An Efficient Graph Codec System 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 this paper we propose an efficient and easily implemented codec system for encoding watermark numbers as reducible permutation flow-graphs. More precisely, in light of our recent encoding algorithms which encode a watermark value w as a self-inverting permutation π^*, we present an efficient algorithm which encodes a 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^*. The whole encoding process takes $O(n)$ time and space, where n is the binary size of the number w or, equivalently, the number of elements of the permutation π^*. We also propose efficient decoding algorithms which extract the permutation π^* from the reducible permutation flow-graph $F[\pi^*]$ within the same time and space complexity. The two main components of our proposed codec system, i.e., the self-inverting permutation π^* and the reducible permutation graph $F[\pi^*]$, incorporate important structural properties which make our codec system resilient to attacks.**

***Index Terms*—software watermarking; codec systems, self-inverting permutations; reducible permutation graphs; encoding/decoding algorithms; performance.**

I. Introduction

Software watermarking is a technique that is currently being studied to prevent or discourage software piracy and copyright infringement. The idea is similar to digital (or, media) watermarking where a unique identifier is embedded in image, audio, or video data through the introduction of errors not detectable by human perception [4], [10].

The *software watermarking problem* can be described as the problem of embedding a structure w into a program P such that w can be reliably located and extracted from P even after P has been subjected to code transformations such as translation, optimization and obfuscation [16].

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, threading, dynamic path techniques (see, [1], [8], [9], [15], [17]).

Recently, several software watermarking algorithms have been appeared in the literature that encode watermarks as graph structures (see Collberg and Nagra [4] for an exposition of the main results). 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 $(\texttt{encode}, \texttt{decode})$ along with the graph G, denoted by $(\texttt{encode}, \texttt{decode})_G$, as *graph codec system* [5].

In 1996, Davidson and Myhrvold [11] proposed the first software watermarking algorithm which is static and embeds the watermark 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 a static algorithm and is called VVS or GTW. In [18] the construction of a directed graph G (or, watermark graph G) is not discussed. Collberg et al. [6] 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) [5], which is a reducible control flow-graph [13], [14] with maximum out-degree of two, mimicking real code. Note that, for encoding integers the $\texttt{GTW}_{\texttt{sm}}$ method uses only those permutations that are self-inverting. The first dynamic watermarking algorithm (CT) was proposed by Collberg and Thomborson [7]; it embeds the watermark through a graph structure which is built on a heap at runtime.

Attacks: A successful attack against the watermarked program P_w prevents the recognizer from extracting the watermark while not seriously harming the performances or correctness of the program P_w. There are four main ways to attack a watermark in a software: (a) Additive attacks, (b) Subtractive attacks, (c) Distortive attacks, and (d) Recognition attacks: Modify or disable the watermark detector, or its inputs, so that it gives a misleading result. For example, an adversary may assert that "his" watermark detector is the one that should be used to prove ownership in a courtroom test.

Attacks against graph-based software watermarking algorithms can mainly occur in the following two ways: (i) Node-modification attacks, and (ii) Edges-modification attacks.

Our Contribution: Recently, we have presented two algorithms, namely Encode_W.to.SIP and Decode_SIP.to.W, for encoding an integer w into a self-inverting permutation π^* and extracting it from π^*; both algorithms run in $O(n)$ time, where n is the length of the binary representation of w [3].

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, we here describe an algorithm for encoding a self-inverting permutation into a directed graph structure having properties capable to match real program graphs. In particular, we propose 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^*. The whole encoding process takes $O(n)$ time and requires $O(n)$ space, where n is the length of the permutation π^*. We also propose the decoding algorithm `Decode_RPG.to.SIP`, which extract the self-inverting permutation π^* from the reducible permutation flow-graph $F[\pi^*]$ by converting first the graph $F[\pi^*]$ into a directed tree $T[\pi^*]$ and then applying DFS-search on $T[\pi^*]$. 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. We point out that the only operations used by the decoding algorithm are edge modifications on $F[\pi^*]$ and DFS-search on trees.

It is worth noting that our codec $(\mathtt{encode}, \mathtt{decode})_{F[\pi^*]}$ system incorporates several important properties which characterize it as an efficient and easily implemented software watermarking component. In particular, the reducible permutation 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. The function `Decode_RPG.to.SIP` is high insensitive to edge-changes and node-changes of $F[\pi^*]$. Moreover, the self-inverting permutation π^* captures important structural properties, due to the bitonic property used in the construction of π^*, which make our codec system resilient to attacks.

II. Preliminaries

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 $N_n = \{1, 2, \ldots, n\}$ [12].

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, $c = (x, x)$.

Let π be a permutation on N_n. We say that an element i of π *dominates* the element j if $i > j$ and $\pi_i^{-1} < \pi_j^{-1}$. An element i *directly dominates* (or, for short, didominates) the element j if i dominates j and there exists no element k in π such that i dominates k and k dominates j. For example, in $\pi = (6, 3, 2, 9, 8, 1, 11, 5, 4, 10, 7)$, the element 9 dominates the elements $8, 1, 5, 4, 7$ and it didominates the element 8.

Definition 2. The domination (resp. didomination) set dom(i) (resp. didom(i)) of the element i of a permutation π is the set of all the elements of π that dominate (resp. didominate) the element i.

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 $p \cap H = \{x\}$ (see, [13], [14]).

Definition 3. A flow-graph is reducible when it does not have a strongly connected subgraph with two (or more) entries.

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

III. Encode Self-inverting Permutations as Reducible Permutation Graphs

Having proposed an efficient method for encoding integers as self-inverting permutations [3], we next describe an algorithm for encoding a self-inverting permutation π^* into a reducible permutation graph $F[\pi^*]$. We also describe a decoding algorithm for extracting the permutation π^* from the graph $F[\pi^*]$.

A. Algorithm Encode_SIP.to.RPG

Given a self-inverting permutation π^* of length n our decoding algorithm works on two phases:

(I) it first uses a strategy to transform the permutation π^* into a directed acyclic graph $D[\pi^*]$ using certain combinatorial properties of the elements of π^*;

(II) then, it constructs a directed graph $F[\pi^*]$ on $n+2$ nodes using the adjacency relation of the nodes of $D[\pi^*]$.

Next, we first describe the main ideas behind the two phases of the encoding algorithm `Encode_SIP.to.RPG` (see, Figure 1).

Phase I: Construction of the DAG $D[\pi^*]$ from π^*: We construct the directed acyclic graph $D[\pi^*]$ by exploiting the didomination relation of the elements of π^*, as follows:

(i) for every element i of π^*, create a vertex v_i and add it in the vertex set $V(D[\pi^*]) = \{v_1, v_2, \ldots, v_n\}$;

(ii) compute the didomination relation of each element i of π^*; that is, the didomination set $didom(i)$ of the element i (see Definition 3);

(iii) for every pair of vertices (v_i, v_j) of the set $V(D[\pi^*])$ do the following: add the edge (v_i, v_j) in $E(D[\pi^*])$ if the element i didominates the element j in π^*;

(iv) create two dummy vertices $s = v_{n+1}$ and $t = v_0$ and add both in $V(D[\pi^*])$; then, add the edge (s, v_i) in $E(D[\pi^*])$, for every v_i with $\text{indeg}(v_i) = 0$, and the edge (v_i, t) in $E(D[\pi^*])$, for every v_i with $\text{outdeg}(v_i) = 0$.

Phase II: Construction of the RPG $F[\pi^*]$ from $D[\pi^*]$: We construct the directed graph $F[\pi^*]$ by exploiting the adjacency relation of the nodes of the dag $D[\pi^*]$, as follows:

(i) for every vertex v_i of $D[\pi^*]$, $0 \leq i \leq n+1$, create a node u_i and add it to $V(F[\pi^*])$; that is, $V(F[\pi^*]) = \{s = u_{n+1}, u_n, u_{n-1}, \ldots, u_1, , u_0 = t\}$;

(ii) for every pair of nodes (u_i, u_{i-1}) of the set $V(F[\pi^*])$ add the directed edge (u_i, u_{i-1}) in $E(F[\pi^*])$, $1 \leq i \leq n+1$; we call it *list pointer*;

(iii) for every vertex v_i of $D[\pi^*]$, $0 \leq i \leq n$, compute $p(v_i)$ to be the maximum-labeled node of the set $P(v_i) = \{v_{i_1}, v_{i_2}, \ldots, v_{i_k}\}$, where $(v_{i_\ell}, v_i) \in E(D[\pi^*])$, $1 \leq \ell \leq k$ and k = $\text{indeg}(v_i)$;

(iv) add the directed edge (u_m, u_i) in $E(F[\pi^*])$ if $(v_m, v_i) \in E(D[\pi^*])$, $1 \leq i \leq n+1$, and v_m is the maximum-labeled node of the set $P(v_i)$, that is, $p(v_i) = v_m$; we call it *max-didomitation pointer*;

Time and Space Performance. The most time- and space-consuming steps of the algorithm are the construction of the directed graph $D[\pi^*]$ (Steps I.i-I.iv) and the computation of the function p for each vertex $v_i \in V(D[\pi^*])$, $1 \leq i \leq n$ (Step II.iii); recall that $p(v_i)$ equals the maximum-labeled vertex v_m of the set $P(v_i)$ containing all the vertices of $D[\pi^*]$ which didominate vertex v_i. On the other hand, the construction of the reducible permutation flow-graph $F[\pi^*]$ (Steps II.ii and II.iv) requires only the list pointers, which can be trivially computed, and the max-didomitation pointers, which can be computed using the function p.

Looking at the permutation π^*, we observe that the element m which corresponds to vertex v_m of $D[\pi^*]$ is the max-indexed element on the left of the element i in π^* that is greater than i. Thus, the function p can be alternatively computed using the input permutation as follows:

(i) insert the element s with value $n+1$ into a stack S; *top_S* is the element on the top of the stack;

(ii) for each element $\pi_i \in \pi^*$, $i = 1, 2, \ldots, n$, do the following:

while *top_S* $< \pi_i$ remove the *top_S* from S;
$p(u_i) =$ *top_S*;
insert π_i in stack S;

Since each element of the input permutation π^* is inserted once in the stack S and is compered once with each new element the whole computation of the function p takes $O(n)$ time and space, where n is the length of the permutation π. Thus, we obtain the following result:

Theorem 1. *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.*

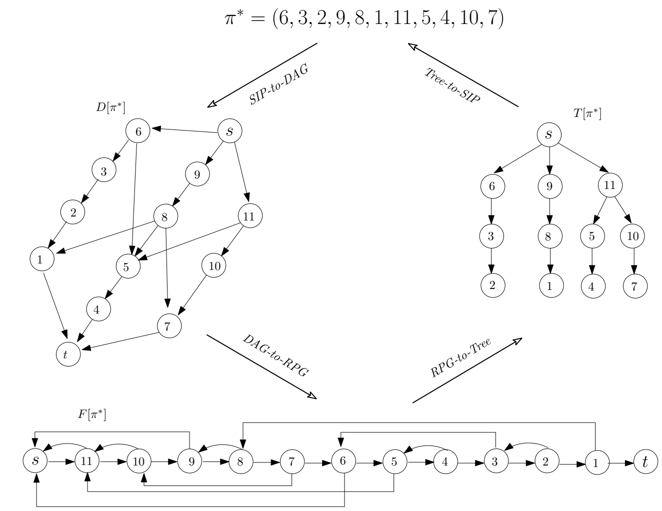

Fig. 1. The main structures used or constructed by the algorithms `Encode_SIP.to.RPG` and `Decode_RPG.to.SIP`; that is, the self-inverting permutation π^*, the dag $D[\pi^*]$, the reducible graph $F[\pi^*]$, and the tree $T[\pi^*]$

B. Algorithm Decode_RPG.to.SIP

Next, we present a decoding algorithm, we call it `Decode_RPG.to.SIP`, which takes as input a reducible permutation flow-graph $F[\pi^*]$ on $n+2$ nodes and produces a self-inverting permutation π^* of length n; it works as follows:

Algorithm `Decode_RPG.to.SIP`

1. Delete the directed edges (v_{i+1}, v_i) from $E(F[\pi^*])$, $1 \leq i \leq n$, and the node $t = v_0$ from $V(F[\pi^*])$;
2. Flip all the remaining directed edges of the graph $F[\pi^*]$; note that, flipping the directed edge (v_i, v_j) results the directed edge (v_j, v_i); Let $T[\pi^*]$ be the resulting tree and let $s = v_0, v_1, v_2, \ldots, v_n$ be the nodes of $T[\pi^*]$;
3. Perform DFS-search on tree $T[\pi^*]$ starting at node s by always proceeding to the min-labeled child and compute the DFS discovery time $d[v]$ of each node v of $T[\pi^*]$;
4. Order the nodes $s = v_0, v_1, v_2, \ldots, v_n$ of the tree $T[\pi^*]$ by their DFS discovery time $d[]$ and let $\pi = (v'_0, v'_1, v'_2, \ldots, v'_n)$ be the resulting order, where $d[v'_i] < d[v'_j]$ for $i < j$, $0 \leq i, j \leq n$;
5. Delete node s from the order π;
6. Return $\pi^* = \pi$;

Time and Space Performance. Our decoding algorithm `Decode_RPG.to.SIP` takes time and space linear in the size of the flow-graph $F[\pi^*]$, that is, $O(n)$; the only operations used by the algorithm are edge modifications on $F[\pi^*]$ and DFS-search on trees. Thus, the following theorem holds:

Theorem 2. *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.*

IV. System's Properties

In this section, we analyze the structures of the two main components of the proposed codec system, that is, the self-inverting permutation (SiP) π^* produced by the algorithm `Encode_W.to.SIP` and the reducible permutation graph $F[\pi^*]$ produced by the algorithm `Encode_SIP.to.RPG`, and present properties which make our codec system resilient to attacks.

A. Properties of permutation π^*

Our codec system encodes an integer w as a SiP π^* using a particular construction technique which captures into π^* important structural properties. These properties enable us to identify with hight probability edge-changes made by an attacker to flow-graph $F[\pi^*]$.

The main structural properties of our permutation π^* produced by the algorithm `Encode_W.to.SIP` are the following three:

- **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 a bitonic sequence (see, [3]), in such a way that the bitonic property is encapsulated in the cycles of π^*.

The above properties can be efficiently used in order to identify whether the graph $F[\pi^*]$ suffer an attack on its edges.

B. Properties of the Flow-graph $F[\pi^*]$

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) 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:

- **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 reducible permutation graph $F[\pi^*]$ has exactly one list out-pointer and exactly one max-didomination out-pointer, any single edge modification, i.e., edge-flip, edge-addition, or edge-deletion, will violate the out-pointer condition of some nodes, and thus the modified edge can be easily identified and corrected. Thus, the graph $F[\pi^*]$ enable us to correct single edge changes;
- **Efficient codecs**: The codec $(\texttt{encode}, \texttt{decode})_{F[\pi^*]}$ has low time and space complexity; indeed, we have showed (see Theorem 1 and Theorem 2) that the encoding algorithm `Encode_SIP.to.RPG` requires $O(n)$ time and space, where n is the size of the input permutation π^*, while the decoding algorithm `Decode_RPG.to.SIP` decodes the flowgraph $F[\pi^*]$ in $O(n)$ time and space.

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

2) Unique Hamiltonian Path: It has been shown that any reducible flow-graph has at most one Hamiltonian path [5]. 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 $F[\pi^*]$.

V. 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.

A. 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 $\{0, 1, \ldots, n+1\}$, 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}, $1 \leq i \leq n$.

Let $F'[\pi^*]$ be the graph which results after making some label modifications on the flow-graph $F[\pi^*]$. 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 $(\texttt{encode}, \texttt{decode})_{F[\pi^*]}$ is susceptible to node modification attacks.

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.

B. Edge Modification

We show that, given a reducible permutation graph $F[\pi^*]$ produced by our codec system (algorithms `Encode_W.to.SIP` and `Encode_SIP.to.RPG`), 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 in a T-incorrect-stage if the following properties hold:

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

(ii) **SiP property**: The permutation π^* of length n produced by algorithm `Decode_RPG.to.SIP` is a self-inverting permutation (SiP);

(iii) **1-cycle SIP property**: The SiP π^* contains only one 1-cycle;

(iv) **Bitonic property**: The 1-cycle SiP π^* has the bitonic property;

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 first show the resilience of the structure of the flow-graph $F[\pi^*]$ in edge changes. To this end, we have produced RPG's $F[\pi^*]$ on $n = 11, 21, 31, \ldots, 91$ 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, $1 \leq i \leq 4$. Figures 2 and 3 depict the high-resilience of the graph $F[\pi^*]$ in edge-changes.

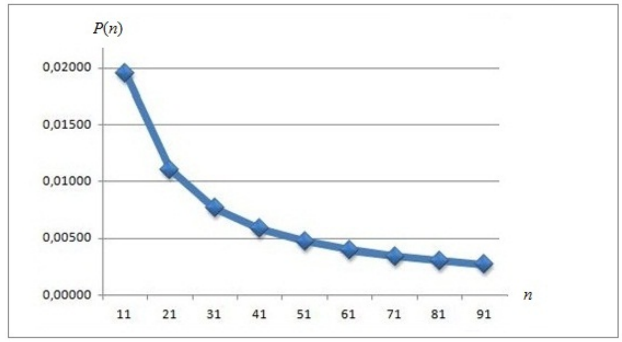

Fig. 2. The probability $P(n)$ for a RPG flow-graph $F[\pi^*]$ on n nodes to have the RPG property after a modification of 1 edge, for $n = 11, 21, \ldots, 91$.

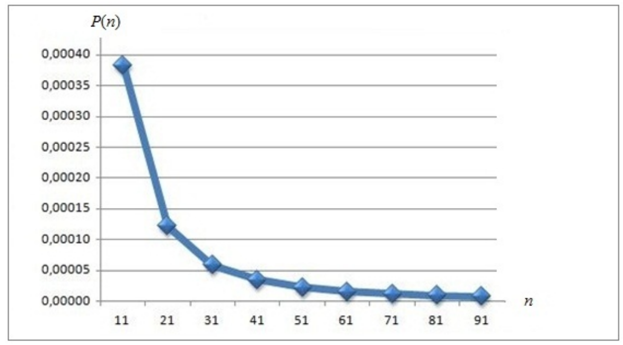

Fig. 3. The probability $P(n)$ for a RPG flow-graph $F[\pi^*]$ on n nodes to have the RPG property after a modification of 2 edges, for $n = 11, 21, \ldots, 91$.

We next consider the scenario where the attacker makes appropriate edge changes to RPG $F[\pi^*]$ so that the resulting graph $F'[\pi^*]$ still has the RPG property. Although the RPG property is maintained in $F'[\pi^*]$, the permutation π^* produced by our decoding algorithm `Decode_RPG.to.SIP` may contain one or more c-cycles ($c \geq 3$) or more than one 1-cycle, or it may not incorporate the bitonic property. In this case, the permutation π^* does not encapsulate one or more of the SiP, 1-cycle SiP, and Bitonic properties, and thus we can conclude that the flow-graph $F[\pi^*]$ has undergone an attack on its edges.

In order to obtain a clear view of the resilience of our codec system to edge attacks, we evaluate it in a simulation environment using the following scenarios:

(1) **Scenario S1**: The attacker knows that the graph $F[\pi^*]$ has the RPG property and makes appropriate edge changes so that it still has the RPG property. We want to compute the probability for the permutation π^* to have the SiP property;

(2) **Scenario S2**: The attacker knows that the graph $F[\pi^*]$ has the RPG property and also the permutation π^* has the SiP property, and makes appropriate edge changes so that $F[\pi^*]$ and π^* still have the RPG and SiP properties, respectively. We want to compute the probability for the SiP π^* to have the 1-cycle SiP property;

(3) **Scenario S3**: The attacker knows that $F[\pi^*]$ has the RPG property and the permutation π^* has the 1-cycle SiP property, and makes appropriate edge changes so that $F[\pi^*]$ and π^* still have the RPG and 1-cycle SiP properties, respectively. We want to compute the probability for the 1-cycle SiP π^* to have the Bitonic property;

We have computed the probability of each scenario by designing the following experiments:

S1: We produce 1.000.000 SiPs of length n, we randomly permute 4 elements in each permutation, and then count the number of the resulting permutations that are still SiPs;

S2: We produce 1.000.000 1-cycle SiPs of length n, we randomly permute 4 elements in each permutation, and then count the number of the resulting permutations that are still 1-cycle SiPs;

S3: We produce 1.000.000 bitonic 1-cycle SiPs of length n, we randomly permute 4 elements in each permutation, and then count the number of the resulting permutations that are still bitonic 1-cycle SiPs;

where, $n = 9, 19, 29, \ldots, 89$.

The experimental results of the above three scenarios are

TABLE I
EXPERIMENTAL RESULTS FOR S1, S2, AND S3 SCENARIOS.

	9	19	29	39	49	59	69	79	89
S1	0,0475	0,0093	0,0037	0,0020	0,0012	0,0009	0,0007	0,0004	0,0003
S2	0,0315	0,0062	0,0025	0,0014	0,0008	0,0006	0,0004	0,0003	0,0002
S3	0,0317	0,0061	0,0025	0,0013	0,0008	0,0006	0,0005	0,0003	0,0002

showed in Table 1. The fist row gives the probability for a SiP π^* of length $9, 19, 29, \ldots, 89$ to remain SiP after permuting 4 elements of π^* (Scenario S1). The second and the third rows give similar results for a 1-cycle SiP and bitonic 1-cycle SiP permutations (Scenarios S1 and S2, respectively).

VI. DISCUSSION

Collberg et al. [5], [7] describe several techniques for encoding watermark integers in graph structures among which an RPG structure; hereafter, we shall refer to their RPG structure as $R[\pi^*]$.

Based on the fact that there is a one-to-one correspondence between self-inverting permutations and isomorphism classes of RPGs, Collberg et al. [5] proposed a polynomial algorithm for encoding any integer w as $R[\pi]$ 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; it does not incorporate any other structural property.

On the other hand, in our codec system an integer w is encoded as self-inverting permutation π^* using a particular construction technique which incorporates into π^* important structural properties. These properties enable us to identify with hight probability edge-changes made by an attacker to flow-graph $F[\pi^*]$; indeed, based on the SiP, 1-cycle, or the Bitonic property we can easily identify with high probability any edge-modification on $F[\pi^*]$.

In addition, since each node in our graph $F[\pi^*]$ has exactly one list out-pointer and exactly one max-didomination out-pointer, any single edge modification, i.e., edge-flip, edge-addition, or edge-deletion, will violate the out-pointer condition of some nodes, and thus the modified edge can be easily identified and corrected. Thus, the graph $F[\pi^*]$ enable us to correct single edge changes. Note that, the graph $R[\pi]$ does not incorporate such a degree property.

It is worth noting that our codec system has low time and space complexity; indeed, we have showed that our encoding and decoding algorithms require time and space linear in the size of the input permutation π^*. The codec system in [5] has polynomial time performance.

Summarizing, 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 make it resilient to attacks by enabling us to identify edge-changes made by an attacker to $F[\pi^*]$. Thus, in light of these properties we could propose $F[\pi^*]$ as a good choice for encoding watermark numbers for practical purposes.

VII. CONCLUDING REMARKS

The evaluation of our codec system in a simulation environment on other types of attacks in order to obtain a more clear view of their practical behavior is a problem for future investigation.

Finally, designing and testing a model for embedding the watermark flow-graph $F[\pi^*]$ into an application program P is also a problem for future investigation (see, [2]).

REFERENCES

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

[2] M. Chroni and S.D. Nikolopoulos, “An embedding graph-based model for software watermarking,” Proc. 8th Int'l Conference on Intelligent Information Hiding and Multimedia Signal Processing (IIH-MSP'12), IEEE Proceedings, 2012.

[3] M. Chroni and S.D. Nikolopoulos, “Encoding watermark integers as self-inverting permutations,” 11th Int'l Conference on Computer Systems and Technologies (CompSysTech'10), ACM ICPS 471, pp. 125–130, 2010.

[4] C. Collberg and J. Nagra, Surreptitious Software, Addison-Wesley, 2010.

[5] 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.

[6] 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.

[7] C. Collberg and C. Thomborson, “Software watermarking: models and dynamic embeddings,” Proc. 26th ACM SIGPLAN-SIGACT Symp. on Principles of Program. Languages (POPL'99), pp. 311–324, 1999.

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

[9] 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.

[10] I. Cox, J. Kilian, T. Leighton, and T. Shamoon, “A secure, robust watermark for multimedia,” Proc. 1st Int'l Workshop on Information Hiding, LNCS 1174, pp. 317–333, 1996.

[11] 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.

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

[13] M.S. Hecht and J.D. Ullman, “Flow graph reducibility,” SIAM J. Computing 1, pp. 188–202, 1972.

[14] M.S. Hecht and J.D. Ullman, “Characterizations of reducible flow graphs,” Journal of the ACM 21, pp. 367–375, 1974.

[15] 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.

[16] G. Myles and C. Collberg, “Software watermarking via opaque predicates: Implementation, analysis, and attacks,” Electronic Commerce Research 6, pp. 155–171, 2006.

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

[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.