A DYNAMIC WATERMARKING MODEL FOR EMBEDDING REDUCIBLE PERMUTATION GRAPHS INTO SOFTWARE

Ioannis Chionis, Maria Chroni, and Stavros D. Nikolopoulos
Department of Computer Science, University of Ioannina, GR-45110 Ioannina, Greece
{*ichionis, mchroni, stavros*}*@cs.uoi.gr*

Keywords: Software watermarking; watermark numbers; self-inverting permutations; reducible permutation graphs; encoding; decoding; graph embedding; call-graphs; control statement; opaque predicates, algorithms.

Abstract: Software watermarking involves embedding a unique identifier or, equivalently, a watermark value, within a software to discourage software theft; towards the embedding process, several graph theoretic watermarking algorithmic techniques encode the watermark values as graph structures and embed them in application programs. Recently, we presented an efficient codec system for encoding a watermark number w as a reducible permutation graph $F[\pi^*]$ through the use of self-inverting permutations π^*. In this paper, we propose a dynamic watermarking model for embedding the watermark graph $F[\pi^*]$ into an application program P. The main idea behind the proposed watermarking model is a systematic use of appropriate calls of specific functions of the program P. More precisely, our model uses the dynamic call-graph $G(P, I_{key})$ of the program P, taken by the specific input I_{key}, and the graph $F[\pi^*]$, and produces the watermarked program P^* having the following key property: its dynamic call-graph $G(P^*, I_{key})$ and the reducible permutation graph $F[\pi^*]$ are isomorphic graphs. Within this idea the program P^* is produced by only altering appropriate real-calls of specific functions of the input program P. Moreover, the proposed watermarking model incorporates such properties which cause it resilient to attacks.

1 INTRODUCTION

The rapid growth of World Wide Web users, the ease of distributing fast and in the original form digital content through internet, as well as the lack of technical measures to assure the intellectual property right of owners, has led to an incensement in copyright infringement. Digital watermarking is a technique for protecting the intellectual property of any digital content. The idea of digital watermarking is the embedding of a unique identifier into the digital image, audio, or video data, software and text through the introduction of errors not detectable by human perception (Cox et al., 1996).

According to the recent Business Software Alliance (BSA) global software piracy study (BSA, 2011) over half of the worlds personal computer users – 57 percent – admit they pirate software. What is more, as the price of hardware drops and the price of licensed software goes up, piracy becomes more popular and lucrative. This fact has led to a more systematic work on protecting the intellectual property as can be seen from a recent research of World Intellectual Property Organization (WIPO) where there is a growth of intellectual property filings (WIPO, 2012).

Software Watermarking. Although digital watermarking has made considerable progress and become a popular technique for copyright protection of multimedia information (Cox et al., 1996), research on software watermarking has recently received sufficient attention. The patent by Davidson and Myhrvold (Davidson and Myhrvold, 1996) presented the first published software watermarking algorithm, where and other patents have been published lately (Rodriguez et al., 2010; Collberg et al., 2011; Horne et al., 2012). The major software watermarking algorithms currently available are based on several techniques, among which the register allocation (Qu and Potkonjak, 1998), spread-spectrum (Zhang et al., 2011), opaque predicate (Arboit, 2002), abstract interpretation (Cousot and Cousot, 2004), dynamic path techniques (Collberg et al., 2004), code re-orderings (Sharma et al., 2011).

The *software watermarking problem* can be described as the problem of embedding a structure w into a program P and, thus, producing a new pro-

gram P_w, such that w can be reliably located and extracted from P_w even after P_w has been subjected to code transformations such as translation, optimization and obfuscation (Myles and Collberg, 2006). More precisely, given a program P, a watermark w, and a key k, the software watermarking problem can be formally described by the following two functions: $\text{embed}(P,w,k) \rightarrow P_w$ and $\text{extract}(P_w,k) \rightarrow w$.

There are two main categories of watermarking algorithms namely *static* and *dynamic* algorithms (Collberg and Thomborson, 1999). A static watermark is stored inside program code in a certain format, and it does not change during the program execution. A dynamic watermark is built during program execution, perhaps only after a particular sequence of input. It might be retrieved by analyzing the data structures built when watermarked program is running. In other cases, tracing the program execution may be required.Further discussion of static and/or dynamic watermarking issues can be found in (Davidson and Myhrvold, 1996; Venkatesan et al., 2001).

Graph-based Codecs and Attacks. Recently, several software watermarking algorithms have been appeared in the literature that encode watermarks as graph structures. In general, such encodings make use of an encoding function *encode* which converts a watermarking number w into a graph G, $encode(w) \rightarrow G$, and also of a decoding function *decode* that converts the graph G into the number w, $decode(G) \rightarrow w$; we usually call the pair $(\text{e}ncode, \text{d}ecode)_G$ as *graph codec system* (Collberg et al., 2003). From a graph-theoretic point of view, we are looking for a class of graphs $\mathcal{G}$ and a corresponding codec $(\text{e}ncode, \text{d}ecode)_{\mathcal{G}}$ with the following properties which cause them resilience to attacks:

- Appropriate graph types: Graphs in $\mathcal{G}$ should be directed having such properties, i.e., nodes with small outdegree, so that matching real program graphs;
- High resiliency: The function $decode(G)$ should be insensitive to small changes of G; that is, if $G \in \mathcal{G}$ and $decode(G) \rightarrow w$ then $decode(G') \rightarrow w$ with $G' \approx G$;
- Small size: The size $|P_w| - |P|$ of the embedded watermark should be small;
- Efficient codecs: The functions *encode* and *decode* should be computed in polynomial time.

Related Work. In 1996, Davidson and Myhrvold (Davidson and Myhrvold, 1996) proposed the first software watermarking algorithm which is static and embeds the watermark by reordering the basic blocks of a control flow-graph; note that a static watermark is stored inside programs' code in a certain format and it does not change during the programs' execution. Based on this idea, Venkatesan, Vazirani and Sinha (Venkatesan et al., 2001) proposed the first graph-based software watermarking algorithm which embeds the watermark by extending a method's control flow-graph through the insertion of a directed subgraph; it is also a static algorithm called VVS or GTW. Collberg et al. (Collberg et al., 2009) proposed an implementation of GTW, which they call GTW_{sm}, and it is the first publicly available implementation of the algorithm GTW. Note that, for encoding integers the GTW_{sm} method uses only those permutations that are self-inverting. The first dynamic watermarking algorithm (CT) was proposed by Collberg and Thomborson (Collberg and Thomborson, 1999); it embeds the watermark through a graph structure which is built on a heap at runtime.

Recently, the authors of this paper (Chroni and Nikolopoulos, 2010; Chroni and Nikolopoulos, 2011) extended the class of software watermarking algorithms and graph structures by proposing an efficient and easily implemented codec system for encoding watermark numbers as reducible permutation flow-graphs; see also (Chroni and Nikolopoulos, 2012).

Our Contribution. Recently, we presented an efficient method for encoding integers as self-inverting permutations (or, for short, SiP) and algorithms for encoding a self-inverting permutation π^* into a reducible permutation flow-graph $F[\pi^*]$ (or, for short, RPG) (Chroni and Nikolopoulos, 2010; Chroni and Nikolopoulos, 2011); the graph $F[\pi^*]$ incorporates properties capable to match real program graphs, that is, it does not differ from the graph data structures built by real programs since its maximum outdegree does not exceed two and it has a unique root node so the program can reach other nodes from the root node.

In this paper, we propose a dynamic watermarking model for embedding the watermark graph $F[\pi^*]$ into an application program P. The main idea behind the proposed watermarking model is a systematic use of appropriate calls of specific functions of the program P. More precisely, our model uses the dynamic call-graph $G(P, I_{key})$ of the program P, taken by the specific input I_{key}, and the graph $F[\pi^*]$, and produces the watermarked program P^* having the following key property: its dynamic call-graph $G(P^*, I_{key})$ and the reducible permutation graph $F[\pi^*]$ are isomorphic graphs. Within this idea the program P^* is produced by only altering appropriate real-calls of specific functions of the input program P. Moreover, the proposed watermarking model incorporates such properties which cause it resilient to attacks.

2 BACKGROUND RESULTS

In this section, we present basic components and background results that are used in the design of our watermarking model. We also briefly discuss properties of dynamic call-graphs which are used as key-objects in our watermarking model for embedding the graph $F[\pi^*]$ into an application program.

2.1 Encode Numbers as RPGs

We consider finite graphs with no multiple edges. For a graph G, we denote by $V(G)$ and $E(G)$ the vertex set and edge set of G, respectively. We also consider permutations over the set $N_n = \{1,2,\ldots,n\}$.

A. Self-inverting Permutation (SiP)

Let π be a permutation over the set N_n. We think of permutation π as a sequence $(\pi_1,\pi_2,\ldots,\pi_n)$, so, for example, the permutation $\pi = (1,4,2,7,5,3,6)$ has $\pi_1 = 1$, $\pi_2 = 4$, etc. Notice that π_i^{-1} is the position in the sequence of the number i.

Definition 2.1. Let $\pi = (\pi_1,\pi_2,\ldots,\pi_n)$ be a permutation over the set N_n. The inverse of π is the permutation $\tau = (\tau_1,\tau_2,\ldots,\tau_n)$ with $\tau_{\pi_i} = \pi_{\tau_i} = i$. A *self-inverting permutation* (or, involution) is a permutation that is its own inverse: $\pi_{\pi_i} = i$.

Notation 2.1. Throughout the paper we denote a self-inverting permutation π over the set N_n as π^*.

B. Reducible Permutation Graphs (RPG)

A flow-graph is a directed graph F with an initial node s from which all other nodes are reachable. A directed graph G is strongly connected when there is a path $x \to y$ for all nodes x, y in $V(G)$. A node u is an *entry* for a subgraph H of the graph G when there is a path $p = (y_1,y_2,\ldots,y_k,x)$ such that $p \cap H = \{x\}$.

Definition 2.2. A flow-graph is reducible when it does not have a strongly connected subgraph with two (or more) entries.

There are at least three other equivalent definitions, as Theorem 2.1 shows. Those definitions use a few more graph-theoretic concepts.

Theorem 2.1. (Hecht and Ullman, 1972; Hecht and Ullman, 1974): Let F be a flow-graph. The following three statements are equivalent:

(1) the graph F is reducible;

(2) the graph F has a unique DFS dag;

(3) the graph F can be transformed into a single node by repeated application of the transformations T_1 and T_2, where T_1 removes a cycle-edge, and T_2 picks a non-initial node y that has only one incoming edge and glue nodes x and y.

The reducible permutation graph $F[\pi^*]$ is directed with descending ordering on its nodes $V(G) = \{s = u_{n+1},u_n,\ldots,u_1,u_0 = t\}$. Throughout the paper, we shall call the edge (u_i,u_j) *forward* if $i > j$ while we shall call (u_i,u_j) *backward* if $i < j$.

C. Codec Algorithms

In (Chroni and Nikolopoulos, 2010) we introduced the notion of *bitonic permutations* and we presented two algorithms, namely Encode_W.to.SiP and Decode_SiP.to.W, for encoding an integer w into an self-inverting permutation π^* and extracting it from π^*; see also (authors paper, 2011) (Chroni and Nikolopoulos, 2011). We have actually proved the following results.

Theorem 2.2. *Let w be an integer and let $b_1b_2\cdots b_n$ be the binary representation of w. The algorithm Encode_W.to.SiP encodes the number w in a self-inverting permutation π^* of length $2n+1$ in $O(n)$ time and space.*

Theorem 2.3. *Let π^* be a self-inverting permutation of length n which encodes an integer w using the algorithm Encode_W.to.SiP. The algorithm Decode_SiP.to.W correctly decodes the permutation π^* in $O(n)$ time and space.*

Recently, we have presented an efficient and easily implemented algorithm for encoding numbers as reducible permutation flow-graphs through the use of self-inverting permutations (Chroni and Nikolopoulos, 2012).

In particular, we have proposed the algorithm `Encode_SiP.to.RPG`, which encodes the self-inverting permutation π^* as a reducible permutation flow-graph $F[\pi^*]$ by exploiting domination relations on the elements of π^* and using an efficient DAG representation of π^*. We also proposed the decoding algorithm `Decode_RPG.to.SIP`, which extract the self-inverting permutation π^* from $F[\pi^*]$ by converting first the graph $F[\pi^*]$ into a directed tree $T[\pi^*]$ and then applying DFS-search on $T[\pi^*]$.

The whole encoding process takes $O(n)$ time and requires $O(n)$ space, where n is the length of the permutation π^*. The decoding process takes time and space linear in the size of the flow-graph $F[\pi^*]$, that is, the algorithm `Decode_RPG.to.SIP` takes $O(n)$ time and space. Our results presented in (authors' paper,

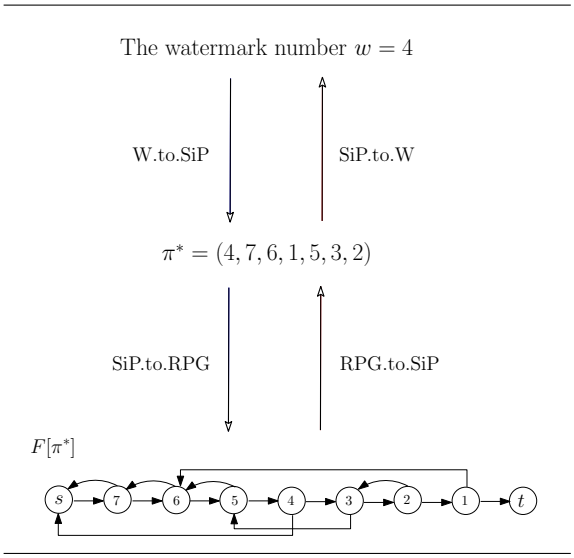

Figure 1: The main data components used by our codec algorithms (i.e., watermark w, SiP π^*, and RPG $F[\pi^*]$) and a flow of the process of encoding a watermark number w into the graph $F[\pi^*]$ and extracting it from $F[\pi^*]$.

2012) are summarized in the following theorems.

Theorem 2.4. *Let π^* be a self-inverting permutation over the set N_n. The algorithm Encode_SiP.to.RPG encodes the permutation π^* into a reducible permutation graph $F[\pi^*]$ in $O(n)$ time and space.*

Theorem 2.5. *Let $F[\pi^*]$ be a reducible permutation graph of order $O(n)$ produced by the algorithm Encode_SiP.to.RPG. The algorithm Encode_RPG.to.SiP correctly extracts the permutation π^* from $F[\pi^*]$ in $O(n)$ time and space.*

Figure 1 depicts the main data components used by our codec algorithms, i.e., the watermark number w, the SiP π^*, and the RPG $F[\pi^*]$. The same figure shows a flow of the process of encoding a watermark number w into the graph $F[\pi^*]$ and extracting it from $F[\pi^*]$ through the use of self-inverting permutations.

2.2 Dynamic Call-graphs

A *call-graph* is a directed graph that represents calling relationships between program units in a computer program. Specifically, the nodes of a call-graph represent functions, procedures, classes, or similar program units and each edge (f_i, f_j) indicates that function f_i calls function f_j; function f_i is called *caller* and function f_j is called *callee*.

Call-graphs can be divided in two main classes of graphs, namely *static* and *dynamic*.

A static call-graph is the structure describing those invocations that could be made from one program unit to another in any possible execution of the program (Xie and Notkin, 2002). The static call-graph can be determined from the program source code; we mention that, its construction is a time consuming process specifically in the case of large scale softwares.

A dynamic call-graph G is a directed graph that includes invocations of caller–callee pairs, over an execution of the program P. A dynamic call-graph can be considered as an instance of the corresponding static call-graph for a specific input sequence I. The call-graph G is the key data structure that dynamic optimizers use to analyze and optimize the whole-program's behavior. Such a graph can be extracted by a profiler. It is fair to mention that the construction of a dynamic call-graph G of a program P is not a time consuming process even if P is a large scale software.

Throughout the paper we denote a call-graph G of the program P over the input I as $G(P,I)$. Figures 2(a) depicts the structure of the dynamic call-graph $G(P,I_{key})$ of an application program P with input I_{key}.

3 THE DYNAMIC WATERMARKING MODEL

Having encoded a watermark number w as reducible permutation graph $F[\pi^*]$, let us now propose a dynamic watermarking model based on which we can efficiently watermark an application program P by embedding the graph $F[\pi^*]$ into P producing thus the watermarked program P^*.

The main idea behind the proposed dynamic watermarking model is the use of the dynamic call-graph $G(P,I_{key})$ of the program P, taken by the specific input I_{key}, and the graph $F[\pi^*]$ in order to produce the watermarked program P^* having the following key property: *its dynamic call-graph $G(P^*,I_{key})$ and the reducible permutation graph $F[\pi^*]$ are isomorphic graphs*. Within this idea the program P^* is produced by only altering appropriate real-calls of specific functions of the input program P.

Figure 2 shows the dynamic call-graph $G(P,I_{key})$ of an application program P, the reducible permutation graph $F[\pi^*]$ which encodes the number $w = 4$ and the dynamic call-graph $G(P^*,I_{key})$ of the watermarked program P^*.

Next, we first describe the data and operational components used by the model and, then, we present the embedding/extracting watermarking algorithms.

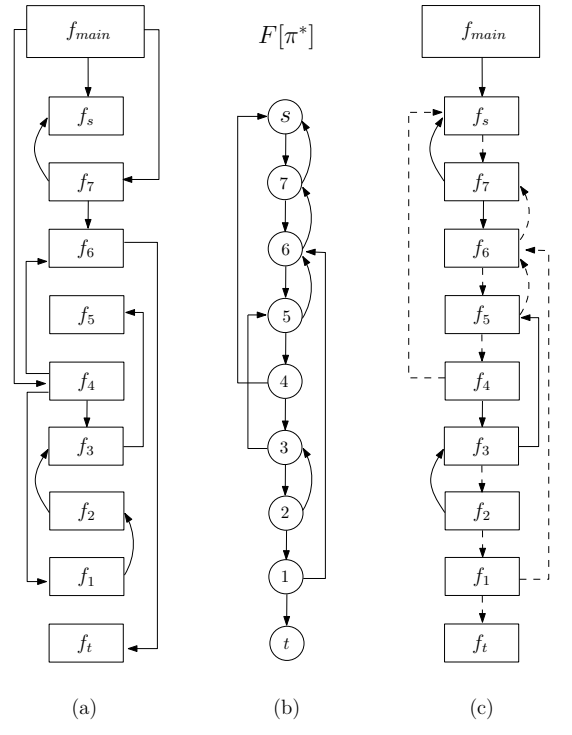

Figure 2: (a) The dynamic call-graph $G(P,I_{key})$ of an application program P. (b) The reducible permutation graph $F[\pi^*]$. (c) The dynamic call-graph $G(P^*,I_{key})$ of the watermarked program P^*.

3.1 Model Components

Our watermarking model uses two main categories of components namely *data* components and *operational* components. The first category includes the dynamic call-graph $G(P,I_{key})$ of the input program P, the watermark graph $F[\pi^*]$, and the dynamic call-graph $G(P^*,I_{key})$ of the watermarked program P^*, while the second category includes call patterns and control statements which are components related to the process of embedding the graph $F[\pi^*]$ into application program P.

We next describe the construction and main properties of the dynamic call-graph $G(P^*,I_{key})$, two call patterns based on which we correspond edges of the call-graph $G(P^*,I_{key})$ to function calls, and specific variables and statements which control the execution of real and water functions.

(I) The Dynamic Call-graph $G(P^*,I_{key})$

Let $F[\pi^*]$ be a watermark-graph on $n+2$ nodes and $G(P,I_{key})$ be the dynamic call-graph of a program P on $n+3$ nodes $f_{main},f_s,f_1,\ldots,f_n,f_t$ taken after running the program P with the input I_{key}. In general, the selection of the input I_{key} is such that it produces the call-graph $G(P,I_{key})$ having structure as "close" as possible to the structure of $F[\pi^*]$. We assign the $n+2$ nodes $f_s=f_{n+1},f_n,\ldots,f_1,f_0=f_t$ of the call-graph $G(P,I_{key})$ to $n+2$ nodes $s=u_{n+1},u_n,\ldots,u_1,u_0=t$ of $F[\pi^*]$ into 1-1 correspondence; the main function f_{main} do not correspond to any node of $F[\pi^*]$.

Let (u_i,u_j) be an edge in graph $F[\pi^*]$ and let (f_i,f_j) be an edge in call-graph $G(P,I_{key})$. We say that the edge (f_i,f_j) corresponds to edge (u_i,u_j) iff the node f_i corresponds to u_i and the node f_j corresponds to u_j, $0\le i,j\le n+1$. Moreover, if (u_i,u_j) is a forward (resp. backward) edge in the graph $F[\pi^*]$ we say that the corresponding edge (f_i,f_j) in graph $G(P,I_{key})$ is a forward (resp. backward) edge.

The dynamic call-graph $G(P^*,I_{key})$ is constructed as follows:

- $V(G(P^*,I_{key}))=V(G(P,I_{key}))$, i.e., it has the same nodes as the call-graph $G(P,I_{key})$;
- $E(G(P^*,I_{key}))=E(F[\pi^*])$, i.e., (f_i,f_j) is an edge in $E(G(P^*,I_{key}))$ iff the corresponding (u_i,u_j) is an edge in $F[\pi^*]$.

The edges of the call-graph $G(P^*,I_{key})$ are divided into two categories namely *real* and *water* edges; note that, the real (resp. water) edges are corresponded to real (resp. water) function calls. An edge (f_i,f_j) of the call-graph $G(P,I_{key})$ is characterized as either

- *real edge* if (f_i,f_j) is an edge in $G(P,I_{key})$, or
- *water edge* if (f_i,f_j) is not an edge in $G(P,I_{key})$.

Figure 2 shows the dynamic call-graph $G(P^*,I_{key})$ along with its real edges (solid arrows) and water edges (dashed arrows); it also depicts the dynamic call-graph $G(P,I_{key})$ and the watermark-graph $F[\pi^*]$.

(II) Call Patterns

In the implementation phase, we modify the source code of program P using specific function call patterns which we describe below.

Let P be an application program, $G(P,I_{key})$ be the dynamic call-graph of the program P with input I_{key}, and $F[\pi^*]$ be a watermark-graph which we have to embed into P. According to our watermarking model, the embedding process relies mainly on altering the execution-flow of appropriate function calls of P such that the execution of the resulting program P^* with the input I_{key} produces a call-graph $G(P^*,I_{key})$ which, after removing the node f_{main}, is isomorphic to watermark-graph $F[\pi^*]$.

Let (f_i,f_j) be an edge of call-graph $G(P^*,I_{key})$ or, equivalently, an edge which we want to appear in

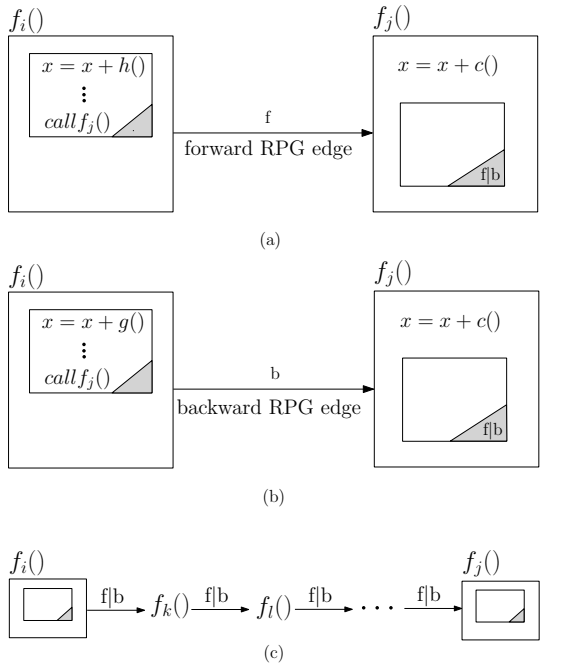

Figure 3: (a) The forward call pattern *f-call*; (b) The backward call pattern *b-call*; (c) The path call pattern *p-call*.

$G(P^*, I_{key})$. Since $G(P^*, I_{key})$ has two types of edges it follows that (f_i, f_j) is either real or water edge. Based on the type of (f_i, f_j), we do the following:

- if (f_i, f_j) is a water edge we add the statement `call(fj)` in the function f_i, while
- if (f_i, f_j) is a real edge we add no call statement since the statement `call`(f_j) exists in f_i.

Based on whether (f_i, f_j) is either a forward or a backward edge we add specific statements in functions f_i and f_j according to the following two call patterns namely *forward* and *backward* call patterns:

(a) if (f_i, f_j) is a forward edge we add the statement $x = x + h()$ in function f_i before the call-site or, equivalently, call-point of the function f_j, and the statement $x = x + c()$ in the function f_j, while

(b) if (f_i, f_j) is a backward edge we add the statement $x = x + g()$ in function f_i before the call-site of the function f_j, and the statement $x = x + c()$ in the function f_j,

where x is a variable of type `A` and $h()$, $g()$ and $c()$ functions which returns values of type `A`. Figure 3(a) depicts the forward call pattern or, for short, *f-call*, while Figure 3(b) depicts the backward call pattern or, for short, *b-call*.

Recall that the direct edge (f_i, f_j) of a call-graph represents a function call operation where f_i is the caller function and f_j the callee function; in other words, it means that in function f_i there exists the statement `call`(f_j). Hereafter, in this case we shall say that (f_i, f_j) is a direct call.

In a call-graph of an application program we usually meet sequences of calls of the form $(f_i, f_{k_1}, f_{k_2}, \ldots, f_{k_m}, f_j)$. For simplicity we set $f_i = f_{k_0}$ and $f_j = f_{k_{m+1}}$ and suppose that each of these calls (f_{k_0}, f_{k_1}), (f_{k_1}, f_{k_2}), …, (f_{k_m}, f_{m+1}) is either forward or backward. We extend the notion of the direct call (f_i, f_j) to indirect call $(f_i \to f_j)$; an indirect call consists of a path of functions $(f_i, f_{k_1}, \ldots, f_j)$ of length $\ell \geq 2$. Using the f-call and b-call patterns, we next define the path call pattern or, for short, *p-call* as follows:

(c) if $(f_{k_i}, f_{k_{i+1}})$ and $(f_{k_{i+1}}, f_{k_{i+2}})$ are two consecutive calls of a call sequence, we apply an f-call or a b-call in $(f_{k_{i+1}}, f_{k_{i+2}})$ by first adding the statement $x = x + h()$ or the statement $x = x + g()$ in the function $f_{k_{i+1}}$ after the call-point of statement $x = x + c()$, and then adding the statement $x = x + c()$ in the function $f_{k_{i+2}}$, $0 \leq i \leq m-1$.

Figure 3 shows the structures of the patterns of an f-call and a b-call of the direct call (f_i, f_j), and the structure of a p-call of an indirect call $(f_i \to f_j)$.

Note that an indirect call $(f_i \to f_j)$ consisting of a path of functions $(f_i, f_{k_1}, \ldots, f_j)$ of length ℓ can be considered as a sequence of ℓ direct calls.

(III) Control Statements

In any watermarking model both the original program P and the watermarked program P^* have to operate identically, that is, the output $O(P, I)$ of the program P must be the same with the output $O(P^*, I)$ of the program P^* for every input I. Thus, since the call-graphs $G(P, I_{key})$ and $G(P^*, I_{key})$ dictate the execution flow of the original program P and the watermarked program P^*, respectively, and since the call-graph $G(P, I_{key})$ is not isomorphic to $G(P^*, I_{key})$ we have to control the flow of selected function calls of P^* so that $O(P, I) = O(P^*, I)$ for every input I.

To do this, we exploit the values of specific variables in a function f_i by using them in some selected or added control statements as part of opaque predicates. More precisely, in our watermarking model we use the values of the variable x of the f-call and b-call patterns and include it in a specific control statement s causing thus an "appropriate execution flow" of the functions of the call-graph $G(P^*, I_{key})$; with the term "appropriate execution flow" we mean that the execution flow of the functions of the call-graph $G(P^*, I_{key})$

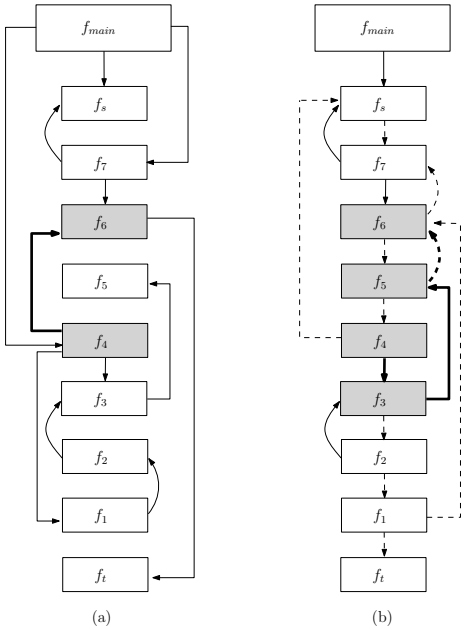

Figure 4: (a) The real-call (f_4, f_6) in the call-graph $G(P, I_{key})$ of a program P; bold arrow. (b) The corresponding path-call (f_4, f_3, f_5, f_6) in the call-graph $G(P^*, I_{key})$ of the watermarked program P^*; bold arrows.

is such that $O(P,I) = O(P^*,I)$ for every input I.

Hereafter, we shall call *cf-statement* the control statement s since it controls the execution flow of the functions of $G(P^*, I_{key})$ and, in an analogous way, we shall call *cf-variable* the variable x of the f-call and b-call patterns.

We next describe the mechanism which ensures an appropriate execution flow of the functions of $G(P^*, I_{key})$ through the altering of the execution flow of the functions of the program P by modifying or adding some specific control statements. In fact, what the mechanism actually does is to modify the conditions or expressions of these control statements by adding opaque predicates.

Definition 3.1. A predicate Q is opaque at a program point p, if at point p the outcome of Q is known at embedding time. If Q always evaluates to *true* we write Q_p^T, for *false* we write Q_p^F, and if Q sometimes evaluates to *true* and sometimes to *false* we write $Q_p^?$.

Let (f_i, f_j) be a direct call in our program P^* or, equivalently, an edge in the call-graph $G(P^*, I_{key})$; it is either real, water, forward, or backward edge. In any case, the proposed mechanism uses the value of the variable x of the f-call or b-call pattern of (f_i, f_j) and does the following:

- In function f_i: create a control statement (if, switch, for, while, etc), add an opaque predicate $Q_p^?$ with respect to cf-variable x in the condition of the control statement, and insert it at a point p before the statement $x = x + h()$ or $x = x + g()$; we can also select an existing control statement at a point p, consider it as cf-statement, and include in its condition part the opaque predicate $Q_p^?$.
- In function f_j: create a control statement (if, switch, for, while, etc), add an opaque predicate $Q_p^?$ with respect to cf-variable x in the condition of the control statement, and insert it at a point p before the statement $x = x + c()$; the main body of f_j is included in a block of a cf-statement the execution of which is depending upon the truthness or falsity of the opaque predicate $Q_p^?$. If f_j is called by another function f_k, that is, if (f_k, f_j) is an edge in the call-graph $G(P^*, I_{key})$, we do not create a new control statement but we use the previous one and add or appropriately modify its existing opaque predicate $Q_p^?$.

Table 1 shows an example of the modification of the condition part of an `if` cf-statement via an opaque predicate; since (f_i, f_j) is a water and forward function call, the statement `call`(f_j) does not exist in function f_i, and thus we add it in f_i, while the cf-statement is the $x = x + h()$. On the other hand, Table 2 shows an example in this case where (f_i, f_j) is a real and backward function call. In this case, the statement `call`(f_j) does exist in f_i while the cf-statement is the $x = x + g()$. Table 3 shows an example of the modification of the function f_j in the case where (f_i, f_j) is a water and forward function call.

Remark 3.1. Based on the structural properties of the watermark graph $F[\pi^*]$ and call-graph $G(P^*, I_{key})$ we can easily prove the following lemma.

Lemma 3.1. *Let $G(P, I_{key})$ and $G(P^*, I_{key})$ be the call-graphs of programs P and P^*, respectively, on input I_{key}, and let (f_i, f_j) be an edge in call-graph $G(P, I_{key})$. Then, there always exists an edge (f_i, f_j) or a path $(f_i, f_{k_1}, f_{k_2}, \ldots, f_{k_m}, f_j)$ in call-graph $G(P^*, I_{key})$.*

Remark 3.2. In our implementation, in the case where (f_i, f_j) is an edge in $G(P, I_{key})$ and (f_i, f_j) is not an edge in $G(P^*, I_{key})$ we have to compute a path $(f_i, f_{k_1}, \ldots, f_j)$ of function calls in $G(P^*, I_{key})$. Such a path is a shortest path from f_i to f_j in the graph $G(P^*, I_{key})$; it may consist of all types of edges, that is, real or water edges and forward or backward edges. Figure 4(a) shows the edge (f_4, f_6) in $G(P, I_{key})$ which is not an edge in $G(P^*, I_{key})$, while Figure 4(b) shows its corresponding shortest path from f_i to f_j, that is, the path (f_4, f_3, f_5, f_6); note that, (f_4, f_3) is a real and

Program P	Program P^*

function $f_i()$
...
 	
if (*condition)*
...
statements;
...
	function $f_i()$
	...
	if (*condition* & $Q_p^?$*)*
	...
	x = x + h();
	...
	call $f_j()$;
	...
	statements;
	...

Table 1: An example of cf-statement modification via opaque predicates in the case where (f_i, f_j) is a water and forward function call.

Program P	Program P^*

function $f_i()$
...
 	
if (*condition)*
...
call $f_j()$;
...
statements;
...
	function $f_i()$
	...
	if (*condition* & $Q_p^?$*)*
	...
	x = x + g();
	...
	call $f_j()$;
	...
	statements;
	...

Table 2: An example of cf-statement modification via opaque predicates in the case where (f_i, f_j) is a real and backwards function call.

Caller of (f_i, f_j)	Callee of (f_i, f_j)

function $f_i()$
...
if (*condition* & $Q_p^?$*)*
...
x = x + h();
...
call $f_j()$;
...
statements;
...
	function $f_j()$
	...
	if (*condition* & $Q_p^?$*)*
	...
	x = x + c();
	...
	if (*condition* & $Q_p^?$*)*
	...
	statements;
	...

Table 3: An example of cf-statement modification via opaque predicates of the function f_j in the case where (f_i, f_j) is a water and forward function call.

forward edge, (f_3, f_5) is a real and backward edge, and (f_5, f_6) is a water and backward edge.

(VI) Execution Rules

We present the rules based on which we control the execution flow of the functions of P^* such that $O(P,I) = O(P^*,I)$ for every input I. In fact, we show in all the cases how the value of $Q_p^?$ dictates the execution flow of functions of $G(P^*, I_{key})$.

Let (f_i, f_j) be a direct call in our program P^* or, equivalently, an edge in the call-graph $G(P^*, I_{key})$. We distinguish the following cases:

- Edge (f_i, f_j) is **real** and **forward/backward**: in this case we modify the functions f_i and f_j as follows:
  - Function f_i: the opaque predicate $Q_p^?$ in the cf-statement before the cf-value $x = x + h()$ or $x = x + g()$ and the `call(`f_j`)` is evaluated to true, that is, Q_p^T.
  - Function f_j: the opaque predicate $Q_p^?$ in the cf-statement before the cf-value $x = x + c()$ is evaluated to true, that is, Q_p^T, while the $Q_p^?$ for the cf-statement which controls the statements of the main body of the function f_j is also evaluated to true, that is, Q_p^T.
- Edge (f_i, f_j) is **water** and **forward/backward**: in this case we modify the functions f_i and f_j as follows:
  - Function f_i: the opaque predicate $Q_p^?$ in the cf-statement before the cf-value $x = x + h()$ or $x = x + g()$ and the `call(`f_j`)` is evaluated to true, that is, Q_p^T.
  - Function f_j: the opaque predicate $Q_p^?$ in the cf-statement before the cf-value $x = x + c()$ is evaluated to true, that is, Q_p^T, while the $Q_p^?$ for the cf-statement which controls the statements of the main body of the function f_j is evaluated to false, that is, Q_p^F.

Remark 3.3. At the execution time of the function f_i of the program P^*, only one opaque predicate $Q_p^?$ in the cf-statements is evaluated to true with respect to the current value of the cf-variable x.

3.2 Embedding an RPG into a Code

Having described the main properties and operations of our dynamic watermarking model, let us now present the algorithm which efficiently watermarks an application program P by embedding the reducible permutation graph $F[\pi^*]$ into an application program P producing thus the watermarked program P^*.

We next present in detail the proposed embedding algorithm, namely `Encode_RPG.to.CODE`, which consists of the following steps:

Embedding Algorithm `Encode_RPG.to.CODE`

1. Take as input the source code of the program P, select an input I_{key}, and construct the call-graph $G(P,I_{key})$; let $f_{main}, f_s, f_n, \ldots, f_1, f_t$ be the functions of call-graph $G(P,I_{key})$; then, construct a watermark graph $F[\pi^*]$ on $n+2$ nodes and let $s = u_{n+1}, u_n, \ldots, u_1, u_0 = t$ be the nodes of $F[\pi^*]$;

2. Remove the node f_{main} from $G(P,I_{key})$ and assign an exact pairing, i.e., one-to-one and onto mapping, of the $n+2$ nodes of $G(P,I_{key})$ to the nodes of $F[\pi^*]$; let $f_i \rightarrow u_i$, $0 \leq i \leq n+1$;

3. Construct the call-graph $G(P^*,I_{key})$ of the watermarked program P^* as follows:
   - $V(G(P^*,I_{key})) = V(G(P,I_{key}))$, i.e., it has the same nodes as the call-graph $G(P,I_{key})$;
   - $E(G(P^*,I_{key})) = E(F[\pi^*])$, i.e., (f_i,f_j) is an edge in $E(G(P^*,I_{key}))$ iff the corresponding (u_i,u_j) is an edge in graph $F[\pi^*]$;

4. Create a call-table $\mathcal{T}$ of dimension $(m \times 3)$, where m is the number of function calls (f_i,f_j) which appear in the execution trace of P with input I_{key}; the 1st column stores the caller functions f_i, the 2nd column stores the callee functions f_j, while the 3rd one stores the invocations of (f_i,f_j);

5. Initially insert into table $\mathcal{T}$ the edges (f_i,f_j) of the call-graph $G(P^*,I_{key})$; then, if (f_i,f_j) does not exist in $G(P,I_{key})$, compute the shortest path $(f_i,f_{k_1},f_{k_2},\ldots,f_{k_m},f_j)$ from node f_i to node f_j in graph $G(P^*,I_{key})$ and insert the edges (f_i,f_{k_1}), (f_{k_1},f_{k_2}), …, (f_{k_m},f_j) into table $\mathcal{T}$; the edges are inserted into table $\mathcal{T}$ in a specific order;

6. Characterize the edges (f_i,f_j) in table $\mathcal{T}$ as either real, water, forward, or backward;

7. For each water edge (f_i,f_j) of the graph $G(P^*,I_{key})$, add the statement $\texttt{call}(f_j)$ in a call-point in the function f_i of the program P;

8. For each edge (f_i,f_j) in the table $\mathcal{T}$, insert the cf-variable x in both functions f_i and f_j by adding the statements $x = x + h()$, $x = x + g()$, or $x = x + c()$ according to f-call and b-call patterns; see, subsection (II) Call Patterns;

9. For each edge (f_i,f_j) in the table $\mathcal{T}$, insert cf-statements which control:
   - the values of the cf-variables x,
   - the function calls $\texttt{call}(f_j)$, and
   - the values of the cf-variables x;

   see, subsection (III) Control Statements;

10. Return the source code of the program P^*;

Remark 3.4. In Step 5 of the embedding algorithm, the edges (f_i,f_j) are included into the table $\mathcal{T}$ in a specific order. This order is determined by the order they appeared in the execution trace of program P with input I_{key}, that is, if call (f_i,f_j) appears before the call (f_k,f_ℓ) in execution trace of P, the edge (f_i,f_j) appears before the edge (f_k,f_ℓ) in table $\mathcal{T}$.

Remark 3.5. Let (f_i,f_j) be an edge which is handled in Step 8 of the embedding algorithm and let the statement $\texttt{call}(f_j)$ appear more that once in function f_i. We point out that in this case we insert both the cf-variable and cf-statement before the call-site of each statement $\texttt{call}(f_j)$ in function f_i.

3.3 Extracting the Watermark RPG from the Code

In this section, we present an algorithm for extracting the graph $F[\pi^*]$ from the program P^* watermarked by the embedding algorithm `Encode_RPG.to.CODE`. The proposed extracting algorithm works as follows:

Extracting Algorithm `Decode_CODE.to.RPG`

1. Take the program P^* watermarked by the embedding algorithm `Encode_RPG.to.CODE` and run it with input I_{key};

2. Construct the call-table $\mathcal{T}$ using the execution trace of the program P^* with input I_{key};

3. Construct the dynamic call-graph $G(P^*,I_{key})$ using the call-table $\mathcal{T}$ as follows:
   - search the table $\mathcal{T}$ and add all the different functions f_i in the set V, where $0 \leq i \leq n+2$,
   - search the table $\mathcal{T}$ row-by-row and select all the different pairs (f_i,f_j), where f_i and f_j belong to the 1st and 2nd columns of the same row of $\mathcal{T}$, respectively; add the selected pairs (f_i,f_j) in the set E;
   - take the vertex set $V(G(P^*,I_{key})) = V$ and the edge set $E(G(P^*,I_{key})) = E$ of the graph $G(P^*,I_{key})$;

4. Remove the node f_{main} from $G(P^*,I_{key})$ resulting the graph $G'(P^*,I_{key})$;

5. Construct a graph $F[P^*]$ isomorphic to $G'(P^*,I_{key})$ and then compute the unique Hamiltonian path HP of the graph $F[P^*]$; let

$$HP = (u_{n+1}, u_n, u_{n-1}, \ldots, u_1, u_0)$$

   be the Hamiltonian path of $F[P^*]$;

6. Relabeling the nodes of the graph $F[P^*]$ according to their order in the HP resulting thus the graph $F[\pi^*]$ with nodes $s = u_{n+1}, u_n, \ldots, u_0 = t$;
7. Return the graph $F[\pi^*]$;

Remark 3.6. In Step 5 of the extracting algorithm, we compute the unique Hamiltonian path of the graph $F[P^*]$. Indeed, it has been shown that the reducible permutation graph $F[\pi^*]$ has always a unique Hamiltonian path, denoted by HP($F[\pi^*]$), and this Hamiltonian path can be found in $O(n)$ time, where n is the number of nodes of $F[\pi^*]$ (author's papers). Since $F[\pi^*]$ is isomorphic to $G'(P^*, I_{key})$ we can compute the unique Hamiltonian path HP of the graph $F[P^*]$ within the same time complexity.

4 MODEL ASSESSMENT

Having designed a static or dynamic software watermarking model, it is very important to evaluate it under various criteria in order to gain information about its practical behavior. A software watermarking model can be evaluated using several criteria (Collberg et al., 2009); these criteria aim to evaluate model's performance and resiliency against malicious users attacks.

4.1 Assessments Criteria

We propose a set of such evaluation criteria presented in a tree structure which we call *Assessment Tree* or, for short, A-tree (see, Figure 5); it consists of two main subtrees namely:

- Performance subtree, and
- Resilience subtree.

The Performance subtree (or, P-tree) contains criteria concerning the performance of the resulting program after embedding of the watermark w, that is, the watermarked program P^*, while the Resilience subtree (or, R-tree) contains criteria concerning the resilience of the embedded watermark w; hereafter, we shall equivalently use the terms P-criteria and R-criteria for the criteria of the P-tree and R-tree, respectively.

4.1.1 Performance

The performance criteria mainly focus on the overhead and the protection of software; in our classification the P-criteria are the following:

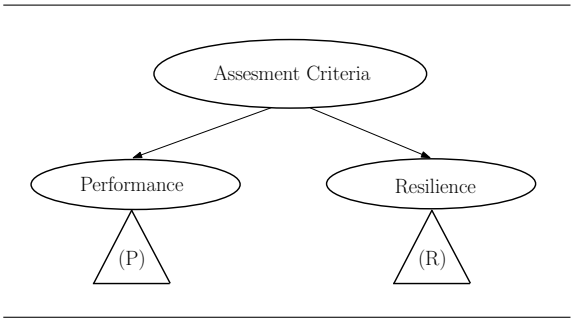

Figure 5: The Assessment tree (ASC) with its two subtrees namely the Performance subtree (P) and the Resilience subtree (R).

- **Data rate**: The ratio of the number of bits of source code (P) to the number of bits of watermark (w) that can be embedded into P.
- **Embedding overhead**: The additional time and space caused by implementing a watermarking model during the execution.
- **Part protection**: The watermarking models should spread the message throughout the entire code of software, in order to decrease the probability that all or part of the message is destroyed during attacks or in a possible partial interception.
- **Credibility**: The detector should minimize the probability to generate false positives and false negative results.

Let us now discuss on the performance of the water-Rpg model and let us first focus on the data rate and the embedding overhead of our model. Both criteria essentially depends on the value of the watermark w or, equivalently, of the size of the embedding graph $F[\pi^*]$. In the case where the code (in bits) of the original program P is huge our model has high data rate and extremely low embedding overhead. This arises from the fact that P^* does not contain any dummy code since we only modify the existing control flow graph of P preserving thus the functionality of the resulting watermarked program P^* by replacing a function call with a pair of function calls with the help of control statements.

Moreover, the more the watermark value w, that is about to be encoded to a reducible permutation graph $F[\pi^*]$, is acceding, the better the part protection is. The reason for that lies on the fact that the number of the code's functions used is greater leading to a 'spread' of the watermark in a large-scaled code. Of course, this also contributes to an inevitable trade-off between the first two criteria i.e. the data rate and the embedding overhead.

The credibility of a watermarking model is dependent on how detectable the watermark is, in order to prove ownership. Concerning our model, we should point out that the rates of false positive and false negative outcomes are noticeably low because even in cases where the watermarked code undergoes attacks, the sequence according to which the call functions are being executed is very hardly distorted.

4.1.2 Resilience

The resilience criteria mainly focus on the stealthiness and the distortion of the watermark; in our classification the R-tree contains two main categories of assessment criteria:

- **Watermark attacks**: In this case the attacker has detected (or, recognized) the watermark embedded in P, this is, the code of program P associated with the watermark w, and thus makes specific operations mainly on that code in order to
  - remove (e.g., by subtracting parts or all of the watermark),
  - destroy (e.g., by applying semantics preserving transformation), or even
  - alter the watermark w.

  In order to alter the watermark the attacker either adds a new w' or modifies the existing w.
- **Code attacks**: In the case where the attacker fails to detect the code of program P associated with the watermark w, he makes attacks in the whole code aiming in this way to distort possible watermarking protections; in this category, our classification includes
  - obfuscation (e.g., reducible to non-reducible flow graphs of methods),
  - optimization (e.g., remove information for debugging with tool named ProGuard),
  - de-compilation (e.g., using the Java Decompiler tool), and
  - language-transformation attacks (e.g., convert a program from C++ to Java).

The watermark attacks take place when the code of program P^* associated with the embedded watermark w is known to the attacker. In such a case, if the attacker makes a modification in a value of a cf-variable in a call-site p, then he has to properly modify all the values of all the cf-variables in every call-site of the execution flow after p and, thus, the watermark w remains unchained.

As described in the previous sections in our model we use appropriate function calls of the application program P and modify the execution flow of P through the use of control statements *cf-statements* and *cf-variables* such that $O(P,I) = O(P^*,I)$ for every input I. In fact what we do is creating dependencies on the data between the original program P and the watermark w making thus the watermark graph part of the computation of the program P. This leads to the failure of a dead code elimination process, because the removal of any of the *cf-variables* would make the program P non operational. What is more, every *cf-statement* produces different results, so our model withstands common subexpression elimination.

Our model uses opaque predicates in specific control statements in order to control the flow of selected function calls of P^* so that the watermarked program P^* have an appropriate execution. It is worth noting that it is hard for an attacker to deduce an opaque predicate at run time. Specifically, the usage of opaque predicates in our model enables us to dictate the execution flow of function calls and also makes the programs' control flow difficult for an attacker to analyze it either statically or dynamically.

Moreover, it is worth noting that if the attacker modifies a real-call then the program P^* is no longer operating. On the other hand, if the attacker modifies a water-call `call`(f_i), that is, modifies a water edge (f_i, f_j), then this modification can be detected by our model WaterRpg due to error-correcting properties of the reducible permutation graph $F[\pi^*]$ which represents the watermark w.

The code attacks are broadly applied in the whole code of the watermarked program P^*. Our watermarking model WaterRpg watermarks an application program P in such a way that it withstands on a relatively large subset of obfuscation and optimization attacks including semantic-preserving transformations, shrinking, re-ordering, and also the time consuming operation namely language transformation (i.e., the attacker rewrites the whole code of P^* in another language).

More precisely, the watermark can be efficiently extracted even the code of P^* has been subjected to some control obfuscation attacks such as expression reordering or loop reordering. It is fair to mention that our model does not properly operate on some other control obfuscation attacks such as aggregation including inline functions, outline functions, etc.

As far as the optimization attacks are concerned we point out that the embedded watermark w of our model withstands on such attacks since any removal of system calls, dead code, or information for debugging does not affect the structural properties of the embedding graph $F[\pi^*]$ which represents the watermark w.

5 CONCLUDING REMARKS

In this paper we presented a dynamic watermarking model for embedding a reducible permutation graph $F[\pi^*]$ into an application program P using appropriate calls of specific functions of P.

The main feature of our model is its ability to embed the graph $F[\pi^*]$ into P using only real functions and thus the size of the watermarked program P^* remains relatively very small. Moreover, the proposed dynamic watermarking model has low time complexity and incorporates such properties which cause it resilient to attacks.

An important property of our model is its ability to not use any mark during the embedding process in order to be able to extract the embedding watermark from the software.

Finally, we point out that it would be very interesting to evaluate our dynamic watermarking model in order to gain information on its practical behavior; we leave it as a problem for future work.

REFERENCES

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

BSA (2011). *BSA Global Software Piracy Study - 2011 Edition*. Retrieved February, 2013, from the site http://portal.bsa.org/globalpiracy2011/.

Chroni, M. and Nikolopoulos, S. (2010). Encoding watermark integers as self-inverting permutations. In *Proc. Int'l Conference on Computer Systems and Technologies (CompSysTech'10)*, volume ACM ICPS 471, pages 125–130.

Chroni, M. and Nikolopoulos, S. (2011). Encoding watermark numbers as cographs using self-inverting permutations. In *Proc. Int'l Conference on Computer Systems and Technologies (CompSysTech'11)*, volume ACM ICPS 578, pages 142–148.

Chroni, M. and Nikolopoulos, S. (2012). An efficient graph codec system for software watermarking. In *36th IEEE Conference on Computers, Software, and Applications (COMPSAC'12 Workshops)*, volume IEEE Proceedings, pages 595–600.

Collberg, C., Carter, E., Debray, S., Huntwork, A., Kececioglu, J., Linn, C., and Stepp, M. (2004). Dynamic path-based software watermarking. In *Proc. ACM Conference on Programming Language Design and Implementation (SIGPLAN'04)*, pages 107–118.

Collberg, C., Carter, E., Kobourov, S., and Thomborson, C. (2003). Error-correcting graphs for software watermarking. In *Proc. 29th Workshop on Graphs in Computer Science (WG'03)*, volume LNCS 2880, pages 156–167.

Collberg, C., Huntwork, A., Carter, E., Townsend, G., and Stepp, M. (2009). More on graph theoretic software watermarks: Implementation, analysis, and attacks. *Information and Software Technology*, 51:56–67.

Collberg, C. and Thomborson, C. (1999). Software watermarking: models and dynamic embeddings. In *Proc. 26th ACM SIGPLAN-SIGACT on Principles of Programming Languages (POPL'99)*, pages 311–324.

Collberg, C., Thomborson, C., Horning, J., Silbert, W., Matheson, L. R., Wright, A., and Owicki, S. (2011). Software watermarking techniques. *US Patent*, 2011/0214188.

Cousot, P. and Cousot, R. (2004). An abstract interpretation-based framework for software watermarking. In *Proc. 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'04)*, pages 173–185.

Cox, I., Kilian, J., Leighton, T., and Shamoon, T. (1996). A secure, robust watermark for multimedia. In *Proc. 1st Int'l Workshop on Information Hiding*, volume LNCS 1174, pages 317–333.

Davidson, R. and Myhrvold, N. (1996). Method and system for generating and auditing a signature for a computer program. *US Patent*, 5.559.884.

Hecht, M. and Ullman, J. (1972). Flow graph reducibility. *SIAM J. Computing*, 1:188–202.

Hecht, M. and Ullman, J. (1974). Flow graph reducibility. *Journal of the ACM*, 21:367–375.

Horne, W., Maheshwari, U., Tarjan, R., Horning, J., Silbert, W., Matheson, L. R., Wright, A., and Owicki, S. (2012). Systems and methods for watermarking software and other media. *US Patent*, 8.140.850.

Myles, G. and Collberg, C. (2006). Software watermarking via opaque predicates: implementation, analysis, and attacks. *Electronic Commerce Research*, 6:155–171.

Qu, G. and Potkonjak, M. (1998). Analysis of watermarking techniques for graph coloring problem. *Proceding of 1998 IEEE/ACM Int. Conference on Computer*, pages 190–193.

Rodriguez, T., MacIntosh, B., and Gustafson, A. (2010). Software watermarking. *US Patent*, 20100095376.

Sharma, B., Agarwal, R., and Singh, R. (2011). An efficient software watermark by equation reordering and fdos. In *SocProS (2)*, volume 131, pages 735–745.

Venkatesan, R., Vazirani, V., and Sinha, S. (2001). A graph theoretic approach to software watermarking. In *Proc. 4th Int'l Workshop on Information Hiding (IH'01)*, volume LNCS 2137, pages 157–168.

WIPO (2012). *World Intellectual Property Indicators - 2012 Edition*. Retrieved February, 2013, from the site http://www.wipo.int/ipstats/en/.

Xie, T. and Notkin, D. (2002). An empirical study of java dynamic call graph extractors. In *University of Washington CSE Technical Report 02-12-03*.

Zhang, X., Zhang, Z., and Zhang, C. (2011). Spread spectrum-based fragile software watermarking. In *15th North-East Asia Symposium on Inform. Technology and Reliability (NASNIT)*, pages 150–154.