Efficient Encoding of Watermark Numbers as Reducible Permutation Graphs

Maria Chroni and Stavros D. Nikolopoulos

Department of Computer Science, University of Ioannina,
GR-45110, Ioannina, Greece.
{mchroni,stavros}@cs.uoi.gr

Abstract. In a software watermarking environment, several graph theoretic watermark methods use numbers as watermark values, where some of these methods encode the watermark numbers as graph structures. In this paper we extended the class of error correcting graphs by proposing an efficient and easily implemented codec system for encoding watermark numbers as reducible permutation flow-graphs. More precisely, we first present an efficient algorithm which encodes a watermark number w as self-inverting permutation π^* and, then, an algorithm 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 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 number w 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 cause them resilience to attacks.

1 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 [12]. The *software watermarking problem* can be described as the problem of embeding 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 [21]. 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: $\texttt{embed}(P, w, k) \rightarrow P'$ and $\texttt{extract}(P', k) \rightarrow w$.

Although digital watermarking has made considerable progress and become a popular technique for copyright protection of multimedia information [12, 28], research on software watermarking has recently received sufficient attention. The

patent by Davidson and Myhrvold [13] presented the first published software watermarking algorithm. The preliminary concepts of software watermarking also appeared in paper [16] and patents [19, 26]. Collberg et al. [7, 8] presented detailed definitions for software watermarking. Authors of papers [30, 31] have given brief surveys of software watermarking research.

Static and Dynamic Watermarking Algorithms: There are two general categories of watermarking algorithms namely *static* and the *dynamic* algorithms [7]. A static watermark is stored inside program code in a certain format, and it does not change during the program execution. A dynamic watermark is built during program execution, perhaps only after a particular sequence of input. It might be retrieved by analyzing the data structures built when watermarked program is running. In other cases, tracing the program execution may be required. Further discussion of static and/or dynamic watermarking issues can be found in [13, 19, 29].

Algorithms and Techniques for Software Watermarking: 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, [2, 4, 10, 11, 20, 22, 24, 27]).

Recently, several software watermarking algorithms have been appeared in the literature that encode watermarks as graph structures. In general, such encodings make use of an encoding function `encode` which converts a watermarking number w into a graph G, $encode(w) \rightarrow G$, and also of a decoding function `decode` that converts the graph G into the number w, $decode(G) \rightarrow w$; we usually call the pair $(\texttt{encode}, \texttt{decode})$ as *graph codec* [5]. 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 the following properties:

- 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 $decode(G) \rightarrow w$ then $decode(G') \rightarrow w$ with $G' \approx G$;
- Small Size: The size $|P_w| - |P|$ of the embedded watermark should be small;
- Efficient Codecs: The functions `encode` and `decode` should be computed in polynomial time.

In 1996, Davidson and Myhrvold [13] 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 [29] 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 [29] 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 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. It is generally assumed that the attacker has access to the algorithm used by the embedder and recognizer. There are four main ways to attack a watermark in a software.

- Additive attacks: Embed a new watermark into the watermarked software, so the original copyright owners of the software cannot prove their ownership by their original watermark inserted in the software;
- Subtractive attacks: Remove the watermark of the watermarked software without affecting the functionality of the watermarked software;
- Distortive attacks: Modify watermark to prevent it from being extracted by the copyright owners and still keep the usability of the software;
- 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 three ways: (i) Edge-flip attacks, (ii) Edges-addition/deletion attacks, and (iii) Node-addition/deletion 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, we first present an efficient algorithm which encodes a number (integer) w as self-inverting permutation π^*. Our algorithm, which we call Encode_W-to-SIP, takes as input an integer w, computes first its binary representation $b_1 b_2 \cdots b_n$, then constructs a bitonic permutation on $2n+1$ numbers, and finally produces a self-inverting permutation π^* of length $2n+1$ in $O(n)$ time and space. We also present a decode algorithm which extracts the integer w from the self-inverting permutation π^* within the same time and space complexity; we call the decode algorithm Decode_SIP-to-W.

Having designed an efficient method for encoding integers as self-inverting permutations, we next 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 an efficient and easily implemented algorithm, the 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 $(\texttt{encode}, \texttt{decode})_{F[\pi^*]}$ 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 small edge-changes and quite insensitive to small node-changes of $F[\pi^*]$, and the graph $F[\pi^*]$ unable us to correct such edge changes. Moreover, the self-inverting permutation π^* captures important structural properties, due to bitonic property used in the construction of π^*, which cause them resilience to attacks.

Finally, we point out that our codec $(\texttt{encode}, \texttt{decode})_{F[\pi^*]}$ 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 $O(n)$.

2 Preliminaries

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. The *neighborhood* $N(x)$ of a vertex x of the graph G is the set of all the vertices of G which are adjacent to x. The *degree* of a vertex x in the graph G, denoted $deg(x)$, is the number of edges incident on x; thus, $d(x) = |N(x)|$. 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 $(v_0, v_k) \in E(G)$.

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, $\pi_4^{-1} = 2$, $\pi_7^{-1} = 4$, $\pi_3^{-1} = 6$, ect [15].

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.

Let π be a permutation on $N = \{1, 2, \ldots, n\}$. We say that an element i of the permutation 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 [23]. For example, in the permutation $\pi = (8, 3, 2, 7, 1, 9, 6, 5, 4)$, the element 7 dominates the elements $1, 6, 5, 4$ and it directly dominates the elements $1, 6$.

Definition 3: The domination (resp. didomination) set $\mathrm{dom}(i)$ (resp. $\mathrm{didom}(i)$) of the element i of a permutation π is the set of all the elements of π that dominate (resp. didominate) the element i.

Definition 4: An undirected graph G with vertices numbered from 1 to n; that is, $V(G) = \{1, 2, \ldots, n\}$, is called a *permutation graph* if there exists a permutation $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$ on N_n such that, $(i, j) \in E(G)$ if and only if $(i - j)(\pi_i^{-1} - \pi_j^{-1}) < 0$.

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

3 Encode Watermark Numbers as Self-inverting Permutations

In this section, we first introduce the notion of *Bitonic Permutations* and then we present 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^*. Both algorithms run in $O(n)$ time, where n is the length of the binary representation of the integer w [3].

3.1 Bitonic Permutations

The key-object in our algorithm for encoding integers as self-inverting permutations is the bitonic permutation: 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. For example, the permutations $\pi_1 = (1, 4, 6, 7, 5, 3, 2)$ and $\pi_2 = (6, 4, 3, 1, 2, 5, 7)$ are both bitonic [3].

In this paper, we consider only bitonic permutations that monotonically increases and then monotonically decreases. Let $\pi = (\pi_1, \pi_2, \ldots, \pi_i, \pi_{i+1}, \ldots, \pi_n)$ be such a bitonic permutation over the set N_n and let π_i, π_{i+1} be the two consecutive elements of π such that $\pi_i > \pi_{i+1}$. Then, the sequence $X = (\pi_1, \pi_2, \ldots, \pi_i)$ is called first increasing subsequence of π and the sequence $Y = (\pi_{i+1}, \pi_{i+2}, \ldots, \pi_n)$ is called first decreasing subsequence of π.

We next give some notations and terminology we shall use throughout the paper. Let w be an integer number. We denote by $B = b_1 b_2 \cdots b_n$ the binary representation of w. If $B_1 = b_1 b_2 \cdots b_n$ and $B_2 = d_1 d_2 \cdots d_m$ be two binary numbers, then the number $B_1 || B_2$ is the binary number $b_1 b_2 \cdots b_n d_1 d_2 \cdots d_m$. The binary sequence of the number $B = b_1 b_2 \cdots b_n$ is the sequence $B^* = (b_1, b_2, \ldots, b_n)$ of length n.

Let $B = b_1 b_2 \cdots bn$ be a binary number. Then, $flip(B) = b'_1 b'_2 \cdots b'_n$ is the binary number such that $b'_i = 0$ (1 resp.) if and only if $b_i = 1 (0 resp.), 1 \leq i \leq n$.

3.2 Algorithm Encode_W-to-SIP

In this section, we present an algorithm for encoding an integer as self-inverting permutation. In particular, our algorithm takes as input an integer w, computes

the binary representation $b_1 b_2 \cdots b_n$ of w, and then produces a self-inverting permutation π^* in $O(n)$ time. We next describe the proposed algorithm:

Algorithm `Encode_W-to-SIP`

1. Compute the binary representation $B = b_1 b_2 \cdots b_n$ of w;
2. Construct the binary number $B' = 00 \cdots 0 || B || 1$ of length $2n+1$, and then the binary sequence $B^* = (b_1, b_2, \ldots, b_{n'})$ of $flip(B')$;
3. Find the sequence $X = (x_1, x_2, \ldots, x_k)$ of the $0's$ positions and the sequence $Y = (y_1, y_2, \ldots, y_m)$ of the $1's$ positions in B^* from left-to-right;
4. Construct the bitonic permutation $\pi^b = X || Y^R$ on $n' = 2n+1$ numbers; let $\pi^b = (x_1, x_2, \ldots, x_k, y_m, y_{m-1}, \ldots, y_1)$
5. Set $(\pi_1, \pi_2, \ldots, \pi_k, \pi_{k+1}, \pi_{k+2}, \ldots, \pi_{n'}) = (x_1, x_2, \ldots, x_k, y_m, y_{m-1}, \ldots, y_1)$, $i = 1$ and $j = n'$;
   while $i < j$ do the following:
   construct the 2-cycle $c_i = (\pi_i, \pi_j)$, and set $i = i + 1$ and $j = j - 1$;
   end-while;
   if $i = j$ then construct the 1-cycle $c_i = (\pi_i)$;
6. Construct the permutation $\pi^* = (\pi_1, \pi_2, \ldots, \pi_{n'})$ on $n' = 2n+1$ numbers such that $\pi_i = i$, $1 \leq i \leq n'$;
7. Let C be the set of all cycles computed at step 5;
   for each 2-cycle $(\pi_i, \pi_j) \in C$ set $\pi_{\pi_i} = \pi_j$ and $\pi_{\pi_j} = \pi_i$;
8. Return the self-inverting permutation π^*;

Example 1: Let $w = 12$ be the input watermark integer in the algorithm `Encode_W-to-SIP`. We first compute the binary representation $B = 1100$ of the number 12; then we construct the binary number $B' = 000011001$ and the binary sequence $B^* = (1, 1, 1, 1, 0, 0, 1, 1, 0)$ of $flip(B')$; we compute the sequences $X = (5, 6, 9)$ and $Y = (1, 2, 3, 4, 7, 8)$, and then construct the bitonic permutation $\pi = (5, 6, 9, 8, 7, 4, 3, 2, 1)$ on $n' = 9$ numbers; since $n' = 9$ odd, we select 4 pairs $(5, 1)$, $(6, 2)$, $(9, 3)$, $(8, 4)$ and the number 7 and then construct the self-inverting permutation $\pi^* = (5, 6, 9, 8, 1, 2, 7, 4, 3)$.

Time and Space Complexity. The encoding algorithm `Encode_W-to-SIP` performs basic operations on sequences of lengths $O(n)$, where n is the number of bits in the binary representation of w (see Figure 1); hereafter, for the number n we shall call the term *binary size* of the integer w. Moreover, all the operations are executed in place, i.e., the algorithm uses no additional space except of a constant number of variables. It is easy to see that the whole encoding process requires $O(n)$ time and space. Thus, the following theorem holds:

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.*

3.3 Algorithm Decode_SIP-to-W

Next, we present an extraction algorithm, that is, an algorithm for decoding a self-inverting permutation. More precisely, our extraction algorithm, which we call Decode_SIP-to-W, takes as input a self-inverting permutation π^* produced by Algorithm Encode_W-to-SIP and returns its corresponding integer w. The time complexity of the decode algorithm is also $O(n)$, where n is the length of the permutation π^*. We next describe the proposed algorithm:

Algorithm `Decode_SIP-to-W`

1. Compute the increasing cycle representation $C = (c_1, c_2, \ldots, c_k)$ of the self-inverting permutation $\pi^* = (\pi_1, \pi_2, \ldots, \pi_{n'})$, where $n' = 2n + 1$, that is, $c_1 \prec c_2 \prec \cdots \prec c_k$;
2. Set $i = 1$ and $j = n'$:
3. Construct the permutation π^b of length n' as follows:

   while the set C is not empty, do the following:

   Select the minimum element c of the sequence C;
   Case 1: the selected cycle c has length 2 and let $c = (a, b)$:
   $\pi_i = b$ and $\pi_j = a$;
   $i = i + 1$ and $j = j - 1$;
   Case 2: the selected cycle c has length 1 and let $c = (a)$:
   $\pi_i = a$ and $i = i + 1$;
   Remove the cycle c from C;
4. Find the first increasing subsequence $X = (\pi_1, \pi_2, \ldots, \pi_k)$ and then the decreasing subsequence $Y = (\pi_{k+1}, \pi_{k+2}, \ldots, \pi_{k'})$ of π;
5. Construct the binary sequence $B^* = (b_1, b_2, \ldots, b_{n'})$ as follows:
   set 0's in positions $\pi_1, \pi_2, \ldots, \pi_k$ and 1's in positions $\pi_{k+1}, \pi_{k+2}, \ldots, \pi_{k'}$;
6. Compute $B' = flip(B^*) = (b_1, b_2, \ldots, b_n, b_{n+1}, \ldots, b_{n'-1}, b_{n'})$;
7. Return the integer w of the binary number $B = b_{n+1} b_{n+2} \cdots b_{n'-1}$;

Example 2: Let $\pi^* = (5, 6, 9, 8, 1, 2, 7, 4, 3)$ be a self-inverting permutation produced by the algorithm `Encode_W-to-SIP`. The cycle representation of π^* is the following: $(1, 5)$, $(2, 6)$, $(3, 9)$, $(4, 8)$, (7); from the cycles we construct the permutation $\pi = (5, 6, 9, 8, 7, 4, 3, 2, 1)$; then, we compute first increasing subsequence $X = (5, 6, 9)$ and the first decreasing subsequence $Y = (8, 7, 4, 3, 2, 1)$; we then construct the binary sequence $B^* = (1, 1, 1, 1, 0, 0, 1, 1, 0)$ of length 9; we flip the elements of B^* and construct the sequence $B' = (0, 0, 0, 0, 1, 1, 0, 0, 1)$; the binary number 1100 is the integer $w = 12$.

Time and Space Complexity. It is easy to see that the decoding algorithm `Decode_SIP-to-W` performs the same basic operations on sequences of lengths $O(n)$ as the encoding algorithm (see Figure 1). Thus, we obtain the following result:

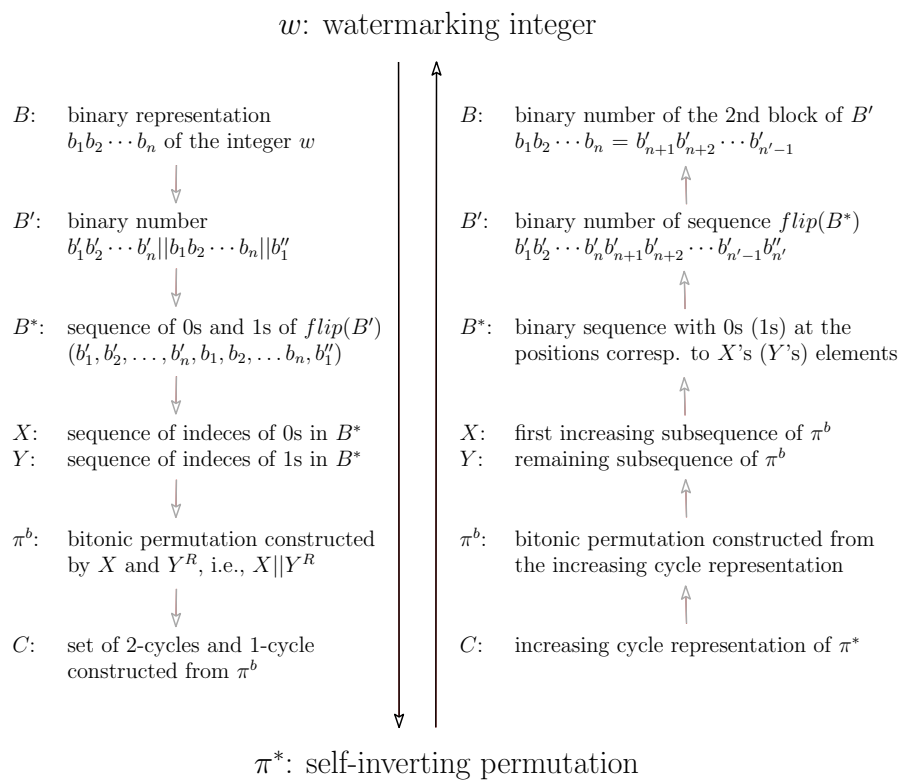

Fig. 1. The main data components used by the algorithms `Encode_W-to-SIP` and `Decode_SIP-to-W`

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.*

4 Encode Self-inverting Permutations as Reducible Permutation Graphs

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

4.1 Algorithm Encode_SIP-to-RPG

We next propose the algorithm `Encode_SIP-to-RPG` which takes as input the self-inverting permutation π^* produced by the algorithm `Encode_W-to-SIP` and constructs a reducible permutation flow-graph $F[\pi^*]$ by using an efficient DAG representation of the self-inverting permutation π^*. The whole encoding process takes $O(n)$ time and requires $O(n)$ space, where n is the length of the input permutation π^*.

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 the dag $D[\pi^*]$.

Next, we first describe the main ideas behind the two phases and then we present in details the whole algorithm.

Construction of the DAG $D[\pi^*]$ from the permutation 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 to the vertex set $V(D[\pi^*])$;

(ii) compute the didomination relation of each element i of π^*; recall that the didomination set $didom(i)$ of the element i contains all the elements j of π^* that are didominated by 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 and t and add both in $V(D[\pi^*])$; then, add the edge (s, v_i) in $E(D[\pi^*])$, for every v_i with $indeg(v_i) = 0$, and the edge (v_i, t) in $E(D[\pi^*])$, for every v_i with $outdeg(v_i) = 0$.

Construction of the RPG $F[\pi^*]$ from the graph $D[\pi^*]$: 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 $D[\pi^*]$, $1 \leq i \leq n$, create a node u_i and add it to $V(F[\pi^*])$; create the nodes u_{n+1} and u_0 and add them to $V(F[\pi^*])$; note that, the nodes u_{n+1} and u_0 correspond to s and t, respectively;

(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 $E(F[\pi^*])$, $1 \leq i \leq n+1$;

(iii) add the directed edge (u_i, u_j) in $E(F[\pi^*])$ if $(v_i, v_j) \in E(D[\pi^*])$, $1 \leq i \leq n+1$, and v_i is the maximum-labeled element of the set $\{v_{i_1}, v_{i_2}, \ldots, v_{i_{indeg(i)}}\}$, where $(v_{i_k}, v_j) \in E(D[\pi^*])$, $1 \leq k \leq indeg(i)$.

Algorithm `Encode_SIP-to-RPG`

1. Construct a directed acyclic graph (dag) $D[\pi^*]$ on n vertices as follows:
   - $V(D[\pi^*]) = \{v_1, v_2, \ldots, v_n\}$;
   - compute the set $didom(i)$ of each element i in π^*, $1 \leq i \leq n-1$;
   - for each $j \in didom(i)$, add the edge (v_i, v_j) in $E(D[\pi^*])$;
   - add two dummy vertices $s = v_{n+1}$ and $t = v_0$ in $V(D[\pi^*])$;
   - add $(s, v_i) \in E(D[\pi^*])$, for every v_i with $indeg(v_i) = 0$;
   - add $(v_i, t) \in E(D[\pi^*])$, for every v_i with $outdeg(v_i) = 0$;
2. For each vertex $v_i \in V(D[\pi^*])$, $1 \leq i \leq n$, do
   - compute the set $P(v_i) = \{v_j \in V(D[\pi^*]) | (v_j, v_i) \in E(D[\pi^*])\}$;
   - select the maximum-labeled vertex v_m from $P(v_i)$;
   - set $p(v_i) = v_m$;
3. Construct a directed graph $F[\pi^*]$ on $n+2$ vertices, as follows:
   - $V(F[\pi^*]) = \{t = u_0, u_1, \ldots, u_n, u_{n+1} = s\}$;
   - for $i = n$ downto 0 do
     add the edge (u_{i+1}, u_i) in $E(F[\pi^*])$; we call it *list pointer*;
4. For each vertex $u_i \in V(F[\pi^*])$, $1 \leq i \leq n$, do
   - add the edge (u_i, u_m) in $E(F[\pi^*])$ if $v_m = p(v_i)$;
     we call it *max-didomitation pointer*;
5. Return the graph $F[\pi^*]$;

Time and Space Complexity. The most time- and space-consuming steps of the algorithm are the construction of the directed graph $D[\pi^*]$ (Step 1) and the computation of the function p for each vertex $v_i \in V(D[\pi^*])$, $1 \leq i \leq n$ (Step 2; 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 3 and 4) 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* $< \pi_i$ do
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

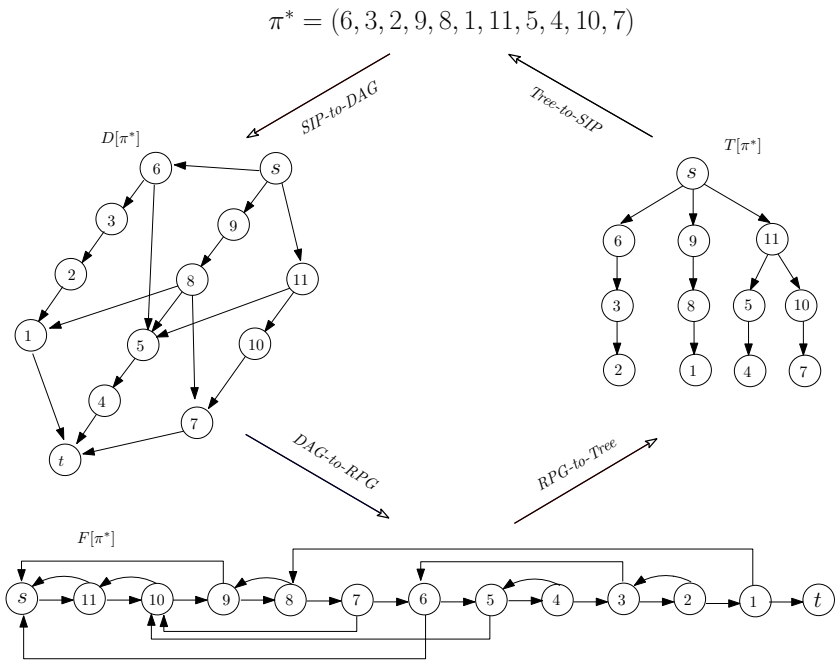

Fig. 2. The main structures used or constructed by the algorithms `Encode_SIP-to-RPG` and `Decode_RPG-to-SIP`; that is, the self-inverting permutation π^*, the dag $D[\pi^*]$, the reducible graph $F[\pi^*]$, and the tree $T[\pi^*]$

function p takes $O(n)$ time and space, where n is the length of the permutation π. Thus, we obtain the following result:

Theorem 3. *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.*

4.2 Algorithm Decode_RPG-to-SIP

The algorithm `Encode_SIP-to-RPG` produces reducible permutation flow-graph $F[\pi^*]$ in which it encodes a self-inverting permutation π^*. Thus, we are interested in designing an efficient and easily implemented algorithm for extracting the permutation π^* from the graph $F[\pi^*]$.

Next, we present such a decoding algorithm, we call it `Decode_RPG-to-SIP`, which is efficient: it takes time and space linear, i.e., $O(n)$, in the size of the flow-graph $F[\pi^*]$, and easily implemented: the only operations used by the algorithm are edge modifications on $F[\pi^*]$ and DFS-search on trees.

The algorithm 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 the edge set $E(F[\pi^*])$, $1 \leq i \leq n$, and the node $t = v_0$ from $V(F[\pi^*])$;
2. Flip all the remaining directed edges of the graph $F[\pi^*]$; the resulting graph is a tree $T[\pi^*]$;
3. Perform DFS-search on tree $T[\pi^*]$ starting at node s by always proceeding to the minimum-labeled child node;
4. Order the nodes $s, v_1, v_2, \ldots, v_n$ of the tree $T[\pi^*]$ by their DFS discovery time $d[]$ and let $\pi = (v_{d[0]}, v_{d[1]}, v_{d[2]}, \ldots, v_{d[n]})$, where $v_{d[0]} = s$ and $d[0] < d[1] < \ldots < d[n]$;
5. Delete node s from the order π;
6. Return $\pi^* = \pi$;

Time and Space Complexity. The size of the reducible permutation graph $F[\pi^*]$ constructed by the algorithm `Encode_SIP-to-RPG` is $O(n)$, where n is the length of the permutation π^*, and thus the size of the resulting tree $T[\pi^*]$ is also $O(n)$. It is well known that the DFS-search on the tree $T[\pi^*]$ takes time linear in the size of $T[\pi^*]$. Thus, the decoding algorithm is executed in $O(n)$ time using $O(n)$ space. Thus, the following theorem holds:

Theorem 4. *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.*

5 Properties and Attacks

In this section, we analyze the structures of the two main components of our proposed codec, that is, the self-inverting permutation 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 discuss their properties with respect to resilience to attacks.

5.1 Properties of permutation π^*

Collberg et al. [7, 5] describe several techniques for encoding watermark integers in graph structures. 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 the RPG corresponding to the wth self-inverting permutation π in this correspondence. In

this encoding the self-inverting permutation has no any other property except that it is its own inverse.

In our codec system proposed in this paper an integer w is encoded as self-inverting permutation π^* using a particular construction technique which captures into π^* important structural properties. These properties unable us to identify any single change (in some cases, multiple 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:

- **Length property**: By construction the self-inverting permutation π^* has always odd length. Thus, any single node-modification, i.e., adding an element in π^* or deleting an element from π^*, can be easily identified;
- **Bitonic property**: Algorithm `Decode_SIP-to-W` decodes the self-inverting permutation π^* to obtain the encoded integer w. During the decoding process two sequences are constructed, that is, the increasing subsequence X and the decreasing subsequence Y (see Step 4), which incorporate the bitonic property of the encoding process. If the permutation π^* has not been produced by our encoding algorithm `Encode_W-to-SIP` then subsequence Y may not be increasing. Thus, an appropriate change to SIP π^* that keeps the SIP property may be identified by checking the subsequence Y;
- **Block property**: The algorithm `Encode_W-to-SIP` takes the binary representation of the integer w and constructs the number B' (see Step 2). The binary representation of B' consists of three parts (or, blocks): (i) the first part contains the first n bits with 0s values, (ii) the second part contains the next n bits which forms the binary representation of the integer w, and (iii) the third part of length one contains a bit 1. This property is encapsulated in the structure of π^* in such a way that during the decoding process the binary sequence B' constructed in Step 6 of the decoding algorithm `Decode_SIP-to-W` is identical to the sequence B' constructed by the encoding algorithm `Encode_W-to-SIP`. If an attacker make appropriate changes to SIP π^* so that the resulting permutation π^* still has the SIP property, then the first block of the binary sequence B' may contain one or more 1s or the third block may be 0.

5.2 Properties of graph $F[\pi^*]$

The reducible permutation graph $F[\pi^*]$ consists of the following three components:

(1) **A header node**: it is a root node with outdegree one from which every other node in the graph $F[\pi^*]$ is reachable. Note that, every control flow-graph has such a node. In the graph $F[\pi^*]$ the header node is denoted by s;

(2) **A footer node**: it is a node with outdegree zero that is reachable from every other node of the graph. Every control flow-graph has such a node, representing the method exit. In the graph $F[\pi^*]$ the footer node is denoted by t;

(3) **A linked list**: 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}, which we call *list pointer*, and the other points to node u_m, which we call *max-didomination pointer*, where $m > i$; note that, $u_m > u_i > u_{i-1}$.

For graph-based algorithms, the watermark is encoded into a graph G in some special kind of graphs. 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 proposed reducible permutation graph $F[\pi^*]$ and a corresponding codec $(\texttt{encode}, \texttt{decode})_{F[\pi^*]}$ have 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 outpointer and exactly one max-didom 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. Thus, the graph $F[\pi^*]$ unable us to correct single edge changes;
- **Small size**: The size $|P_w| - |P|$ of the embedded watermark is small;
- **Efficient codecs**: The codec $(\texttt{encode}, \texttt{decode})_{F[\pi^*]}$ has low time and space complexity; more precisely, we have showed (see Theorem 3 and Theorem 4) that the algorithm `Encode_SIP-to-RPG` for encoding the permutation π^* in $F[\pi^*]$ requires $O(n^2)$ time and $O(n)$ space, where n is the size of the input permutation π^*, while the algorithm `Decode_RPG-to-SIP` decodes the flowgraph $F[\pi^*]$ in $O(n)$ time and space, where n is the size of $F[\pi^*]$.

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

6 Concluding Remarks

In this paper we extended the class of error correcting graphs by proposing efficient and easy to implement graph encodings. In particular, we proposed an

efficient and easily implemented codec system for encoding watermark numbers as graph structures.

More precisely, we first presented the algorithm `Encode_W-to-SIP` which encode an integer w as SIP (self-inverting permutation) π^* in $O(n)$ time and space, where n is the number of bits in the binary representation of w, and the corresponding decoding algorithm `Decode_SIP-to-W` which extracts the watermark number w from the SIP π^* also in $O(n)$ time and space.

We next presented the algorithm `Encode_SIP-to-RPG` which encodes the SIP π^* as a reducible flow-graph $F[\pi^*]$ in $O(n)$ time and space by exploiting didomination relations on the elements of π^*, and the corresponding decoding algorithm `Decode_RPG-to-SIP` which extracts the SIP π^* from the graph $F[\pi^*]$ in $O(n)$ time and space by converting first the graph $F[\pi^*]$ into a directed tree $T[\pi^*]$ and then applying DFS-search on $T[\pi^*]$.

The main features of our proposed encoding and decoding algorithms can be summarized as follows:

- **Algorithms** `Encode_W-to-SIP` **and** `Decode_SIP-to-W`: use basic data structures; apply elementary operations on sequences; have low time and space complexity; have an easy implementation;
- **Algorithms** `Encode_SIP-to-RPG` **and** `Decode_RPG-to-SIP`: use domination relations on permutations; construct dags and lists; use DFS-search on directed trees; have low time and space complexity; have an easy implementation;

An interesting property of our encoding approach is that of enabling 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 the smallest value of n' is $O(\log n)$.

It is worth noting that 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, due to bitonic property encapsulated in π^* and the reducible property of $F[\pi^*]$, which cause them resilience to attacks. In particular, these properties unable us to identify any single change (in some cases, multiple changes) made by an attacker to π^* and $F[\pi^*]$.

Thus, in light of our two codec components π^* and $F[\pi^*]$ proposed in this paper 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. Another interesting open question with practical value 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.

Finally, we leave as an open problem the evaluation of our codec algorithms and structures in a simulation environment in order to obtain detailed information about their practical behaviour. For future investigation, we also leave as an open problem the analysis of our codec algorithms under other software watermarking measurements.

References

1. A.V. Aho, R. Sethi, and J.D. Ullman: Compilers, Principles, Techniques, and Tools. Addison-Wesley, (1986)
2. G. Arboit: A method for watermarking Java programs via opaque predicates. 5th International Conference on Electronic Commerce Research(ICECR-5) (2002)
3. M. Chroni and S.D. Nikolopoulos: Encoding watermark integers as self-inverting permutations. International Conference on Computer Systems and Technologies (CompSysTech'10), ACM ICPS 471, 125–130, (2010)
4. 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, 107–118 (2004)
5. C. Collberg, E. Carter, S. Kobourov, and C. Thomborson: Error-correcting graphs for software watermarking. Proc. 29th Workshop on Graphs in Computer Science (WG'03), LNCS 2880, 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, 56–67 (2009)
7. C. Collberg and C. Thomborson: Software watermarking: models and dynamic embeddings. Proc. 26th ACM SIGPLAN-SIGACT on Principles of Programming Languages (POPL'99), 311–324 (1999)
8. C. Collberg, C. Thomborson, and D. Low: On the limits of software watermarking. Department of Computer Science, The University of Auckland, Technical Report No 164 (1998)
9. C.S. Collberg, C. Thomborson, and G.M. Townsend: Dynamic graph-based software fingerprinting. ACM Transactions on Programming Languages and Systems 29, 35:1-67 (2007)
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), 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'07), 145–148 (2003)
12. 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, 317–333 (1996)
13. 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)
14. R. Ghiya and L.J. Hendren: Is it a tree, a DAG, or a cyclic graph? a shape analysis for heapdirected pointers in c. Proc. 23rd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL'96), 1–15 (1996)
15. M.C. Golumbic, Algorithmic Graph Theory and Perfect Graphs, Academic Press, New York (1980). Second edition, Annals of Discrete Math. 57, Elsevier (2004)
16. D. Grover: The Protection of Computer Software - Its Technology and Applications. Cambridge University Press, New York (1997)
17. M.S. Hecht and J.D. Ullman: Flow graph reducibilit. SIAM J. Computing 1, 188–202 (1972)

18. M.S. Hecht and J.D. Ullman: Characterizations of reducible flow graphs. Journal of the ACM 21, 367–375 (1974)
19. S.A. Moskowitz and M. Cooperman: Method for stegacipher protection of computer code. US Patent 5.745.569 (1996)
20. 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), 191–197 (2000)
21. G. Myles and C. Collberg: Software watermarking via opaque predicates: Implementation, analysis, and attacks. Electronic Commerce Research 6, 155–171 (2006)
22. J. Nagra and C. Thomborson: Threading software watermarks. Proc. 6th Int'l Workshop on Information Hiding (IH'04), LNCS 3200, 208-223 (2004)
23. S.D. Nikolopoulos: Coloring permutation graphs in parallel. Discrete Applied Mathematics 120, 165–195 (2002)
24. 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, 190–193 (1998)
25. G. Ramalingam: The undecidability of aliasing. ACM Transactions on Programming Languages and Systems 16, 1467–1471 (1994)
26. P. Samson: Apparatus and method for serializing and validating copies of computer software. US Patent 5.287.408 (1994)
27. J. Stern, G. Hachez, F. Koeune, and J. Quisquater: Robust object watermarking: Application to code. Proc. 3rd Int'l Workshop on Information Hiding (IH'99), LNCS 1768, 368–378 (1999)
28. H. Tamada, M. Nakamura, A. Monden, and K. Matsumoto: Design and evaluation of birthmarks for detecting theft of Java programs. Proc. Int'l Conference on Software Engineering (IASTED SE'04), 569–575 (2004)
29. 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, 157–168 (2001)
30. L. Zhang, Y. Yang, X. Niu, and S. Niu: A survey on software watermarking. Journal of Software 14, 268–277 (2003)
31. W. Zhu, C. Thomborson, and F.Y. Wang: A survey of software watermarking. Proc. IEEE Int'l Conference on Intelligence and Security Informatics (ISI'05), LNCS 3495, 454–458 (2005)