DESIGN AND EVALUATION OF A GRAPH CODEC SYSTEM FOR SOFTWARE WATERMARKING

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

Keywords: Software watermarking; Codec systems, Self-inverting permutations; Reducible permutation graphs; Encoding algorithms; Decoding algorithms; Performance; Experimental Evaluation.

Abstract: In this paper, we propose an efficient and easily implemented codec system for encoding watermark numbers as graph structures thought the use of self-inverting permutations. More precisely, based on the fact that a watermark number w can be efficiently encoded as self-inverting permutation π^*, we present an efficient encoding algorithm which encodes a self-inverting permutation π^* as a reducible flow-graph $F[\pi^*]$ and a decoding algorithm which extracts the permutation π^* from the graph $F[\pi^*]$. Our codec algorithms are very simple, use elementary operations on sequences and linked structures, and the produced flow-graph $F[\pi^*]$ does not differ from the graph data structures built by real programs. Moreover, our codec algorithms have very low time and space complexity and the flow-graph $F[\pi^*]$ incorporates important structural properties which cause it resilient to attacks. We have evaluated several components of our codec system in a simulation environment in order to obtain a clear view of their practical behaviour; the experimental results show that we can decide with high probability whether the graph $F[\pi^*]$ suffer an attack on its edges.

1 INTRODUCTION

Software watermarking is a technique for protecting the intellectual property of an application program; the idea is similar to digital watermarking where a unique identifier is embedded in image, audio, or video data through the introduction of errors not detectable by human perception (Cox et al., 1996). The *software watermarking problem* can be described as the problem of embedding a structure w into a program P and, thus, producing a new program 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 $\text{extract}(P_w,k) \rightarrow w$.

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. 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 (Arboit, 2002; Cousot and Cousot, 2004; Curran et al., 2003; Grover, 1997; Monden et al., 2000; Nagra and Thomborson, 2004; Qu and Potkonjak, 1998; Stern et al., 1999); see also, Collberg and Nagra (Collberg and Nagra, 2010) and (Zhang et al., 2003; Zhu et al., 2005) for an exposition of the main results.

We should mention that there are two general categories of watermarking algorithms namely *static* and the *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, while a dynamic watermark is built during program execution, perhaps only after a particular sequence of input. Further discussion of static and/or dynamic watermarking issues can be found in (Davidson and Myhrvold, 1996; Moskowitz and Cooperman, 1996; Venkatesan et al., 2001).

Codec Systems 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, $encode(w) \to G$, and also of a decoding function decode that converts the graph G into the number w, $decode(G) \to w$; we usually call the pair $(encode, decode)_G$ 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 $(encode, decode)_{\mathcal{G}}$ 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, i.e., insertions or deletions of a constant number of nodes or/and edges; that is, if $G \in \mathcal{G}$ and $decode(G) \to w$ then $decode(G') \to w$ 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 program code in a certain format and it does not change during the program 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 and is called VVS or GTW. In (Venkatesan et al., 2001) the construction of a directed graph G (or, watermark graph G) is not discussed. 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. In GTW$_{sm}$ the watermark is encoded as a reducible permutation graph (RPG) (Collberg et al., 2003), which is a reducible control flow-graph with maximum out-degree of two, mimicking real code. 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 (or, hereafter, RPG). They presented an efficient algorithm which encodes a watermark number w as self-inverting permutation π^* and, also, an efficient algorithm which encodes the permutation π^* as a reducible permutation flow-graph. The construction of the flow-graph is made by exploiting domination relations on the elements of π^* and using an efficient DAG representation of the permutation π^*; in the same paper, the authors also proposed efficient decoding algorithms. The main components of their codec system incorporate important structural properties which cause them resilient to attacks; we mention that attacks against graph-based software watermarking structures can mainly occur in the following two ways: (i) Node-modification attacks, and (ii) Edges-modification attacks.

Our Contribution. In this paper we present an efficient and easily implemented algorithm for encoding numbers as reducible permutation flow-graphs through the use of self-inverting permutations (or, for short, SiP). More precisely, having designed an efficient method for encoding integers as self-inverting permutations (Chroni and Nikolopoulos, 2010; Chroni and Nikolopoulos, 2011), we describe an algorithm for encoding a self-inverting permutation π^* into a directed graph structure $F[\pi^*]$ having properties capable to match real program graphs; it is simpler than that proposed in (Chroni and Nikolopoulos, 2011), uses elementary operations on sequences and linked structures (trees and graphs), and the produced flow-graph $F[\pi^*]$ 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. We also describe an efficient decoding algorithm which extract the self-inverting permutation π^* from the reducible permutation flow-graph $F[\pi^*]$.

We have evaluated several components of our codec system in a simulation environment in order to obtain a clear view of their practical behaviour; the experimental results show that we can decide with high probability whether the graph $F[\pi^*]$ suffer an attack on its edges.

It is worth noting that our codec system $(encode, decode)_{F[\pi^*]}$ has very low time and space complexity, and the flow-graph $F[\pi^*]$ incorporates important structural properties which enable us to identify with hight probability any single or multiple changes made by an attacker to flow-graph $F[\pi^*]$. We have evaluated several components of our codec system in a simulation environment in order to obtain a clear view of their practical behaviour; the experimental results are presented in Section 6.

Finally, we should point out that our graph codec system has very low time and space complexity which is $O(n)$ where n is the number of bits in a binary representation of the watermark integer w. Indeed, both functions Encode_W.to.SiP and Decode_SiP.to.W are computed in time and space linear in the binary size of the watermark integer w. Moreover, the functions Encode_SiP.to.RPG and Decode_RPG.to.SiP are also computed in linear time and space; in particular, the function Encode_SiP.to.RPG is computed in time and space linear in the length of the self-inverting permutation π^* which is $O(n)$, while the function Decode_RPG.to.SiP is computed in time and space linear in the size of the flow-graph $F[\pi^*]$ which is also $O(n)$, where $n = O(\log w)$ and w is the watermark.

Road Map. The paper is organized as follows: In Section 2 we establish the notation and related terminology, we present background results, and we prove important properties on self-inverting permutations. In Section 3 we present an efficient algorithm for encoding the self-inverting permutation π^* as a reducible permutation flow-graph $F[\pi^*]$, which we call Encode_SIP.to.RPG, and the corresponding decoding algorithm Decode_RPG.to.SIP. In Section 4 we analyze the structure of the flow-graph $F[\pi^*]$ and show properties which prevent several edge and/or node modifications attacks. In Section 5 we show that the malicious intentions of an attacker to lead an RPG in incorrect-stage by modifying node-labels and/or edges of the flow-graph $F[\pi^*]$ can be efficiently detected. Finally, in Section 6 we conclude the paper and discuss possible future extensions.

2 THEORETICAL FRAMEWORK

In this section, we present basic components and background results that are used in the design of our codec system. In particular, we first define the main components namely self-inverting permutations (SiP) and reducible permutation graphs (RPG), and then we study self-inverting permutations and prove properties which are used as key-objects in our algorithms for encoding numbers as reducible permutation graphs.

2.1 Data Components

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; in our example, $\pi_4^{-1} = 2$, $\pi_7^{-1} = 4$, $\pi_3^{-1} = 6$, etc.

Definition 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 1. Throughout the paper we denote a self-inverting permutation π over the set $N_n = \{1, 2, \ldots, n\}$ as π^*;

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 iff all its cycles are of length 1 or 2; hereafter, we shall denote a 2-cycle by $c = (x, y)$ and by 1-cycle as $c = (x)$, or, equivalently, $c = (x, x)$.

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 $x \rightarrow y$ 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. 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 A shows. Those definitions use a few more graph-theoretic concepts. An edge (x, x) (for some node x) is a cycle-edge. A depth first search (DFS) of a flow-graph partitions its edges into tree edges (making up a spanning tree known as a DFS tree), forward edges (pointing to a successor in the spanning tree), back edges (pointing to a predecessor in the spanning tree, plus cycle-edges), and cross edges (the others). Tree edges, forward edges, and cross edges form a dag known as a DFS dag.

Theorem A (Hecht and Ullman, 1972; Hecht and Ullman, 1974): Let F be a flow-graph. The following three statements about the flow-graph F 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.

C. Encode Numbers as SiPs

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 (Chroni and Nikolopoulos, 2011). We have actually proved the following results.

Theorem 1. *Let w be an integer and let $b_1 b_2 \cdots b_n$ 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. *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.*

2.2 System Components

Our codec system uses of three main data components: (i) the watermark number w, (ii) the self-inverting permutation π^*, and (iii) the reducible permutation graph $F[\pi^*]$; theses three components are depicted in Figure 1.

The same figure also shows the two main faces of our system's process:

(I) **Face W–SiP:** it uses two algorithms, namely Encode_W.to.SiP and Decode_SiP.to.W, and encodes a watermark number w into an self-inverting permutation π^* and extracting it from π^*;

(II) **Face SiP–RPG:** this face also uses two algorithms, namely Encode_SiP.to.RPG and Decode_RPG.to.SiP, and encodes a self-inverting permutation π^* into a reducible permutation graph $F[\pi^*]$ and extracting it from $F[\pi^*]$.

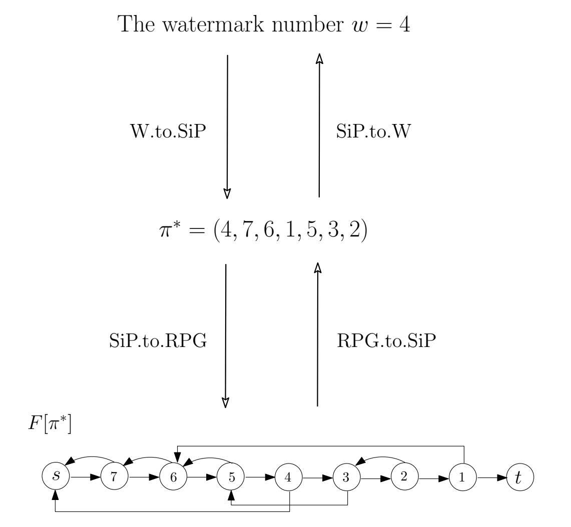

Figure 1: The main data components used by the algorithms of our codec system: (i) the watermark number w, (ii) the self-inverting permutation π^*, and (iii) the reducible permutation graph $F[\pi^*]$.

Recall that our contribution in this paper is concentrated on face SiP–RPG. Indeed, we design and analyze algorithms for encoding a SiP π^* as a reducible permutation flow-graph $F[\pi^*]$ and the corresponding decoding algorithm; we also show properties which prevent several edge and/or node modifications attacks.

2.3 Properties of SIPs

In this section, we study self-inverting permutations and prove properties which are used as key-objects in our algorithms for encoding numbers as reducible permutation graphs.

Lemma 1. *Let $c_i = (x, y)$ and $c_j = (z, w)$ be two 2-cycles of a self-inverting permutation π^* such that $x > y$ and $z > w$. If $x > z > y$ and $\pi_x^{*-1} < \pi_z^{*-1} < \pi_y^{*-1}$, then $w > y$ and $\pi_w^{*-1} < \pi_y^{*-1}$.*

Lemma 2. *Let $c_i = (x, y)$ and $c_j = (z, w)$ be two 2-cycles of a self-inverting permutation π^* such that $x > y$ and $z > w$. If $x < z$ and $\pi_x^{*-1} < \pi_z^{*-1}$, then $w > y$ and $\pi_w^{*-1} > \pi_y^{*-1}$.*

Let $\pi^* = (\pi_1^*, \pi_2^*, \ldots, \pi_n^*)$ be a permutation over the set N_n. A *subsequence* of π^* is a sequence $\alpha^* = (\pi^*_{i_1}, \pi^*_{i_2}, \dots, \pi^*_{i_k})$ such that $i_1 < i_2 < \cdots < i_k$. If, in addition, $\pi^*_{i_1} < \pi^*_{i_2} < \cdots < \pi^*_{i_k}$, then we say that α^* is an *increasing subsequence* of π^*, while if $\pi^*_{i_1} > \pi^*_{i_2} > \cdots > \pi^*_{i_k}$, then we say that α^* is a *decreasing subsequence* of π^*.

A *left-to-right maximum* (resp. *left-to-right minimum*) of π^* is an element π^*_i, $1 \le i \le n$, such that $\pi^*_i > \pi^*_j$ (resp. $\pi^*_i < \pi^*_j$) for all $j < i$. The increasing (resp. decreasing) subsequence $\alpha^* = (\pi^*_{i_1}, \pi^*_{i_2}, \dots, \pi^*_{i_k})$ is called a *left.to.right maxima (resp. minima) subsequence* if it consists of all the left.to.right maxima (resp. minima) of π^*; clearly, $\pi^*_{i_1} = \pi^*_1$. For example, the left.to.right maxima subsequence of the permutation $\pi^* = (4,2,6,1,9,3,7,5,12,11,8,10)$ is $(4,6,9,12)$, while the left.to.right minima subsequence of π^* is $(4,2,1)$.

The 1st increasing (resp. decreasing) subsequence S_1 of a permutation π^* is defined to be the left.to.right maxima (resp. minima) subsequence of π^*. The *i*th increasing (resp. decreasing) subsequence S_i of π^* is defined to be the left.to.right maxima (resp. minima) subsequence of π', where π' results from π^* after having ignored the elements of the $1st, 2ed, \dots, (i-1)st$ increasing (resp. decreasing) subsequences of π. For example, the decreasing subsequences of the permutation $\pi^* = (4,2,6,1,9,3,7,5,12,11,8,10)$ are $S_1 = (4,2,1)$, $S_2 = (6,3)$, $S_3 = (9,7,5)$, $S_4 = (12,11,8)$ and $S_5 = (10)$.

Theorem 3. *Let $S_i = (x_1, x_2, \dots, x_k)$ be the ith decreasing subsequence of a self-inverting permutation π^*. Then,*

(i) if k is an even number, the following pairs (x_1, x_k), $(x_2, x_{k-1}), \dots, (x_{\frac{k}{2}}, x_{\frac{k}{2}+1})$ form $\frac{k}{2}$ 2-cycles of π^*;

(ii) if k is an odd number, the following pairs (x_1, x_k), $(x_2, x_{k-1}), \dots, (x_{\lfloor\frac{k}{2}\rfloor}, x_{\lceil\frac{k}{2}\rceil+1})$ form $\lfloor\frac{k}{2}\rfloor$ 2-cycles and $(x_{\frac{k}{2}+1})$ forms an 1-cycle of π^*.

3 CODEC ALGORITHMS

It has been shown that a watermark number w can be efficiently encoded as self-inverting permutation π^* and efficiently decoded form it (Chroni and Nikolopoulos, 2010; Chroni and Nikolopoulos, 2011). In this section, we concentrate on system's face SiP–RPG and present efficient algorithms for encoding a self-inverting permutation π^* into a reducible permutation graph $F[\pi^*]$ and also decoding the graph $F[\pi^*]$.

3.1 Algorithm Encode_SiP.to.RPG

The proposed encoding algorithm, which we call Encode_SiP.to.RPG, takes as input the self-inverting permutation π^* of length n and constructs a reducible permutation flow-graph $F[\pi^*]$ by using the properties of the decreasing subsequences of π^* described in Theorem 3. The algorithm takes $O(n)$ time and requires $O(n)$ space; it works on two phases:

(I) it first computes the decreasing subsequences $S_1, S_2, \dots, S_k$ of the permutation π^* and, then

(II) it constructs a directed graph $F[\pi^*]$ on $n+2$ nodes using the next relation on the elements of the decreasing subsequences $S_1, S_2, \dots, S_k$; we note that, if $(x_1, x_2, \dots, x_i, x_{i+1}, \dots, x_m)$ is a subsequence, then the elements $\{x_i, x_{i+1}\}$ are next related, that is, $\text{next}(x_i) = x_{i+1}$, $1 \le i \le m-1$.

Next, we present in details the proposed encoding algorithm (see, Figure 2).

```
Algorithm Encode_SiP.to.RPG
Input: a self-inverting permutation $\pi^*$;
Output: a reducible permutation flow-graph $F[\pi^*]$;
1. Compute the decreasing subsequences $S_1$, $S_2$, $\dots, S_k$ of $\pi^*$ as follows:
   ◦ construct $k$ queues $Q_1, Q_2, \dots, Q_k$, initially empty; let $last(i)$ be the last inserted element in queue $Q_i$, $1 \le i \le k$;
   ◦ insert the first element $\pi_1$ of $\pi^*$ in queue $Q_1$;
   ◦ insert the $ith$ element $\pi_i$, $i = 2,3,\dots,n$, of $\pi^*$ in queue $Q_j$ if
         (i) $\pi_i < last(j)$ or $Q_j$ is empty, and
         (ii) $\pi_j > last(j-1)$;
   ◦ compute $S_i$ from the elements of $Q_i$, $1 \le i \le k$;
2. Construct a directed graph $F[\pi^*]$ on $n+2$ vertices, as follows:
   ◦ $V(F[\pi^*]) = \{t = u_0, u_1, \dots, u_n, u_{n+1} = s\}$;
   ◦ for $i = n$ downto 0 do: add the edge $(u_{i+1}, u_i)$ in $E(F[\pi^*])$; we call it list pointer;
3. For each vertex $u_i \in V(F[\pi^*])$, $1 \le i \le n$, do
   ◦ add the edge $(u_i, u_m)$ in $E(F[\pi^*])$ if
         (i) $\pi_m > \pi_i$ and $\text{next}(\pi_m) = \pi_i$ in
             a subsequence $S_j$, or
         (ii) $u_m = u_{n+1}$ and $\pi_i$ is the first element
             of $S_j$, $1 \le j \le k$; we call it
             permutation pointer;
4. Return the graph $F[\pi^*]$;
```

Time and Space Complexity. The encoding algorithm Encode_SiP.to.RPG constructs the reducible permutation flow-graph $F[\pi^*]$ in $O(n \log n)$ time, where n is the length of π^*; it uses binary search to insert the elements of π^* in $Q_1, Q_2, \ldots, Q_k$ and requires $O(n \log n)$ time. The sequences $S_1, S_2, \ldots, S_k$ of π^* can also be computed in $O(n)$ time using counting sort. Thus, the following theorem holds.

Theorem 4. *Let π^* be a self-inverting permutation of length n. The algorithm Encode_SiP.to.RPG for encoding the permutation π^* as a reducible permutation flow-graph $F[\pi^*]$ requires $O(n)$ time and space.*

3.2 Algorithm Decode_RPG.to.SiP

Having designed the efficient encoding algorithm Encode_SiP.to.RPG, we next present the decoding algorithm Decode_RPG.to.SiP which takes as input a flow-graph $F[\pi^*]$ and extracts the self-inverting permutation π^* from the graph $F[\pi^*]$ (see, Figure 2); it works as follows:

```
Algorithm Decode_RPG.to.SiP
Input: a reducible permutation flow-graph $F[\pi^*]$;
Output: the self-inverting permutation $\pi^*$;
1. Delete the directed edges $(v_{i+1}, v_i)$ from the edge set $E(F[\pi^*])$, $1 \le i \le n$, and the node $t = v_0$ from $V(F[\pi^*]) = \{t = v_0, v_1, \ldots, v_n, v_{n+1} = s\}$;
2. Flip all the remaining directed edges of the graph $F[\pi^*]$; the resulting graph is a tree $T[\pi^*]$ rooted at $s = v_{n+1}$;
3. While the root $s$ of $T[\pi^*]$ has at least one child $v_i$, do the following:
     ◦ find the leaf $v_j$ of $T[\pi^*]$ which is reachable from node $v_i$;
     ◦ set $P_m = (v_i, v_j)$ and delete both $v_i$ and $v_j$ from $T[\pi^*]$;
4. Construct the permutation $\pi^* = (\pi_1, \pi_2, \ldots, \pi_n)$ over the set $N_n$ such that $\pi_i = i$, $1 \le i \le n$;
5. Let $P$ be the set of all pairs $P_1, P_2, \ldots, P_k$ computed at step 3; then,
     ◦ for each pair $(v_i, v_j) \in P$ set $\pi_{\pi_i} = \pi_j$ and $\pi_{\pi_j} = \pi_i$;
6. Return the self-inverting permutation $\pi^*$;
```

Time and Space Complexity. The size of the tree $T[\pi^*]$ is $O(n)$ since the input graph $F[\pi^*]$ constructed by the algorithm Encode_SiP.to.RPG has $O(n)$ nodes. Based

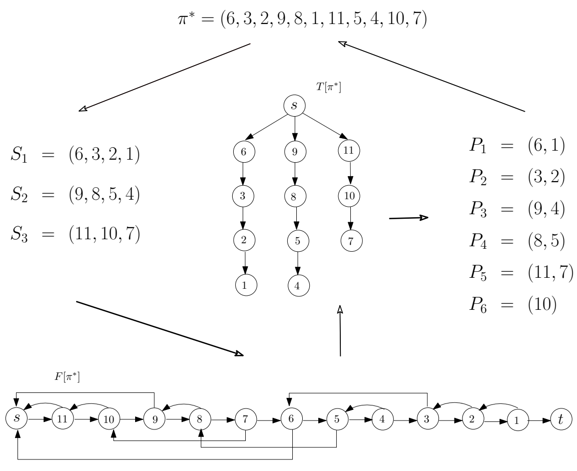

Figure 2: The main structures used or constructed by the algorithms Encode_SiP.to.RPG and Decode_RPG.to.SiP; that is, the self-inverting permutation π^*, the decreasing subsequences of π^*, the graph $F[\pi^*]$, the tree $T[\pi^*]$, and the elements of π^* in pairs.

on the structure of the tree $T[\pi^*]$ we can compute the pairs $P_1, P_2, \ldots, P_k$ in $O(n)$ time using $O(n)$ space. Thus, we can obtain the following result.

Theorem 5. *Let $F[\pi^*]$ be a reducible permutation flow-graph of size $O(n)$ produced by the algorithm Encode_SiP.to.RPG. The algorithm Decode_RPG.to.SiP decodes the flow-graph $F[\pi^*]$ in $O(n)$ time and space.*

4 PROPERTIES OF OUR CODEC SYSTEM

In this section, we analyze the structures of the two main components of the proposed codec system, that is, the self-inverting permutation π^* and the reducible permutation graph $F[\pi^*]$, and present properties which make our codec system resilient to attacks.

4.1 Properties of SiP π^*

In our codec system $(encode, decode)_{F[\pi^*]}$ an integer w is encoded as self-inverting permutation π^* using a particular construction technique which captures into π^* important structural properties. These properties enable us to identify edge changes made by an attacker to π^*.

The main structural properties of our self-inverting permutation π^* produced by the algorithm

Encode_W.to.SiP can be summarized into the following three categories:

- **SiP property**: By construction the permutation π^* is self-inverting permutation of odd length;
- **1-cycle property**: The self-inverting permutation π^* always contains one, and only one, cycle of length 1;
- **Bitonic property**: The self-inverting permutation π^* is constructed from the bitonic sequence $\pi^b = X||Y^R$, where X and Y are increasing subsequences (see, authors' work in (Chroni and Nikolopoulos, 2010; Chroni and Nikolopoulos, 2011)), and thus the bitonic property is encapsulated in the cycles of π^*.

The above properties can be efficiently used in order to identify whether our system has been attacked. In closing, we should mention that our encoding approach enables us to encode the integer $w = b_1 b_2 \cdots b_n$ as self-inverting permutation π^* of any length; indeed, π^* can be constructed over the set $N_{n'} = \{1, 2, \ldots, n'\}$, where $n' \geq 2\lceil \log w \rceil + 1$.

4.2 Properties of Flow-graph $F[\pi^*]$

We next describe the main properties of the reducible permutation graph $F[\pi^*]$ with respect to graph-based software watermarking attacks.

Structural Properties. In graph-based encoding algorithms, the watermark w is encoded into some special kind of graphs G. Generally, the watermark graph G should not differ from the graph data structures built by real programs. Important conditions are that the maximum outdegree of G should not exceed two or three, and that the graph G have a unique root node so the program can reach other nodes from the root node. Moreover, G should be resilient to attacks against edge and/or node modifications. Finally, G should be efficiently constructed.

The reducible permutation graph $F[\pi^*]$ produced by our codec system has all the above properties; in particular, the graph $F[\pi^*]$ and the corresponding codec have the following properties: (i) Appropriate graph types, (ii) High resiliency, and (iii) Efficient codecs. It is also worth noting that our encoding and decoding algorithms use basic data structures and basic operations, and, thus, they can be easily implemented.

Unique Hamiltonian Path. It is well-known that any acyclic digraph G has at most one Hamiltonian path (HP); G has one HP if the subgraphs $G_0, G_1, \ldots, G_n$ have only one node with indegree zero, where $G_0 = G$ and $G_i = G \backslash \{v_1, v_2, \ldots, v_i\}$, $1 \leq i \leq n-1$ (recall that n denotes the number of nodes in G). Furthermore, it has been shown that any reducible flow-graph has at most one Hamiltonian path (Collberg et al., 2003).

We next show that the reducible permutation graph $F[\pi^*]$ produced by the algorithm Encode_SiP.to.RPG 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 flow-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 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 $F[\pi^*]$ by their DFS discovery time $d[]$ and let HP($F[\pi^*]$) $=$ $(u'_0, u'_1, \ldots, u'_{n+1})$ be the resulting order, where $d[u'_i] < d[u'_j]$ for $i < j$ and $0 \leq i, j \leq n+1$;

4. Return HP($F[\pi^*]$);
```

Since the graph $F[\pi^*]$ contains n nodes and $m = O(n)$ edges, it is easy to see that 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. Thus, we have the following result.

Theorem 6. *Let $F[\pi^*]$ be a reducible permutation graph of size $O(n)$ produced by the algorithm Encode_SiP.to.RPG. The algorithm Unique_HP correctly computes the unique Hamiltonian path of $F[\pi^*]$ in $O(n)$ time and space.*

5 DETECTING ATTACKS

In this section, we show that the malicious intentions of an attacker to lead a reducible permutation graph $F[\pi^*]$ in incorrect-stage by modifying some node-labels or edges of the graph $F[\pi^*]$ can be efficiently detected.

5.1 Node-label Modification

By construction, our reducible flow-graph $F[\pi^*]$ is a labeled graph. Indeed, the labels of $F[\pi^*]$ are numbers of the set $\{0,1,\dots,n+1\}$, where the label $n+1$ is assigned to header node $s = u_{n+1}$, the label 0 is assigned to footer node $t = u_0$, and the label $n-i$ is assigned to the ith body node u_{n+1-i}, $1 \le i \le n$.

Let $F'[\pi^*]$ be the graph which results after making some label modifications on the flow-graph $F[\pi^*]$; a label modification attacker may be performs swapping of the labels of two nodes, altering the value of the label of a node, or even removing all the labels of the graph $F[\pi^*]$ resulting an unlabeled graph. Since the extraction of the watermark w relies on the labels of the flow-graph $F[\pi^*]$ (see algorithm Decode_RPG.to.SiP), it follows that our codec system $(encode, decode)_{F[\pi^*]}$ is susceptible to node modification attacks.

Thus, we are interested in finding a way to extract the watermark w efficiently from $F[\pi^*]$ without relying on its labels; for example, to extract w efficiently from the graph $F'[\pi^*]$. We show that, after any node-label modification attack on graph $F[\pi^*]$, we can efficiently reassign the initial labels to nodes of $F[\pi^*]$ using the structure of the unique Hamiltonian path HP$(F[\pi^*])$. More precisely, given the graph $F'[\pi^*]$ we can construct the flow-graph $F[\pi^*]$ in $O(n)$ time and space. In addition, if $F'[\pi^*]$ is the unlabeled graph of the flow-graph $F[\pi^*]$ we can also construct the graph $F[\pi^*]$ in $O(n)$ time and space.

The above properties imply that we are able to extract a watermark w in linear time from a modified or unlabeled flow-graph $F[\pi^*]$; this can be simply done by assigning labels to nodes of $F[\pi^*]$ just prior the use of the decoding algorithm Decode_RPG.to.SiP. Thus, we obtain the following result.

Lemma 3. *Let $F[\pi^*]$ be a reducible permutation graph of size $O(n)$ produced by the algorithm Encode_SiP.to.RPG and let $F'[\pi^*]$ be the graph resulting from $F[\pi^*]$ after modifying or deleting its node-labels. Given the graph $F'[\pi^*]$, the flow-graph $F[\pi^*]$ can be constructed in $O(n)$ time and space.*

5.2 Edge Modification

In this section we show that, given a reducible permutation graph $F[\pi^*]$ produced by our codec system we can decide with high probability whether the graph $F[\pi^*]$ suffer an attack on its edges.

Let $F[\pi^*]$ be a flow-graph which encodes the integer w and let $F'[\pi^*]$ be the graph resulting from $F[\pi^*]$ after an edge modification. Then, we say that $F'[\pi^*]$ is either in a F-incorrect-stage (False) or in a T-incorrect-stage (True): $F'[\pi^*]$ is in F-incorrect-stage if our codec system fails to return an integer w from the graph $F'[\pi^*]$, while $F'[\pi^*]$ is in T-incorrect-stage if our codec system extracts from the graph $F'[\pi^*]$ and returns an integer $w' \neq w$.

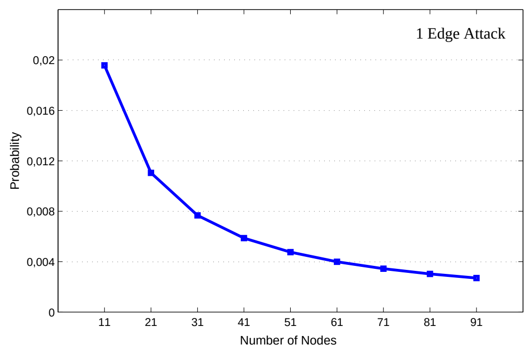

Figure 3: The probability for the RPG $F[\pi^*]$ to have the RPG property after a modification of 1 edge.

Let $F'[\pi^*]$ be the graph resulting from a flow-graph $F[\pi^*]$ after an edge modification and let $F'[\pi^*]$ be in a T-incorrect-stage. Then, the following properties hold:

(i) **RPG property**: $F'[\pi^*]$ is a directed graph on $n+2$ nodes $s, u_1, u_2, \dots, u_n, t$; node s (resp. t) has indegree 0 (resp. 1) and outdegree 1 (resp. 0), and each internal node u_i has outdegree exactly two, $1 \le i \le n$;

(ii) **SiP property**: The permutation π^* of length n produced by algorithm Decode_RPG.to.SiP is a self-inverting permutation (SiP). Thus, any single edge-modification can be easily identified;

(iii) **1-cycle property**: The SiP π^* contains only one 1-cycle. Thus, any random rearrangement of some elements of π^* can be identified with high probability (if we rearrange some elements of π^* it is unlike to produce a SiP with only one 1-cycle);

(iv) **Bitonic property**: The 1-cycle SiP π^* has the bitonic property. Recall that the self-inverting permutation π^* is constructed from the bitonic sequence $\pi^b = X||Y^R$, where X and Y are increasing subsequences (see, authors' work in (Chroni and Nikolopoulos, 2010; Chroni and Nikolopoulos, 2011)), and thus this property of π^b is encapsulated in the cycles of π^*. Thus an appropriate change to SiP π^* that keeps the 1-cycle SiP property may be identified during the decoding process by checking the subsequence Y (if a SiP permutation π^* has not been produced by our encoding algorithm Encode_W.to.SiP then subsequence Y may not be increasing).

The graph $F'[\pi^*]$ is in F-incorrect-stage if one of the above properties does not hold. Based on these properties, we next show that the malicious intentions of an attacker to lead a flow-graph $F[\pi^*]$ in F-incorrect-stage by modifying some of its edges can be detected with high probability.

We experimentally evaluated the resilience of the main component of our system, which is the flow-graph $F[\pi^*]$, in edge changes. To this end, we have produced RPG's $F[\pi^*]$ on $n = 11, 21, 31, \dots, 91$ nodes and computed the probability for the graph $F_i[\pi^*]$ to be in F-incorrect-stage, where $F_i[\pi^*]$ is the graph resulting from $F[\pi^*]$ after a modification of i edges, $1 \leq i \leq 4$. The experimental results show that we can decide with high probability whether the flow-graph $F[\pi^*]$ suffer an attack on its edges; Figures 3 and 4 depict the high-resilience structure of the graph $F[\pi^*]$.

6 CONCLUDING REMARKS

In this paper we proposed an efficient and easily implemented codec system for encoding watermark numbers as graph structures . In particular, we proposed an efficient codec method for encoding a watermark number w as self-inverting permutation π^* and then embedding it into a flow-graph $F[\pi^*]$; the proposed flow-graph $F[\pi^*]$ can be efficiently used for software watermarking.

Our codec algorithms are very simple, use elementary operations on sequences and linked structures, have very low time and space complexity, and the flow-graph $F[\pi^*]$ incorporates important structural properties which enable us to identify with high probability edge changes made by an attacker to $F[\pi^*]$.

In light of the two data components π^* and $F[\pi^*]$ of our codec system for software watermarking it would be very interesting to come up with new efficient codec algorithms and structures having "better" properties with respect to resilience to attacks; we leave it as an open question.

An interesting question for further investigation is whether the class of reducible permutation graphs can be extended so that it includes other classes of graphs with structural properties capable to efficiently encode watermark numbers.

Another interesting question with practical value is whether we can produce more than one reducible flow-graphs $F_1[\pi^*], F_2[\pi^*], \dots, F_n[\pi^*]$ which encode the same self-inverting permutation π^* using different subsequences or other relations on the elements of π^*; we also leave it as an open question.

Finally, the evaluation of our codec algorithms and structures under other software watermarking measurements in order to obtain detailed information about their practical behaviour is a interesting problem for future study.

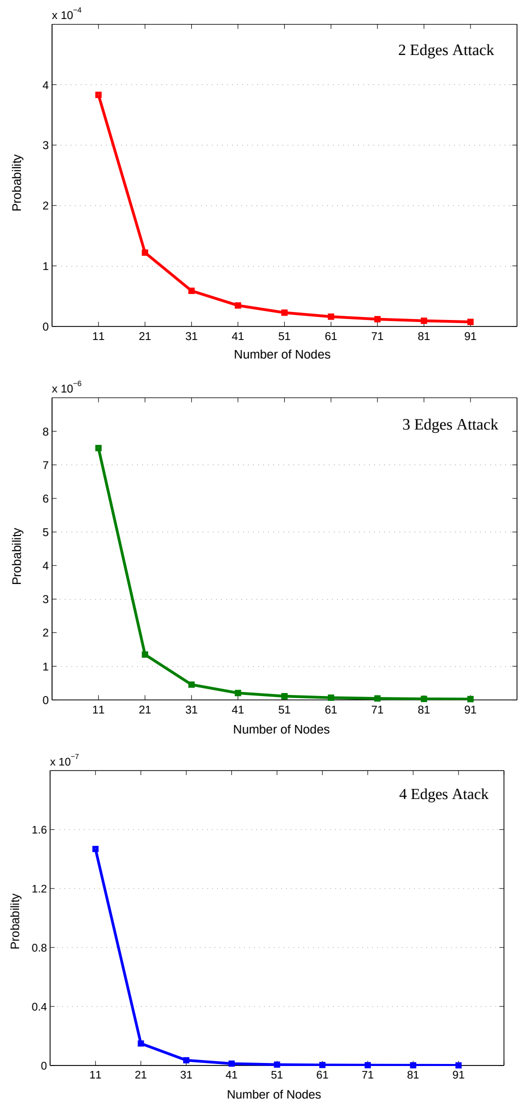

Figure 4: The probability for the RPG $F[\pi^*]$ to have the RPG property after a modification of 2 edges (top graph), 3 edges (middle graph) and 4 edges (bottom graph).

REFERENCES

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

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.

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 Nagra, J. (2010). *Surreptitious Software*. Addison-Wesley.

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.

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.

Curran, D., Hurley, N., and Cinneide, M. (2003). Securing java through software watermarking. In *Proc. Int'l Conference on Principles and Practice of Programming in Java (PPPJ'03)*, pages 145–148.

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

Grover, D. (1997). *The Protection of Computer Software - Its Technology and Applications*. Cambridge University Press, New York.

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

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

Monden, A., Iida, H., Matsumoto, K., Inoue, K., and Torii, K. (2000). A practical method for watermarking java programs. In *Proc. 24th Computer Software and Applications Conference (COMPSAC'00)*, pages 191–197.

Moskowitz, S. and Cooperman, M. (1996). Method for stegacipher protection of computer code. *US Patent*, 5.745.569.

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

Nagra, J. and Thomborson, C. (2004). Threading software watermarks. In *Proc. 6th Int'l Workshop on Information Hiding (IH'04)*, volume LNCS 3200, pages 208–223.

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

Stern, J., Hachez, G., Koeune, F., and Quisquater, J. (1999). Robust object watermarking: Application to code. In *Proc. 3rd Int'l Workshop on Information Hiding (IH'99)*, volume LNCS 1768, pages 368–378.

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.

Zhang, L., Yang, Y., Niu, X., and Niu, S. (2003). A survey on software watermarking. *Journal of Software*, 14:268–277.

Zhu, W., Thomborson, C., and Wang, F. (2005). A survey of software watermarking. In *Proc. IEEE Int'l Conference on Intelligence and Security Informatics (ISI'05)*, volume LNCS 3495, pages 454–458.