it is a translated instruction. If so, it interprets the instruction according to A_u, otherwise it interprets the instruction normally.

In the phase of watermark extraction, we extract frequency vector v and d from the original program P and the program P_t needs to be tested respectively. Then we compare $d-v$ to w. If they are similar enough (according to some similarity measure), we output "marked" else "unmarked". And this phase can be formalized by Algorithm 2 as follows.

Algorithm 2 (Watermark Extraction)

Input: original program P, program P_t needs to be tested, vector $S=(s_1,\ldots,s_n)$ of instruction groups.

Output: "marked" or "unmarked".

Steps:

1) Set a detection threshold σ, ($0<\sigma<1$).
2) For each instruction group s_i in S, count the frequency v_i of the group in the original program P, and form the frequency vector $v=(v_1,\ldots,v_n)$.
3) For each instruction group s_i in S, count the frequency d_i of the group in the program P_t that needs to be tested, and form the frequency vector $d=(d_1,\ldots,d_n)$.
4) Compute a similarity measure Q between $d-v$ and w.
5) If Q is higher than σ then return "marked", else return "unmarked".

III. Implementation of the Proposed Watermarking Scheme

In this study, we implemented our watermarking scheme for Java programs. The code book building, watermark embedding and extraction were all implemented within the SandMark environment, which is a tool developed for research into software watermarking, tamper-proofing, and code obfuscation. The platform that we chose to implement obfuscated interpretation is Mysaifu JVM, which is an open source Java Virtual Machine running on the Windows mobile platform.

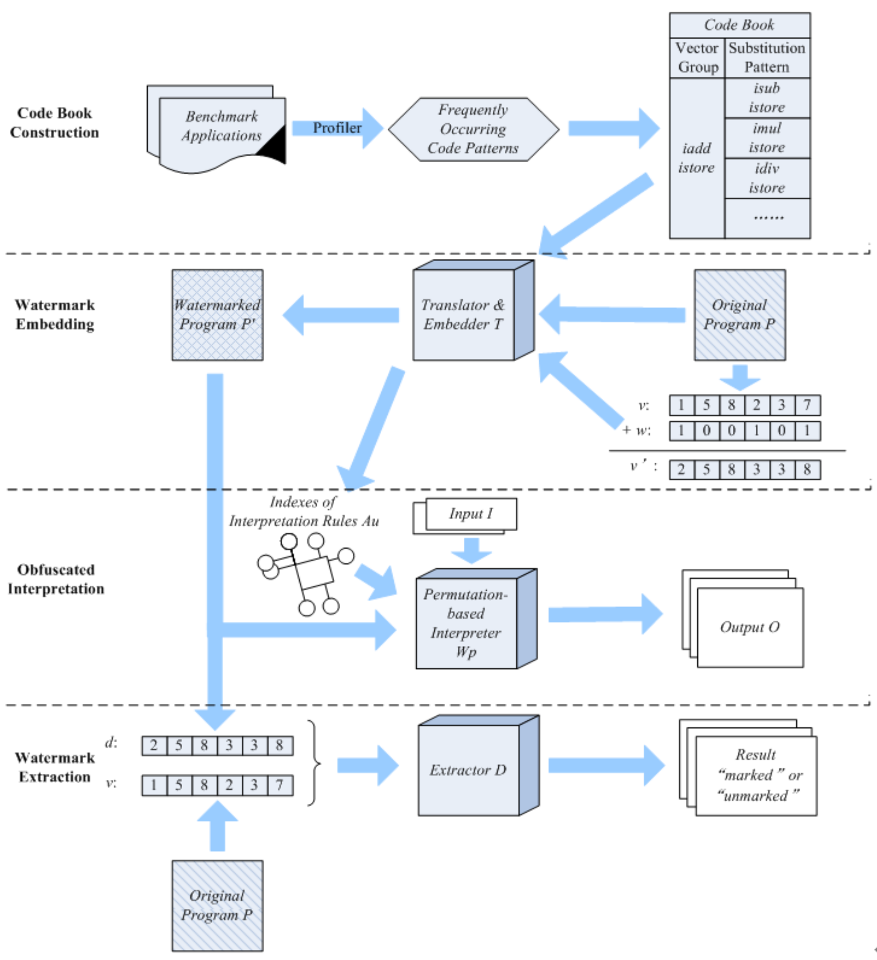


Figure 4. The framework of the obfuscated interpretation based watermarking scheme

A. *Building the Code Book*

As we show above, our watermarking scheme makes use of a code book. It defines a set of vector instruction groups and a set of substitution patterns for these groups. The code book is assumed to be kept secret from attackers.

To ensure that the watermarked program is similar to the code that occurs in typical applications so that does not arouse suspicion, the code book is constructed based on what are frequently occurring code sequences in real programs. To collect this information, we collected statistics from different programs and identified which instruction groups have more likelihood to occur in real code. We profiled over different programs such as specJVM benchmark, JFIG and SandMark. To simplify the watermark embedding procedure, we were only interested in instruction groups of size 2. The frequently used instruction groups obtained from profiling form the potential candidates for vector instruction groups.

Since watermarks are embedded in the frequencies of the vector instruction groups in our watermarking scheme, we selected frequently occurring instruction groups which can be transformed to manipulate the frequencies of instruction groups. And from the discussion in Section II, we observe that the translation procedure of obfuscated interpretation can be used to modify code and manipulate the frequencies of instruction groups. Thus, we took advantage of the translation rules for instructions in obfuscated interpretation to select vector instruction groups from the profiling instruction groups and define the substitution patterns for these vector instruction groups.

In the translation procedure, to make sure that the translated program can pass Java's bytecode verifier, instructions are divided into several groups according to their operand type and operand number so that instructions in the same group can substitute each other without causing any stack or syntactic errors. Table I shows six groups of instructions which can be obfuscated interpreted. Instructions within each group can be randomly translated into one of the instructions in the same group. For example, instruction *fadd* can be translated into *fsub* , *fmul* , *fdiv* , *frem* randomly.

Therefore, the restrictions on the selecting of vector instruction groups are as follows. We refer to the instructions labeled with * in Table I. as key instructions.

- The size of the instruction groups is 2.
- Each vector instruction group should contain one key instruction.

And the substitution patterns for these vector instruction groups were defined according to the translation rules of the containing key instruction. We reserve one instruction in each group in Table I. not to be used to define the vector instruction groups for the purpose of providing enough available substitution patterns in the program that needs to be watermarked.

To illustrate these points we will next consider a simple example. Assume the existence of the instruction group [*iload,iadd*]. As it contains the key instruction *iadd* in group 3 of Table I, and is a frequently occurring instruction group of size 2, we then define it as a vector instruction group. And according to the seven translation rules for *iadd* , we can define seven substitution patterns for this vector instruction group [*iload,iadd*], which are [*iload,isub*], [*iload,imul*], [*iload,idiv*], [*iload,irem*], [*iload,iand*], [*iload,ior*], [*iload,ixor*]. If one of those instruction groups is found in the code, it can be replaced with [*iload,iadd*]. After a substitution has taken place we have increased the frequency of the instruction group [*iload,iadd*] by one. And as long as we record the corresponding interpretation rule, the obfuscated interpreter can interpret the instructions correctly.

TABLE I. GROUPS OF INSTRUCTIONS

No.	Instructions	No.	Instructions	No.	Instructions
1	iconst_m1 *	2	ladd *	3	iadd *
	iconst_0 *		lsub *		isub *
	iconst_1 *		lmul *		imul *
	iconst_2 *		ldiv *		idiv *
	iconst_3 *		lrem *		irem *
	iconst_4 *		land *		iand *
	iconst_5		lor *		ior *
			lxor		ixor
4	fadd *	5	dadd *	6	if_icmpeq *
	fsub *		dsub *		if_icmpne *
	fmul *		dmul *		if_icmplt *
	fdiv *		ddiv *		if_icmple *
	frem		drem		if_icmpgt *
					if_icmpge

B. Embedding by Translation

In order to solve the problem of finding a large set of semantically equivalent groups of instructions, we proposed an alternative simple way to embed a watermark using the translation procedure of obfuscated interpretation. Since the vector instruction groups and their substitution patterns are defined according to the translation procedure of obfuscated interpretation, embedding a watermark in the frequency of a vector instruction group can be done by translating a key instruction. The details are given in the following algorithm.

Algorithm 3 (Embedding by Translation)

Input: program P, vector instruction group s_i whose frequency needs to be increased by one, the set $C_{s_i} = \{c_1, \cdots, c_m\}$ of substitution patterns for the vector instruction group s_i, watermark vector $w = (w_1, \ldots, w_n)$ that need to be embedded, vector $S = (s_1, \ldots, s_n)$ of the vector instruction groups.

Output: program P, and "*success*" or "*failure*".

Steps:

```
1) Set a maximum trying threshold $Tc$, and initialize
   a counter $r = 0$.
2) If $r > Tc$, then go to 7); else randomly select a pattern
   $c_k$ from the set $C_{s_i}$, and increase $r$ by one.
3) Search $c_k$ in set $S$, if $c_k$ is found, then go to 2);
   else go to 4).
4) Search the program for a sequence of instructions
   $Seq$ matched to the selected pattern $c_k$. If $Seq$ is
   found, then go to 5); else go to 2).
5) Check whether translate the key instruction in $Seq$
   will affect another vector instruction group or not,
   if so, then go to 2); else go to 6).
6) Translate the key instruction in $Seq$, which means
   to replace $Seq$ with $s_i$, and record the index of the
   interpretation rule related to the translated key instruction.
   Decrease $w_i$ by one, and return program
   $P$ and "success".
7) Return program $P$ and "failure".
```

Embedding by translation is our default embedding scheme, and only when there is no further scope for translation do we turn to the other embedding scheme. However, there is a main complication in this embedding scheme that is this embedding scheme has as a side effect, and it may affect more than one vector instruction group. For instance, assume that there are two vector instruction groups A and B whose frequencies both need to be increased. If A is translated into B to increase the frequency of B, then the frequency of A will decrease. And in the three known implementation of the SHKQ algorithm, similar problem also existed. To face this problem, different approaches were proposed by researchers.

Hachez followed a random hill-climbing approach, which intends to change multiple vector components at each step of modification, so that there are always side effects, and the vector frequency updates might not converge to the targeted vector, or the convergence may be time consuming.

Collberg et al. shifted the complexity onto the code book, and followed a directed hill-climbing approach. Restrictions are applied on the design of the code book to make sure that a code substitution or insertion only affects one vector instruction group. However, this further increases the difficulty of finding appropriate substitution patterns for the vector instruction groups and the design of the code book.

In our implementation, we also revert to the directed approach, which means at each step we move closer to the final result so that guarantees the convergence of the vector frequency updates. But we did not shift the complexity on the code book; instead we added some controls in the embedding procedure. Before we commit to translate an instruction, we first verify that the transformation only affect one vector instruction group. And this is done in Step 3) and 5) of Algorithm 3.

Specifically, let us consider a concrete example to illustrate this embedding procedure. Assume a vector instruction group [*iload,iadd*], together with seven substitution patterns [*iload,isub*], [*iload,imul*], [*iload,idiv*], [*iload,irem*], [*iload,iand*], [*iload,ior*], [*iload,ixor*]. There are two situations that we should take into consideration.

- If one of the substitution pattern is also a vector instruction group, for example [*iload,isub*], then we cannot use it to translate the vector instruction group [*iload,iadd*], since that will affect the frequency of [*iload,isub*] as well.
- If the key instruction that we are trying to translate forms another vector instruction group in the program, for example as showed in the following case. Assume that there is a sequence of instructions in the program as below:

  iload

  iand

  bispush

  If [*iand, bipush*] is also a vector instruction group, then translating [*iload,iand*] into [*iload,iadd*] will affect the frequency of [*iand, bipush*] as well. And we should verify that this case does not happen before we commit the translation.

C. Embedding by Code Insertion

There are cases when not all the components of the watermark vector can be embedded by translation. In these cases, we need to embed the watermark by inserting dumb code. The detail is described in the following algorithm.

Algorithm 4 (Embedding by Code Insertion)

Input: program P , a vector instruction group s_i whose frequency needs to be increased by one, watermark vector $w=(w_1,\ldots,w_n)$ that need to be embedded.

Output: program P .

Steps:

```
1) Randomly select a location in the code of program $P$.
2) Insert a true opaque predicate in the selected location.
3) Add codes that contain the vector instruction group $s_i$ to the false branch of the newly inserted true opaque predicate, and decrease $w_i$ by one.
4) Return program $P$.
```

D. Embedding and Translation Procedure

In our watermarking scheme, the watermark embedding and translation procedure are combined together. We have two embedding approaches: embedding by translation and embedding by code insertion. Both these two approaches are directed hill-climbing approaches, i.e. each code modification takes up at most one vector instruction group. And the embedding and translation procedure can be summarized as follows.

Algorithm 5 (Embedding and Translation Procedure)

Input: program P, watermark vector $w=(w_1,\ldots,w_n)$ that need to be embedded.

Output: program P .

Steps:

```
1) Attempt to embed watermark vector $w$ by translation, according to Algorithm 3.
2) If all vector component frequencies have reached their desired level, then go to 4); else go to 3).
3) Embed the rest vector component frequencies that have not reached their desired level by code insertion, according to Algorithm 4.
4) Randomly translate the rest key instructions that have not been embedded watermark components.
5) Encrypt the indexes of the interpretation rules of the translated instructions with a stream cipher, and add an attribute which contains the encrypted information in the attributes table of a code attribute.
6) Return program $P$.
```

Note that stream cipher is used to encrypt the indexes of the interpretation rules of the translated instructions, so that different ciphertexts can be obtained every time the same plaintext is encrypted and thus keep this information secret from attackers. And we use SIM card number as the key to encrypt the information.

E. Extraction and Obfuscated Interpretation Procedure

Our watermarking scheme is static and informed. Provided with the watermark, the target program to be examined and the original program, the watermark extractor returns a “marked/unmarked” whether the given watermark is present in the target program. The extractor is implemented to first retrieve the frequency vectors from the original program and the target program, and then calculate the normalized linear correlation between the two vectors to detect the watermark. The normalized linear correlation between two vectors v and w is as follows:

$$z_{nc}=\frac{1}{N}\sum\tilde{v}_i\tilde{w}_i \ , \tag{1}$$

where v_i and w_i denote the *i*th vector component in their respective vectors, N denotes the number of vector components in each vector, $\tilde{v}=v/|v|$, and $\tilde{w}=w/|w|$. We assume that a correlation ratio greater than 0.9 implies watermark found, and less than 0.6 implies watermark not found. And we are not quite sure whether the watermark is present or not when the correlation ratio is in the range of 0.6-0.9.

For a program, the embedding and translation procedure changes its instructions and hides the functionality of the program. The semantics of the instructions are not static any more, they are determined during the program execution. To execute the program, a permutation-based interpreter is needed. We implemented our permutation-based interpreter on the basis of Mysaifu JVM. Specifically, the obfuscated interpretation procedure is described in Algorithm 6.

Algorithm 6 (Obfuscated Interpretation Procedure)

Input: program P .

Output: the execution of program P .

Steps:

```
1) Decide whether the program is translated. If so, then go to 2); else go to 4).
```

2) Decrypt the attribute info in the program to obtain the indexes of the interpretation rules of the translated instructions. We refer to this information as A_u .
3) Interpret the instructions in the program according to information A_u , and return.
4) Interpret the instructions in the program normally, and return.

IV. WATERMARK EVALUATION

In this section, we evaluate our watermarking scheme according to the following criteria:

- Data rate, which specifies the ratio of the number of bits of watermark that can be embedded in the target program to the size of the program.
- Embedding overhead, which refers to the amount of time and space the watermarking scheme added to the program.
- Resilience against semantics-preserving transformations, which measures the ability of the watermark to survive transformations such as code obfuscation and code optimization.
- Resilience against collusive attacks, which checks whether the location of the watermark can be determined by comparing two or more differently watermarked copies of the same program.
- False positive rate, which is the probability that the watermark extractor recognize a random value as a valid watermark.

A. Data Rate

Since each vector instruction group encodes certain number of bits of watermark, the data rate of the watermark essentially depends on the number of vector instruction groups defined in the code book. By profiling over a set of different programs, we found that there are more than 1000 instruction groups that can be the potential candidates for vector instruction groups. And according to the translation rules of the key instructions, we can easily build the substitution patterns for the vector instruction groups. Thus, we can define a large set of vector instruction groups in the code book.

However, small programs may not provide enough scope for embedding by translation. In other words, there may not be any available instruction sequence that matches to the substitution patterns for the vector instruction groups. But as the program size increases, more bits can be embedded by translation.

B. Embedding Overhead

As for all the substitution patterns defined according to the translation rules, the original and the substituted code segments are of the same length. Thus, embedding by translation has a minimal impact on the size and performance of the watermarked program. However, not all frequencies of the vector instruction groups can be changed by translation. In order to embed a watermark, additional methods are needed to be constructed, such as insertion dumb code. Embedding by code insertion will somehow increase code size, but the effect on execution time is minimal. In addition, the obfuscated interpretation used in our watermarking scheme also has some effect on execution time.

We applied our watermarking scheme on several java programs. Table II summarizes the effects of watermark embedding on code size and load time. The Mysaifu JVM is set as follows

- Max heap size: 2M;
- Java stack size: 32KB;
- Native stack size: 160KB:
- Verifier: Off.

And execution environment is Pocket PC 2003 SE Emulator. The results show that the program size is increased by maximum 6% and the max load delay is less than 11%.

TABLE II. EFFECTS OF WATERMARK EMBEDDING ON CODE SIZE AND LOAD TIME

Program	Size (bytes)			Load Time (sec)		
	Bef	Aft	%Incr	Bef	Aft	%Incr
Example	111580	113524	1.7	29.8	30.9	3.7
ImageViewer	4137	4212	1.8	20.6	21.4	5.0
Jode	530491	561782	5.9	62	65.1	5.0
Jbubblebreaker	187795	190150	1.3	53	55.2	4.2
JHEditor	77036	79982	3.8	52	57.6	10.8

C. Resilience against semantics-preserving transformations

Semantics-preserving transformations are the most serious threat to software watermarks. To destroy the embedded watermarks, automated attacks can be done by using tools that perform code obfuscation or code optimization. And there exist several such tools, such as Debray, BLOAT, SmokeScreen and SandMark. These tools are free and commonly available, thus it is essential for a software watermark to be resilient against semantics-preserving transformations.

To evaluate the resilience of our watermark against semantics-preserving transformation, we used various obfuscation algorithms implemented in SandMark as attacks on watermarks. These obfuscations include class level modifications as well as method level modifications. We compared the resilience of our watermark against various obfuscations on a set of Java programs to that of the SHKQ watermark. Table III shows the results of our evaluation. The embedded watermark was 45572689, and the watermark detection threshold was set to 0.9. Columns “a” and columns “b” in Table III show the evaluation results of the SHKQ watermark and our watermark respectively. “√” means that the watermark is found in the program after obfuscation. “×” means that the watermark is destroyed by obfuscation. “∇” means that the program is crashed by obfuscation.

From the evaluation results showed in Table III, we observe that both the SHKQ watermark and our watermark survive the high-level obfuscations that affect classes, fields, and method signature. However, for obfuscations that affect codes in methods, watermarks may not survive.

The “Method Merger” obfuscation merges all of the public static methods that have the same signature in each class into one large master method. Thus, this transformation may disturb some of the vector instruction groups and destroy the embedded watermark.

The “Promote Locals” and “Primitive Promoter” obfuscations both belong to a class of local variable promotion obfuscations, which convert primitive types like int and float to their equivalent layers of abstraction, such as java.lang.Integer and java.lang.Float. These two obfuscations manipulate and change instructions that depend on retrieving and storing int or float values. For example, assume that there exists an original instruction:

iload < index > .

The two obfuscations will manipulate this instruction and replace it with the following instructions:

aload < index >

invokevirtual java/lang/Integer/intValue(V)I .

Hence, they can alter the frequency vector considerably and destroy the watermark.

The “Append Bogus Code” obfuscation adds bogus codes onto the end of a java method. Therefore it may affect the frequencies of some vector instruction groups and destroy the watermark.

According to the results in Table III, we can see that the SHKQ watermark was destroyed by some obfuscating transformations, such as “Method Merger”, “Promote Locals”, “Primitive Promoter” and “Append Bogus Code”.

TABLE III. EFFECTS OF OBFUSCATORS ON A SET OF WATERMARKED PROGRAMS

Obfuscation	Image viewer		Jode		Jbubble breaker		JHEditor	
	a	b	a	b	a	b	a	b
Variable Reassigner	√	√∇	√	√∇	√	√∇	√	√∇
Bogus Arguments	√	√∇	√	√∇	√	√∇	√	√∇
Name Obfuscator	√	√	√	√	√	√	√	√
Method Merger	√	√∇	√	√∇	×	×∇	×	√∇
Publicizer	√	√∇	√	√∇	√	√∇	√	√∇
Parameter Reorder	√	√∇	√	√∇	√	√∇	√	√∇
Promote Locals	×	√∇	×	×∇	×	×∇	×	√∇
Set Fields Public	√	√∇	√	√∇	√	√∇	√	√∇
Primitive Promoter	√	√∇	×	×∇	×	√∇	×	√∇
Constant Pool Reorder	√	√∇	√	√∇	√	√∇	√	√∇
Local Variable Reorder	√	√∇	√	√∇	√	√∇	√	√∇
Buggy Code	√	√∇	√	√∇	√	√∇	√	√∇
Append Bogus Code	√	√∇	×	√∇	√	√∇	√	√∇
Variable Splitter	√	√∇	√	√∇	√	√∇	√	√∇

a: programs watermarked by the SHKQ algorithm, b: programs watermarked by o proposed sch eme √: watermark found, ×: watermark destroyed, ∇: program crashed.

As for the test programs watermarked by our watermarking scheme, since they were protected by obfuscated interpretation, obfuscations that affect the instructions in the programs would make obfuscated interpretation of the programs incorrect. Hence, although our watermark was damaged by several obfuscations, the watermarked programs were seriously damaged by the obfuscations as well. In other words, destroying our watermark by these obfuscations significantly affected the functionalities of the watermarked programs. Therefore, these obfuscations were not successful attacks on our watermark.

Note that the “Name Obfuscator” obfuscation did not affect the functionality of the test program. And this is because that this obfuscation only renames the fields and methods of a program to unique identifiers, and it does not disturb the instructions in the program. However, this obfuscation is weak. Since it does not alter the vector instruction groups, watermarks can survive this obfuscation.

D. Resilience against collusive attacks

A collusion attack is a common attack on watermarks, specifically fingerprints, which attempts to find the watermark location by comparing two or more differently watermarked programs. Once the watermark locations have been found, a manual inspection of the code may reveal useful details about the watermark scheme [12]. Thus hiding watermark locations is essential for a software watermarking scheme to improve its resilience against collusion attacks.

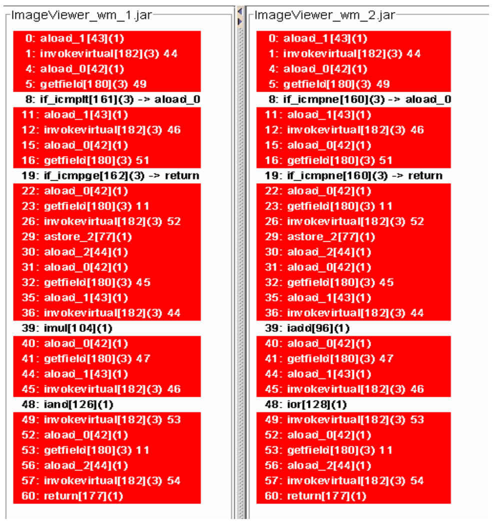
As pointed out in [12], in the case of the SHKQ algorithm, a collusion attack may be able to reveal the patterns of the code book. Thus we introduce obfuscated interpretation in our watermarking scheme to improve the resilience of the watermark against collusion attacks. Instructions in the program are modified by watermark embedding as well as the translation procedure of obfuscated interpretation, so that attackers can not decide whether a difference between two differently watermarked programs is caused by watermark embedding and find the watermark locations.

We tested the resilience of our watermarking scheme against collusion attacks by SandMark which is equipped with a Java bytecode comparison tool. Fig. 5 shows an example of a collusion attack on our watermark. Two different watermarks were embedded in the java program “ImageViewer”. The two columns show the results of the “mouseDragged” method in the two differently watermarked programs. The differences in the code have been highlighted in Fig. 5. There are totally four differences in the two methods. But only the “*imul*” in the left column and “*iadd*” in the right column were embedded a bit of the watermark. The other three differences were all made by the random translation of obfuscated interpretation. And we observed that the two differently watermarked programs differed not just at the locations of the watermark, but also at some other locations. Thus, it is difficult for attackers to locate the watermark and deuce useful information.

E. False positive rate

Besides to be resilient against various attacks, it is also important for a watermarking scheme to have a low probability of false detections. The false positive rate essentially depends on the length of the watermark vector and the range of values that the watermark vector components take.

Since we use the translation rules of key instructions to construct the code book, the size of the set of the vector instruction groups is not limited by the lack of available substitution patterns for the vector instruction groups. Hence while in Collberg et al.’s implementation of the SHKQ algorithm, the length is only set to 8 due to the difficulty of finding appropriate substitution patterns for instruction groups, we set the length of the watermark vector to 1000. And it makes that our watermark is spread over a large mount of frequencies.


ImageViewer_wm_1.jar

```
 0: aload_1[43](1)
 1: invokevirtual[182](3) 44
 4: aload_0[42](1)
 5: getfield[180](3) 49
 8: if_icmplt[161](3) -> aload_0
11: aload_1[43](1)
12: invokevirtual[182](3) 46
15: aload_0[42](1)
16: getfield[180](3) 51
19: if_icmpge[162](3) -> return
22: aload_0[42](1)
23: getfield[180](3) 11
26: invokevirtual[182](3) 52
29: astore_2[77](1)
30: aload_2[44](1)
31: aload_0[42](1)
32: getfield[180](3) 45
35: aload_1[43](1)
36: invokevirtual[182](3) 44
39: imul[104](1)
40: aload_0[42](1)
41: getfield[180](3) 47
44: aload_1[43](1)
45: invokevirtual[182](3) 46
48: iand[126](1)
49: invokevirtual[182](3) 53
52: aload_0[42](1)
53: getfield[180](3) 11
56: aload_2[44](1)
57: invokevirtual[182](3) 54
60: return[177](1)
```

ImageViewer_wm_2.jar

```
 0: aload_1[43](1)
 1: invokevirtual[182](3) 44
 4: aload_0[42](1)
 5: getfield[180](3) 49
 8: if_icmpne[160](3) -> aload_0
11: aload_1[43](1)
12: invokevirtual[182](3) 46
15: aload_0[42](1)
16: getfield[180](3) 51
19: if_icmpne[160](3) -> return
22: aload_0[42](1)
23: getfield[180](3) 11
26: invokevirtual[182](3) 52
29: astore_2[77](1)
30: aload_2[44](1)
31: aload_0[42](1)
32: getfield[180](3) 45
35: aload_1[43](1)
36: invokevirtual[182](3) 44
39: iadd[96](1)
40: aload_0[42](1)
41: getfield[180](3) 47
44: aload_1[43](1)
45: invokevirtual[182](3) 46
48: ior[128](1)
49: invokevirtual[182](3) 53
52: aload_0[42](1)
53: getfield[180](3) 11
56: aload_2[44](1)
57: invokevirtual[182](3) 54
60: return[177](1)
```

Figure 5. Example of a collusive attack on our proposed watermark

We evaluated the false positive rate of our watermarking scheme by giving a random value to the watermark extractor to see whether the random value is recognized as a valid watermark or not. We embedded a watermark 74893657 into the program "Jbubblebreaker", which is a bubble breaker game written in Java. And then we generated several random watermarks and tested whether the extractor recognized them as valid watermarks. Fig. 6 shows the experimental results of detection of false watermarks. For these values, the correlation ratio all lay below the 0.9 upper threshold. In several cases, the correlation ratio was even below the 0.6 lower threshold. And compared to the SHKQ algorithm, the false positive rate of our watermarking scheme is lower.

According to the above evaluation of our obfuscated interpretation-based watermarking scheme, we have the following results

- The data rate of our proposed scheme is improved by introducing translation into watermark embedding.
- The embedding overhead is low. For moderately sized java programs, the program size is increased by maximum 6% and the max load delay is less than 11%.
- The watermarks are resilient to semantics-preserving transformations. Code obfuscations are either too simple to destroy the watermarks, or so strong that the functionalities of the watermarked program are affected.
- The watermarks are resilient to collusion attacks. The translation introduced in watermark embedding makes two differently watermarked versions of the same program different everywhere not just at the watermark locations, so that it is difficult for attackers to recognize the watermark location by comparing.
- The false positive rate is low. In our implementation, the length of the watermark vector is large, which makes the probability of recognizing a random number as a valid watermark low.

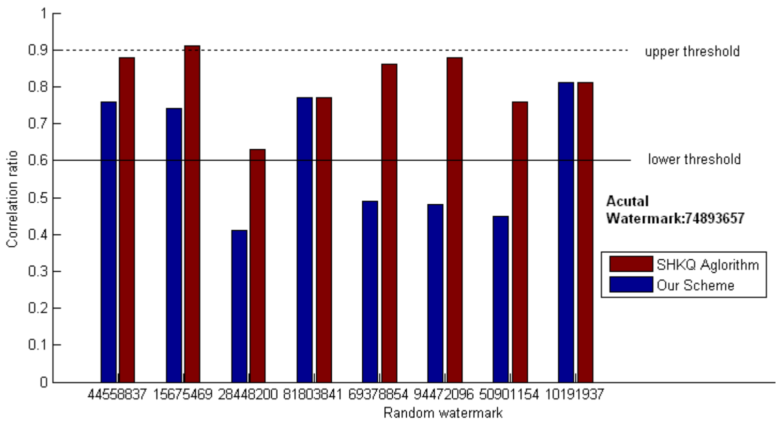


Figure 6. Detection results of false watermarks by the extractor

V. Conclusions

Based on obfuscated interpretation, this paper has proposed a robust software watermarking scheme which is resilient against various semantics-preserving transformations and collusion attacks as well. The proposed watermarking scheme has the following characteristics:

1) The proposed watermarking scheme introduces the translation procedure in obfuscated interpretation into watermark embedding, so that makes it simple to define substitution patterns for vector instruction groups, and improves the efficiency of the software watermarking scheme.
2) The proposed watermarking scheme is combined with obfuscated interpretation, which is a software protection technique that hides the functionality of the protected program. Hence, watermarks embedded by our scheme are resilient to semantics-preserving transformations.

3) Since instructions in a program are modified by both watermark embedding and translation, two differently watermarked versions of the same program are different everywhere not just at the watermark locations. And as a result, it makes the proposed watermarking scheme resilient against collusion attacks.

Experimental results have showed that the proposed scheme have a high data rate and a low embedding overhead. Moreover, it can resist against various semantics-preserving transformations and collusion attacks as well. Also the false positive rate of the proposed scheme is lower than that of the SHKQ algorithm. However the watermark extraction is not blind, and so research on blind software watermarking scheme will be our next step.

This paper is an extended version of our previous work [26]. Changes were made to provide more detailed explanation of our watermarking scheme and to present a comparison with the SHKQ algorithm proposed by Stern et al.

ACKNOWLEDGMENT

The authors wish to thank B. Lu and L. Chen for comments and suggestions. This work was supported in part by grants from the National Natural Science Foundation of China under Grant Nos. 60970141, 60902102 and Technology Innovation Excellent Youth Foundation of Henan Scientific Committee under Grant No. 094200510008.

REFERENCES

[1] Business Software Alliance, "*Seventh Annual BSA/IDC Global Software 09 Piracy Study*," http://portal.bsa.org/globalpiracy2009/studies/globalpiracystudy2009.pdf, May. 2010.

[2] C. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation-Tools for software protection," *IEEE Trans. Softw. Eng.*, vol. 28, no. 8, pp. 735-746, Aug. 2002.

[3] S. A. Moskowitz and M. Cooperman, "Method for stega-cipher protection of computer code," US Patent 5,745,569, Assignee: The Dice Company, 1996.

[4] 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.

[5] K. Hattanda and S. Ichikawa, "The evaluation of Davidson's digital signature scheme," *IEICE Transactions*, vol. 87-A, no. 1, pp. 224-225, 2004.

[6] G. Myles, C. Collberg, Z. Heidepriem, and A. Navabi, "The evaluation of two software watermarking algorithm," *Software: Practice and Experience*, vol. 35, no. 10, pp. 923-938, 2005.

[7] G. Qu and M. Potkonjak, "Hiding signatures in graph coloring solutions," in *Proceedings of the 3rd International Workshop on Information Hiding*, 1999, vol. 1768, pp. 348-367.

[8] R. Venkatesan, V. Vazirani, and S. Sinha, "A graph theoretic approach to software watermarking," in *Proceedings of the 4th International Workshop on Information Hiding*, 2001, vol. 2137, pp. 157-168.

[9] C. Collberg, A. Huntwork, E. Carter, and G. Townsend, "Graph theoretic software watermarks: implementation, analysis, and attacks," in *Proceedings of the 6th International Workshop on Information Hiding*, 2004, vol. 3200, pp. 192-207.

[10] C. Collberg, A. Huntwork, E. Carter, G. Townsend, and M. Stepp, "More on graph theoretic software watermarks: implementation, analysis, and attacks," *Inf. Softw. Technol.*, vol. 51, no. 1, pp. 56-57, Jan. 2009.

[11] J. P. Stern, G. Hachez , F. Koeune, and J. J. Quisquater, "Robust object watermarking: application to code," in *Proceedings of the 3rd International Workshop on Information Hiding*, 1999, vol. 1768, pp. 368-378.

[12] C. Collberg and T. R. Sahoo, "Software watermarking in the frequency domain: implementation, analysis, and attacks," *Journal of Computer Security*, vol. 13, no. 5, pp. 721-755, Dec. 2005.

[13] G. Hachez, "A comparative study of software protection tools suited for E-commerce with constributions to software watermarking and smart cards," Ph. D. thesis, Universite Catholique de Louvain, 2003.

[14] C. Collberg, and J. Nagra, *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*, 1st ed. Boston: Addison Wesley, 2009.

[15] C. Collberg and C. Thomborson, "Software watermarking: Models and dynamic embeddings," in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium Principles of Programming Languages*, 1999, pp. 311-324.

[16] C. Collberg, C. Thomborson, and G. Townsend, "Dynamic graph-based software fingerprinting," *ACM Trans. Program. Lang. Syst.*, vol. 29, no. 6, article 35, 67 pages, Oct. 2007.

[17] J. Nagra, "Threading software watermarks," in *Proceedings of the 6th International Workshop on Information Hiding*, 2004, vol. 3200, pp. 208-233.

[18] J. Nagra, "Threading Software Watermarks," Ph. D. thesis, University of Auckland, 2007.

[19] C. Collberg, E. Carter, S. Debray, J. Kececioglu, A. Huntwork, C. Linn, and M. Stepp, "Dynamic path-based software watermarking," in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2004, vol. 39, no. 6, pp. 107-118.

[20] P. Cousot, R. Cousot, "An abstract interpretation-based framework for software watermarking," in *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium Principles of Programming Languages*, 2004, vol. 39, no. 1, pp. 173-185.

[21] F. L. Liu, B. Lu, and X. Y. Luo, "A chaos-based robust software watermarking," in *Proceedings of Information Security Practice and Experience Conference*, 2006, pp. 355-366.

[22] I. Kamel and Q. Albuwi, "A robust software watermarking for copyright protection," *Computer & Security*, vol. 28, no. 6, pp. 395-409, Sep. 2009.

[23] I. J. Cox, J. Kilian, F. T. Leighton, and T. Shamoon, "Secure spread spectrum watermarking for multimedia," *IEEE Transactions on Image Processing*, vol. 6, no. 12, pp. 1673-1687, Dec. 1997.

[24] A. Monden, A. Monsifrot, and C. Thomborson, "A framework for obfuscated interpretation," in *Proceedings of Australasian Information Security Workshop*, 2004, vol. 32, pp. 7-16.

[25] X. S. Zhang, F. L. He, and W. L. Zuo, "A framework for mobile phone java software protection," in *Proceedings of International Conference on Convergence and Hybrid Information Technology*, 2008, vol. 2, pp. 527-532

[26] Y. Zeng, F. L. Liu, X. Y. Luo, C. F. Yang, "Robust software watermarking scheme based on obfuscated interpretation," in *Proceedings of the International Conference on Multimedia Information Networking and Security*, 2010, pp. 671-675.

Ying Zeng received her B.S. and M.S. from the Zhengzhou Institute of Information Science and Technology, in 2004 and 2007, respectively. Now, she is a doctoral candidate of Computer Applications of Zhengzhou Information Science and Technology Institute. Her current research interest is software watermarking, code obfuscation and software birthmarking technique.

Fenlin Liu received his B.S. from Zhengzhou Institute of Information Science and Technology in 1986, M.S. from Harbin Institute of Technology in 1992, and Ph.D. from the Northeast University in 1998. Now, he is a professor of Zhengzhou Institute of Information Science and Technology. His research interests include information hiding and security theory. He is the author or coauthor of more than 90 refereed international journal and conference papers. He obtained the support of the National Natural Science Foundation of China and the Fund of Innovation Scientists and Technicians Outstanding Talents of Henan Province of China.

Xiangyang Luo received his B.S., M.S. and Ph. D. from Zhengzhou Institute of Information Science and Technology, in 2001, 2004 and 2010, respectively. He has been with Zhengzhou Information Science and Technology Institute since July 2004. From 2006 to 2007, he was a visiting scholar of the Department of Computer Science and Technology of Tsinghua University. He is the author or co-author of more than 40 refereed international journal and conference papers. His research interest includes image steganography and steganalysis. He obtained the support of the National Natural Science Foundation of China and the Basic and Frontier Technology Research Program of Henan Province.

Chunfang Yang received his B.S. and M.S. from the Zhengzhou Institute of Information Science and Technology, in 2005 and 2008, respectively. Now, he is a doctoral candidate of Computer Applications of Zhengzhou Information Science and Technology Institute. His current research interest is image steganography and steganalysis.