Encoding Watermark Integers as Self-inverting Permutations

Maria Chroni and Stavros D. Nikolopoulos

Abstract: *In a software watermarking environment, several graph theoretic watermark methods use integers as watermark values, where some of these methods encode the watermark integers as reducible permutation graphs (RPG; these are reducible control-flow graphs with a maximum out-degree of two). Since there is a one-to-one correspondence between self-inverting permutations and isomorphic classes of RPGs, for encoding watermark integers most of the watermarking methods use only those permutations that are self-inverting. In this paper we present an efficient algorithm for encoding integers as self-inverting permutations. More precisely, our algorithm takes as input an integer w, computes its binary representation* $b_1b_2 \ldots b_n$*, and then produces a self-inverting permutation* π^* *in O(n) time. Moreover, we also present an algorithm for decoding a self-inverting permutation; our algorithm takes as input a self-inverting permutation* π^* *produced by the encoding algorithm and returns its corresponding integer w in O(n) time, where n is the length of the input permutation.*

Key words: *Watermark, permutations, self-inverting permutations, encoding, decoding, graphs, algorithms.*

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 [6].

The Software Watermarking problem can be described as follows: Embed 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. More precisely, a Software Watermarking System can be defined as follows [10]: Given a program P, a watermark w, and a key k, a software watermarking system consists of functions:

- embed(P, w, k) $\rightarrow$ P'
- recognize(P', k) $\rightarrow$ w

Although digital watermarking has made considerable progress and become a popular technique for copyright protection of multimedia information [6, 12], research on software watermarking has recently received sufficient attention. The patent by Davidson and Myhrvold [7] presented the first published software watermarking algorithm. The preliminary concepts of software watermarking also appeared in paper [8] and patents [9, 11]. Collberg et al. [4, 5] presented detailed definitions for software watermarking. Authors of papers [14, 15] have given brief surveys of software watermarking research.

There are two general categories of watermarking algorithms: the static and the dynamic algorithms [4]. A static watermark is stored inside program code in a certain format, and it does not change during the program execution. According to the representation of watermark information, there are two types of static watermarks: data watermarks and code watermarks. A data watermark stores watermark information as program data, and can be stored anywhere inside a program, such as in comments or in variables.

CompSysTech'10, June 17–18, 2010, Sofia, Bulgaria.

A code watermark is represented by choosing a particular sequence of instructions in cases (and these are common), where more than one sequence of instructions has an equivalent effect. A static code watermark may also be stored in "dead code" (which is never executed); any sequence of instructions may be used with equivalent effect in a dead-code area. For example, in a Java program, a particular order of cases in a switch statement can be used to represent a watermark number. Further discussion of static watermarking issues can be found in [7, 9, 13]**.**

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. There are three kinds of dynamic watermarks: *Easter Eggs*, *Execution Trace Watermarks*, and *Dynamic Data Structure Watermarks* [4].

In 1990, Davidson and Myhrvold [7] 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 et al. [13] proposed an algorithm which embeds the watermark by extending a method's control flow graph (CFG) through the insertion of a subgraph. The first dynamic watermarking algorithm (CT) was proposed by Collberg et al. [4]; it embeds the watermark through a graph structure which is built on a heap at runtime.

Venkatesan et al. [13] propose a software watermarking scheme which is called GTW; in such a scheme an executable program is marked by the addition of code for which the topology of the control flow graph (CFG) encodes a watermark. More precisely, the GTW process is as follows: The watermark value W is encoded as a directed graph G which, in turn, is converted into control flow graph (CFG). In [13] the construction of a directed graph G (or, watermark graph G) is not discussed. Collberg et al. [2] proposed an implementation of GTW, which they call GTWsm, and it is the first publicly available implementation of the algorithm GTW. In GTWsm the watermark is encoded as a reducible permutation graph (RPG) [3], which is a reducible control-flow graph with maximum out-degree of two, mimicking real code.

In GTWsm implementation a watermark value (integer) is encoded as a RPG; in particular, in the enumeration of Collberg et al. [3], an integer n is encoded as the RPG corresponding to the nth self-inverting permutation. Note that the is a one-to-one correspondence between self-inverting permutations and isomorphic classes of RPGs. Thus, for encoding integers the GTWsm methods uses only those permutations that are self-inverting.

In this paper we propose an efficient algorithm for encoding integers as self-inverting permutations. More precisely, our algorithm takes as input an integer w, computes its binary representation $b_1b_2\dots b_n$, and then produces a self-inverting permutation π^* in $O(n)$ time. Moreover, we also propose an algorithm for decoding a self-inverting permutation; our algorithm takes as input a self-inverting permutation π^* produced by the encoding algorithm and returns its corresponding integer w in $O(n)$ time, where n is the length of the input permutation.

The paper is organized as follows: In Section 2 we define the reducible permutation graphs and the self-inverting permutations. In Section 3 we introduce the notion of the bitonic permutations which is the key-object in our algorithm for encoding integers as self-inverting permutations. In Section 4 we present our algorithm Encode-Integers-as-SIP which takes as input an integer w and produces a self-inverting permutation π^*, while in the same section we also present a recognition algorithm, that is, an algorithm which takes a self-inverting permutation π^* produced by algorithm Encode-Integers-as-SIP and returns the integer w. Finally, Section 5 concludes the paper and discuses futures research directions.

REDUCIBLE PERMUTATION GRAPHS

Several graph theoretic watermarking methods encodes a watermark value in the topology of a control-flow graph (CFG) [1] and embed it in an application program P [2]. Note that, each node of a CFG represents a basic block which consists of instructions with a single entry and a single exit; two basic blocks are connected by a directed edge if, during the execution, control can pass from one basic block to the other. Moreover, note that a CFG itself also has a single entry and a single exit.

In a graph theoretic watermarking environment, the GTW method [2] forms a typical watermark embedding process. We next give an overview of the GTW method consisting of the following steps:

1. The watermark value w is spited into several values $w_1, w_2, \ldots, w_k$;
2. The values $w_1, w_2, \ldots, w_k$ are encoded as directed graphs $G_1, G_2, \ldots, G_k$;
3. The generated graphs $G_1, G_2, \ldots, G_k$ are converted into CFGs $W_1, W_2, \ldots, W_k$ by generating executable code for each CFG;
4. Each $W_1, W_2, \ldots, W_k$ is marked to indicate whether it is part of the watermark, and is embedded in a specific location in the application code P;

Having embedded a watermark value w in an application program P, we are interested in designing an efficient reverse method, that is, a method which takes as input the watermarked code P and produces the watermark value w; such a reverse method is usually called *recognition method*. The recognition method for the above described embedding method is a follows:

1. Identify the marked CFGs $W_1, W_2, \ldots, W_k$ in the application program P;
2. Each CFG $W_1, W_2, \ldots, W_k$ is decoded to compute a value;
3. The individual values are combined to yield the watermark value w;

Some graph theoretic watermarking methods, like the GTW method, use integers as watermark values and encode them as reducible permutation graphs (RPG) [2]; these are reducible control-flow graphs with a maximum out-degree of two.
More precisely, an RPG is a reducible control-flow graph with a Hamiltonian path consisting of four pieces: (a) *a header node*, (b) *the preamble*, (c) *the body*, and (d) *a footer* [2].

There is a one-to-one correspondence between self-inverting permutations and isomorphic classes of RPGs. Thus, for encoding integers some watermarking methods uses only those permutations that are self-inverting.

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 $(\pi^{-1})_i$, denoted here as p_i, is the position in the sequence of the number *i*; in our example, $p_4 = 2$, $p_7 = 4$, $p_3 = 6$, 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.

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.

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 $B = b_1b_2\ldots b_n$ the binary representation of w, where b_i is either 1 or 0 ($1 \leq i \leq n$). If $B_1 = b_1b_2\ldots b_n$ and $B_2 = d_1d_2\ldots d_m$ be two binary numbers, then the number $B_1||B_2$ is the binary number $b_1b_2\ldots b_nd_1d_2\ldots d_m$; for example, if $B_1 = 10101$ and $B_2 = 110$ are the integers 21 and 6, respectively, then the binary number $B_1||B_2 = 10101110$ is the integer 174. The binary sequence of the number $B = b_1b_2\ldots b_n$ is the sequence $B^* = (b_1, b_2, \ldots, b_n)$ of length n.

Let $B = b_1b_2\ldots b_n$ be a binary number. Then, $\text{flip}(B) = b'_1b'_2\ldots b'_n$ is the binary number such that $b'_i = 0$ (1) if and only if $b_i = 1$ (0), $1 \leq i \leq n$.

ENCODING INTEGERS

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 $b_1b_2\ldots b_n$ of w, and then produces a self-inverting permutation π^* in O(n) time. We next describe the propose algorithm:

Algorithm Encode-Integers-as-SIP
Input: a watermark integer w;
Output: the self-inverting permutation π^*;

1. Compute the binary representation $B = b_1b_2\ldots b_n$ of w;
2. Construct the binary number $B' = 00\ldots0||B||1$ 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 positions of 0's and the sequence $Y = (y_1, y_2, \ldots, y_m)$ of the positions of 1's in B* from left-to-right;
4. Construct the bitonic permutation $\pi = (x_1, x_2, \ldots, x_k, y_m, y_{m-1}, \ldots, y_1)$ on n' = 2n+1 numbers;
5. Let $(z_1, z_2, \ldots, z_k, z_{k+1}, z_{k+2}, \ldots, z_{n'}) = (x_1, x_2, \ldots, x_k, y_m, y_{m-1}, \ldots, y_1)$;
   Case 1: n' even: select n'/2 pairs $(z_1, z_{n'})$, $(z_2, z_{n'-1})$, …, $(z_{n'/2}, z_{(n'+3)/2})$;
   for each selected pair (z_i, z_j), do the following:
   $\pi_{z_j} = z_i$ and $\pi_{z_i} = z_j$;
   Case 2: n' odd: select $\lfloor n'/2 \rfloor$ pairs $(z_1, z_{n'})$, $(z_2, z_{n'-1})$, …, $(z_{\lfloor n'/2 \rfloor}, z_{\lfloor n'/2 \rfloor+2})$ and the number $z_{\lfloor n'/2 \rfloor+1}$;
   for each selected pair (z_i, z_j), do the following:
   $\pi_{z_j} = z_i$ and $\pi_{z_i} = z_j$;
   $\pi_{z_{\lfloor n'/2 \rfloor+1}} = z_{\lfloor n'/2 \rfloor+1}$;
6. Return the self-inverting permutation $\pi^* = (\pi_1, \pi_2, \ldots, \pi_{n'})$ on n' = 2n+1 numbers;

Example 1: Let w = 12 be the input watermark integer in the Algorithm Encode-Integers-as-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 π = (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 π^* = (5, 6, 9, 8, 1, 2, 7, 4, 3).

Next, we present a recognition algorithm, that is, an algorithm for decoding a self-inverting permutation. More precisely, our recognition algorithm, which we call Decode-SIP, takes as input a self-inverting permutation π^* produced by Algorithm Encode-Integers-as-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 propose algorithm:

Algorithm Decode-SIP
Input: a self-inverting permutation π^* produced from Algorithm Encode-Integers-as-SIP;
Output: an integer w;

1. Compute the cycle representation C = $c_1 c_2 \dots c_k$ of the self-inverting permutation $\pi^* = (\pi_1, \pi_2, \dots, \pi_{n'})$, where n' = 2n+1;
2. Initially, i = 1, j = n' and all the cycles of C are unmarked;
3. While there exists an unmarked cycle c in C, do the following:
   Select the first unmarked cycle c of C from left-to-right;
   Case 1: the selected cycle c has length 2 and let c = (a, b):
   π_i = a and π_j = b;
   mark cycle c; i = i+1 and j = j -1;
   Case 2: the selected cycle c has length 1 and let c = (a):
   π_i = a; mark cycle c; i = i+1;
4. Find the first increasing subsequence X = $(x_1, x_2, \dots, x_k)$ and then the first decreasing subsequence Y = $(y_1, y_2, \dots, y_m)$ of π;
5. Construct the binary sequence $B^* = (b_1, b_2, \dots, b_{n'})$ as follows:
   set 0's in positions $x_1, x_2, \dots, x_k$ and 1's in positions $y_1, y_2, \dots, y_m$;
6. Compute B' = flip(B*) = $(b_1, b_2, \dots, b_n, b_{n+1}, \dots, b_{2n}, b_{2n+1})$;
7. Return the integer w of the binary number B = $b_{n+1}, b_{n+2}, \dots, b_{2n}$;

Example 2: Let π^* = (5, 6, 9, 8, 1, 2, 7, 4, 3) be a self-inverting permutation produced from Algorithm Encode-Integers-as-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 π = (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;

CONCLUSIONS AND FUTURE WORK

In this paper we presented an efficient algorithm for encoding watermark integers as self-inverting permutations. Our algorithm takes as input an integer w and produces a self-inverting permutation π^* in O(n) time, where n is the number of bits in the binary representation $b_1 b_2 \dots b_n$ of w. We also presented a decoding algorithm; it takes as input a self-inverting permutation π^* produced by the encoding algorithm and returns its corresponding integer w in O(n) time, where n is the length of the input permutation. Both algorithms are simple, easy implemented and very fast.

It is worth noting that our approach enable us to encode the integer w = $b_1 b_2 \dots b_n$ as self-inverting permutation π^* of any length; indeed, π^* can be constructed over the set $N_{n'}$ = {1, 2, …, n'}, where the smallest value of n' is O(logn).

REFERENCES

[1] A.V. Aho, R. Sethi, J.D. Ullman, Compilers, Principles, Techniques, and Tools, Addison-Wesley, 1986.

[2] 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.

[3] C. Collberg, E. Carter, S. Kobourov, and C. Thomborson, “Error-correction graphs,” Workshop on Graphs in Computer Science (WG'03), 2003.

[4] C. Collberg and C. Thomborson, "Software watermarking: models and dynamic embeddings," 26th Symposium on Principles of Programming Languages (POPL '99), ACM, 1999.

[5] C. Collberg, C. Thomborson, and D. Low, “On the limits of software watermarking,” Technical Report No 164, Department of Computer Science, The University of Auckland, 1998.

[6] I. Cox, J. Kilian, T. Leighton, and T. Shamoon, “A secure, robust watermark for multimedia,” LNCS 1174, 1996, pp. 317–333.

[7] 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, Sep 1996.

[8] D. Grover, The Protection of Computer Software - Its Technology and Applications, Cambridge University Press New York, 1997.

[9] S. A. Moskowitz and M. Cooperman, "Method for stegacipher protection of computer code," US Patent 5.745.569, The Dice Company, Jan 1996.

[10] G. Myles and C. Collberg, “Software watermarking via opaque predicates: Implementation, analysis, and attacks,” Electron Commerce Res 6 (2006) 155-171.

[11] P. Samson, “Apparatus and method for serializing and validating copies of computer software,” US Patent 5.287.408, 1994.

[12] H. Tamada, M. Nakamura, A. Monden, and K. Matsumoto, “Design and evaluation of birthmarks for detecting theft of java programs,” International Conference on Software Engineering (IASTED SE'04), 2004, pp. 569–575.

[13] R. Venkatesan, V. Vazirani, and S. Sinha, "A graph theoretic approach to software watermarking," 4th International Information Hiding Workshop (IH'01), LNCS 2137, 2001, pp. 157-168.

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

[15] W. Zhu, C. Thomborson, and F.-Y. Wang, “A survey of software watermarking,” in IEEE ISI 2005, LNCS 3495, 2005, pp. 454–458.

ABOUT THE AUTHORS

Maria Chroni, MSc, PhD Candidate, Department of Computer Systems, University of Ioannina, Phone: +30 265 100 8832, E-mail: mchroni@cs.uoi.gr

Stavros D. Nikolopoulos, Professor, PhD, Department of Computer Systems, University of Ioannina, Phone: +30 265 100 8801, E-mail: stavros@cs.uoi.gr