Encoding Watermark Numbers as Cographs using Self-inverting Permutations

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 encode the watermark values as graph structures and embed them in application programs. In this paper we extended the class of graphs which can be efficiently used in a software watermarking system by proposing an efficient codec system, i.e., encoding and decoding algorithms that emendextract watermark values intofrom cographs through the use of self-inverting permutations. More precisely, we present a codec system which takes as input an integer w as watermark value, converts it into a self-inverting permutation π^*, and then encodes the permutation π^* as a cograph. The main property of our codec system is its ability to encode the same integer w, using a self-inverting permutation π^*, into more than one cograph. This property cause our system resilience to attacks since it can embed multiple copies of the same watermark number w into an application program. Moreover, the proposed codec system has low time complexity and can be easily implemented.

Key words: software watermarking, watermark numbers, self-inverting permutations, encoding, decoding, cographs, cotrees, algorithms.

1 Introduction

Software watermarking is a technique for protecting the intellectual property of an application program. The software watermarking can be viewed 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]. More formally, 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 $\texttt{extract}(P', k) \rightarrow w$.

A lot of research has been done on software watermarking. The major software watermarking algorithms currently available are based on several techniques, among which the register allocation, spread-spectrum, opaque predicate, abstract interpretation, dynamic path techniques (see, [1, 5, 10, 11, 15, 17, 18]).

In the last decade, several software watermarking algorithms have been appeared in the literature that encode watermarks as graph structures. In general, such encodings make use of an encoding function `encode` which converts a watermarking number w into a graph G, $\texttt{encode}(w) \rightarrow G$, and also of a decoding function `decode` that converts the graph G into the number w, $\texttt{decode}(G) \rightarrow w$; we usually call the pair $(\texttt{encode}, \texttt{decode})$ as *graph codec* [6]. From a graph-theoretic point of view, we are looking for a class of graphs $\mathcal{G}$ and a corresponding codec $(\texttt{encode}, \texttt{decode})_{\mathcal{G}}$ with such properties which cause them resilience to attacks.

In 1996, Davidson and Myhrvold [12] proposed the first static software watermarking algorithm which embeds the watermark into an application program by reordering the basic blocks of a control flow-graph. Based on this idea, Venkatesan, Vazirani and Sinha [19] 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 [19] the construction of a directed graph G (or, watermark graph G) is not discussed. Collberg et al. [7] 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) [6], which is a reducible control flow-graph with maximum out-degree of two, mimicking real code. Note that, for encoding integers the $\texttt{GTW}_{\texttt{sm}}$ method uses self-inverting permutations.

Recently, Chroni and Nikolopoulos [3, 4] extended the class of software watermarking algorithms and graph structures by proposing an efficient and easily implemented codec system for encoding watermark numbers as reducible permutation flow-graphs. They presented an efficient algorithm which encodes a watermark number w as self-inverting permutation π^* and, also, an algorithm which encodes the permutation π^* as a reducible permutation flow-graph $F[\pi^*]$ by exploiting domination relations on the elements of π^* and using an efficient DAG representation of π^*; in the same paper, the authors also proposed efficient decoding algorithms. The two main components of their 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.

In this paper, we extended the class of graphs which can be efficiently used in a software watermarking system by proposing efficient encoding and decoding algorithms that emend watermark values into cographs through the use of self-inverting permutations (or, for short, SIP) and extract them from the graph structures. More precisely, we present an encoding algorithm, we call it `Encode_SIP-to-Cograph`, which takes as input a self-inverting permutation π^* corresponding to watermark integer w [3, 4], and encodes the permutation π^* into a cograph. We also present a decoding algorithm which we call `Decode_Cograph-to-SIP`; it takes as input a cograph G produced by our encoding algorithm and extracts the self-inverting permutation π^* from G. The main property of our encoding algorithm is its ability to encode the same integer w, through the use of a self-inverting permutation π^*, into more than one cograph. This property cause our codec system resilience to attacks since it can embed multiple copies of the same watermark number w into an application program. Moreover, the proposed codec system has low time complexity and can be easily implemented.

The paper is organized as follows: In Section 2 we give the definition and properties on the structure of self-inverting permutations. In Section 3 we define the class of cographs and describe its tree representation. In Section 4 we present the codec algorithms of our codec system, i.e., the encoding algorithm `Encode_SIP-to-Cograph` and decoding algorithm `Decode-Cograph-to-SIP`. Finally, Section 5 concludes the paper and gives futures research directions.

2 Watermark Numbers and Self-inverting Permutations

In this section, we give some definitions that are key-objects in our algorithms for encoding watermark numbers as complement reducible graphs, also known as cographs [13].

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.

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.

For encoding integers some recently proposed watermarking methods uses only those permutations that are self-inverting. Collberg et al. [8, 6] based on the fact that there is a one-to-one correspondence between self-inverting permutations and isomorphism classes of RPGs, Collberg et al. [6] proposed a polynomial algorithm for encoding any integer w as the RPG corresponding to the wth self-inverting permutation π in this correspondence. This encoding exploits only the fact that a self-inverting permutation is its own inverse.

In [3] Chroni and Nikolopoulos proposed an efficient algorithm which encodes an integer w as self-inverting permutation π^* thought the use of bitonic permutations; recall that a permutation $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$ over the set N_n is called bitonic if either monotonically increases and then monotonically decreases, or else monotonically decreases and then monotonically increases. In this encoding the self-inverting permutation incorporates important structural properties which cause it resilience to attacks.

3 Cographs and Cotrees

The complement reducible graphs, also known as cographs, are defined as the class of graphs formed from a single vertex under the closure of the operations of union and complement [14]. More precisely, the class of cographs is defined recursively as follows:

(i) a single-vertex graph is a cograph;
(ii) the disjoint union of cographs is a cograph;
(iii) the complement of a cograph is a cograph.

Cographs have arisen in many disparate areas of applied mathematics and computer science and have been independently rediscovered by various researchers under various. Cographs are perfect graphs and in fact form a proper subclass of permutation graphs and distance hereditary graphs; they contain the class of quasi-threshold graphs and, thus, the class of threshold graphs [17, 18]. Furthermore, cographs are precisely the graphs which contain no induced subgraph isomorphic to a P4 (i.e., a chordless path on four vertices).

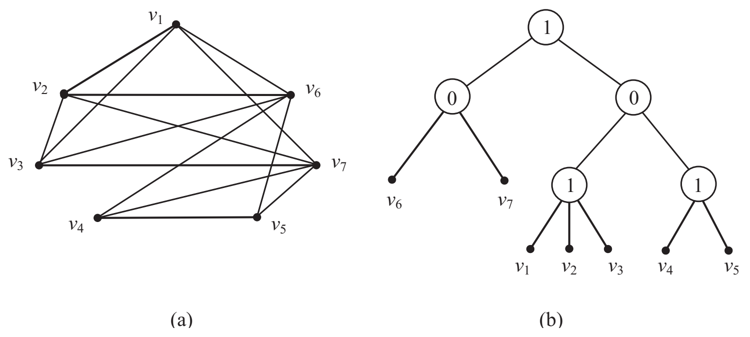

Fig. 1. (a) A cograph on 7 vertices, and (b) the corresponding cotree.

Cographs were introduced in the early 1970s by Lerchs [14] who studied their structural and algorithmic properties. Along with other properties, Lerchs has shown that the cographs admit a unique tree representation, up to isomorphism, called a cotree. The cotree of a cograph G is a rooted tree such that:

(i) each internal node, except possibly for the root, has at least two children;
(ii) the internal nodes are labeled by either 0 (0-nodes) or 1 (1-nodes); the internal nodes that are children of a 1-node (0-node resp.) are 0-nodes (1-nodes resp.), i.e., 1-nodes and 0-nodes alternate along every path from the root to any (internal) node of the cotree;
(iii) the leaves of the cotree are in a 1-to-1 correspondence with the vertices of G, and two vertices v_i, v_j are adjacent in G if and only if the least common ancestor of the leaves corresponding to v_i and v_j is a 1-node.

Lerchs' definition required that the root of a cotree be a 1-node; if however we relax this condition and allow the root to be a 0-node as well, then we obtain cotrees whose internal nodes all have at least two children, and whose root is a 1-node if and only if the corresponding cograph is connected.

The study of cographs led naturally to constructive characterizations that implied several linear-time recognition algorithms that also enabled the construction of the corresponding tree representation (cotree) in linear time [13]. The first linear-time recognition and cotree-construction algorithm was proposed by Corneil, Perl, and Stewart in 1985 [9]. Recently, Bretscher et. al [2] presented a simple linear-time recognition algorithm which uses a multisweep LexBFS approach; their algorithm either produces the cotree of the input graph or identifies an induced P_4.

4 Encoding Self-inverting Permutations as Cographs

In this section, we present an algorithm for encoding a self-inverting permutation as a cograph. In particular, our algorithm takes as input a self-inverting permutation π^* of length $2n+1$ produced by algorithm `Encode_W-to-SIP` (see [3, 4]), and then constructs an arbitrary cograph $C[\pi^*]$ on $2n+1$ vertices by preserving the cycle relation of permutation π^*; recall that, by construction π^* has length $2n+1$ and contains one 1-cycle (x, x) and n 2-cycles (x_1, y_1), (x_2, y_2), $\ldots$, (x_n, y_n). We next describe the proposed algorithm:

```
Algorithm Encode_SIP-to-Cograph
Input: a self-inverting permutation $\pi^* = (\pi_1, \pi_2, \ldots, \pi_n)$;
Output: the cograph $C[\pi^*]$;
1. Construct a graph $H$ such that:
   $V(H) = \{\pi_1, \pi_2, \ldots, \pi_n\}$;
   $E(H) = \{(\pi_i, \pi_j)$ is an edge if $(\pi_i, \pi_j)$ is a 2-cycle in $\pi^*\}$;
2. Compute the connected components $H_1, H_2, \ldots, H_k$ of the graph $H$;
3. $S = \{H_1, H_2, \ldots, H_k\}$; the graphs in $S$ are connected cographs
4. While $|S| > 1$ do
       Select two arbitrary cographs $H_i$, $H_j$ from $S$;
       Remove $H_i$ and $H_j$ from the set, i.e., $S = S - \{H_i, H_j\}$;
       Compute the complements $\overline{H_i}$ and $\overline{H_j}$ of the connected cographs
           $Hi$ and $Hj$, and set $H_i = \overline{H_i}$ and $H_j = \overline{H_j}$;
           the cographs $H_i$ and $H_j$ are now disconnected;
       Compute the disjoint union $H_i + H_j$ of the disconnected cographs
           $H_i$ and $H_j$, and set $H_i = H_i + H_j$;
       Add the cograph $H_i$ in the set $S$, i.e., $S = S \cup H_i$;
   end-while;
5. Return the cograph $G = H_i$, where $H_i$ is the only cograph in $S$;
```

Encode Example: Let $\pi^* = (3, 5, 1, 7, 2, 6, 4)$ be the input self-inverting permutation in the algorithm `Encode_SIP-to-Cograph` which corresponds to watermark number w. The algorithm first constructs the graph H having $V(H) = \{v_1,$

v_2, v_3, v_4, v_5, v_6, $v_7\}$ and $E(H) = \{(v_1, v_3), (v_2, v_5), (v_4, v_7)\}$ and then computes its connected components $H_1 = H[v_1, v_3]$, $H_2 = H[v_2, v_5]$, $H_3 = H[v_4, v_7]$ and $H_4 = H[v_6]$; note that $H_1 = H[v_1, v_3]$ is the subgraph of H induced by the nodes v_1 and v_3.

Construction of the cograph of Figure 2(a): 1st iteration of step 4: the algorithm takes H_1 and H_2, computes the disjoint union $U(1, 2)$ of $\overline{H_1}$ and $\overline{H_2}$, and then sets $H_1 = U(1, 2)$ and removes subgraph H_2; 2nd iteration of step 4: it takes H_1 and H_4, computes the disjoint union $U(1, 4)$ of $\overline{H_1}$ and $\overline{H_4}$, and then sets $H_1 = U(1, 4)$ and removes subgraph H_4; 3rd iteration of step 4: it takes H_1 and H_3, computes the disjoint union $U(1, 3)$ of $\overline{H_1}$ and $\overline{H_3}$, and then sets $H_1 = U(1, 3)$ and removes subgraph H_3; it returns H_1 which is the cograph of Figure 2(a).

Construction of the cograph of Figure 2(b): in a similar way, the algorithm constructs the graph of Figure 2(b) by taking first the subgraphs H_1 and H_2, then the subgraphs H_3 and H_4, and finally the subgraphs H_1 and H_3;

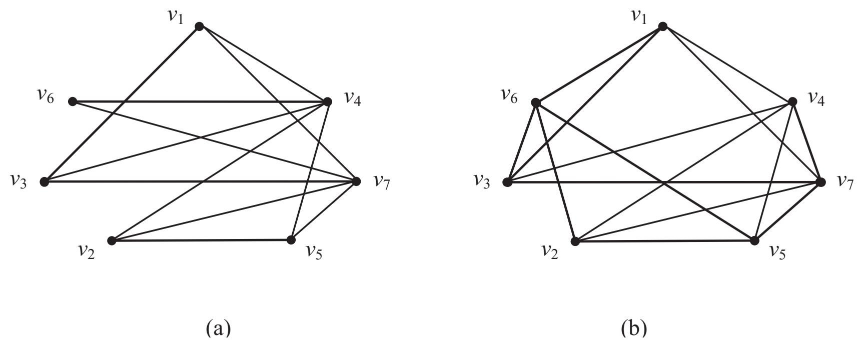

Fig. 2. (a) & (b): Two cographs on 7 vertices which encode the same watermark number w.

Next, we present a decoding algorithm for extracting a self-inverting permutation from a cograph. Our algorithm, which we call `Decode_Cograph-to-SIP`, takes as input a cograph $C[\pi^*]$ produced by algorithm `Encode_SIP-to-Cograph` and extracts the self-inverting permutation π^* from $C[\pi^*]$ by constructing first its cotree $T[\pi^*]$ and then finding the pairs of nodes (x_1, y_1), (x_2, y_2), …, (x_n, y_n) such that the nodes x_i and y_i, $1 \leq i \leq n$, have the same internal node (0-node or 1-node) as parent; these pairs correspond to 2-cycles of the permutation π^*. We next describe the decoding algorithm:

Algorithm `Decode_Cograph-to-SIP`
Input: a cograph $G = C[\pi^*]$ constructed from Algorithm `Encode-SIP-to-Cograph`;
Output: a self-inverting permutation $\pi^* = (\pi_1, \pi_2, \ldots, \pi_n)$;

```
1. Compute the cotree $T(G)$ of the input cograph $G$;
   Let $V = \{v_1, v_2, \ldots, v_n\}$ be the set of its terminal vertices;
2. While $|V| > 0$ do
       Select a vertex $v$ from the set $V$ and remove it from $V$, i.e., $V = V - \{v\}$;
       Find the child $u \neq v$ of the parent $p(v)$ of the vertex $v$;
       If $u$ is a vertex of $V$ then
           construct a 2-cycle $(v, u)$ and $V = V - \{v, u\}$;
       else
           construct a 1-cycle $(v)$ and $V = V - \{v\}$;
   end-while;
3. Construct the identity permutation $\pi^* = (\pi_1, \pi_2, \ldots, \pi_n)$, i.e., $\pi_i^{-1} = i$,
   $1 \leq i \leq n$;
4. For each 2-cycle $(v, u)$ do the following:
       $\pi_v = u$ and $\pi_u = v$;
5. Return the self-inverting permutation $\pi^*$;
```

Decode Example: Let $C_a[\pi^*]$ and $C_b[\pi^*]$ be two cographs produced by the encoding algorithm `Encode_SIP-to-Cograph`. The decoding algorithm constructs first the corresoding cotrees $T_a[\pi^*]$ and $T_b[\pi^*]$, and then computes the pairs of nodes (v_4, v_7), (v_1, v_3) and (v_2, v_5) (see Figure 3). Then it constructs the identity permutation $\pi^* = (1, 2, 3, 4, 5, 6, 7)$, which maps every element of the set N_n to itself, and changes the positions of element 4 and 7, 1 and 3, and 2 and 5; it returns the self-inverting permutation $\pi^* = (3, 5, 1, 7, 2, 6, 4)$ which corresponds to watermark number w.

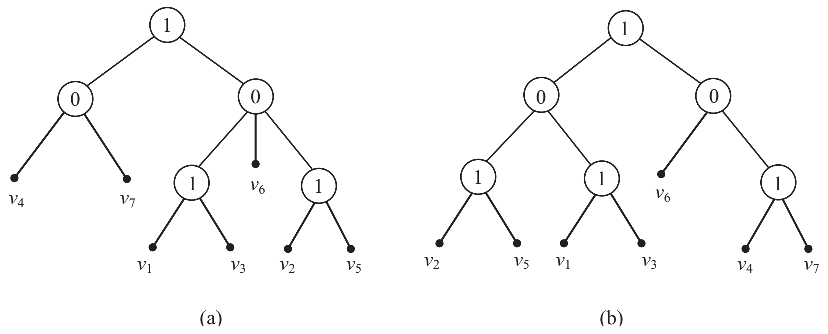

Fig. 3. (a) & (b): The corresponding cotrees of the cographs of figure 2.

5 Concluding Remarks

In this paper we presented an efficient codec system for encoding self-inverting permutations as cographs. The main property of our codec system is its ability to encode the same self-inverting permutation π^*, which corresponds to integer w, into more than one cograph. This property causes our system resilience to attacks. Moreover, the proposing codec system has low time complexity and can be easily implemented.

In light of our codec system's encoding algorithm `Encode_SIP-to-Cograph` which can encode the self-inverting permutation π^* of a watermark integer w into several different cographs it would be very interesting to study the possibility of finding an efficient transformation of a cograph into a reducible permutation graph [3, 4]; we leave it as an open question.

References

1. G. Arboit, "A method for watermarking Java programs via opaque predicates," 5th International Conference on Electronic Commerce Research (ICECR-5), 2002.
2. A. Bretscher, D. Corneil, M. Habib, and C. Paul, "A simple linear time LexBFS cograph recognition algorithm," SIAM J. Discrete Math. 22 (2008) 1277–1296.
3. M. Chroni and S.D. Nikolopoulos, "Encoding watermark integers as self-inverting permutations," Int' l Conference on Computer Systems and Technologies (CompSysTech'10), ACM ICPS 471, 2010, pp. 125–130.
4. M. Chroni and S.D. Nikolopoulos, "Efficient encoding of watermark numbers as reducible permutation graphs," Department of Computer Science, University of Ioannina, Technical Report TR–2011–03, 2011.
5. C. Collberg, E. Carter, S. Debray, A. Huntwork, J. Kececioglu, C. Linn and M. Stepp, "Dynamic path-based software watermarking," Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, SIGPLAN 39, 2004, pp. 107–118.
6. 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, 2003, pp. 156–167.
7. 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 (2009) 56–67.
8. C. Collberg and C. Thomborson, "Software watermarking: models and dynamic embeddings," Proc. 26th ACM SIGPLAN-SIGACT on Principles of Programming Languages (POPL'99), 1999, pp. 311–324.
9. D.G. Corneil, Y. Perl, and L.K. Stewart, "A linear recognition algorithm for cographs," SIAM J. Comput. 14 (1985) 926–984.
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), 2004, pp. 173–185.
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), 2003, pp. 145–148.
12. R.L. Davidson and N. Myhrvold, "Method and system for generating and auditing a signature for a computer program," US Patent 5.559.884, Microsoft Corporation, 1996.
13. M.C. Golumbic, Algorithmic Graph Theory and Perfect Graphs, Academic Press, New York (1980); Second edition, Annals of Discrete Math. 57, Elsevier (2004).

14. H. Lerchs, “On cliques and kernels,” Department of Computer Science, University of Toronto, March 1971.
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), 2000, pp. 191–197.
16. G. Myles and C. Collberg, “Software watermarking via opaque predicates: Implementation, analysis, and attacks,” Electronic Commerce Research 6 (2006) 155–171.
17. J. Nagra and C. Thomborson, “Threading software watermarks,” Proc. 6th Int’l Workshop on Information Hiding (IH’04), LNCS 3200, 2004, pp. 208–223.
18. 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, 1998, pp. 190–193.
19. R. Venkatesan, V. Vazirani, and S. Sinha, “A graph theoretic approach to software watermarking,” Proc. 4th Int’l Information Hiding Workshop (IH’01), LNCS 2137, 2001, pp. 157–168.