An Embedding Graph-based Model for Software Watermarking

Maria Chroni and Stavros D. Nikolopoulos
Department of Computer Science
University of Ioannina
Ioannina, Greece
{mchroni, stavros}@cs.uoi.gr

***Abstract*—In a software watermarking environment, several graph theoretic watermark methods encode the watermark values as graph structures and embed them in application programs. In this paper we first present an efficient codec system for encoding a watermark number w as a reducible permutation graph $F[\pi^*]$ through the use of the self-inverting permutation π^* which encodes the number w and, then, we propose a method for embedding the watermark graph $F[\pi^*]$ into a program P. The main idea behind the proposed embedding method is a systematic use of appropriate calls of specific functions of the program P. That is, our method embeds the graph $F[\pi^*]$ into P using only real functions and thus the size of the watermarked program P^* remains very small. Moreover, the proposed codec system has low time complexity, can be easily implemented, and incorporates such properties which cause it resilient to attacks.**

***Index Terms*—software watermarking, watermark numbers, self-inverting permutations, reducible permutation graph, encoding, decoding, graph embedding, call-graphs, algorithms.**

I. Introduction

In the last decade, 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, $\texttt{encode}(w) \rightarrow G$, and also of a decoding function decode that converts the graph G into the number w, $\texttt{decode}(G) \rightarrow w$; we usually call the pair $(\texttt{encode}, \texttt{decode})$ as *graph codec* [7]. From a graph-theoretic point of view, we are looking for a class of graphs $\mathcal{G}$ and a corresponding codec $(\texttt{encode}, \texttt{decode})_{\mathcal{G}}$ with such properties which cause them resilient to attacks.

A lot of research has been done on software watermarking. The major software watermarking algorithms currently available are based on several techniques, among which the register allocation, spread-spectrum, opaque predicate, abstract interpretation, dynamic path techniques (see, [1], [5], [10], [11], [14], [15], [16]).

In 1996, Davidson and Myhrvold [12] proposed the first static software watermarking algorithm which embeds the watermark into an application program by reordering the basic blocks of a control flow-graph. Based on this idea, Venkatesan, Vazirani and Sinha [18] 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 called VVS or GTW.

In [18] the construction of a directed graph G (or, watermark graph G) is not discussed. Collberg et al. [8] proposed an implementation of GTW, which they call $\texttt{GTW}_{\texttt{sm}}$, and it is the first publicly available implementation of the algorithm GTW. In $\texttt{GTW}_{\texttt{sm}}$ the watermark is encoded as a reducible permutation graph (RPG) [7], which is a reducible control flow-graph with maximum out-degree of two, mimicking real code. Note that, for encoding integers the $\texttt{GTW}_{\texttt{sm}}$ method uses self-inverting permutations.

Recently, Chroni and Nikolopoulos [3], [4] 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. They presented an efficient algorithm which encodes a watermark number w as self-inverting permutation π^* and, also, an algorithm which encodes the 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^*; in the same paper, the authors also proposed efficient decoding algorithms.

In this paper, we first present efficient encoding and decoding algorithms that encode watermark values into reducible permutation graphs through the use of self-inverting permutations (or, for short, SiP) and extract them from the graph structures. More precisely, we present an efficient codec system for encoding a watermark number w as a reducible permutation graph $F[\pi^*]$ through the use of the self-inverting permutation π^* which encodes the number w [3]. Then, we propose a method for embedding the watermark graph $F[\pi^*]$ into an application program P by using appropriate calls of specific functions of the program P. The main feature of our embedding method 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 very small. Moreover, the proposed codec system has low time complexity, can be easily implemented, and incorporates such properties which cause it resilience to attacks.

II. Background Results

We consider finite graphs with no multiple edges. For a graph G, we denote by $V(G)$ and $E(G)$ the vertex (or, node) set and edge set of G, respectively. The *degree* of a vertex x in the graph G, denoted $deg(x)$, is the number of edges incident on node x; for a node x of a directed graph G, the number of head-endpoints of the directed edges adjacent to x is called the indegree of the node x, denoted $indeg(x)$, and the number of tail-endpoints is its outdegree, denoted $outdeg(x)$.

A *path* in a graph G of length k is a sequence of vertices $(v_0, v_1, \ldots, v_k)$ such that $(v_{i-1}, v_i) \in E(G)$ for $i = 1, 2, \ldots, k$. A path is called *simple* if none of its vertices occurs more than once. A path (simple path) $(v_0, v_1, \ldots, v_k)$ is a *cycle* (*simple cycle*) of length $k+1$ if $(v_0, v_k) \in E(G)$.

A. Self-inverting Permutations – SiP

Next, we introduce some definitions that are key-objects in our algorithms for encoding numbers as graphs. Let π be a permutation over the set $N_n = \{1, 2, \ldots, n\}$. 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 $\pi_1 = 1$, $\pi_2 = 4$, ect. Notice that π_i^{-1} is the position in the sequence of the number i; in our example, $\pi_4^{-1} = 2$, $\pi_7^{-1} = 4$, $\pi_3^{-1} = 6$, etc [13].

Definition 1. The inverse of a permutation $(\pi_1, \pi_2, \ldots, \pi_n)$ is the permutation $(q_1, q_2, \ldots, q_n)$ with $q_{\pi_i} = \pi_{q_i} = i$. A *self-inverting permutation* (or, involution) is a permutation that is its own inverse: $\pi_{\pi_i} = i$.

By definition, every permutation has a unique inverse, and the inverse of the inverse is the original permutation. Clearly, a permutation is a self-inverting permutation if and only if all its cycles are of length 1 or 2; hereafter, we shall denote a 2-cycle as $c = (x, y)$ and an 1-cycle as $c = (x)$, or, equivalently, $c = (x, x)$.

Definition 2. Let $C_{1,2} = \{c_1 = (x_1, y_1), c_2 = (x_2, y_2), \ldots, c_k = (x_k, y_k)\}$ be the set of all the cycles of a self-inverting permutation π such that $x_i < y_i$ $(1 \leq i \leq k)$, and let $\prec$ be a linear order on $C_{1,2}$ such that $c_i \prec c_j$ if $x_i < x_j$, $1 \leq i, j \leq k$. A sequence $C = (c_1, c_2, \ldots, c_k)$ of all the cycles of a self-inverting permutation π is called *increasing cycle representation* of π if $c_1 \prec c_2 \prec \cdots \prec c_k$. The cycle c_1 is the minimum element of the sequence C.

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

III. Encode Watermark Numbers as RPGs

For encoding integers some recently proposed watermarking methods use only those permutations that are self-inverting. Collberg et al. [7] based on the fact that there is a one-to-one correspondence between self-inverting permutations and isomorphism classes of RPGs proposed a polynomial algorithm for encoding any integer w as the RPG, corresponding to the wth self-inverting permutation π in this correspondence. This encoding exploits only the fact that a self-inverting permutation is its own inverse.

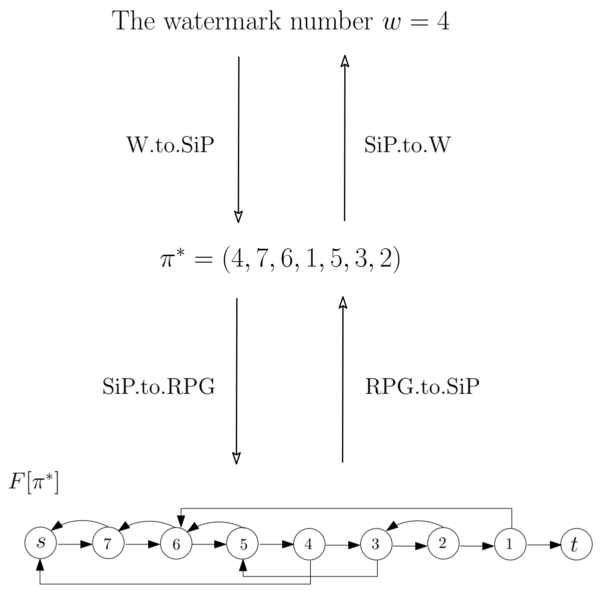

Fig. 1. The main data components used by the algorithms of the codec system $(\texttt{encode}, \texttt{decode})_{F[\pi^*]}$.

In [4] Chroni and Nikolopoulos proposed an efficient algorithm which encodes an integer w as self-inverting permutation π^* thought the use of bitonic permutations; recall that a permutation $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$ over the set N_n is called bitonic if either monotonically increases and then monotonically decreases, or else monotonically decreases and then monotonically increases. In this encoding the self-inverting permutation incorporates important structural properties which cause it resilient to attacks.

We next describe the main properties of our codec system $(\texttt{encode}, \texttt{decode})_{F[\pi^*]}$; we mainly focus on the properties of the reducible permutation graph $F[\pi^*]$ with respect to graph-based software watermarking attacks.

1) Components of the graph $F[\pi^*]$: The reducible permutation graph $F[\pi^*]$ consists of the following three components: the **header node** (it is a root node with outdegree one from which every other node in the graph $F[\pi^*]$ is reachable; in the graph $F[\pi^*]$ the header node is denoted by s); the **footer node** (it is a node with outdegree zero that is reachable from every other node of the graph; in the graph $F[\pi^*]$ the footer node is denoted by t); and the **body** (it consists of n nodes $u_1, u_2, \ldots, u_n$ each with outdegree two. In particular, each node u_i $(1 \leq i \leq n)$ has exactly two outpointers: one points to node u_{i-1} and the other points to node u_m, where $m > i$).

2) Structural Properties: The proposed reducible permutation graph $F[\pi^*]$ and a corresponding codec system $(\texttt{encode}, \texttt{decode})_{F[\pi^*]}$ have the following properties:

- **Appropriate graph types**: The graph $F[\pi^*]$ is directed on $n+2$ nodes with outdegree exactly two; that is, it has low max-outdegree, and, thus, it matches real program graphs;
- **High resiliency**: Since each node in the graph $F[\pi^*]$ has exactly one outpointer and exactly one outpointer, any single edge modification, i.e., edge-flip, edge-addition, or edge-deletion, will violate the outpointer condition of some nodes, and thus the modified edge can be easily identified and corrected;
- **Small size**: The size $|P^*| - |P|$ of the embedded watermark is small;
- **Efficient codecs**: The codec $(\texttt{encode}, \texttt{decode})_{F[\pi^*]}$ has low time and space complexity [3].

It is worth noting that our encoding and decoding algorithms use basic data structures and basic operations, and, thus, they can be easily implemented.

3) Unique Hamiltonian Path: We next show 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 $F[\pi^*]$. The following algorithm, which we call `Unique_HP`, takes as input a graph $F[\pi^*]$ on n nodes and produces its unique Hamiltonian path HP($F[\pi^*]$).

```
Algorithm Unique_HP
1. Find the node $u_0$ of the reducible permutation graph
   $F[\pi^*]$ with outdegree one;
2. Perform DFS-search on graph $F[\pi^*]$ starting at node $u_0$
   and compute the DFS discovery time $d[u]$ of each node
   $u$ of $F[\pi^*]$;
3. Order the nodes $u_0, u_1, \ldots, u_{n+1}$ of the graph $F[\pi^*]$
   by their DFS discovery time $d[]$ and let HP($F[\pi^*]$) =
   $(u'_0, u'_1, \ldots, u'_{n+1})$ be the resulting order of the nodes,
   where $d[u'_i] < d[u'_j]$ for $i < j$, $0 \leq i, j \leq n+1$;
4. Return HP($F[\pi^*]$);
```

Since the graph $F[\pi^*]$ contains n nodes and $m = O(n)$ edges, both finding the node of $F[\pi^*]$ with outdegree one and performing DFS-search on $F[\pi^*]$ take $O(n)$ time and require $O(n)$ space.

IV. Embedding a RPG into a Code

Having encoded a watermark number w as Reducible Permutation Graph $F[\pi^*]$, let us now propose a method which embeds the watermark graph $F[\pi^*]$ into an application program. The main idea behind the proposed method is a systematic use of appropriate calls of specific functions of the program. More specifically we present a method for encoding our RPG in the call graph of a program.

A call graph of a program represents calling relations between procedures in a program. It has a distinguished root node, corresponding to the highest-level procedure and representing an abstraction of the whole system [17]. These graphs are used in interprocedural program optimization and for reverse engineering of software systems [2], [9].

The nodes of a call graph represent the procedures being either callees or callers; their edges represent the calling relations between the procedures. Since call graphs are directed graphs, every edge has an explicit source and target node, representing the calling and called procedure, respectively; note that, cycles in a call graph represent recursion.

A. Embedding Model

We next present the method for embedding the graph $F[\pi^*]$ into an application program P (see, Figure 1); the proposed embedding method consists of the following steps:

```
Embedding Method RPG.to.CODE
1. Take as input the source code of the program $P$ and
   the watermark RPG $F[\pi^*]$ and let $f_0, f_1, \ldots, f_k$ be the
   functions of $P$ and $s = u_0, u_1, \ldots, u_n, u_{n+1} = t$ be the
   nodes of $F[\pi^*]$, $k \geq n+2$;
2. Select $n+2$ functions of $P$, say, $f^*_0, f^*_1, \ldots, f^*_{n+1}$, and
   assign the node $u_i$ of the graph $F[\pi^*]$ to function $f^*_i$,
   $0 \leq i \leq n+1$;
3. Construct the call-graph (CG) of the input program $P$
   by finding the real-calls of each function of $P$;
4. Construct the control-flow graph (CFG) of each function
   $f^*_0, f^*_1, \ldots, f^*_{n+1}$ of the program $P$ and locate two "call-
   points" in the control-flow graph of each function $f^*_i$,
   $0 \leq i \leq n+1$;
5. Add the statement call($f^*_j$) in a call-point of the
   function $f^*_i$ if the corresponding nodes $u_i$ and $u_j$ form
   a directed edge $(u_i, u_j)$ in the graph $F[\pi^*]$, $0 \leq i, j \leq
   n+1$, and mark this call as water-call;
6. Return the source code of the program $P^*$;
```

B. Extracting the Watermark RPG from the Code

In this section, we present an algorithm for extracting the graph $F[\pi^*]$ from program P^* watermarked by the embedding method `RPG.to.CODE`; the proposed extracting method is the following:

```
Extracting Method RPG.from.CODE
1. Take as input the source code of the watermarked
   program $P^*$;
2. Construct the call-graph CG[$P^*$] of the input program
   $P^*$ by finding the real-calls and the water-calls of each
   function of $P^*$;
3. Delete all the edges from the graph CG[$P^*$] which
   corresponds to real-calls and, then, delete all its isolated
   nodes; let CG[$P'$] be the resulting graph;
```

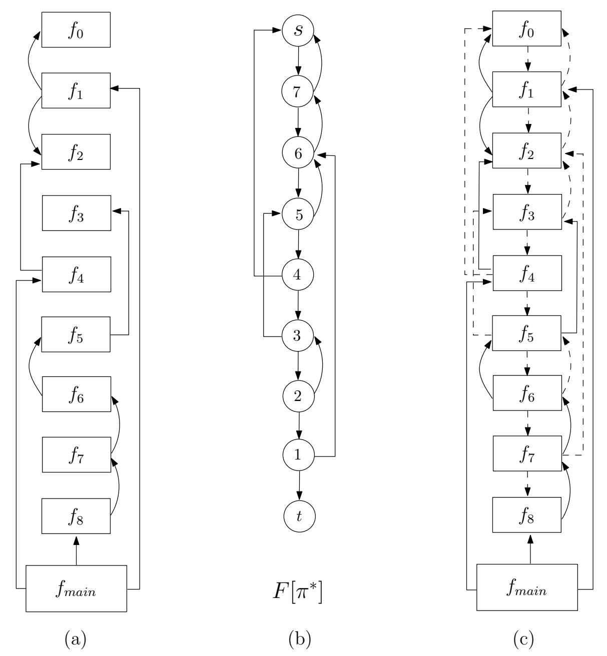

Fig. 2. (a) The call-graph CG[P] of the input application program P. (b) The reducible permutation graph $F[\pi^*]$. (c) The call-graph CG[P^*] of the watermarked program P^*.

4. Construct a graph $F[P^*]$ isomorphic to CG[P'] and let $s = u_0, u_1, \ldots, u_n, u_{n+1} = t$ be its nodes;
5. Compute the unique Hamiltonian path of the graph $F[P^*]$ and, then, construct the flow-graph $F[\pi^*]$;
6. Return the graph $F[\pi^*]$;

In order to ensure that the water-calls do not affect the execution of the program, we propose a method which alter the source code by inserting control flow statements, say, an `if-else-statement`, and an extra parameter, say, w, in each function f^*. During the execution of the program P^*, if a function f water-calls the function f^* then we pass by reference an appropriate value as an argument into f^* through the parameter w; the type of the parameter may be either boolean, number, or a string. Once the condition of the `if-else-statement` is satisfied, an appropriate statement is executed (for example, $w = -w$) and the function f^* returns; otherwise, the function f^* is normally executed. Note that, argument's value does not affect program's functionality.

In addition, we may perform an obfuscating technique in P^* in order to prevent reverse engineering. Roughly speaking, the goal of obfuscation is to hide the secrets inside a program while preserving its functionality.

V. Concluding Remarks

In this paper we first presented a codec system which encodes a watermark number w as reducible permutation graph $F[\pi^*]$ and, then, we proposed a model for embedding the graph $F[\pi^*]$ into an application program P using appropriate calls of specific functions of P. The main feature of our embedding 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 very small. Moreover, the proposed codec system has low time complexity, can be easily implemented, and incorporates such properties which cause it resilient to attacks.

In light of our embedding method it would be very interesting to implement the method in order to obtain a clear view of its practical behaviour; we leave it as a problem for future investigation.

References

[1] G. Arboit, "A method for watermarking Java programs via opaque predicates," 5th International Conference on Electronic Commerce Research (ICECR-5) (2002).

[2] E. J. Chikofsky, and J. H. Cross II, "Reverse engineering and design recovery: A taxonomy," IEEE Software, pp. 1317 (1990).

[3] M. Chroni and S.D. Nikolopoulos, "Efficient Encoding of Watermark Numbers as Reducible Permutation Graphs," CoRR abs/1110.1194 (October 2011).

[4] M. Chroni and S.D. Nikolopoulos, "Encoding watermark integers as self-inverting permutations," Proc. Int'l Conference on Computer Systems and Technologies (CompSysTech'10), ACM ICPS 471, 125–130 (2010).

[5] C. Collberg, E. Carter, S. Debray, A. Huntwork, J. Kececioglu, C. Linn and M. Stepp, "Dynamic path-based software watermarking." Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM SIGPLAN 39, pp. 107–118 (2004).

[6] C. Collberg and J. Nagra, "Surreptitious Software," Addison-Wesley (2010).

[7] C. Collberg, S. Kobourov, E. Carter, and C. Thomborson, "Error-correcting graphs for software watermarking," Proc. 29th Workshop on Graph-Theoretic Concepts in Computer Science (WG'03), LNCS 2880, pp. 156–167 (2003).

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

[9] K. D. Cooper, and K. Kennedy, "Efficient computation of flow insensitive interprocedural summary information," In Proceedings of the ACM SIGPLAN84 Symposium on Compiler Construction, pp. 247258 (1984).

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

[11] D. Curran, N. Hurley and M. Cinneide, "Securing Java through software satermarking, Proc," Int'l Conference on Principles and Practice of Programming in Java (PPPJ'03), pp. 145–148 (2003).

[12] R.L. Davidson and N. Myhrvold, "Method and system for generating and auditing a signature for a computer program" US Patent 5.559.884, Microsoft Corporation (1996).

[13] M.C. Golumbic, "Algorithmic Graph Theory and Perfect Graphs," Academic Press, New York (1980). Second edition, Annals of Discrete Math. 57, Elsevier (2004).

[14] A. Monden, H. Iida, K. Matsumoto, K. Inoue and K. Torii, "A practical method for watermarking Java programs," Proc. 24th Computer Software and Applications Conference (COMPSAC'00), pp. 191–197 (2000).

[15] J. Nagra and C. Thomborson, "Threading software watermarks," Proc. 6th Int'l Workshop on Information Hiding (IH'04), LNCS 3200, pp. 208-223 (2004).

[16] G. Qu and M. Potkonjak, "Analysis of watermarking techniques for graph coloring problem," Proc. IEEE/ACM Int'l Conference on Computer-aided Design (ICCAD'98), ACM Press, pp. 190–193 (1998).

[17] B. Ryder,"Constructing the Call Graph of a Program ," IEEE Transactions on Software Engineering SE-5(3), pp. 216–225 (1979).

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