Αλγοριθμικές Τεχνικές Κωδικοποίησης Μεταθέσεων και Μεταθετικών Γραφημάτων για Υδατοσύμανση Λογισμικού, Εικόνας, Ήχου, και Κειμένου

Η ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

υποβάλλεται στην

ορισθείσα από την Γενική Συνέλευση Ειδικής Σύνθεσης

του Τμήματος Μηχανικών Η/Υ και Πληροφορικής

Εξεταστική Επιτροπή

από την

Μαρία Γ. Χρόνη

ως μέρος των Υποχρεώσεων για τη λήψη του

ΔΙΔΑΚΤΟΡΙΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΣΤΗΝ ΠΛΗΡΟΦΟΡΙΚΗ

Δεκέμβριος 2014

Τριμελησ Συμβουλευτικη Επιτροπη

- **Σταύρος Δ. Νικολόπουλος**, Καθηγητής, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων (επιβλέπων).
- **Γεώργιος Α. Παπαδόπουλος**, Καθηγητής, Τμήμα Πληροφορικής, Πανεπιστήμιο Κύπρου.
- **Ιωάννη Σταματίου**, Αναπληρωτής Καθηγητής, Τμήμα Οργάνωσης και Διοίκησης Επιχειρήσεων, Πανεπιστήμιο Πατρών.

Επταμελησ Εξεταστικη Επιτροπη

1. **Σταύρος Δ. Νικολόπουλος**, Καθηγητής, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων (επιβλέπων).
2. **Γεώργιος Α. Παπαδόπουλος**, Καθηγητής, Τμήμα Πληροφορικής, Πανεπιστήμιο Κύπρου.
3. **Ιωάννη Σταματίου**, Αναπληρωτής Καθηγητής, Τμήμα Οργάνωσης και Διοίκησης Επιχειρήσεων, Πανεπιστήμιο Πατρών.
4. **Ιωάννης Μανωλόπουλος**, Καθηγητής, Τμήμα Πληροφορικής, Αριστοτέλειο Πανεπιστήμιο Θεσσαλονίκης.
5. **Στέφανος Γκρίτζαλης**, Καθηγητής, Τμήμα Μηχανικών Πληροφοριακών και Επικοινωνιακών Συστημάτων, Πανεπιστήμιο Αιγαίου.
6. **Λεωνίδας Παληός**, Αναπληρωτής Καθηγητής, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων.
7. **Λουκάς Γεωργιάδης**, Επίκουρος Καθηγητής, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων.

Dedication

Στη μητέρα μου...

...Γεωργία

To my mother...

...Georgia

You were my strength when I was weak
You were my voice when I couldn't speak
You were my eyes when I couldn't see
You lifted me up when I couldn't reach
You gave me faith because you believed
You were always there for me
I'm blessed and grateful for been my mother!

Ευχαριστιες

Η εκπόνηση μιας διδακτορικής διατριβής αποτελεί μια πολυετή διαδικασία, ένα ταξίδι γεμάτο συγκινήσεις, συντροφιά με ανθρώπους που μοιράζεσαι κοινά ενδιαφέροντα και στόχους, αλλά και με ανθρώπους που απλά βρίσκονται δίπλα σου και κάνουν το ταξίδι αυτό πιο όμορφο και ασφαλές. Το δικό μου ταξίδι ολοκληρώθηκε και για αυτό θα ήθελα να ευχαριστήσω ξεχωριστά όλους αυτούς τους ανθρώπους που ήταν συνταξιδιώτες στο ταξίδι αυτό ως ελάχιστο δείγμα ευγνωμοσύνης και αναγνώρισης της συμβολής τους.

Θα ήθελα να ευχαριστήσω μέσα από την καρδιά μου τον κύριο συνταξιδιώτη μου, τον επιβλέποντα καθηγητή μου, Σταύρο Δ. Νικολόπουλο για την δυνατότητα που μου έδωσε να ακολουθήσω τα όμορφα και συναρπαστικά μονοπάτια της έρευνας, για την εμπιστοσύνη που μου έδειξε, και για τη στήριξη και καθοδήγηση που μου παρείχε καθ' όλη τη διάρκεια των σπουδών μου. Με έμαθε να έχω πίστη, προσήλωση στους στόχους μου, και να ξεπερνώ τα εμπόδια. Θα ήθελα να τον ευχαριστήσω γιατί μέσα από αυτό το ταξίδι όλων αυτών των ετών μου άνοιξε νέους ορίζοντες και μου μετέφερε με το καλύτερο τρόπο πλήθος γνώσεων, δεξιοτήτων και στάσεων που θα με ακολουθούν και θα με οδηγούν στη ζωή μου. Αποτελεί για μένα πρότυπο ακαδημαϊκού, δασκάλου, και ανθρώπου.

Ευχαριστώ θερμά τα μέλη της εξεταστικής επιτροπής μου για τη τιμή που μου έκαναν να συμμετέχουν σε αυτή και για την ουσιαστική συμβολή τους στην ποιοτική αξιολόγηση των ερευνητικών μου πεπραγμένων. Ευχαριστώ τους μεταπτυχιακούς φοιτητές, και πλέον φίλους μου, Άγγελο Φυλάκη και Ιωάννη Χιόνη για την ειλικρινή ερευνητική συνεργασία που είχαμε, και για το κλίμα εμπιστοσύνης, σεβασμού, και πίστης στους κοινούς ερευνητικούς στόχους που θέσαμε. Η συνεργασία αυτή, εκτός από τα όποια σημαντικά ερευνητικά αποτελέσματα, μου έδωσε πρωτίστως δύο αξιόλογους, καλούς και αγαπημένους φίλους.

Ιδιαιτέρως θα ήθελα να ευχαριστήσω τους ανθρώπους του Τμήματος Μηχανικών Η/Υ και Πληροφορικής του Πανεπιστημίου Ιωαννίνων, για τη στήριξή τους, την εμπιστοσύνη τους, και τη δυνατότητα που μου έδωσαν να είμαι σε ένα Τμήμα πρότυπο, με άριστη επιστημονική υποδομή και άψογο κλίμα συνεργασίας, και να εργαστώ σε ένα αντικείμενο που πραγματικά αγαπώ και που αισθάνομαι ότι προσφέρω με τις γνώσεις που το ίδιο αυτό Τμήμα κάποτε μου έδωσε.

Ευχαριστώ τους φίλους μου που αποδέχτηκαν την απουσία μου μέσα στις ατέλειωτες ώρες που χανόμουν στο μαγικό αυτό κόσμο της έρευνας. Τέλος οφείλω ένα μεγάλο ευχαριστώ στην οικογένειά μου, για την αγάπη τους, την καθοδήγησή τους, την αποδοχή και στήριξη όλων των στόχων μου, των στόχων των σπουδών μου, των στόχων της ζωής μου.

Μαρία Γεωργίου Χρόνη
Δεκέμβριος 2014

. . . ένα νέο ταξίδι ξεκινά!

Table of Contents

1 Introduction **1**

1.1 Information Hiding . . . 1
1.2 Digital Watermarking . . . 3
1.2.1 Watermarking Classification . . . 5
1.2.2 Properties . . . 6
1.2.3 Attacks . . . 6
1.2.4 Watermarking Techniques . . . 8
1.3 Motivation . . . 13
1.4 The Structure of the Thesis . . . 14
1.5 Main Results . . . 15

2 Encode Watermark Numbers as Self-inverting Permutations **19**

2.1 Introduction . . . 19
2.2 Preliminaries . . . 21
2.3 Self-inverting Permutations (SiP) . . . 22
2.4 Encode Watermark Numbers as SiPs . . . 23
2.4.1 Algorithm Encode_W.to.SiP . . . 24
2.4.2 Algorithm Decode_SiP.to.W . . . 25
2.5 The Structure of Permutation π^* . . . 27
2.6 Properties of Permutation π^* . . . 29
2.7 Concluding Remarks . . . 30

3 Encoding SiPs as Reducible Permutation Graphs **31**

3.1 Introduction . . . 31
3.2 Preliminaries . . . 35
3.3 Reducible Permutation Graphs (RPG) . . . 35
3.4 The Structure of our Codec System . . . 36
3.5 Encode SiPs as Reducible Permutation Graphs . . . 37
3.5.1 Encode SiPs as Reducible Permutation Graphs - I . . . 37
3.5.2 Encode SiPs as Reducible Permutation Graphs - II . . . 41
3.6 Properties of the Flow-graph $F[\pi^*]$ . . . 44
3.6.1 Structural Properties . . . 45
3.6.2 Unique Hamiltonian Path . . . 45
3.7 Detecting Attacks . . . 46

3.7.1 Node-label Modification . . . 46
3.7.2 Edge Modification . . . 47
3.8 Concluding Remarks . . . 48

4 Multiple Encoding of a Number into RPGs using Cographs 49
4.1 Introduction . . . 49
4.2 Background Results . . . 50
4.2.1 Cographs and Cotrees . . . 52
4.3 Multiple Encoding of a SiP into Cographs . . . 53
4.3.1 Algorithm Encode_SiP.to.Cograph . . . 53
4.3.2 Algorithm Decode_Cograph.to.SiP . . . 54
4.4 Encoding Cographs as RPGs . . . 56
4.4.1 Algorithm Encode_Cotree.to.RPG . . . 56
4.4.2 Algorithm Decode_RPG.to.SiP . . . 58
4.5 Concluding Remarks . . . 60

5 Software Watermarking 61
5.1 Introduction . . . 61
5.2 Background Results . . . 65
5.2.1 Encode Numbers as RPGs . . . 66
5.2.2 Call-graphs . . . 66
5.3 The Dynamic Watermarking Model WaterRPG . . . 67
5.3.1 Operational Framework . . . 67
5.3.2 Model Components . . . 67
5.3.3 Embedding an RPG into a Code . . . 74
5.3.4 Extracting the RPG from the Code . . . 79
5.4 Implementation . . . 79
5.5 Model Evaluation . . . 84
5.5.1 Performance . . . 85
5.5.2 Resilience . . . 89
5.6 Concluding Remarks . . . 92

6 Image and Audio Watermarking 93
6.1 Introduction . . . 93
6.2 Background Results . . . 97
6.2.1 2D Representation of SiP . . . 97
6.2.2 1D Representation of SiP . . . 99
6.2.3 Color Images . . . 100
6.2.4 The Discrete Fourier Transform . . . 101
6.3 Image Watermarking in the Spatial Domain . . . 101
6.3.1 Embed Watermark into Image - S . . . 101
6.3.2 Extract Watermark from Image - S . . . 105
6.3.3 Performance . . . 106
6.4 Image Watermarking in the Frequency Domain . . . 108

6.4.1 Embed Watermark into Image - F . . . 109
6.4.2 Extract Watermark from Image - F . . . 111
6.4.3 Function f . . . 112
6.4.4 Experimental Evaluation . . . 113
6.4.5 Performance . . . 113
6.4.6 Attack Issues . . . 114
6.5 Audio Watermarking . . . 118
6.5.1 Embed Watermark into Audio . . . 118
6.5.2 Extract Watermark from Audio . . . 122
6.5.3 Experimental Results . . . 123
6.6 Concluding Remarks . . . 126

7 Text Watermarking **129**
7.1 Introduction . . . 129
7.2 Background Results . . . 132
7.2.1 Structure of a PDF Documents . . . 134
7.3 Watermarking PDF Documents . . . 135
7.3.1 Embed Watermark into PDF - I . . . 136
7.3.2 Embed Watermark into PDF - II . . . 137
7.3.3 Embed an RPG into a PDF . . . 139
7.4 Concluding Remarks . . . 141

8 Conclusions and Future Work **143**
8.1 Encoding Numbers as SiPs and RPGs . . . 143
8.2 Software Watermarking . . . 144
8.3 Image and Audio Watermarking . . . 145
8.4 Text Watermarking . . . 146

List of Figures

1.1 A formal view of digital watermarking. . . . 3
1.2 An abstract view of software and digital media watermarking. . . . 4
1.3 Digital watermarking classification. . . . 5
1.4 Digital watermarking attacks. . . . 7
1.5 The Structure of the 8 Chapters of the Thesis. . . . 15
1.6 Algorithms of the Chapters 2, 3, and 4 (Part II: Encodings) and Chapters 5, 6, and 7 (Part III: Watermarking). . . . 16

2.1 The main data components used by the algorithms of our codec system: (i) the watermark number w and (ii) the self-inverting permutation π^*. . . . 21
2.2 The main data components used by our two codec Algorithms `Encode_W.to.SiP` and `Decode_SiP.to.W`. . . . 26

3.1 The main data components used by the algorithms of our codec system: (i) the watermark number w, (ii) the self-inverting permutation π^*, and (iii) the reducible permutation graph $F[\pi^*]$. . . . 34
3.2 The DAG $D[\pi^*]$ of the self-inverting permutation π^* and the corresponding Dmax-tree $T_d[\pi^*]$. . . . 37
3.3 The main structures used or constructed by the algorithms `Encode_SiP.to.RPG-I` and `Decode_RPG.to.SiP-I`, that is, the self-inverting permutation π^*, the values of function $P()$, the reducible permutation graph $F[\pi^*]$, and the Dmax-tree $T_d[\pi^*]$. . . . 40
3.4 The main structures used or constructed by Algorithms `Encode_SiP.to.RPG-II` and `Decode_RPG.to.SiP-II`, i.e., the self-inverting permutation π^*, the decreasing subsequences of π^*, the graph $F[\pi^*]$, the tree $T_s[\pi^*]$, and the elements of π^* in pairs. . . . 43
3.5 The probability for the RPG $F[\pi^*]$ to have the RPG property after a modification of (a) 1 edge, (b) 2 edges, (c) 3 edges, and (d) 4 edges. Note the different scaling of the four diagrams. . . . 47

4.1 The main data components used by the algorithms of our codec system for multiple encoding the same watermark number $w = 4$ into several RPGs using Cographs. . . . 51
4.2 (a) A cograph on 7 vertices, and (b) the corresponding cotree. . . . 52
4.3 Two cographs $C_1[\pi^*]$ and $C_2[\pi^*]$ on 7 vertices which encode the same watermark number w, corresponding to permutation $\pi^* = (3, 5, 1, 7, 2, 6, 4)$, along with their corresponding cotrees. . . . 55
4.4 The cotree $T_1[\pi^*]$ and the resulting RPG-tree $R_1[\pi^*]$; trees $T'[\pi^*]$ and $T''[\pi^*]$ show the contraction process. . . . 57

4.5 Two RPG-trees $R_1[\pi^*]$ and $R_2[\pi^*]$ and the corresponding reducible permutation graphs $F_1[\pi^*]$ and $F_2[\pi^*]$, respectively, produced by the algorithm `Encode_Cotree.to.RPG`. . . . 59

5.1 (a) The dynamic call-graph $G(P, I_{key})$ of an application program P. (b) The reducible permutation graph $F[\pi^*]$. (c) The dynamic call-graph $G(P^*, I_{key})$ of the watermarked program P^*. . . . 68
5.2 (a) The real-call (f_4, f_6) in the call-graph $G(P, I_{key})$ of a program P; bold arrow. (b) The corresponding path-call (f_4, f_3, f_5, f_6) in the call-graph $G(P^*, I_{key})$ of the watermarked program P^*; bold arrows. . . . 70
5.3 (a) The forward call pattern *f-call*; (b) The backward call pattern *b-call*; (c) The path call pattern *p-call*. The boxes with marked corners are the f|b-blocks. . . . 71
5.4 An example of cf-statement modification via opaque predicates in the case where (f_i, f_j) is a water-forward function call. . . . 72
5.5 An example of cf-statement modification via opaque predicates in the case where (f_i, f_j) is a real-backward function call. . . . 72
5.6 An example of cf-statement modification via opaque predicates of the function f_j in the case where (f_i, f_j) is a water-forward function call. . . . 73
5.7 A block diagram of the main operations of the embedding algorithm. . . . 75
5.8 (a) The dynamic call-graph $G(\texttt{Shortest\_Path}, I_{key})$. (b) The reducible permutation graph $F[\pi^*]$ which encodes the watermark $w = 2$, where $\pi^* = (3, 5, 1, 4, 2)$. (c) The dynamic call-graph $G(\texttt{Shortest\_Path}^*, I_{key})$. . . . 77
5.9 The call-tables $\mathcal{T}$ and $\mathcal{T}^*$ of the programs `Shortest_Path` and `Shortest_Path`*, respectively, the edge characterization table $\mathcal{C}^*$, and the values of the cf-variable. 78
5.10 The function `up`() of the original program `Laser` watermarked with the naive approach; the functions `down`() and `health`() are both water functions and belong to category $\mathcal{B}_{callee}$, i.e., both are functions of $G(\texttt{Laser}, I_{key})$. . . . 82
5.11 The function `up`() of the original program `Laser` watermarked with a stealthy approach; the functions `down`() and `health`() are both water functions and belong to category $\mathcal{B}_{callee}$, i.e., both are functions of $G(\texttt{Laser}, I_{key})$. . . . 83
5.12 Graphical representation of the results for parameters (i) Execution time, (ii.a) Disk usage and (ii.b) Heap space usage of the original program P, and the corresponding watermarked program under both the Naive P_N^* and Stealthy P_S^* approaches. . . . 88

6.1 (a) A 2D representation of the self-inverting permutation $\pi = (5, 6, 9, 8, 1, 2, 7, 4, 3)$; (b) A 2DM representation of permutation π. . . . 98
6.2 The 1DM representations of the self-inverting permutation $\pi = (4, 7, 6, 1, 5, 3, 2)$. . . . 99
6.3 The range of colors represented on the Cartesian 3-dimensional system. . . . 100
6.4 The brightness k_{ij}^{ℓ} of the central and cross pixels p_{ij}^{ℓ} of the grid-cell $C_{ij}(I)$, $0 \leq \ell \leq 4$, and the brightness $k_{ij}^{\ell m}$ of the cycle-cross pixels $p_{ij}^{\ell m}$, $1 \leq \ell \leq 4$ and $m = 1, 2, 3$. . . . 103
6.5 (a) The original image I; (b) The watermarked image I_w. . . . 104
6.6 The embedding process. . . . 109
6.7 The original image of Lena and its two watermarked images with $c = c_{max}$ and $c = c_{opt}$; the watermark corresponds to SiP (6,3,2,4,5,1). . . . 114

6.8 Some original images and their corresponding watermarked ones; for each image, its size and its c_{opt}, and PSNR and SSIM values are also shown, for $Q = 55$. 115
6.9 The original image of Lena and its watermarked images with $\sigma^2 = 0.01$, $\sigma^2 = 0.001$ and $\sigma^2 = 0.0001$. 116
6.10 (a) Watermarked image of Lena, (b) 90 degrees angled image, (c) 180 degrees angled image, and (d) cropped image. 117
6.11 Segmentation of the S's signal into specific frames according to 1DM-representation of the permutation π^*. 119
6.12 The DFT representation of a marked frame. 120
6.13 The "Red" and "Blue" segments on DFT. 121
6.14 The encoding process of audio signal watermarking. 122

7.1 Three different representations of permutation $\pi^* = (4, 7, 6, 1, 5, 3, 2)$. 133
7.2 Components of a PDF file. 134
7.3 (a) The structure of a PDF file; (b) The code of a PDF file containing, in object `50obj`, the text "Hello World". 135
7.4 (a) The main structural components of a PDF file; (b) The document structure of PDF file. 136
7.5 The initial PDF document T and watermarked PDF document T_w using the 1D representation of permutation $\pi^* = (4, 7, 6, 1, 5, 3, 2)$. 137
7.6 The initial PDF document T and watermarked PDF document T_w using the 2D representation of permutation $\pi^* = (4, 7, 6, 1, 5, 3, 2)$. 138
7.7 The watermarked $DS(T^*)$ which encodes the RPG of $\pi^* = (4, 5, 3, 1, 2)$. 140

List of Tables

5.1 General properties of watermarking models and the properties of our WarerRPG model. . . . 65
5.2 Execution Time (msec) . . . 87
5.3 Disk Usage (Kb) . . . 87
5.4 Heap Space Usage (Mb) . . . 87
5.5 Three Group of Bytecode Instructions . . . 89
5.6 Indicative Bytecode Instructions of each Group . . . 89

6.1 The PSNR and SSIM values of the original and watermarked images, for compression of qualities $Q = 85$, $Q = 75$, $Q = 65$, and $Q = 55$. . . . 115
6.2 The PSNR values of the original and watermarked images, for Gaussian noise with variance values $\sigma^2 = 0.01$, $\sigma^2 = 0.001$, and $\sigma^2 = 0.0001$. . . . 116
6.3 The PSNR values of the watermarked audio signals. . . . 124
6.4 The Hamming distance of the watermark w and the extracted watermark w^* after common signal attacks. . . . 126

Abstract

Maria G. Chroni. PhD, Department of Computer Science & Engineering, University of Ioannina, Greece; Graduation December, 2014.

PhD thesis: Algorithmic Techniques for Encoding Permutations and Permutation Graphs for Watermarking Software, Image, Audio, and Text.

Supervisor: Stavros D. Nikolopoulos, Professor.

Internet technology, in modern communities, has become an indispensable tool for everyday life since most people use it on a regular basis and do many daily activities online. This frequent use of the internet means that measures taken for internet security are indispensable since the web is not risk-free. One of those risks is the fact that the web is an environment where intellectual property is under threat since a huge amount of digital data are transferred every day, and thus such data may end up on a user who falsely claims ownership.

Digital watermarking (or, simply, watermarking) is a technique for protecting the intellectual property of a digital object; the idea is simple: a unique identifier, which is called *watermark*, is embedded into a digital object which may be used to verify its authenticity or the identity of its owners. A digital object may be audio, picture, video, text, or software, and the watermark is embedded into object's data through the introduction of errors not detectable by human perception; note that, if the object is copied then the watermark also is carried in the copy. Efficient watermarking techniques, should be able to embed and successfully extract the watermark, even after the digital object has been attacked, i.e., it has been subjected to transformations by malicious users that aim to mislead the watermark extractor.

The issues addressed in this thesis concern the design of efficient and easily implementable codec systems for watermarking software and digital media, such as image, audio, and text. Previous works on digital watermarking propose specific encoding techniques for a specific type $\mathcal{O}$ of a digital object, i.e., the main idea of a proposed technique for watermarking a digital object of type $\mathcal{O}$ cannot be efficiently applied to a different digital object of type $\mathcal{O}'$; for example, the main idea of a proposed technique for watermarking a software (application program) $\mathcal{P}$, cannot be efficiently applied to image $\mathcal{I}$, audio (signal) $\mathcal{S}$ or even text $\mathcal{T}$. In our work, we overcome such a drawback by proposing algorithmic techniques for encoding a watermark w into a self-inverting permutation (or, SiP for short) π^*, and then embedding the self-inverting permutation π^* into different digital objects, i.e., software, image, audio, and text, by using different representations of the same SiP π^*. The data structures used to represent the SiP, as well as the encoding techniques, encompasses important properties allowing us to design a codec system which efficiently detect attacks.

In the first part of the thesis, presents the basic research on encoding watermark members as graph structures through the use of self-inverting permutations (SiP) and algorithms for multiple encodings. We introduce the notion of a bitonic permutation, and present our algorithm `Encode_W.to.SiP` for encoding an integer w as a self-inverting permutation π^*, along with the corresponding decoding algorithm `Decode_SiP.to.W`, and discuss important properties of the self-inverting permutation π^*. Then, we define the main graph-based data component of our codec system, namely reducible permutation graphs (or, PRG for short), describe the two operational phases of our codec system, and present the structure of our system's reducible permutation graph $F[\pi^*]$. We next present the two algorithms, namely `Encode_SiP.to.RPG-I` and `-II` for encoding the self-inverting permutation π^* as a reducible permutation flow-graph $F[\pi^*]$ along with the corresponding decoding algorithms `Decode_RPG.to.SiP-I` and `-II`. Finally, we present the properties of the reducible permutation flow-graph $F[\pi^*]$ and show that node-label or edge modifications on the graph $F[\pi^*]$ can be efficiently detected.

We extended the class of graphs that encode a watermark by proposing a randomized encoding algorithm which takes as input a self-inverting permutation π^* and encodes the same permutation π^* into several cographs $C_1[\pi^*], C_2[\pi^*], \ldots, C_n[\pi^*]$. Then, we present the algorithm `Encode_Cograph.to.RPG`, along with its corresponding decoding algorithm, which embeds a cograph into an RPG by exploiting the structure and some important algorithmic properties of its cotree. Thus, having such encoding algorithms, we can encode a watermark number w into many RPGs $F_1[\pi^*], F_2[\pi^*], \ldots, F_n[\pi^*]$, $n \geq 2$. A digital object can be made more resilient to attacks if multiple copies of the same watermark w are embedded into it.

The second part of the thesis, presents how the different components of our codec system can be used for watermarking software, digital images and audio, as well as, digital text. Initially, we present our dynamic software watermarking model WaterRPG; we first describe its structural and operational components and then the embedding algorithm `Embed_RPG.to.CODE` and the extracting algorithm `Extract_CODE.to.RPG`. The main idea behind the proposed watermarking model is a systematic modification of appropriate function calls of the program P, through the use of control statements and opaque predicates, so that the execution of the watermarked program P_w with a specific input gives a dynamic call-graph from which the watermark graph $F[\pi^*]$ can be easily constructed. Then, we implement our watermarking model in real Java application programs and show two main watermarking approaches supported by the WaterRpg model, namely naive and stealthy. We also evaluate our model under several software watermarking assessment criteria.

Next, we present our image watermarking technique where a watermark w or, equivalently, a self-inverting permutation π^* of length n^*, is transformed from a numerical form to a 2D form (i.e., 2D-representation) through the exploitation of self-inverting permutation properties. The 2D-representation can be efficiently marked on an image I resulting thus the watermarked image I_w. The main idea of the proposed algorithms is that a self-inverting permutation π^* is embedded into an image I by first mapping the elements of π^* into an $n^* \times n^*$ matrix A^* and then using the information stored in A^* to mark specific areas of image I in the frequency domain resulting the watermarked image I_w. We have evaluated the embedding and extracting algorithms by testing them on various and different in characteristics images that were initially in JPEG format and we had positive results as the watermark was successfully extracted even if the image was converted back into JPEG format with various compression ratios.

Similarly, since an audio signal is one dimensional object, we present a transformation of a watermark w or, equivalently, a self-inverting permutation π^* of length n^*, from a numerical form to a 1D form (i.e., 1D-representation) and then an algorithm which embeds w into an audio signal. More precisely, our proposed algorithm embeds a self-inverting permutation π^* over the set N_{n^*} into an audio signal S by first mapping the elements of π^* into an 1D-matrix B^* of length $n' = n^* \times n^*$, and then, based on the information stored in B^*, marking specific areas of audio S in the frequency domain resulting thus the watermarked audio S_w. An efficient algorithm extracts the embedded self-inverting permutation π^* from the watermarked audio S_w by locating the positions of the marks in S_w; it enables us to reconstruct the 1D representation of π^* and, then, obtain the watermark w.

Based on the three different representations of self-inverting permutation (SiP), namely 1D-representation, 2D-representation, and RPG-representation (i.e., the encoding of permutation π^* as a reducible permutation graph $F^*[\pi^*]$), we present the algorithms `Embed_SiP.to.PDF-I`, `Embed_SiP.to.PDF-II`, and `Embed_RPG.to.PDF`, respectively, along with the corresponding extracting algorithms, for embedding a watermark number (or, equivalently, a self-inverting permutation π^* or a reducible permutation graph $F^*[\pi^*]$) into a PDF document file T. We have evaluated the embedding and extracting algorithms by testing them on various and different in characteristics PDF documents.

Finally, we conclude the thesis by summarizing our results, discussing possible future extensions, and proposing open problems for further research in the aria of digital watermarking and, in general, in the aria of information hiding.

Περιληψη

Μαρία Γ. Χρόνη. PhD, Τμήμα Μηχανικών Η/Υ & Πληροφορικής, Πανεπιστήμιο Ιωαννίνων. Αποφοίτηση, Δεκέμβριος 2014.

Διδακτορική Διατριβή: Αλγοριθμικές Τεχνικές Κωδικοποίησης Μεταθέσεων και Μεταθετικών Γραφημάτων για Υδατοσήμανση Λογισμικού, Εικόνας, Ήχου και Κειμένου.

Επιβλέπων: Σταύρος Δ. Νικολόπουλος, Καθηγητής.

Στη σύγχρονη εποχή το διαδίκτυο αποτελεί αναπόσπαστο τεχνολογικό μέσο βοήθειας των δραστηριοτήτων της καθημερινής μας ζωής, καθώς χρησιμοποιείται για τη διεκπεραίωση σύνθετων, και συχνά χρονοβόρων, επαγγελματικών και καθημερινών εργασιών. Η συνεχής και συχνή χρήση του διαδικτύου, ο όγκος της ψηφιακής πληροφορίας που διακινείται μέσω αυτού, το εύρος της ηλικιακής διαστρωμάτωσης χρήσης του, και επιπλέον οι πολλαπλοί κίνδυνοι της προσωπικής ασφάλειας των χρηστών του, απαιτούν αυξημένα και τεχνολογικά ευφυή μέτρα προστασίας αυτού.

Η ψηφιακή υδατοσήμανση (ή, υδατοσήμανση) είναι μια τεχνική για την προστασία της πνευματικής ιδιοκτησίας ενός ψηφιακού αντικειμένου. Η ιδέα είναι απλή: ένα μοναδικό αναγνωριστικό, το οποίο ονομάζεται υδατόσημα, ενσωματώνεται στο ψηφιακό αντικείμενο προκειμένου να χρησιμοποιηθεί για την απόδειξη της αυθεντικότητας ή αναγνώριση της ταυτότητας του ψηφιακού αντικειμένου από τους ιδιοκτήτες του. Ένα ψηφιακό αντικείμενο μπορεί να είναι ψηφιακός ήχος, εικόνα, κείμενο, ή λογισμικό, και το υδατόσημα ενσωματώνεται στο ψηφιακό αντικείμενο εισάγοντας σε αυτό τροποποιήσεις που δεν είναι ορατές και δεν γίνονται αντιληπτές. Για να είναι αποδοτική μια προτεινόμενη τεχνική υδατοσήμανσης, θα πρέπει να ενσωματώνει αποτελεσματικά το υδατόσημα στα ψηφιακά δεδομένα του αντικειμένου και να το εξάγει επιτυχώς, ακόμη και αν το αντικείμενο αυτό έχει υποστεί τροποποιήσεις, δηλαδή έχει δεχθεί επιθέσεις από κακόβουλους χρήστες με σκοπό την μη-εφικτή ή ανεπιτυχή εξαγωγή του υδατόσημου.

Η παρούσα διδακτορική διατριβή πραγματεύεται θέματα σχετικά με την σχεδίαση αποτελεσματικών και εύκολα υλοποιήσιμων συστημάτων κωδικοποίησης για την υδατοσήμανση λογισμικού και ψηφιακών μέσων, όπως είναι η εικόνα, ο ήχος, και το κείμενο. Στα έως σήμερα ερευνητικά αποτελέσματα ψηφιακής υδατοσήμανσης, κάθε προτεινόμενη τεχνική κωδικοποίησης και ενσωμάτωσης εφαρμόζεται συνήθως σε ένα συγκεκριμένο είδος ψηφιακού αντικειμένου $\mathcal{O}$. Πιο συγκεκριμένα, τεχνικές κωδικοποίησης που έχουν εφαρμοστεί για την υδατοσήμανση ενός ψηφιακού αντικειμένου $\mathcal{O}$, δεν μπορούν να εφαρμοστούν, τουλάχιστον εύκολα, σε ένα διαφορετικό αντικείμενο $\mathcal{O}'$. Για παράδειγμα, η βασική ιδέα μιας τεχνικής που χρησιμοποιήθηκε στην υδατοσήμανση ενός λογισμικού $\mathcal{P}$, δεν μπορεί να εφαρμοστεί αποτελεσματικά στην εικόνα $\mathcal{I}$, στον ήχο $\mathcal{S}$ ή ακόμα σε ένα κείμενο $\mathcal{T}$. Η σημαντική συνεισφορά της παρούσας διατριβής έγκειται στη σχεδίαση αποτελεσματικών αλγοριθμικών τεχνικών κωδικοποίησης ενός υδατόσημου w σε μια αυτοαναστρέφουσα μετάθεση

(ή, SiP) π^*, καθώς και στην ενσωμάτωση της αυτοανατρέφουσας μετάθεσης π^* σε διαφορετικά ψηφιακά αντικείμενα, όπως λογισμικό, εικόνα, ήχος, και κείμενο, χρησιμοποιώντας διαφορετικές αναπαραστάσεις της ίδιας μετάθεσης π^*. Οι δομές δεδομένων που χρησιμοποιούνται για την αναπαράσταση της SiP, όπως επίσης και οι τεχνικές κωδικοποίησης, ενσωματώνουν σημαντικές ιδιότητες που δίνουν τη δυνατότητα σχεδίασης συστημάτων κωδικοποίησης που ανιχνεύουν αποτελεσματικά ένα πλήθος κακόβουλων επιθέσεων.

Στο πρώτο μέρος της διατριβής, παρουσιάζονται τα βασικά ερευνητικά αποτελέσματα για την κωδικοποίηση αριθμητικών υδατοσημάτων σε γραφήματα μέσω της αυτοαναστρέφουσας μετάθεσης (SiP) π^*, καθώς και αλγόριθμοι πολλαπλής κωδικοποίησης. Εισάγουμε αρχικά την έννοια των διτονικών (bitonic) μεταθέσεων και στη συνέχεια παρουσιάζουμε τον αλγόριθμο `Encode_W.to.SiP` για την κωδικοποίηση ενός ακεραίου αριθμού w σε μια αυτοαναστρέφουσα μετάθεση π^*, όπως επίσης και τον αντίστοιχο αλγόριθμο αποκωδικοποίησης `Decode_SiP.to.W`, με παράλληλη αναφορά στις σημαντικές ιδιότητες μιας αυτοαναστρέφουσας μετάθεσης π^*. Στη συνέχεια, ορίζουμε τη βασική γραφοθεωρητική συνιστώσα του προτεινόμενου συστήματος κωδικοποίησης, την οποία ονομάζουμε αναγώγιμο μεταθετικό γραφήματα (reducible permutation graphs ή, PRG), περιγράφουμε τις φάσεις κωδικοποίησης ενός υδατόσημου σε αναγώγιμο μεταθετικό γράφημα $F[\pi^*]$, καθώς και τη δομή του γραφήματος $F[\pi^*]$. Πιο συγκεκριμένα, παρουσιάζουμε τους αλγορίθμους `Encode_SiP.to.RPG-I` και `-II` για την κωδικοποίηση μιας αυτοαναστρέφουσας μετάθεσης π^* σε ένα αναγώγιμο μεταθετικό γράφημα $F[\pi^*]$, καθώς και τους αντίστοιχους αλγορίθμους αποκωδικοποίησης `Decode_RPG.to.SiP-I` και `-II`. Τέλος, αναφέρουμε τις ιδιότητες του αναγώγιμου μεταθετικού γραφήματος $F[\pi^*]$ και αποδεικνύουμε ότι κακόβουλες επιθέσεις στο γράφημα $F[\pi^*]$, όπως μετονομασία κόμβων και τροποποίηση ακμών, μπορούν να ανιχνευτούν αποτελεσματικά.

Επεκτείνουμε τη κλάση των γραφημάτων που κωδικοποιούν ένα υδατόσημα προτείνοντας έναν πιθανοτικό αλγόριθμο κωδικοποίησης (randomized algorithm), ο οποίος δέχεται ως είσοδο μια αυτοαναστρέφουσα μετάθεση π^* και κωδικοποιεί αυτή σε διαφορετικά συμπληρωματικά-αναγώγιμα γραφήματα (complement-reducible graphs) ή cographs $C_1[\pi^*], C_2[\pi^*], \ldots, C_n[\pi^*]$. Στη συνέχεια, παρουσιάζουμε τον αλγόριθμο `Encode_Cograph.to.RPG`, καθώς και τον αντίστοιχο αλγόριθμο αποκωδικοποίησης, ο οποίος μετατρέπει ένα cograph $C_i[\pi^*]$ σε ένα αναγώγιμο μεταθετικό γράφημα $F_i[\pi^*]$ χρησιμοποιώντας τη δομή του και σημαντικές αλγοριθμικές ιδιότητες της μοναδιαίας δενδρικής αναπαράστασης (cotree) ενός cograph. Επομένως, μπορούμε να κωδικοποιήσουμε ένα αριθμητικό υδατόσημα w σε πολλά αναγώγιμα μεταθετικά γραφήματα $F_1[\pi^*], F_2[\pi^*], \ldots, F_n[\pi^*]$, $n \geq 2$. Η ενσωμάτωση πολλαπλών αντιγράφων που κωδικοποιούν το ίδιο υδατόσημα σε ένα ψηφιακό αντικείμενο καθιστά αυτό πιο ανθεκτικό σε κακόβουλες επιθέσεις.

Στο δεύτερο μέρος της διατριβής, παρουσιάζονται αλγοριθμικές τεχνικές υδατοσήμανσης λογισμικού, ψηφιακής εικόνας, ήχου, και κειμένου, οι οποίες βασίζονται στα δομικά στοιχεία που αναπτύχτηκαν και παρουσιάστηκαν στο πρώτο μέρος . Αρχικά, παρουσιάζεται το μοντέλο δυναμικής υδατοσήμανσης λογισμικού, το οποίο ονομάζουμε WaterRPG, και αναλύονται οι δομικές και λειτουργικές συνιστώσες του, και στη συνέχεια παρουσιάζεται ο αλγόριθμο ενσωμάτωσης `Embed_RPG.to.CODE` του υδατοσήματος στο κώδικα ενός προγράμματος P, προκύπτοντας έτσι το υδατοσημασμένο πρόγραμμα P_w , καθώς και ο αλγόριθμος εξαγωγής `Extract_CODE.to.RPG` του υδατοσήματος από τον κώδικα του προγράμματος P_w. Η βασική ιδέα του προτεινόμενου συστήματος υδατοσήμανσης βασίζεται στη συστηματική τροποποίηση κατάλληλων κλήσεων συναρτήσεων ενός προγράμματος P, χρησιμοποιώντας συνθήκες ελέγχου και αδιαφανή κατηγορήματα, έτσι ώστε η εκτέλεση του υδατοσημασμένου προγράμματος P_w με μια συγκεκριμένη είσοδο I_{key} να

επιστρέφει ένα δυναμικό γράφημα κλήσεων από το οποίο μπορεί εύκολα να κατασκευαστεί το γράφημα $F[\pi^*]$. Το προτεινόμενο μοντέλο έχει υλοποιηθεί σε προγράμματα που έχουν αναπτυχθεί στη γλώσσα προγραμματισμού Java, και η υλοποίησή του εμπεριέχει δύο προσεγγίζεις: την naive και την stealthy προσέγγιση. Τέλος, το προτεινόμενο μοντέλο αξιολογήθηκε χρησιμοποιώντας διάφορα κριτήρια για την εκτίμηση της αποτελεσματικότητάς του.

Στη συνέχεια, παρουσιάζουμε τεχνικές υδατοσήμανσης ψηφιακής εικόνας βασιζόμενες στο μετασχηματισμό ενός υδατοσήματος w ή, ισοδύναμα, μια αυτοαναστρέφουσα μετάθεση π^* μήκους n^* από αριθμητική μορφή σε δισδιάστατη (2Δ-αναπαράσταση) χρησιμοποιώντας τις ιδιότητες της αυτοαναστρέφουσας μετάθεσης π^*. Η 2Δ-αναπαράσταση μιας αυτοαναστρέφουσας μετάθεσης μπορεί να ενσωματωθεί αποτελεσματικά σε μια ψηφιακή εικόνα, καθώς η εικόνα αποτελεί ένα διασδιάστατο αντικείμενο. Η βασική ιδέα των προτεινόμενων αλγορίθμων υδατοσήμανσης εικόνας έγκειται στην απεικόνιση της αυτοαναστρέφουσας μετάθεσης π^* σε έναν $n^* \times n^*$ πίνακα A^*, στη χρήση της πληροφορίας που είναι αποθηκευμένη σε συγκεκριμένες θέσεις του πίνακα A^*, και στην τροποποίηση των αντίστοιχων περιοχών της ψηφιακής εικόνας I στο πεδίο των συχνοτήτων, παράγοντας έτσι την υδατοσημασμένη εικόνα I_w. Οι αλγόριθμοι ενσωμάτωσης και εξαγωγής του υδατοσήματος σε μια ψηφιακή εικόνα αξιολογήθηκαν πειραματικά σε ένα σύνολο ψηφιακών εικόνων τύπου JPEG, με διαφορετικά χαρακτηριστικά, διαφορετικά μεγέθη, και διαφορετικές αναλογίες, δίνοντας θετικά αποτελέσματα καθώς το υδατόσημα εξάγεται επιτυχώς ακόμη και στην περίπτωση που η εικόνα υφίσταται υψηλή συμπίεση.

Έχοντας παρουσιάσει τις τεχνικές υδατοσήμανσης ψηφιακών εικόνων και καθότι ο ψηφιακός ήχος είναι ένα μονοδιάστατο σήμα, παρουσιάζουμε ένα μετασχηματισμό του υδατοσήματος w ή, ισοδύναμα, της αυτοαναστρέφουσας μετάθεσης π^* μήκους n^*, από την αριθμητική του μορφή στη 1Δ μορφή (1Δ-αναπαράσταση) και στη συνέχεια παρουσιάζουμε τον αλγόριθμο που ενσωματώνει το w στο ψηφιακό ήχο. Πιο συγκεκριμένα, οι προτεινόμενοι αλγόριθμοι ενσωματώνουν μια αυτοαναστρέφουσα μετάθεση π^* μήκους n^* σε ένα ηχητικό σήμα S, απεικονίζοντας τα στοιχεία του π^* σε ένα μονοδιάστατο πίνακα B^* μήκους $n' = n^* \times n^*$, και στη συνέχεια με βάση τα στοιχεία του πίνακα B^* τροποποιούμε το ηχητικό σήμα S στο πεδίο των συχνοτήτων, επιστρέφοντας τελικά το υδατοσημασμένο ηχητικό σήμα S_w. Ο αλγόριθμος εξαγωγής του υδατοσήματος βασίζεται στον εντοπισμό των σημείων στο S_w που έχουν τροποποιηθεί, γεγονός που μας δίνει τη δυνατότητα να ανακατασκευάσουμε την 1Δ-αναπαράσταση του π^*, και επομένως να πάρουμε το υδατόσημα w.

Βασιζόμενοι στις τρεις διαφορετικές αναπαραστάσεις μιας αυτοαναστρέφουσας μετάθεσης, δηλαδή την 1Δ-αναπαράσταση, τη 2Δ-αναπαράσταση, και την RPG-αναπαράσταση (ήτοι η κωδικοποίηση της μετάθεσης π^* σε ένα αναγώγιμο μεταθετικό γράφημα $F^*[\pi^*]$), παρουσιάζουμε τους αλγορίθμους `Embed_SiP.to.PDF-I`, `Embed_SiP.to.PDF-II`, και `Embed_RPG.to.PDF`, αντίστοιχα, για την ενσωμάτωση ενός υδατοσήματος w (ή ισοδύναμα μιας αυτοαναστρέφουσας μετάθεσης π^* ή ενός αναγώγιμου μεταθετικού γραφήματος $F^*[\pi^*]$) σε ένα ψηφιακό κείμενο T τύπου PDF. Οι αλγόριθμοι ενσωμάτωσης και εξαγωγής αξιολογήθηκαν πειραματικά σε διαφορετικά PDF αρχεία.

Το κείμενο της διατριβής ολοκληρώνεται συνοψίζοντας τα ερευνητικά αποτελέσματά μας, προτείνοντας μελλοντικές επεκτάσεις, καθώς και ανοιχτά ερευνητικά προβλήματα στην περιοχή της ψηφιακής υδατοσήμανσης και γενικότερα στην περιοχή της απόκρυψης πληροφορίας.

Chapter 1

Introduction

1.1 Information Hiding

1.2 Digital Watermarking

1.3 Motivation

1.4 The Structure of the Thesis

1.5 Main Results

1.1 Information Hiding

Information hiding, steganography, and watermarking are three closely related fields that have a great deal of overlap and share many technical approaches. However, there are fundamental philosophical differences that affect the requirements, and thus the design, of a technical solution. In this section, we discuss these differences, we give some historical examples on how steganography used in ancient times and present the importance of information hiding techniques in current era.

Information hiding (or data hiding) is a general term encompassing a wide range of problems beyond that of embedding messages in content. The term hiding here can refer to either making the information imperceptible (as in watermarking) or keeping the existence of the information secret. Information hiding can thus be thought as yet another tool to convey information and provide privacy, ownership proof, and deter piracy and copyright infringement.

Staganography, as a branch of information hiding, is not a new method. The technical term itself is derived from the Greek words steganos (στεγανός), which means "covered", and graphi (γραφή), which means "writing". Steganography is the art of concealed communication. The very existence of a message is secret. The first use of steganography is reported by Herodotus, the so-called father of history, who mentions that in ancient Greek, hidden text was written on wax tablets. When Demeratus wanted to notify Sparta that the king of Persia, Xerxes, intended to invade Greece, he wrote this message on a tablet and covered it with wax. To recover the message

the other people in Sparta simply had to scrape the wax off the tablet. Aeneas the Tactician mentions in his documents a lot of other steganographic schemes. Secret letters can be hidden in the messengers' shoe soles or women's ear rings, secret text could be written on wood tables and then whitewashed or one could use pigeons to carry secret notes. Aeneas also suggested some schemes which are very similar to those. One of these suggested techniques included hiding text by making very small holes below or above letters or by changing the heights of letter-strokes in a cover text. Another ingenious method was to shave the head of a messenger and to paint the secret letters on the messenger's head [43].

In ancient China, people used paper masks to agree about the locations of the secret letters. Both, the sender and the receiver had the same paper mask with a number of holes cut at random locations. To conceal a message, the sender places this mask over a sheet of paper, writes the secret message into the holes and fills up the other locations by composing a cover-text. The receiver can read the secret message by placing his mask over the complete message. Of course, this technique assumes that the cover text does not cause the suspicion of a third party. It seems surprising that Cardan, an Italian mathematician, reinvented this scheme in the 16th century and that a British bank recommended to its customers in 1992 (!) to conceal the personal information number (for the cash machine) using a similar system. The idea behind this paper mask scheme is to "camouflage" the secret part in an innocent sounding message.

Very famous is a message which a German spy sent in World War II:

"Apparently neutrals protest is thoroughly discounted and ignored. Isman hard hit. Blockade issue affects pretext for embargo on by products, ejecting suets and vegetable oils."

Taking the second letter in each word reveals the following secret message:

"Pershing sails from NY June 1"

It becomes apparent that people from ancient times need to communicate and exchange secret messages, ensuring the privacy and authentication of their content. The digital revolution has penetrated every aspect of our lives, and has formed a new way of communication. Everyday people communicate through world wide web by exchanging digital text, images, video, or audio. A vast amount of digital data are transferred every day. The ease of distributing fast and in the original form digital content through internet, has led to an increment in privacy infringement from unwanted parties. This has significantly affected not only the global economy but also people's personal lives, since personal data are used or transmitted without the consent of the owner. It is estimated that digital piracy costs the global economy somewhere around $75 billion annually. Staganography and watermarking appears in digital era as means of privacy protection, and what it makes them to distinguish from the ancient methods is the cover for secret data, i.e., the human skin, game, etc. has been replaced by digital text, image, audio, video, software.

Steganography is the process where the digital object is changed by the addition of a secret message in a way that only the sender and the intended recipient is able to detect the message sent through it. It is an invisible message, and thus the detection is not easy. It is a better way of sending secret messages than encoded messages as cryptography does.

Watermarking is used to verify the identity and authenticity of the owner of a digital image. The term "digital watermark" was first coined in 1993 by Andrew Tirkel et al. [116]. It is a process in which the information which verifies the owner is embedded into the digital object;

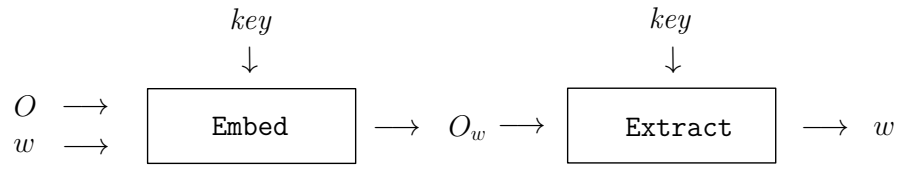


Figure 1.1: A formal view of digital watermarking.

note that it is embedded in a way that it is inseparable from the data and so that it is resistant to many operations not degrading the host object. These objects could be either video, picture, audio, or software. For example, famous artists watermark their pictures and images, ensuring thus that every copy of the image is a watermarked copy.

It is worth noting that staganography and watermarking although they are similar concepts, they present some basic differences. The information hidden by a watermarking system is always associated to the digital object to be protected or to its owner while steganographic systems just hide any information. "Robustness" criteria are also different, since

- steganography is mainly concerned with detection of the hidden message, while
- watermarking concerns potential removal by a pirate.

Steganographic communications are usually point-to-point (between sender and receiver) while watermarking techniques are usually one-to-many.

Another concept which is related to watermarking is cryptography. A number of analogies to cryptographic concepts have been made about watermarking, and according to [45] these analogies are misleading or incorrect. Cryptography is defined as the art and science of secret writing. The word itself comes from Greek where the words kruptos (κρυπτός) and graphen (γραφήν) mean secret and writing, respectively. The focus in cryptography is to protect the content of the message and to keep it secure from unintended audiences. Encryption of digital objects prevents an intruder from accessing the contents without a proper decryption key. But once the data is decrypted, it can be duplicated and distributed illegally. In digital watermarking if somebody tries to copy the digital object, the watermark is copied along with it. Additionally, in cryptography, the message is usually scrambled and unreadable. However, when the communication happens, it is known or noticed. Although the information is hidden in the cipher, an interception of the message can be damaging, as it still shows that there is communication between the sender and receiver. Digital watermarking requires imperceptibility, i.e., the watermark is an invisible mark that is embedded in the digital object, without making it's presence noticeable.

1.2 Digital Watermarking

Paper watermarks appeared in the art of handmade papermaking nearly 700 years ago. The oldest watermarked paper found in archives dates back to 1292 and has its origin in Fabriano, Italy, which is considered the birthplace of watermarks. The analogy between paper watermarks,

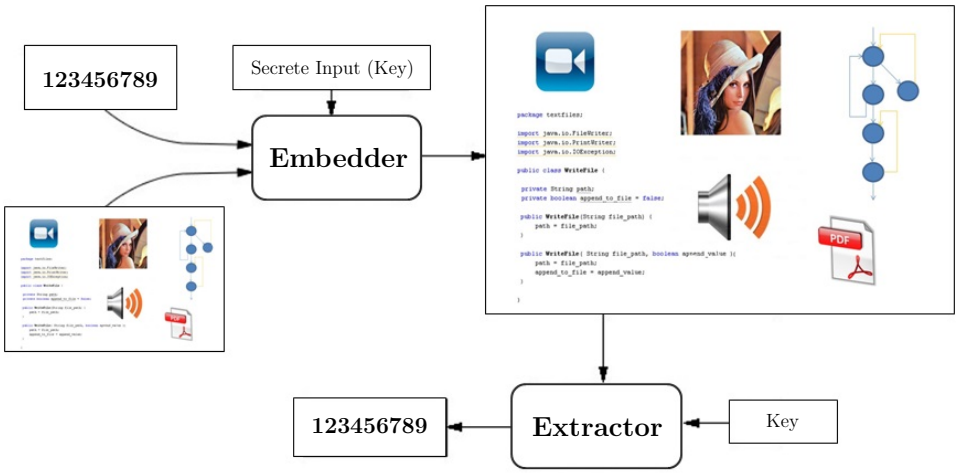


Figure 1.2: An abstract view of software and digital media watermarking.

steganography, and digital watermarking is obvious, and in fact, paper watermarks in money bills or stamps [135] actually inspired the first use of the term watermarking in the context of digital data.

In 1961, Emil Hembrooke of the Muzac Corporation, in his patent entitled "Identification of sound and like signals" [65] described a method for imperceptibly embedding an identification code into music for the purpose of proving ownership. The term identification code for ownership proving was later replaced by term "digital watermark" in works of the authors [116] and [71]. It took a few more years until 1996 before watermarking received remarkable attention. Since then, digital watermarking has gained a lot of attention and has evolved very quickly, and while there are a lot of topics open for further research, practical working methods and systems have been developed.

Digital watermarking (or, simply, watermarking) is a technique for protecting the intellectual property of a digital object; the idea is simple: a unique identifier, which is called *watermark*, is embedded into a digital object which may be used to verify its authenticity or the identity of its owners [29]. A digital object may be audio, picture, video, text, or software, and the watermark is embedded into object's data through the introduction of errors not detectable by human perception [15]; note that, if the object is copied then the watermark also is carried in the copy.

Formally, the watermarking problem can be described as the problem of embedding a watermark w into an object O and, thus, producing a new object O_w, such that w can be reliably located and extracted from O_w even after O_w has been subjected to transformations; for example, compression in case the object is an image or optimization in case the object is software. An abstract view of watermarking with side information is depicted in Figure 1.1 (see also, Figure 1.2).

In the following sections we present a digital watermarking classification, various attacks on watermarks, and we give an overview of the watermarking techniques proposed in the literature in software, image, audio, and text watermarking.

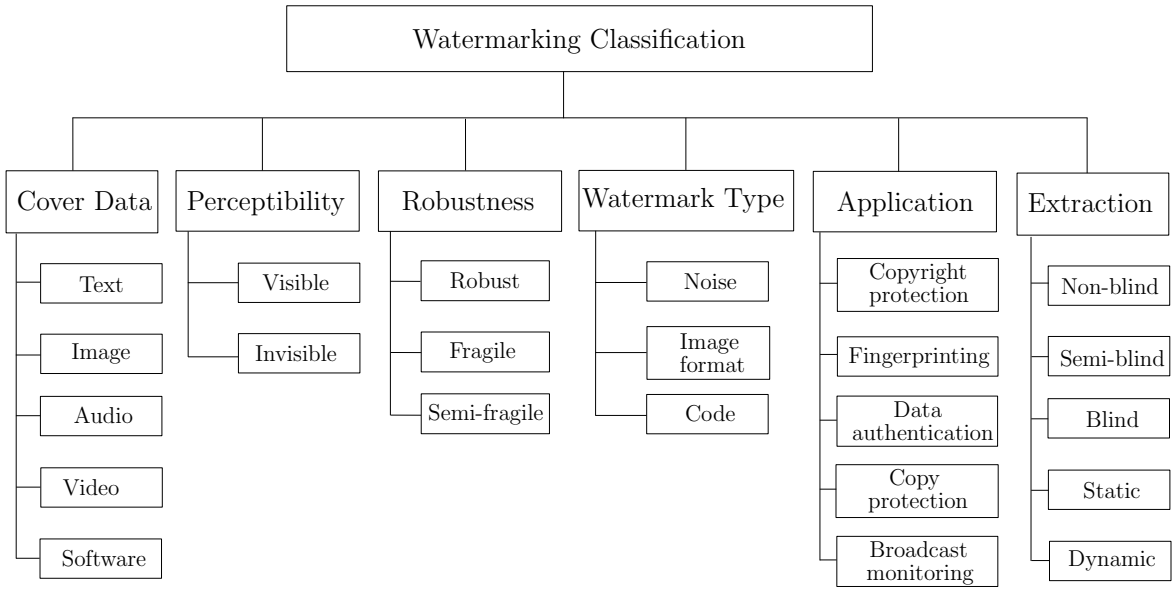


Figure 1.3: Digital watermarking classification.

1.2.1 Watermarking Classification

Digital watermarking has some very basic characteristics, according to the purpose they serve. Figure 1.3 shows classification of watermarking techniques.

Digital Watermarking can be applied into several digital objects such as text, image, audio, video, and software. According to perceptivity, a watermark is either visible or invisible. An example of visible watermark is the logo. Invisible watermarks will not appear as a legible image to the end-user but can be extracted by some algorithm in the end-user's direct control. Authors of [43] define robustness as the "ability to detect the watermark after common signal processing operations". In case where a watermark tolerates some of the operations it is characterized as semi-fragile. Fragile watermarks will always be destroyed when the digital object has been changed. In image, audio, and video watermarking techniques, the watermark is represented in the form of a noise signal, such as pseudo noise, gaussian noise. Another common type of a watermark, which is met and in text watermarking, is image type, where a binary image, a stamp, or a logo is inserted into the cover data. The watermark of code type, is used in software watermarking.

Copyright protection is the most important application of watermarking. The objective is to embed information identifies the copyright owner of the digital media, in order to prevent other parties from claiming the copyright. This application requires a high level of robustness to ensure that embedded watermark cannot be removed without causing a significant distortion in digital media. In fingerprinting a different watermark embedded into each distributed copy, in order to identify single distributed copies of digital object. It is very similar to the serial number of software product. Data authentications' objective is to detect modification of data. This can be achieved with so called fragile watermark that have a low robustness to certain modifications. Copy protection tries to find a mechanism to disallow unauthorized copy of digital media. In

broadcast monitoring an identification code is embedded in the digital data being broadcasted. A computer-base monitoring system can detect the embedded watermark, to ensure that they receive all of the airtime they purchase from the broadcasters.

Finally, in order to extract the watermark information, blind, semi-blind, non-blind, static, or dynamic techniques are used. Non-blind techniques require at least an original media. It extracts a watermark from the possibly distorted image and the original media. Semi-blind techniques do not require an original media for detection, whereas blind techniques require neither an original media nor the embedded watermark. It is also referred to as public watermarking. Static or dynamic techniques refer to software watermarking [32]. A static software watermark is one inserted in the data area or the text of codes. The extraction of such watermark needs not run the software. A dynamic software watermark is inserted in the execution state of a software object. More precisely, in dynamic software watermarking, what has been embedded is not the watermark itself but some codes which cause the watermark to be expressed, or extracted, when the software is run.

1.2.2 Properties

There are three main requirements of digital watermarking. They are transparency, robustness, and capacity, which are presented below.

- **Transparency** (or Fidelity or Imperceptibility). The digital watermark should not affect the quality of the original digital object after it is watermarked. In [43] transparency is defined as "perceptual similarity between the original and the watermarked versions of the cover work". Watermarking should not introduce visible distortions because if such distortions are introduced it reduces the commercial value of the image.
- **Robustness**. Authors of [43] define robustness as the "ability to detect the watermark after common signal processing operations". Watermarks could be removed intentionally or unintentionally by simple processing operations. Hence watermarks should be robust against variety of attacks.
- **Capacity** (or Data Payload). Capacity or data payload is defined as "the number of bits a watermark encodes within a unit of time or work". This property describes how much data should be embedded as a watermark to successfully detect during extraction. Watermark should be able to carry enough information to represent the uniqueness of the digital object [43].

1.2.3 Attacks

A watermarked object is likely to be subjected to certain manipulative processes before it reaches the receiver. Common signal processing functions such as analog-to-digital conversion, digital-to-analog conversion, sampling, quantization, requantization, recompression, linear and nonlinear filtering, low-pass and high-pass filtering, addition of Gaussian and non Gaussian noise are common manipulations. An attack is any processing that impairs or misleads the watermark detector. The performance of a watermarking algorithm against these attacks reflects its quality.Figure 1.4 shows classification of watermarking attacks.

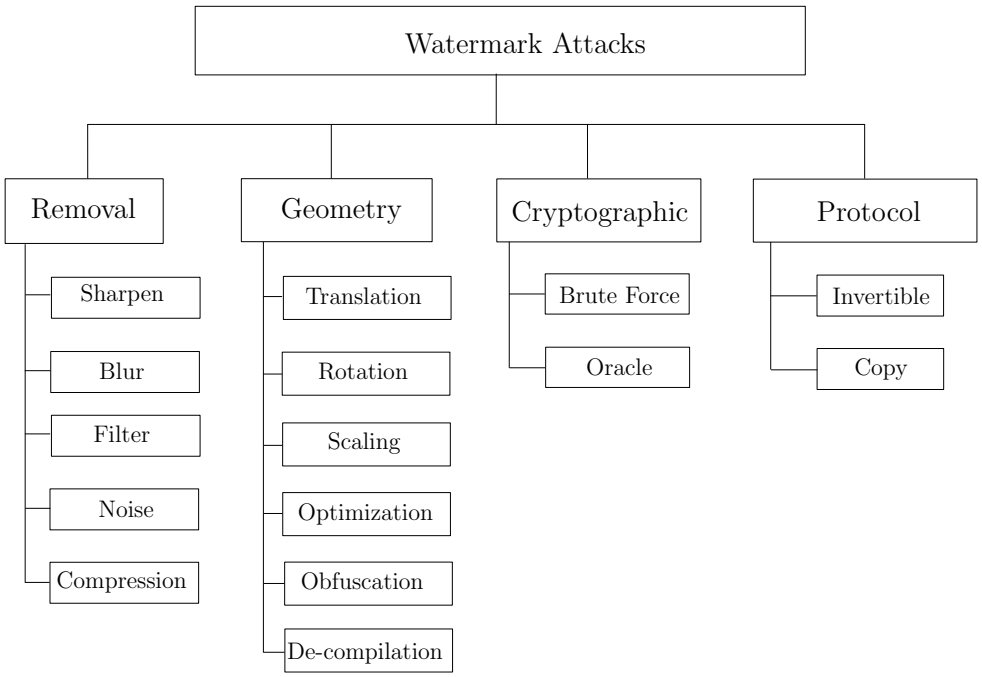


Figure 1.4: Digital watermarking attacks.

- **Removal and interference attacks**. Removal attacks intend to remove the watermark data from the watermarked object. Such attacks exploit the fact that the watermark is usually an additive noise signal present in the host signal. Moreover, interference attacks are those which add additional noise to the watermarked object. Lossy compression, quantization, collusion, denoising, remodulation, averaging, and noise storm are some examples of this category of attacks. The collusion attack occurs when a number of authorized recipients of the multimedia object come together to generate an un-watermarked object by averaging all the different watermarked objects. In software watermarking one of the first things that an adversary may do in an attempt to eliminate a watermark is decompile the application. Once the code has been decompiled the attacker can search for aspects of the code that look suspicious such as dummy methods.

- **Geometric attacks**. Geometric attacks do not actually remove the watermark, but manipulate the watermarked object in such a way that the detector cannot find the watermark data, i.e., distort the watermark. This type of attack includes affine transformations such as rotation, translation, and scaling. Warping, line/column removal and cropping are also included in this family of attacks. Another example of geometric attack is the mosaic attack. As far as it concerns software watermarking, distortive attacks are any semantics preserving code transformation, such as code obfuscation or optimization algorithms. This type of attack is used to distort a watermark such that it is unrecoverable. The advantage of this attack over subtractive attacks is that the adversary need not know the exact location of the watermark. Rather, he can apply the transformation indiscriminately over the application.

- **Cryptographic attacks**. The aim of cryptographic attacks is to crack the security methods in watermarking schemes and thus find a way to remove the embedded watermark information or to embed misleading watermarks. For example, finding the secret watermarking key using exhaustive brute force method is a cryptographic attack. Another example of this type of attack is the oracle attack. In the oracle attack, a non-watermarked object is created when a public watermark detector device is available. These attacks are similar to the attacks used in cryptography.

- **Protocol attacks**. Protocol attacks add the attacker's own watermark onto the data in question. This results in ambiguities on the true owners question. The first protocol attack was proposed by Craver et al. [46]. They introduced the concept of invertible watermarks and showed that for copyright protection purpose watermarks need to be non-invertible. The idea behind inversion is that the attacker subtracts his own watermark from the watermarked data and claims to be the owner of the watermarked data. This can create ambiguity with respect to the true ownership of the data. It has been shown that for copyright protection applications, watermarks need to be non-invertible. Another protocol attack is the copy attack: instead of destroying the watermark, the copy attack estimates a watermark from watermarked data and copies it to some other data, called target data. This attack, in fact, embeds one or several additional watermarks such that it is unclear which the watermark of the original owner was.

1.2.4 Watermarking Techniques

In this section we present a brief overview on the watermarking techniques currently available in the literature on software, image, audio, and text watermarking.

(A) Software Watermarking

The most important software watermarking algorithms currently available in the literature are based on several techniques. In 1996, Davidson and Myhrvold [48] presented the first patented static software watermarking algorithm,where a program is watermarked by reordering its basic blocks. The preliminary concepts of software watermarking also appeared in the paper [51] and the patents [85, 105]. Register allocation algorithm proposed by Qu and Potkonjak [101], inserts a watermark into the interference graph of a program. Each vertex in the graph represents a variable and an edge between two variables indicates that their live ranges overlap. The graph is colored in order to minimize the number of registers required and ensure that two live variables do not share a register. In [38, 106], authors propose a spread-spectrum algorithm which represents a program as a vector and modifies each component of the vector with a small random amount. Algorithms have been also proposed, where pieces of a watermark are encoded as constants within opaque predicates [1, 86]; an opaque predicate is a predicate whose outcome is known a priori. Collberg et al. [35] proposed the path-based algorithm that inserts a watermark in the runtime branch structure of a program to be watermarked. It is based on the observation that the branch structure is an essential part of a program and that it is difficult to analyze such a structure completely because it captures so much of the semantics of the program. Other software watermarking techniques rely on abstract interpretation [37], code re-orderings [112]; see, also

[29, 130, 131] for an exposition of the main results. It is worth noting that many algorithmic techniques on software watermarking have also received patent protection [34, 48, 103, 114].

The algorithm of Davidson and Myhrvold [48] embeds the watermark into a program by reordering the basic blocks of a control flow-graph; note that a static watermark is stored inside programs' code in a certain format and it does not change during the programs' execution. Based on this idea, Venkatesan, Vazirani and Sinha [120] proposed the first graph-based software watermarking algorithm which embeds the watermark by extending a method's control flow-graph through the insertion of a directed subgraph; it is also a static algorithm called `VVS` or `GTW`. In [120], the construction of the directed watermark graph G is not discussed. Collberg et al. [31] 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 (or, for short, RPG) [30], which is a reducible control flow-graph with a 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 [32]; it embeds the watermark through a graph structure which is built on a heap at runtime.

Recently, several software watermarking algorithms have been appeared in the literature that encode watermarks as graph structures [30, 31, 48, 120]. 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* [30]. 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.

Five classes of graph structures have been proposed in the literature for encoding a watermark as graph, the Oriented Parent-Pointer Tree, the Radix Graphs, the Permutation Graphs, the Planted Plane Cubic Trees, and the Reducible Permutation Graphs.

- **Oriented Parent-Pointer Trees**. In oriented parent-pointer trees each node has just one pointer field referencing its parent. In [69], is presented an extended theoretical analysis of enumeration of trees by the exploitation of oriented parent-pointer trees. The parent-pointer data structure has almost no error-correcting properties. An adversary who adds a single node or an edge to a parent-pointer graph may radically change the watermark value it represents [30].

- **Radix-list Watermark Graphs**. Radix-list watermark graphs G_r of order n have the following structure: they are a singly-linked circular list of n nodes, in which each node has an additional pointer field that may point either to *NULL* or to any other node in the list. The idea is simple. First, it is constructed a circular linked list o length k. Then, extra pointer fields are added to each node representing the base-k digit. These graphs have poor error-correcting properties. Node and edge addition attacks on such graphs will force the decoder to enumerate all Hamiltonian subgraphs, in order to construct all possible watermark graphs that could have existed prior to the attacks [30].

- **Permutation Graphs**. Permutation graphs use the same structure as radix graphs, i.e. use the same singly linked circular list, where the extra k pointers encode the permutation of integers $[0, 1, \ldots, k-1]$. A number n is first converted to a permutation π and then the permutation is encoded to a graph. In [29], the algorithms that encode an integer n to permutation π, and the inverse, are presented. The specific class of graphs is more resilient to the modification of an edge in comparison to radix graphs.

- **Planted Plane Cubic Tree**. Planted plane cubic tree (or, for short, PPCT), is essentially a binary tree with one extra node called origin, which has a pointer to the root of the binary tree. Moreover, all leaves are linked into a circular linked list which includes the origin. Each leaf has a self-pointer and every node in a PPCT has two outgoing pointers. The underlying undirected graph is thus cubic (uniform degree 3) except at its leaves and its root. Such data structures are used in any program involving binary search trees, and superficially similar structures arise in any program that has a data type with exactly two pointer fields. According to Catalan number theory [50], a PPCT with n leaves ($2n$ nodes) can represent any integer in the following set: $0, 1, 2, \ldots, \frac{1}{n}C_{n-2}^{n-1}$. The Catalan number $c(n)$ gives the number of unique trees for each n. An integer is encoded as one of the trees.

- **Reducible Permutation Graphs**. Reducible permutation graphs (or, for short, RPG) [29, 30] are very similar to permutation graphs but they closely resemble control flow graphs as they are reducible-flow graphs.

  A flow graph G is reducible if and only if we can partition the edges into two disjoint groups, often called forward edges and back edges, with the following two properties:

  (i) The forward edges from an acyclic graph in which every node can be reached from the initial node of G.

  (ii) The back edges consist only of edges whose heads dominate their tails.

  The reducibility of this family of graphs means that they resemble control-flow graphs constructed from programming constructs such as if, while etc. [29]. RPGs, like CFGs, contain a unique entry node and a unique exit node, a preamble which contains zero or more nodes from which all other nodes can be reached and a body which encodes a watermarking using a self-inverting permutation

(B) Image and Audio Watermarking

Digital image and audio watermarking schemes mainly fall into two broad categories: a) the spatial-domain or time-domain technique, and b) the frequency domain technique. The first

category is the most straightforward way to hide the watermark within the host signal. Since audio signal is an one-dimensional signal, the technique is called time-domain watermarking, whereas spatial-domain technique refers to digital image watermarking, since image is a two-dimension signal. In frequency techniques, image and audio are represented in terms of its frequencies, and watermark is embedded by modifying the frequency coefficients of the host signal.

- **Spatial/Time-Domain Techniques**. According to this method the watermark is embedded into original signal. Least Significant Bits (LSB) is the simplest approach, because the least significant bit carries less relevant information and their modification does not cause perceptible changes. LSB technique is easy to implement to embed the watermark and there are two basic ways of doing this: the lower order bits of the digital audio signal can be fully substituted with a pseudo random (PN) sequence that contains the watermark message m, or the PN-sequence can be embedded into the lower order bit stream. The major disadvantage of LSB techniques is the poor robustness to signal processing operations. Another technique have been proposed, the spread spectrum technique,which encodes the watermark data prior to insertion into the image or audio frequency domain data. Spread spectrum encoding allows multiple narrow band communications signals to be spread across a broad band with out interference.

- **Frequency Domain Techniques**. Compared to spatial domain techniques, frequency domain techniques are more applied. The target of this technique is to insert the watermarks in the spectral coefficients of the image. The most commonly used transforms are the Discrete Cosine Transform (DCT), Discrete Fourier Transform (DFT), and Discrete Wavelet Transform(DWT). The discrete wavelet transforms (DWT) and the discrete cosine transforms (DCT) are implemented very effectively in numerous digital images watermarking scheme. In this new era Singular Value Decomposition (SVD) is also implementing very effectively in the digital image watermarking scheme.

Spatial domain watermarking techniques for images include works of [100, 115], and [123]. Some of the earliest techniques [115] and [123] embed M-sequence into the LSB of the host signal to provide an effective transparent embedding technique. In [123] the authors reshape the M-sequence into two-dimensional watermark blocks which are added and detected on a block-by-block basis. Another spatial-domain technique is proposed in [73], where the blue component of an image in RGB format is watermarked to ensure robustness while remaining fairly insensitive to human visual system (HVS) factors.

Initial research on audio watermarking dates back to the mid-nineties where Bender et al. [8] presented data hiding techniques for audio signals. Several audio watermarking algorithms in time-domain have been proposed. One of the simplest techniques under this category is Least Significant Bit (LSB) alteration. In this technique, LSB of each sample value of the host audio signal is made 0 or 1 depending upon the watermark bit to be embedded [125]. Echo hiding is another audio watermarking technique in time domain which embeds the watermark by introducing an echo [72]. Another category of watermarking techniques in time-domain is Quantization Index Modulation (QIM) watermarking methods [47].

The concept of spread spectrum used in communication systems is also employed in digital watermarking. The basic idea of spread spectrum is to encode audio signal by spreading the

watermark information [44] across as much of the audible spectrum as possible. Using this technique, a watermark can be embedded robustly into a host audio signal without destroying its perceptual quality.

Various frequency-domain audio watermarking techniques were proposed. These techniques can be DFT [126, 127], DCT or DWT [124]. In these approaches, the amplitude or phase of the transformed coefficients are modified with some specified amount in order to carry watermark data.

A lot of work have been proposed in image watermarking in the frequency-domain [43, 98]. DCT domain watermarking can be classified into Global DCT watermarking and Block based DCT watermarking. One of the first algorithms presented by [44] used global DCT approach to embed a robust watermark in the perceptually significant portion of the Human Visual System (HVS). Embedding in the perceptually significant portion of the image has its own advantages because most compression schemes remove the perceptually insignificant portion of the image.

Another domain exploited for embedding the watermark is the wavelet domain. The DWT (Discrete Wavelet Transform) separates an image into a lower resolution approximation image (LL) as well as horizontal (HL), vertical (LH) and diagonal (HH) detail components. The process can then be repeated to computes multiple "scale" wavelet decomposition. DWT is much preferred because it provides both a simultaneous spatial localization and a frequency spread of the watermark within the host image.

Discrete Fourier transform (DFT) transforms a continuous function into its frequency components. It has robustness against geometric attacks like rotation, scaling, cropping, translation etc. DFT shows translation invariance. An extensive analysis of the different techniques proposed in the literature can be found at [98, 104].

(C) Text Watermarking

The previous work on digital text watermarking can be classified in the following categories; an image based approach, a syntactic approach and a semantic approach.

- **Image-based Techniques**. In Image-based approach towards digital text watermarking, text document image is used to embed the watermark. Text is difficult to watermark because of its simplicity, sensitiveness, and low capacity for watermark embedding. Initially attempts in text watermarking tried to treat text as image. Watermark was embedded in the layout and appearance of the text image.

  Brassil, et al. were the first to propose a few text watermarking methods utilizing text image [9, 10]; they also developed document watermarking schemes based on line shifts, word shifts as well as slight modifications to the characters [11]. Maxemchuk, et al. [87, 88, 89] analyzed the performance of these methods, while later Low et al. [80, 81] further analyzed their efficiency. Huang and Yan [64] proposed a text watermarking method based on an average inter-word distance in each line. The distances are adjusted according to the sine-wave of a specific phase and frequency. Feature and pixel level algorithms were also developed which mark the documents by modifying the stroke features such as width or serif [6].

- **Syntactic Techniques**. Text is made up of characters, words, and sentences. Sentences have different syntactic structures. Applying syntactic transformations on text structure

to embed watermark has also been one of the approaches towards text watermarking in the past. In syntactic approach, the syntactic structure of text has been used to embed watermark. Atallah, et al. [5] proposed several methods of natural language watermarking, which opened up a brand-new and challenging research direction for text watermarking. Meral et al. performed morpho-syntactic alterations to the text to watermark it [90]. They also provided an overview of available syntactic tools for text watermarking [91].

- **Semantic Techniques**. In semantic approach, semantics of text are used to embed the watermark in text. The semantic watermarking schemes focus on using the semantic structure of text to embed the watermark. Text contents, verbs, nouns, words and their spellings, acronyms, sentence structure, grammar rules, etc. have been exploited to insert watermark in the text but none of these proved to be resilient and degrade the quality of the text to a large extent. Atallah et al. were the first to propose the semantic watermarking schemes [5]. Later, the synonym substitution method was proposed, in which watermark was embedded by replacing certain words with their synonyms [118]. Sun, et al. [111] proposed noun-verb based technique for text watermarking which used nouns and verbs parsed by semantic networks. Topkara, et al. proposed an algorithm of the text watermarking by using typos, acronyms and abbreviation in the text to embed the watermark [119]. Algorithms were developed to watermark the text using the linguistic approach of presuppositions [92] in which the discourse structure, meaning, and representations are observed and utilized to embed watermark bits. The text pruning and the grafting algorithms were also developed in the past. Another algorithm based on text meaning representation (TMR) strings has also been proposed [82].

- **Structural Techniques**. The structural approach is the most recent approach used for copyright protection of text documents. In this approach, text is not altered, rather it is used to logically embed watermark in it. A text watermarking algorithm for copyright protection of text using occurrences of double letters (aa-zz) in text has recently been proposed [67]. Recently, a significant number of techniques have been proposed in the literature which use Portable Document Format (PDF) files as cover media in order to hide data [12, 13, 76, 77, 78, 79, 133].

1.3 Motivation

Today storing information and data such as documents, images, video, and audio in digital formats is very common. For many people transferring digital files via the internet is a daily activity. Owing to the rapid development of digital technology and the widespread use of the Internet, life becomes increasingly more convenient than previously. However, this increase in internet usage has motivated copyright concerns.

As is well known, due to the nature of digital information, it is easy to make unlimited lossless copies from the original digital source, to modify the content, and to transfer the copies rapidly over the Internet. Creators and owners of Intellectual Property designs want assurances that their content will not be illegally redistributed by consumers, and consumers want assurances that the content they buy is legitimate.

The term intellectual property (IP) refers to a creation of a mind for which a set of exclusive rights are recognized [56]. That creation may have any possible form; for example, it may be a work of art, an invention, literary or artistic work, a discovery or even a phrase. More precisely, IP can be divided into two categories: *industrial property*, which includes inventions (patents), trademarks, industrial designs, and geographic indications of source; and *copyright*, which includes literary and artistic works such as novels, poems, plays, films, video games, software applications, musical works, drawings, paintings, photographs, sculptures, and architectural designs.

The objective of recognizing intellectual property is to encourage innovation. That is because people won't have the incentive to create if they are not legally protected in order to get the social value that they deserve from their creations [83]. Of course the world's evolution and economic growth depends on creations and inventions and that makes intellectual property such an important and vital aspect [68].

Over the last years the internet has been expanding very rapidly and, thus, information is now spread freely, easily and cost-efficiently and that gives a greater importance to intellectual property. Because of the internet, the distribution of intellectual material went out of control. Just the fact that nearly every intellectual material that is produced today is published in digital form or can be transformed into digital form means that it can be easily transmitted free via the internet, without any permission from the creator.

All that urged the adoption of new laws and the development of systems for the protection of intellectual property [49]. But still the cyberspace is chaotic nowadays and that makes it extremely difficult to have any kind of control over it. The figures talk by themselves; according to IFPI (International Federation of the Phonographic Industry) 95% of music downloads are pirated. What is more, a survey from Digital Life America showed us that things aren't any better for the movies. If we also take into account the fact that the internet population is consisted of nearly seven billion we may realize that its power is greater than the law and the systems for protection. Therefore, the demands of copyright protection, ownership demonstration, and tampering verification for digital data are becoming more and more urgent. Among the solutions for these problems, digital watermarking [43] is the most popular one.

1.4 The Structure of the Thesis

The current PhD Thesis consists of 8 Chapters which can be partitioned, according to research issues addressed, into the following four Parts I-IV, namely:

(I) **Background Results** (Chapter 1),

(II) **Encodings** (Chapters 2-3-4),

(III) **Watermarking** (Chapters 5-6-7), and

(IV) **Conclusions** (Chapters 8).

We next briefly discuss our contribution and the main results of the chapters of each of the four Parts I, II, III, and IV.

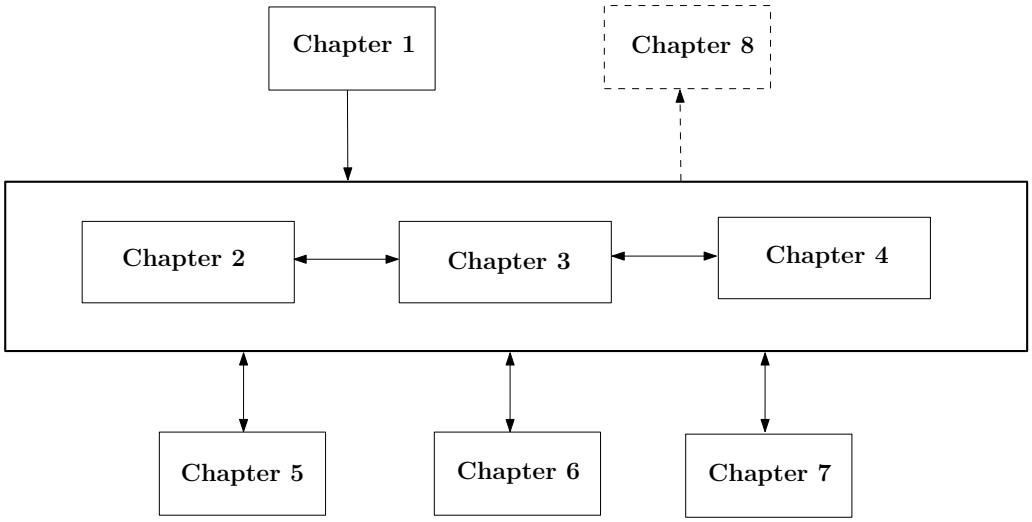


Figure 1.5: The Structure of the 8 Chapters of the Thesis.

- **Part I:** The part, which contains the Chapter 1, provides an introduction to current watermarking techniques and an outline of the problems under consideration, while it briefly presents our research contribution.
- **Part II:** This is the part containing the basic research on encoding watermark members as graph structures through the use of self-inverting permutations (Chapters 2-3) and algorithms for multiple encodings (Chapter 4).
- **Part III:** The three Chapters of this part propose efficient and easily implementable codec algorithms for software watermarking (Chapter 5), image and audio watermarking (Chapter 6) and text watermarking (Chapter 7).
- **Part IV:** This part consists of the Chapter 8, the last chapter of the PhD thesis, which summarizes the main results presented in Chapters 2 to 7, and discusses possible future extensions.

Figure 1.5 schematically depicts the eight Chapters of the Thesis and organizes them in levels and groups according to Parts I-IV.

1.5 Main Results

From the above partitioning of the 8 Chapters and the brief description of the 4 Parts, it becomes obvious that the main research results of our Thesis are presented in Part II and Part III. We next give the organization of the chapters of theses two parts emphasizing the main features of the proposed codec techniques.

Part II, Chapters 2-3-4. In Chapter 2, we define a main data component of our codec system, namely, the self-inverting permutations (SiP), we introduce the notion of a bitonic permutation,

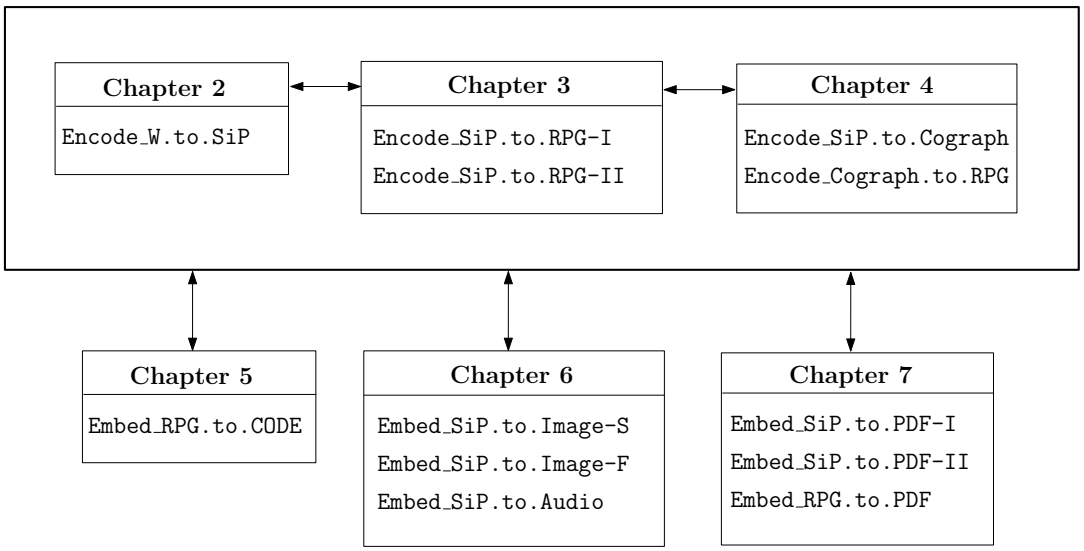


Figure 1.6: Algorithms of the Chapters 2, 3, and 4 (Part II: Encodings) and Chapters 5, 6, and 7 (Part III: Watermarking).

we present our algorithm `Encode-W.to.SiP` for encoding an integer w as a self-inverting permutation π^*, along with the corresponding decoding algorithm `Decode-SiP.to.W`, and finally discuss important properties of the self-inverting permutation π^*.

In Chapter 3, we first define the main graph-based data component of our codec system, namely, the reducible permutation graphs (PRG), we describe the two operational phases of our codec system, and present the structure of our system's reducible permutation graph $F[\pi^*]$. Then, we present the two algorithms, namely `Encode-SiP.to.RPG-I` and `-II` for encoding the self-inverting permutation π^* as a reducible permutation flow-graph $F[\pi^*]$ along with the corresponding decoding algorithms `Decode-RPG.to.SiP-I` and `-II`. Finally, we present the properties of the reducible permutation flow-graph $F[\pi^*]$ and show that node-label or edge modifications on the graph $F[\pi^*]$ can be efficiently detected.

In Chapter 4, we first present the randomize encoding algorithm `Encode_SiP.to.Cograph`, along with its corresponding decoding algorithm, which takes as input a self-inverting permutation π^* and encodes the permutation π^* into a cograph $C[\pi^*]$. Then, we present the algorithm `Encode_Cograph.to.RPG`, along with its corresponding decoding algorithm, which embeds a cograph into an RPG by exploiting the structure and some important algorithmic properties of its cotree.

Part III, Chapters 5-6-7. In Chapter 5, we present our dynamic watermarking model WaterRPG; we first describe its structural and operational components and then the embedding algorithm `Embed_RPG.to.CODE` and the extracting algorithm `Extract_CODE.to.RPG`. Then, we implement our watermarking model in real Java application programs and show two main watermarking approaches supported by the WaterRpg model, namely naive and stealthy. We also evaluate our model under several software watermarking assessment criteria.

In Chapter 6, we describe our primary work on image watermarking in the spatial domain and

give the `Embed_SiP.to.Image-S` and `Extract_SiP.from.Image-S` algorithms. We first present our codec algorithms, `Embed_SiP.to.Image-F` and `Extract_SiP.from.Image-F`, for watermarking images in the frequency domain, and then we expand our idea on image watermarking by applying it in audio watermarking and present our audio watermarking algorithms, i.e., the embedding algorithm `Embed_SiP.to.Audio` and the extracting algorithm `Extract_SiP.from.Audio`.

In Chapter 7, based on the three different representations of self-inverting permutation (SiP), i.e. the two dimensional (2D-representation), the one dimensional (1D-representation), and the RPG-representation (the encoding of permutation π^* as a reducible permutation graph $F[\pi^*]$), we present the algorithms `Embed_SiP.to.PDF-I`, `Embed_SiP.to.PDF-II`, and `Embed_RPG.to.PDF`, along with the corresponding extracting algorithms, for embedding a watermark number (or, equivalently, a self-inverting permutation π^* or a reducible permutation graph $F[\pi^*]$) into a PDF document file.

Chapter 2

Encode Watermark Numbers as Self-inverting Permutations

2.1 Introduction

2.2 Preliminaries

2.3 Self-inverting Permutations (SiP)

2.4 Encode Watermark Numbers as SiPs

2.5 The Structure of Permutation π^*

2.6 Properties of Permutation π^*

2.7 Concluding Remarks

2.1 Introduction

Digital (or, media) watermarking is a technique that is currently being studied to prevent or discourage digital media piracy and copyright infringement. The main idea is simple: a unique identifier is embedded in software, image, audio, or video data through the introduction of errors not detectable by human perception [15].

The Digital Watermarking problem can be described as follows: Embed a structure w into a digital object O, resulting the watermarked object O_w, such that w can be reliably located and efficiently extracted from O_w even after O_w has been subjected to typical transformations. More precisely, a Digital Watermarking System can be defined by the following two functions:

- $\texttt{embed}(O, w, k) \longrightarrow O_w$
- $\texttt{extract}(O_w, k) \longrightarrow w$

where, O is the digital object, w is a watermark, and k is a key; throughout the thesis, we equivalently use the functions $\texttt{decode}(\cdot)$ or $\texttt{recognize}(\cdot)$ to denote the extraction process.

Digital watermarking has made considerable progress and become a popular technique for copyright protection of multimedia information [15, 113]. Recently, research on software watermarking has also received sufficient attention. We next briefly discuss on software watermarking issues since self-inverting permutations have been used in an graph-based watermarking codec system proposed by Collberg et al. [30].

In 1990, Davidson and Myhrvold [48] 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. [120] 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 [120] the construction of a directed graph G (or, watermark graph G) is not discussed. Collberg et al. [31] proposed an implementation of algorithm `GTW`, which they call $\texttt{GTW}_{\texttt{sm}}$, and it is the first publicly available implementation of `GTW`. In $\texttt{GTW}_{\texttt{sm}}$ the watermark is encoded as a reducible permutation graph (RPG) [30], which is a reducible control-flow graph with maximum out-degree of two, mimicking real code.

In $\texttt{GTW}_{\texttt{sm}}$ implementation a watermark value (integer) is encoded as a RPG; in particular, in the enumeration of Collberg et al. [31], an integer n is encoded as the RPG corresponding to the n-th self-inverting permutation. Note that there is a one-to-one correspondence between self-inverting permutations and isomorphic classes of RPGs. Thus, for encoding integers the $\texttt{GTW}_{\texttt{sm}}$ methods uses only those permutations that are self-inverting.

Contribution. Collberg et al. [29, 31] use an enumeration of self-inverting permutations and correspond an integer n to the n-th self-inverting permutation in their enumeration. In this chapter, we present an efficient, resilient to attacks, and easily implementable system for encoding numbers as self-inverting permutations (or SiP, for short); see, Figure 2.1.

More precisely, we present an efficient algorithm which encodes a watermark number (integer) w as a self-inverting permutation π^*. Our algorithm, which we call `Encode_W.to.SiP`, takes as input an integer w, computes its binary representation $b_1 b_2 \cdots b_n$, then constructs a bitonic permutation on $n^* = 2n + 1$ numbers, and finally produces a self-inverting permutation π^* of length n^* in $O(n^*)$ time and space. We also present the corresponding decoding algorithm `Decode_SiP.to.W`, which extracts the integer w from the self-inverting permutation π^* within the same time and space complexity.

The self-inverting permutation π^* encompasses important structural properties, due to the bitonic property used in the construction of π^*, which makes our codec system resilient to attacks. Finally, we point out that our codec system has very low time and space complexity which is $O(n)$, where n is the number of bits in the binary representation of the watermark integer w.

We mention here that, having designed an efficient method for encoding integers as self-inverting permutations, in Chapter 3 we present algorithms for encoding a self-inverting permutation π^* as a reducible permutation flow-graph.

Road Map. The chapter is organized as follows: In Section 2.2 we establish the notation and related terminology. In Section 2.3 we define a main data component of our codec system, namely, the self-inverting permutations (SiP). In Section 2.4 we first introduce the notion of a bitonic

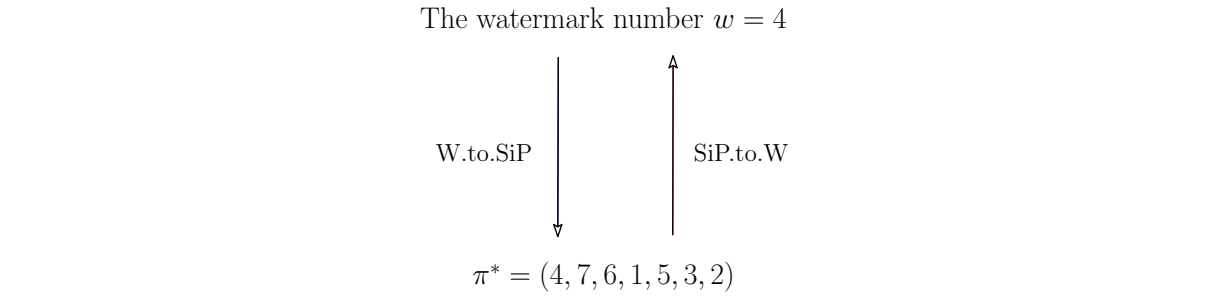


Figure 2.1: The main data components used by the algorithms of our codec system: (i) the watermark number w and (ii) the self-inverting permutation π^*.

permutation, and then present our algorithm `Encode-W.to.SiP` for encoding an integer w as a self-inverting permutation π^* and the corresponding decoding algorithm `Decode-SiP.to.W`. In Section 2.5 we analyze the structure of a self-inverting permutation π^* produced by algorithm `Encode_W.to.SiP`. In Section 2.6 we summarize the properties of the self-inverting permutation π^*. Finally, in Section 2.7 we conclude the chapter and discuss possible future extensions.

2.2 Preliminaries

We next introduce some definitions that are key to our algorithms for encoding numbers as self-inverting permutations. 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 $\pi_1 = 1$, $\pi_2 = 4$, etc. By π_i^{-1} we denote the position in the sequence of number $i \in N_n$; in our example, $\pi_4^{-1} = 2$, $\pi_7^{-1} = 4$, $\pi_3^{-1} = 6$, etc [52]. The *length* of a permutation π is the number of elements in π. The *reverse* of π, denoted π^R, is the permutation $\pi^R = (\pi_n, \pi_{n-1}, \ldots, \pi_1)$. The *inverse* of π is the permutation $\tau = (\tau_1, \tau_2, \ldots, \tau_n)$ with $\tau_{\pi_i} = \pi_{\tau_i} = i$. Clearly, every permutation has a unique inverse, and the inverse of the inverse is the original permutation.

A *subsequence* of a permutation $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$ is a sequence $\alpha = (\pi_{i_1}, \pi_{i_2}, \ldots, \pi_{i_k})$ such that $i_1 < i_2 < \cdots < i_k$. If, in addition, $\pi_{i_1} < \pi_{i_2} < \cdots < \pi_{i_k}$, then we say that α is an *increasing subsequence* of π, while if $\pi_{i_1} > \pi_{i_2} > \cdots > \pi_{i_k}$ we say that α is a *decreasing subsequence* of π; the length of a subsequence α is the number of elements in α.

A *cycle* of π is a sequence $c = (\pi_{i_1}, \pi_{i_2}, \ldots, \pi_{i_p})$ such that $\pi_{i_1}^{-1} = \pi_{i_2}$, $\pi_{i_2}^{-1} = \pi_{i_3}$, $\cdots$, $\pi_{i_p}^{-1} = \pi_{i_1}$. For example, the permutation $\pi = (4, 7, 1, 6, 5, 3, 2)$ has three cycles $c_1 = (4, 1, 3, 6)$, $c_2 = (7, 2)$, and $c_3 = (5)$ of lengths 4, 2, and 1, respectively. In general, a permutation π contains ℓ cycles, where $1 \leq \ell \leq n$; for example, the identity permutation λ over the set N_n contains n cycles of length 1. Throughout the thesis, a cycle of length k is referred to as a k-cycle.

A *left-to-right maximum* (*left-to-right minimum*, resp.) of π is an element π_i, $1 \leq i \leq n$, such that $\pi_i > \pi_j$ ($\pi_i < \pi_j$, resp.) for all $j < i$. The increasing (decreasing, resp.) subsequence $\alpha = (\pi_{i_1}, \pi_{i_2}, \ldots, \pi_{i_k})$ is a *left-to-right maxima (minima, resp.) subsequence* if it consists of all the left-to-right maxima (minima, resp.) of π; clearly, $\pi_{i_1} = \pi_1$. For example, the left-to-right maxima subsequence of the permutation $\pi = (4, 2, 6, 1, 9, 3, 7, 5, 12, 11, 8, 10)$ is $(4, 6, 9, 12)$, while

the left-to-right minima subsequence of π is $(4, 2, 1)$.

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

We say that an element i of a permutation π over the set N_n *dominates* the element j if $i > j$ and $\pi_i^{-1} < \pi_j^{-1}$. An element i *directly dominates* (or d-dominates, for short) the element j if i dominates j and there exists no element k in π such that i dominates k and k dominates j [94]. 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 directly dominates the elements $1, 6$.

2.3 Self-inverting Permutations (SiP)

We next define the main component of our codec system, namely, the self-inverting permutations (SiP), and prove properties that are used as key-objects in our algorithms for encoding numbers as reducible permutation graphs.

Definition 2.1. Let $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$ be a permutation over the set N_n. A *self-inverting permutation* (or involution) is a permutation that is its own inverse: $\pi_{\pi_i} = i$.

The definition of the inverse of a permutation implies that a permutation is a self-inverting permutation iff all its cycles are of length 1 or 2; hereafter, we shall denote a 2-cycle by (x, y) with $x > y$ and a 1-cycle by (x) or, equivalently, (x, x).

Definition 2.2. A sequence $C = (c_1, c_2, \ldots, c_k)$ of all the 2- and 1-cycles of a self-inverting permutation π is a *decreasing cycle representation* of π if $c_1 \succ c_2 \succ \cdots \succ c_k$ where $c_i = (a_i, b_i) \succ c_j = (a_j, b_j)$ (with $a_i \geq b_i$ and $a_j \geq b_j$) if $b_i > b_j$, $1 \leq i, j \leq k$. The cycle c_k containing the smallest element among the elements of the cycles is the *minimum element* of the sequence C.

Lemma 2.1. *Let $c_i = (x, y)$ and $c_j = (z, w)$ be two 2-cycles of a self-inverting permutation π such that $x > y$ and $z > w$.*

(i) If $x > z > y$ and $\pi_x^{-1} < \pi_z^{-1} < \pi_y^{-1}$, then $w > y$ and $\pi_w^{-1} < \pi_y^{-1}$. Symmetrically, if $x > w > y$ and $\pi_x^{-1} < \pi_w^{-1} < \pi_y^{-1}$, then $x > z$ and $\pi_x^{-1} < \pi_z^{-1}$. In either case, it holds that $x > z > w > y$ and $\pi_x^{-1} < \pi_z^{-1} < \pi_w^{-1} < \pi_y^{-1}$.

(ii) If $x > z > w$ and $\pi_x^{-1} < \pi_z^{-1} < \pi_w^{-1}$, then $w > y$ and $\pi_w^{-1} < \pi_y^{-1}$. Symmetrically, if $z > w > y$ and $\pi_z^{-1} < \pi_w^{-1} < \pi_y^{-1}$, then $x > z$ and $\pi_x^{-1} < \pi_z^{-1}$. In either case, it holds that $x > z > w > y$ and $\pi_x^{-1} < \pi_z^{-1} < \pi_w^{-1} < \pi_y^{-1}$.

Proof. The lemma follows from the fact that in a self-inverting permutation π (of size n) for every $i = 1, 2, \ldots, n$, $\pi_{\pi_i} = i$ or equivalently for any 2-cycle (a, b) it holds that $a = \pi_b^{-1}$ and $b = \pi_a^{-1}$. ∎

Theorem 2.1. *Let $S_i = (x_1, x_2, \ldots, x_{k_i})$ be the ith decreasing subsequence of a self-inverting permutation π. Then,*

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

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

Proof. We use induction on i. For the basis case, we consider $i = 1$, that is, the 1st decreasing subsequence $S_1 = (x_1, x_2, \ldots, x_{k_1})$. Then, for each $j = 1, 2, \ldots, \lfloor \frac{k_1}{2} \rfloor$ it suffices to show that (i) if $x_j \in S_1$ belongs to a 2-cycle $(x_j, x_{j'})$ then $x_{j'}$ belongs to S_1, and (ii) in fact, $x_{j'}$ is the $(k_1 - j + 1)$-st element in S_1. We use induction on j. Statements (i) and (ii) clearly hold for $j = 1$, since the first and the smallest element of π belong to the 1st decreasing subsequence of π, they form a 2-cycle of π, and in fact they are the first and the last element of S_1, respectively. For the inductive hypothesis, suppose that statements (i) and (ii) hold for $j = j_0 - 1$ where $1 \leq j_0 - 1 < \lfloor \frac{k_i}{2} \rfloor$; we next show that they hold for $j = j_0$ where $2 \leq j_0 \leq \lfloor \frac{k_i}{2} \rfloor$. Let (x_{j_0}, x') be the 2-cycle of the self-inverting permutation π to which x_{j_0} belongs; then, Lemma 2.1(i) for the 2-cycles $(x_{j_0-1}, x_{k_1-j_0+2})$ and (x_{j_0}, x') implies that $x' > x_{k_1-j_0+1}$ and $\pi^{-1}_{x'} < \pi^{-1}_{x_{k_1-j_0+1}}$.

Now, suppose for contradiction that $x' \notin S_1$. Since $x' \notin S_1$, there must exist t in π such that

$$\pi^{-1}_{x_{j_0}} \; < \; \pi^{-1}_t \; < \; \pi^{-1}_{x'} \qquad \text{and} \qquad t \; < \; x'.$$

These inequalities imply that $\pi^{-1}_t > \pi^{-1}_{x_{j_0}} = x' > t$, that is, the element t cannot belong to an 1-cycle. Thus, t belongs to a 2-cycle of π; let it be (s, t) with $s > t$ since $s = \pi^{-1}_t > x' > t$. Lemma 2.1(ii) for the 2-cycles (x_{j_0}, x') and (s, t) implies that $s < x_{j_0}$ and $\pi^{-1}_s < \pi^{-1}_{x_{j_0}}$, which in turn imply that x_{j_0} should not belong to S_1, a contradiction. Therefore, x' belongs to S_1 as well.

Moreover, $x' = x_{k_1-j_0+1}$. If not, let $t' = x_{k_1-j_0+1}$. Then

$$x' \; > \; t' \; > \; x_{k_1-j_0+2} \qquad \text{and} \qquad \pi^{-1}_{x'} \; < \; \pi^{-1}_{t'} \; < \; \pi^{-1}_{x_{k_1-j_0+2}}.$$

These inequalities imply that $\pi^{-1}_{t'} > \pi^{-1}_{x'} = x_{j_0} > x' > t'$; hence, t' does not belong to an 1-cycle. Therefore, t' belongs to a 2-cycle (s', t') with $s' > t'$. Then, Lemma 2.1(ii) for the 2-cycles (x_{j_0}, x') and (s', t') implies that $s' > x_{j_0}$ and $\pi^{-1}_{s'} < \pi^{-1}_{x_{j_0}}$. Additionally, Lemma 2.1(i) for the 2-cycles $(x_{j_0-1}, x_{k_1-j_0+2})$ and (s', t') implies that $x_{j_0-1} > s'$ and $\pi^{-1}_{x_{j_0-1}} < \pi^{-1}_{s'}$. However, these contradict the fact that x_{j_0-1} and x_{j_0} are in consecutive positions in the decreasing subsequence S_1. Thus, $x' = x_{k_1-j_0+1}$, as desired.

The inductive step for the induction on i can be easily proved by taking into account (i) that the ith decreasing subsequence of a permutation π is precisely the 1st decreasing subsequence of the permutation that results from π after having removed from it the elements of the 1st, 2nd, ..., $(i-1)$st decreasing subsequence of π, and (ii) that both elements of a 2-cycle get removed while removing a decreasing subsequence. ∎

2.4 Encode Watermark Numbers as SiPs

In this section, we first introduce the notion of a *Bitonic Permutation* and then we present two algorithms, namely `Encode_W.to.SiP` and `Decode_SiP.to.W`, for encoding an integer w into a

self-inverting permutation π^* and for extracting it from π^*, respectively. Both algorithms run in $O(n)$ time, where $n = \log_2 w$ is the length of the binary representation of the integer w [28].

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 it either monotonically increases and then monotonically decreases, or 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.

Let $\pi = (\pi_1, \pi_2, \ldots, \pi_i, \pi_{i+1}, \ldots, \pi_n)$ be a bitonic permutation over the set N_n that first monotonically increases and then monotonically decreases and let π_i be the *leftmost* element of π such that $\pi_i > \pi_{i+1}$; note that π_i is the maximum element of π. Then, we call the sequence $X = (\pi_1, \pi_2, \ldots, \pi_{i-1})$ *the increasing subsequence* of π and the sequence $Y = (\pi_i, \pi_{i+1}, \ldots, \pi_n)$ *the decreasing subsequence* of π.

Notations: We next give some notations and terminology we shall use throughout the chapter. If $B_1 = b_1 b_2 \cdots b_n$ and $B_2 = d_1 d_2 \cdots d_m$ are 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. For a binary number $B = b_1 b_2 \cdots b_n$, $flip(B) = b'_1 b'_2 \cdots b'_n$ is the binary number such that $b'_i = 0$ (1, resp.) if and only if $b_i = 1$ (0, resp.), $1 \leq i \leq n$. Note that for any binary number B, $flip(flip(B)) = B$.

2.4.1 Algorithm Encode_W.to.SiP

We next present an algorithm for encoding an integer as a self-inverting permutation without having to consult a list of all self-inverting permutations. Our algorithm takes as input an integer w, computes its binary representation, and then produces a self-inverting permutation π^* in time linear in the length of the binary representation of w. The proposed algorithm is the following:

Algorithm `Encode_W.to.SiP`

1. Compute the binary representation B of w and let n be the length of B;

2. Construct the binary number $B' = \underbrace{00\cdots0}_{n}||B||0$ of length $n^* = 2n+1$, and then the binary sequence B^* of $flip(B')$;

3. Construct the sequence $X = (x_1, x_2, \ldots, x_k)$ of the 0s' positions and the sequence $Y = (y_1, y_2, \ldots, y_m)$ of the 1s' positions in B^* from left to right, where $k + m = n^*$;

4. Construct the bitonic permutation $\pi^b = X||Y^R = (x_1, x_2, \ldots, x_k, y_m, y_{m-1}, \ldots, y_1)$ over the set $N_{n^*} = N_{2n+1}$;

5. for $i = 1, 2, \ldots, n = \lfloor n^*/2 \rfloor$ do
{*construct a 2-cycle with the i-th element of π^b from left and the i-th from right*}
construct the 2-cycle $c_i = (\pi^b_i, \pi^b_{n^*-i+1})$;
construct the 1-cycle $c_i = (\pi^b_{n+1})$;

6. Initialize the permutation π^* to the identity permutation $(1, 2, \ldots, 2n+1)$;
   for each 2-cycle (π_i, π_j) computed at Step 5, set $\pi^*_{\pi_i} = \pi_j$ and $\pi^*_{\pi_j} = \pi_i$;

7. Return the permutation π^* (which by construction is self-inverting).

Example 2.1 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 $B' = 000011000$ and the binary sequence $B^* = (1, 1, 1, 1, 0, 0, 1, 1, 1)$ of $flip(B')$; we compute the sequences $X = (5, 6)$ and $Y = (1, 2, 3, 4, 7, 8, 9)$, and then construct the bitonic permutation $\pi^b = (5, 6, 9, 8, 7, 4, 3, 2, 1)$ on $n^* = 9$ numbers; since $n^* = 9$ is odd, we form 4 2-cycles $(5, 1)$, $(6, 2)$, $(9, 3)$, $(8, 4)$ and the 1-cycle (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 $O(n)$ length, where n is the number of bits in the binary representation of w (see Figure 2.2). Thus, the whole encoding process requires $O(n)$ time and space, and the following theorem holds:

Theorem 2.2. *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.*

2.4.2 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. Its time complexity is linear in the length of the permutation π^*. We next describe the proposed algorithm:

Algorithm `Decode_SiP.to.W`

1. Compute the decreasing cycle representation $C = (c_1, c_2, \ldots, c_k)$ of the self-inverting permutation $\pi^* = (\pi_1, \pi_2, \ldots, \pi_{n^*})$, where $n^* = 2n + 1$;

2. Construct the permutation π^b of length n^* as follows:
   set $i = 1$ and $j = n^*$;
   while the set C is not empty, do the following:
   select the minimum element c of the set C, i.e., the cycle containing the minimum among
   the elements of all the cycles in C;
   Case 1: the selected cycle c has length 2 and let $c = (a, a')$ with $a > a'$:
   set $\pi^b_i = a$ and $\pi^b_j = a'$;
   $i = i + 1$ and $j = j - 1$;
   Case 2: the selected cycle c has length 1 and let $c = (a)$:

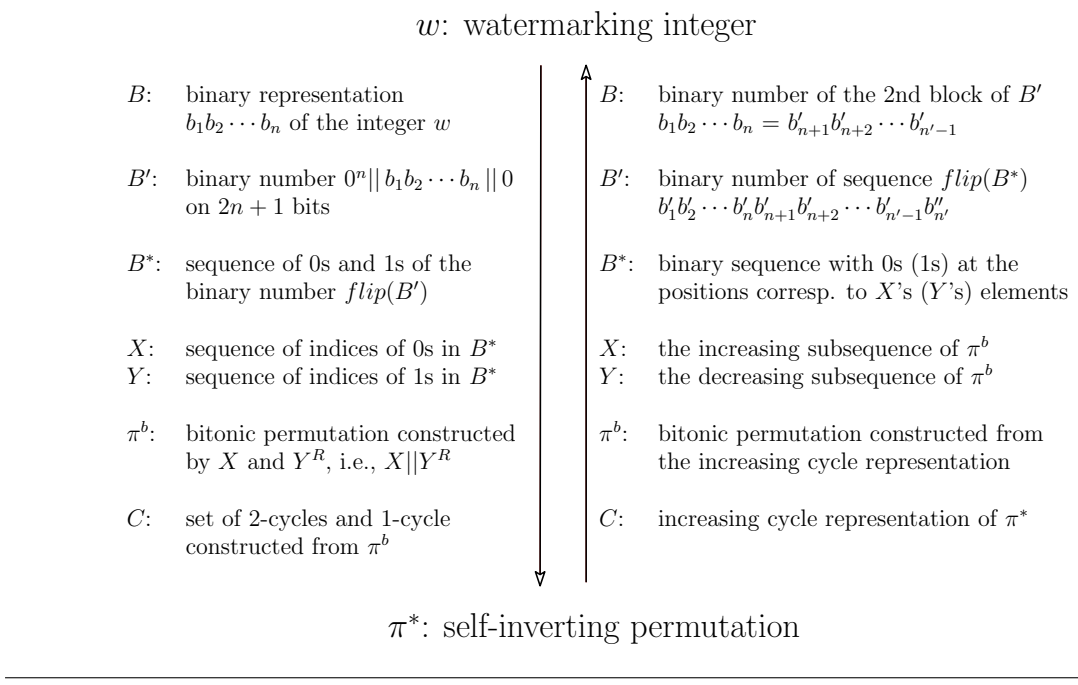


Figure 2.2: The main data components used by our two codec Algorithms `Encode_W.to.SiP` and `Decode_SiP.to.W`.

set $\pi^b_i = a$ and $i = i + 1$;
remove the cycle c from C;

3. Find the increasing subsequence $X = (\pi^b_1, \pi^b_2, \ldots, \pi^b_k)$ of π^b and then the decreasing subsequence $Y = (\pi^b_{k+1}, \pi^b_{k+2}, \ldots, \pi^b_{n^*})$;

4. Construct the binary sequence $B^* = (b_1, b_2, \ldots, b_{n^*})$ by setting 0 in positions $\pi^b_1, \pi^b_2, \ldots, \pi^b_k$ and 1 in positions $\pi^b_{k+1}, \pi^b_{k+2}, \ldots, \pi^b_{n^*}$;

5. Compute $B' = flip(B^*) = (b'_1, b'_2, \ldots, b'_n, b'_{n+1}, \ldots, b'_{n^*-1}, b'_{n^*})$;

6. Return the decimal value w of the binary number $B = b'_{n+1}b'_{n+2}\cdots b'_{n^*-1}$.

The decoding algorithm is essentially the reverse of the encoding algorithm; its correctness relies on the fact that $flip(flip(B)) = B$.

Example 2.2 Let $\pi^* = (5,6,9,8,1,2,7,4,3)$ be a self-inverting permutation produced by Algorithm `Encode_W.to.SiP`. The decreasing cycle representation of π^* is the sequence $C = \big((7), (8,4), (9,3), (6,2), (5,1)\big)$; these cycles are processed in order from right to left in C and we construct the permutation $\pi^b = (5,6,9,8,7,4,3,2,1)$; then, we compute the increasing subsequence $X = (5,6)$ and the decreasing subsequence $Y = (9,8,7,4,3,2,1)$; we then construct the binary sequence $B^* = (1,1,1,1,0,0,1,1,1)$ of length 9; we flip the elements of B^* and construct the sequence $B' = (0,0,0,0,1,1,0,0,0)$; the decimal value of 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 $O(n)$ length as the encoding algorithm (see, Figure 2.2). Thus, we obtain the following result:

Theorem 2.3. *Let w be an integer (whose binary representation has length n) and let π^* be the self-inverting permutation of length $n^* = 2n+1$ produced by algorithm* `Encode_W.to.SiP` *to encode w. Algorithm* `Decode_SiP.to.W` *correctly extracts w from π^* in $O(n^*) = O(n)$ time and space.*

2.5 The Structure of Permutation π^*

We next analyze the structure of a self-inverting permutation π^* produced by the encoding algorithm `Encode_W.to.SiP`. We distinguish the following two cases.

Special Case: Suppose that $w = 2^n - 1$, that is, all the bits in w's binary representation are 1. Then, according to algorithm `Encode_W.to.SiP`, $B' = 0^n 1^n 0$, $\pi^b = (n+1,\, n+2,\, \ldots,\, 2n+1,\, n,\, n-1,\, \ldots,\, 2,\, 1)$, and $\pi^* = (n+1,\, n+2,\, \ldots,\, 2n,\, 1,\, 2,\, \ldots,\, n,\, 2n+1)$. Note that π^* is the concatenation of the (increasing) sub-permutations π_1^* and π_2^* where

$$\pi_1^* \;=\; (n+1,\, n+2,\, \ldots,\, 2n) \qquad \text{and} \qquad \pi_2^* \;=\; (1,\, 2,\, \ldots,\, n,\, 2n+1). \tag{2.1}$$

General Case: Suppose that $w \neq 2^n - 1$. Then, the concatenation of the binary representation of w with a trailing 0 consists of a_1 1s, followed by b_1 0s, followed by a_2 1s, followed by b_2 0s, and so on, followed by a_ℓ 1s, followed by b_ℓ 0s where $\ell \geq 1$, $a_i, b_i > 0$, and $a_1 < n$.

For convenience, let $A_j = \sum_{t=1}^{j} a_t$ and $\Gamma_j = \sum_{t=1}^{j}(a_t + b_t)$ for $0 \leq j \leq \ell$; note that $A_0 = 0$, $\Gamma_0 = 0$, $\Gamma_\ell = n+1$, and A_ℓ is equal to the number of 1s in the binary representation of w. Additionally, let $B_j = \sum_{t=1}^{j} b_t$ for $0 \leq j \leq \ell$, and $\overline{B}_j = \left(\sum_{t=1}^{\ell} b_t\right) - B_j$, which imply that $B_0 = \overline{B}_\ell = 0$, $\overline{B}_{j-1} = \overline{B}_j + b_j$, and $\Gamma_\ell = A_\ell + B_\ell = A_\ell + \overline{B}_0$.

Then, according to algorithm `Encode_W.to.SiP`, $B' = 0 \cdots 0 || B || 1$ where B is the binary representation of w, and $\pi^b = X \,||\, Y^R$ where X is the sequence of the 0s' positions in $flip(B')$ and Y^R is the reverse of the sequence of the 1s' positions in $flip(B')$, that is,

$$\begin{aligned} X \;&=\; X_1 \,||\, X_2 \,||\, \ldots \,||\, X_i \,||\, \ldots \,||\, X_\ell \\ &=\; \Big(\underbrace{n+1, \ldots, n+a_1}_{a_1},\; \underbrace{n+\Gamma_1+1, \ldots, n+\Gamma_1+a_2}_{a_2},\; \ldots,\; \underbrace{n+\Gamma_{i-1}+1, \ldots, n+\Gamma_{i-1}+a_i}_{a_i}, \\ &\qquad \ldots,\; \underbrace{n+\Gamma_{\ell-1}+1, \ldots, n+\Gamma_{\ell-1}+a_\ell}_{a_\ell}\Big) \end{aligned}$$

and

$$\begin{aligned} Y^R \;&=\; Y_\ell^R \,||\, \ldots \,||\, Y_i^R \,||\, \ldots \,||\, Y_1^R \,||\, Y_0^R \\ &=\; \Big(\underbrace{n+\Gamma_\ell, \ldots, n+\Gamma_\ell - b_\ell + 1}_{b_\ell},\; \ldots,\; \underbrace{n+\Gamma_i, \ldots, n+\Gamma_i - b_i + 1}_{b_i},\; \ldots, \\ &\qquad \underbrace{n+\Gamma_1, \ldots, n+\Gamma_1 - b_1 + 1}_{b_1},\; \underbrace{n,\, n-1, \ldots, 2,\, 1}_{n}\Big). \end{aligned}$$

(Note that $\Gamma_{i-1} + a_i = \Gamma_i - b_i$.)

Next, because the total length of the concatenation of $X_1, X_2, \ldots, X_\ell, Y_\ell^R, \ldots, Y_1^R$ is $n+1$ whereas the length of Y_0^R is n, the cycles constructed in Step 5 of algorithm `Encode_W.to.SiP` are:

- *2-cycles*: the 2-cycles in increasing order of their second elements from 1 to n are (note that $A_0 = \overline{B}_\ell = 0$, $A_\ell + \overline{B}_0 - 1 = \Gamma_\ell - 1 = (n+1) - 1 = n$, and $\Gamma_{i-1} + a_i = \Gamma_i - b_i$):
  $(n+1,\ A_0+1),\ (n+2,\ A_0+2),\ \ldots,\ (n+a_1,\ A_1)$,
  $(n+\Gamma_1+1,\ A_1+1),\ (n+\Gamma_1+2,\ A_1+2),\ \ldots,\ (n+\Gamma_1+a_2,\ A_2)$,
  $\ldots$
  $(n+\Gamma_{i-1}+1,\ A_{i-1}+1),\ (n+\Gamma_{i-1}+2,\ A_{i-1}+2),\ \ldots,\ (n+\Gamma_{i-1}+a_i,\ A_i)$,
  $\ldots$
  $(n+\Gamma_{\ell-1}+1,\ A_{\ell-1}+1),\ (n+\Gamma_{\ell-1}+2,\ A_{\ell-1}+2),\ \ldots,\ (n+\Gamma_{\ell-1}+a_\ell,\ A_\ell)$,
  $(n+\Gamma_\ell,\ A_\ell+\overline{B}_\ell+1),\ (n+\Gamma_\ell-1,\ A_\ell+\overline{B}_\ell+2),\ \ldots,\ (n+\Gamma_\ell-b_\ell+1,\ A_\ell+\overline{B}_{\ell-1})$,
  $\ldots$
  $(n+\Gamma_i,\ A_\ell+\overline{B}_i+1),\ (n+\Gamma_i-1,\ A_\ell+\overline{B}_i+2),\ \ldots,\ (n+\Gamma_i-b_i+1,\ A_\ell+\overline{B}_{i-1})$,
  $\ldots$
  $(n+\Gamma_2,\ A_\ell+\overline{B}_2+1),\ (n+\Gamma_2-1,\ A_\ell+\overline{B}_2+2),\ \ldots,\ (n+\Gamma_2-b_2+1,\ A_\ell+\overline{B}_1)$,
  $(n+\Gamma_1,\ A_\ell+\overline{B}_1+1),\ (n+\Gamma_1-1,\ A_\ell+\overline{B}_1+2),\ \ldots,\ (n+\Gamma_1-b_1+2,\ A_\ell+\overline{B}_0-1)$;
- *1-cycle*: the 1-cycle involves the last element of Y_1^R, that is, it is $(n+\Gamma_1-b_1+1) = (n+a_1+1)$.

Therefore, the self-inverting permutation π^* is the concatenation of π_1^* and π_2^*, where

$$\pi_1^* \ = \ X_1 \,||\, X_2 \,||\, \ldots \,||\, X_\ell \,||\, Y_\ell^R \,||\, \ldots \,||\, Y_2^R \,||\, \Psi_1^R \tag{2.2}$$

with $\Psi_1^R = (n+\Gamma_1, \ldots, n+\Gamma_1-b_1+2)$ (note that Ψ_1^R is empty if $b_1 = 1$ otherwise it is equal to Y_1^R without its last element $n+\Gamma_1-b_1+1 = n+a_1+1$) and

$$\begin{aligned}
\pi_2^* \ = \ \Big(& \underbrace{1, 2, \ldots, A_1}_{a_1},\ n+A_1+1,\ \underbrace{n, n-1, \ldots, n-B_1+2}_{b_1-1}, \\
& \underbrace{A_1+1, A_1+2, \ldots, A_2}_{a_2},\ \underbrace{n-B_1+1, n-B_1, \ldots, n-B_2+2}_{b_2}, \\
& \ldots, \\
& \underbrace{A_{i-1}+1, A_{i-1}+2, \ldots, A_i}_{a_i},\ \underbrace{n-B_{i-1}+1, n-B_{i-1}, \ldots, n-B_i+2}_{b_i}, \\
& \ldots, \\
& \underbrace{A_{\ell-1}+1, A_{\ell-1}+2, \ldots, A_\ell}_{a_\ell},\ \underbrace{n-B_{\ell-1}+1, n-B_{\ell-1}, \ldots, n-B_\ell+2}_{b_\ell} \Big)
\end{aligned} \tag{2.3}$$

(note that the last element of π_2^* is $n-B_\ell+2 = (\Gamma_\ell-1)-B_\ell+2 = (A_\ell+B_\ell-1)-B_\ell+2 = A_\ell+1$). It is interesting to note that π_1^* is a permutation of the numbers $n+1, n+2, \ldots, 2n+1$ except for $n+a_1+1$; in turn, π_2^* is a permutation of the numbers $1, 2, \ldots, n$ and $n+a_1+1$. Additionally, π_2^* consists of the a_1 numbers $1, 2, \ldots, A_1$ followed by b_1 numbers larger than $A_\ell+1$, followed by the a_2 numbers $A_1+1, A_1+2, \ldots, A_2$, followed by b_2 numbers larger than $A_\ell+1$, and so on, up to the a_ℓ numbers $A_{\ell-1}+1, A_{\ell-1}+2, \ldots, A_\ell$ that are followed by the b_ℓ numbers $A_\ell+b_\ell, A_\ell+b_\ell-1, \ldots, A_\ell$

(note that $n-B_{\ell-1}+1 = (\Gamma_\ell-1)-B_{\ell-1}+1 = (A_\ell+B_\ell-1)-B_{\ell-1}+1 = A_\ell+(B_\ell-B_{\ell-1}) = A_\ell+b_\ell$).

Example 2.3 Consider $w = 220$. The binary representation of w is 11011100, and hence $n = 8$. From the concatenation of the binary representation of w with the additional trailing 0, we have that $\ell = 2$, $a_1 = 2$, $b_1 = 1$, $a_2 = 3$, $b_2 = 3$; in turn, $A_1 = a_1 = 2$, $A_2 = a_1 + a_2 = 5$, $B_1 = \overline{B}_1 = b_1 = 1$, $B_2 = \overline{B}_0 = b_1 + b_2 = 4$, $\Gamma_1 = A_1 + B_1 = 3$, and $\Gamma_2 = A_2 + B_2 = 9$. Then,

$$\pi^b = (\underbrace{0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0}_{n=8},\ \underbrace{1,\ 1,\ 0,\ 1,\ 1,\ 1,\ 0,\ 0}_{n=8},\ 0)$$

and $\pi_1^* = (9,\ 10,\ 12,\ 13,\ 14,\ 17,\ 16,\ 15)$ and $\pi_2^* = (1,\ 2,\ 11,\ 3,\ 4,\ 5,\ 8,\ 7,\ 6)$; thus,

$$\pi^* = (9,\ 10,\ 12,\ 13,\ 14,\ 17,\ 16,\ 15,\ 1,\ 2,\ 11,\ 3,\ 4,\ 5,\ 8,\ 7,\ 6).$$

2.6 Properties of Permutation π^*

In this section, we analyze the structure of the self-inverting permutation π^* produced by the algorithms `Encode_W.to.SiP` and discuss its properties with respect to resilience to attacks.

Collberg et al. [30, 32] describe several techniques for encoding watermark integers in graph structures. Based on the fact that there is a one-to-one correspondence, say, C, between self-inverting permutations and isomorphism classes of reducible permutation graphs (or, for short, RPG), Collberg et al. [30] proposed a polynomial-time algorithm for encoding the integer w as the RPG corresponding to the wth self-inverting permutation π in C. This encoding exploits only the inversion property of a self-inverting permutation; it does not incorporate any other structural properties.

In our codec system an integer w is encoded as a self-inverting permutation π^* using a construction technique which captures into π^* important structural properties. The main properties of our self-inverting permutation π^* produced by the algorithm `Encode_W.to.SiP` can be summarized into the following four categories:

- **Odd-length property**: By construction, the self-inverting permutation π^* has always odd length.

- **One-cycle property**: The self-inverting permutation π^* always contains one, and only one, cycle of length 1;

- **Bitonic property**: The self-inverting permutation π^* is constructed from the bitonic sequence $\pi^b = X||Y^R$, where X and Y are increasing subsequences (see, Step 4 of our encoding algorithm `Encode_W.to.SiP`), and thus the bitonic property of π^b is encapsulated in the cycles of π^*.

- **Block property**: The algorithm `Encode_W.to.SiP` takes the binary representation of the integer w and initially constructs the binary number B' (see, Step 2). The binary representation of B' consists of three parts (or, blocks):

  (i) the first part contains the leftmost n bits, each equal to 0,

(ii) the second part contains the next n bits which form the binary representation of the integer w, and

(iii) the third part of length one contains a bit 0.

This property affects the construction of both subsequences X and Y (see, Bitonic property), and thus the cycles of π^*.

The above properties enable us to identify any single change (in some cases, multiple changes) made by an attacker to permutation π^* produced by our encoding algorithm `Encode_W.to.SiP`.

2.7 Concluding Remarks

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 of w. We also presented the corresponding decoding algorithm; it takes as input a self-inverting permutation π^* produced by the encoding algorithm and returns the encoding 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 encoding approach enable us to encode any integer w as self-inverting permutation π^* of any length $n^* \geq 3$; indeed, π^* can be constructed over the set N_{n^*}, where $n^* = 2\lceil \log w \rceil + 1$.

Chapter 3

Encoding SiPs as Reducible Permutation Graphs

3.1 Introduction

3.2 Preliminaries

3.3 Reducible Permutation Graphs (RPG)

3.4 The Structure of Our Codec System

3.5 Encode SiPs as Reducible Permutation Graphs

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

3.7 Detecting Attacks

3.8 Concluding Remarks

3.1 Introduction

Although digital watermarking has made considerable progress and become a popular technique for copyright protection of multimedia information [15, 113], research on software watermarking has recently received considerable attention. Software watermarking has been studied for about 10-15 years and has been pioneered by Christian Collberg and his research team. 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, 35, 37, 38, 86, 95, 101, 106]).

Recently, several software watermarking algorithms have appeared in the literature that encode watermarks as graph structures among which our algorithms proposed in this thesis which encode watermarks numbers as Reducible Permutation Graphs (this chapter) or Cographs (Chapter 5). In general, such encodings make use of an encoding function `encode` which converts a

watermarking number w into a graph G and of a decoding function `decode` that converts the graph G into the number w, that is,

- $\texttt{encode}(w) \rightarrow G$
- $\texttt{decode}(G) \rightarrow w$

We usually call the pair $(\texttt{encode}, \texttt{decode})_G$ as a *graph codec* [30]. 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:

(i) Appropriate Graph Types: Graphs in class $\mathcal{G}$ should be directed having appropriate properties (e.g., nodes with small outdegree) so that their structure resembles that of real program graphs;

(ii) High Resiliency: The function $decode(G)$ should be insensitive to small changes of G (e.g., insertions or deletions of a constant number of nodes or/and edges); that is, if $G \in \mathcal{G}$ and $\texttt{decode}(G) \rightarrow w$ then $\texttt{decode}(G') \rightarrow w$ with $G' \approx G$;

(iii) Small Size: The size $|P_w| - |P|$ of the embedded watermark should be small, where P_w and P are the watermarked and the original program, respectively;

(iv) Efficient Codecs: The functions `encode` and `decode` should be computed in polynomial time.

We briefly mention here that there are two general categories of software watermarking algorithms: the static and the dynamic algorithms [32]. A static watermark is stored inside program code P 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, and it might be retrieved by analyzing the data structures built when watermarked program is running. Further discussion of static and dynamic watermarking issues can be found in [48, 85, 120] and [32].

In 1996, Davidson and Myhrvold [48] presented the first published static software watermarking algorithm; it embeds the watermark by reordering the basic blocks of a control flow-graph, whereas the first dynamic watermarking algorithm `CT` was proposed by Collberg and Thomborson [32]; it embeds the watermark through a graph structure which is built on a heap at runtime.

Based on the idea of Davidson and Myhrvold algorithm, Venkatesan, Vazirani, and Sinha [120] 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 [120] the construction of a directed graph G (or watermark graph G) is not discussed. Collberg et al. [31] proposed an implementation of `GTW`, which they call $\texttt{GTW}_{\texttt{sm}}$; this 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) [30], 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.

Attacks: A successful attack against the watermarked program P_w prevents the recognizer from extracting the watermark while not seriously harming the performance 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 an application program.

- Additive attacks: Embed a new watermark into the watermarked software, so that 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, so that the detector gives a misleading result.

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.

Contribution. For encoding integers some recently proposed watermarking methods uses only those permutations that are self-inverting. Collberg et al. [30, 32] based on the fact that there is a one-to-one correspondence between self-inverting permutations and isomorphism classes of RPGs, proposed a polynomial algorithm for encoding any integer w as the RPG corresponding to the w-th self-inverting permutation π in this correspondence [30]. This encoding exploits only the fact that a self-inverting permutation is its own inverse.

In this chapter, we present an efficient and easily implementable system for encoding numbers as reducible permutation graphs, whose structure resembles that of real program graphs, through the use of self-inverting permutations (or SiP, for short); see, Figure 3.1.

More precisely, having designed an efficient method for encoding integers as self-inverting permutations (see, Chapter 3), we next describe algorithms for encoding a self-inverting permutation π^* of length n^* as a reducible permutation flow-graph $F[\pi^*]$. In particular, we propose the algorithm `Encode_SiP.to.RPG-I` which exploits domination relations on the elements of π^* and properties of a DAG representation of π^*, and the algorithm `Encode_SiP.to.RPG-II` which exploits neighborhood relations on specific decreasing subsequences of permutation π^*, and both produce a reducible permutation flow-graph $F[\pi^*]$ on $n^* + 2$ nodes; in both approaches, the whole encoding process takes $O(n^*)$ time and requires $O(n^*)$ space. The corresponding decoding algorithm `Decode_RPG.to.SiP-I` extracts the self-inverting permutation π^* from the reducible permutation graph $F[\pi^*]$ by first converting the graph $F[\pi^*]$ into a directed tree $T_d[\pi^*]$ and then applying DFS-search on $T_d[\pi^*]$, while the decoding algorithm `Decode_RPG.to.SiP-II` extracts π^* from $F[\pi^*]$ by also first converting $F[\pi^*]$ into a directed tree $T_s[\pi^*]$ and then finding pairs of nodes on specific paths in $T_s[\pi^*]$. The decoding process takes time and space linear in the size of the flow-graph $F[\pi^*]$, that is, both decoding algorithms take $O(n^*)$ time and space; recall that the length of the permutation π^* and the size of the flow-graph $F[\pi^*]$ are both $O(n^*) = O(n)$, where $n = \lceil \log_2 w \rceil$.

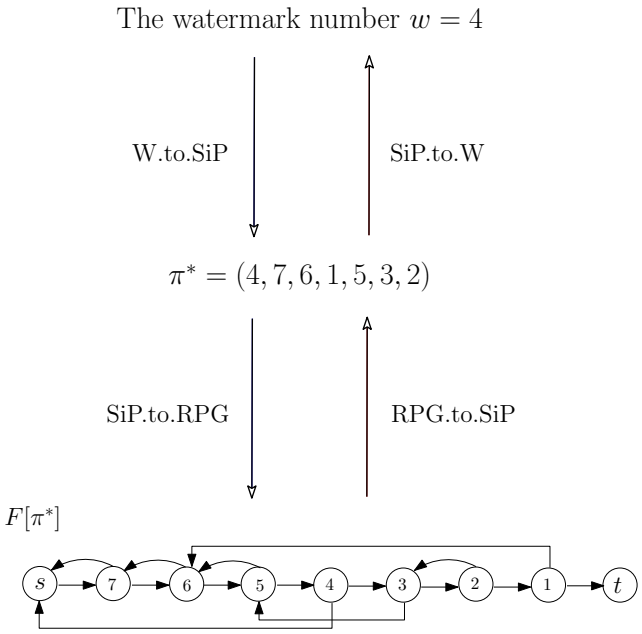


Figure 3.1: The main data components used by the algorithms of our codec system: (i) the watermark number w, (ii) the self-inverting permutation π^*, and (iii) the reducible permutation graph $F[\pi^*]$.

Our codec $(\texttt{encode}, \texttt{decode})_{F[\pi^*]}$ system incorporates several important properties and characteristics which make it appropriate for use in a real software watermarking environment. In particular, the reducible permutation flow-graph $F[\pi^*]$ resembles the graph data structures built by real programs since its maximum outdegree does not exceed two and it has a unique root node. The flow-graph $F[\pi^*]$ is highly insensitive to small edge-changes and fairly insensitive to small node-changes of $F[\pi^*]$; on the other hand, the properties of the graph $F[\pi^*]$ enable us to correct such edge changes.

Road Map. The chapter is organized as follows: In Section 3.2 we establish the notation and related terminology, and we present background results. In Section 3.3 we define the main graph-based data component of our codec system, namely, the reducible permutation graphs (PRG). In Section 3.4 we describe the two operational phases of our codec system and present the structure of our system's reducible permutation graph $F[\pi^*]$. In Section 3.5 we present the two algorithms, namely `Encode-SiP.to.RPG-I` and `-II` for encoding the self-inverting permutation π^* as a reducible permutation flow-graph $F[\pi^*]$ along with the corresponding decoding algorithms `Decode-RPG.to.SiP-I` and `-II`. In Section 3.6 we present the properties of the reducible permutation flow-graph $F[\pi^*]$, while in Section 3.7 we show that node-label or edge modifications on the graph $F[\pi^*]$ can be efficiently detected. Finally, in Section 3.8 we conclude the chapter and discuss possible future extensions.

3.2 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, $deg(x) = |N(x)|$. For a node x of a directed graph G, the number of directed edges coming in x is called the *indegree* of x and the number of directed edges leaving x is its *outdegree*.

A *path* in an undirected 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.

Let T be a rooted tree. The parent of a node x of T is denoted by $p(x)$, whereas the node set containing the children of x in T is denoted by $ch(x)$. The root node of a tree is said to be at *level* 0, while the level of any other node x is equal to the level of x's parent increased by 1. Let L_i denote the set of nodes at the i-th level of T, for each value of i from 0 to the height of the tree T.

3.3 Reducible Permutation Graphs (RPG)

A *flow-graph* is a directed graph with an initial node from which all other nodes are reachable. A directed graph G is *strongly connected* if for every pair of vertices x, y of G there is a directed path in G from x to y. A node y is an *entry* for a subgraph H of the graph G if there is an edge (x, y) in G such that $y \in H$ and $x \notin H$.

Definition 3.1. A flow-graph is *reducible* if it does not have a strongly connected subgraph with two (or more) entries.

There are at least two other equivalent definitions, as Theorem 3.1 shows. These definitions use a few more graph-theoretic concepts. An edge (x, x) (for some node x) is a *cycle-edge*. A depth first search (DFS) traversal of a flow-graph partitions its edges into *tree edges* (making up a spanning tree known as a *DFS tree*), *forward edges* (pointing to a successor in the spanning tree), *back edges* (pointing to a predecessor in the spanning tree, plus the cycle-edges), and *cross edges* (the remaining edges). Tree edges, forward edges, and cross edges form a dag known as a *DFS dag*.

Theorem 3.1. *[59, 60] Let F be a flow-graph. The following three statements about F are equivalent:*

(i) the graph F is reducible;

(ii) the graph F has a unique DFS dag;

(iii) the graph F can be transformed into a single node by repeated application of the transformations $\mathcal{T}_1$ and $\mathcal{T}_2$, where $\mathcal{T}_1$ removes a cycle-edge, and $\mathcal{T}_2$ picks a non-initial node y that has only one incoming edge (x, y) and glues nodes x and y.

3.4 The Structure of our Codec System

In this section, we describe the main structural and data components of our codec system which encodes an integer w into a reducible permutation graph through the use of self-inverting permutations produced the algorithm `Encode_W.to.SiP` (see, Chapter 2).

For a watermark number w, our codec system uses two main data components: (i) the self-inverting permutation π^* and (ii) the reducible permutation graph $F[\pi^*]$; these components are depicted in Figure 3.1. The same figure also shows the two main phases of our system's process:

(I) **Phase W–SiP:** it uses two algorithms, one for encoding the watermark number w into a self-inverting permutation π^* and the other for extracting w from π^*;

(II) **Phase SiP–RPG:** this phase uses two algorithms as well, one for encoding the self-inverting permutation π^* into a reducible permutation graph $F[\pi^*]$ and the other for extracting π^* from $F[\pi^*]$.

Our codec system encodes an integer w as a self-inverting permutation π^* using a construction technique which captures into π^* important structural properties. As we discussed in Chapter 3, these properties enable an attack-detection system to identify edge and/or node modifications made by an attacker to π^*; we briefly mention here, the odd-length property (the self-inverting permutation π^* has always odd length), the one-cycle property (π^* contains always one, and only one, cycle of length 1), the bitonic property, and the block property (see, Section 2.6). Moreover, the encoding approach adopted by our system enables it to encode any integer w as a self-inverting permutation π^* of length $n^* = 2n + 1$, where $n = 2\lceil \log_2 w \rceil + 1$.

The reducible permutation graph $F[\pi^*]$ produced by our system's algorithms consists of $n^* + 2$ nodes, say, $u_{n^*+1}, u_{n^*}, \ldots, u_i, \ldots, u_0$, which include:

(A) **A header node**: it is a root node with outdegree one from which all other nodes of the graph $F[\pi^*]$ are reachable; note that every control flow-graph has such a node. In $F[\pi^*]$ the header node is denoted by $s = u_{n^*+1}$;

(B) **A footer node**: it is a node with outdegree zero that is reachable from all other nodes of the graph $F[\pi^*]$. Every control flow-graph has such a node representing the exit of the method. In $F[\pi^*]$ the footer node is denoted by $t = u_0$;

(C) **The body**: it consists of n^* nodes $u_{n^*}, u_{n^*-1}, \ldots, u_i, \ldots, u_1$ each with outdegree two. In particular, each node u_i $(1 \leq i \leq n^*)$ has exactly two outgoing pointers: one points to node u_{i-1} and the other points to a node u_m with $m > i$; recall that $u_{n^*+1} = s$ and $u_0 = t$.

By construction, the reducible permutation graph $F[\pi^*]$ is of order (i.e., number of nodes) $n^* + 2$ and size (i.e., number of edges) $2n^* + 1$. Thus, since $n^* = 2n + 1$, both the order and size of graph $F[\pi^*]$ are of $O(n)$, where $n = 2\lceil \log_2 w \rceil + 1$.

Recall that our contribution in this chapter has to do with both the W–SiP and the SiP–RPG phase. We design and analyze algorithms for encoding a watermark number w as a SiP π^* and algorithms for encoding a SiP π^* as a reducible permutation flow-graph $F[\pi^*]$ along with the corresponding decoding algorithms; we also show properties of our codec system that prevent edge and/or node modification attacks.

$$\pi^* = (6, 3, 2, 9, 8, 1, 11, 5, 4, 10, 7)$$

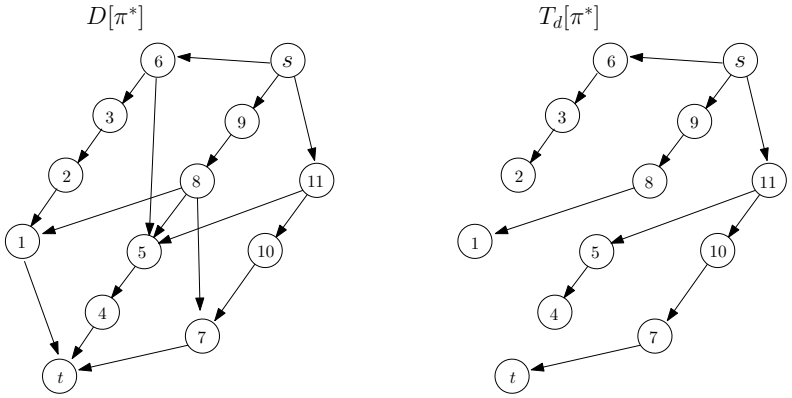


Figure 3.2: The DAG $D[\pi^*]$ of the self-inverting permutation π^* and the corresponding Dmax-tree $T_d[\pi^*]$.

3.5 Encode SiPs as Reducible Permutation Graphs

In this section, we concentrate on the system's phase SiP–RPG and present efficient algorithms for encoding a self-inverting permutation π^* into a reducible permutation graph $F[\pi^*]$ as well as the corresponding decoding algorithms for extracting the permutation π^* from the graph $F[\pi^*]$.

We present two such encoding algorithms yielding two different reducible permutation graphs. The former applies to any permutation π and relies on domination relations on the elements of π whereas the latter applies to a self-inverting permutation π^* produced in any way and relies on the decreasing subsequences of π^*. We mention that the restricted structure of a self-inverting permutation π^* produced by algorithm `Encode_W.to.SiP` (as discussed in Chapter 2) enables us to obtain the decreasing subsequences of π^* in time linear in its size and, more importantly, makes the encoding of π^* into graph $F[\pi^*]$ robust and resilient to attacks.

In light of our encoding algorithm `Encode_W.to.SiP`, the two proposed encoding algorithms provide two different ways to encode the same watermark value w into two different reducible permutation graphs and thus can be very useful for multiple watermarking.

3.5.1 Encode SiPs as Reducible Permutation Graphs - I

The proposed algorithm, which we call `Encode_SiP.to.RPG-I`, 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 a DAG representation $D[\pi^*]$ of the permutation π^* [23]; in fact, it uses a parent-relation of a tree obtained from the graph $D[\pi^*]$ defined below. The whole encoding process takes $O(n^*)$ time and requires $O(n^*)$ space, where n^* is the length of the input self-inverting permutation π^*.

Next, we first describe the main ideas and the structures behind our encoding algorithm. In particular, given a self-inverting permutation π^* we construct a directed acyclic graph, a directed

tree and define specific d-domination relations on the elements of π^*.

DAG Representation $D[\pi^*]$: We construct the directed acyclic graph $D[\pi^*]$ by exploiting the d-domination relation of the elements of π^* as follows: (i) for every element i of π^*, we create a corresponding vertex v_i and we add it into the vertex set $V(D[\pi^*])$ of $D[\pi^*]$; (ii) for every pair of vertices (v_i, v_j) where $v_i, v_j \in V(D[\pi^*])$, we add the directed edge (v_i, v_j) in $E(D[\pi^*])$ if the element i d-dominates the element j in π^*; (iii) we create two dummy vertices $s = v_{n^*+1}$ and $t = v_0$ and we add them both in $V(D[\pi^*])$; then, we add in $E(D[\pi^*])$ the directed edge (s, v_i) for every v_i with indegree equal to 0, and the edge (v_j, t) for every v_j with outdegree equal to 0.

Figure 3.2 depicts the graph $D[\pi^*]$ of the permutation $\pi^* = (6, 3, 2, 9, 8, 1, 11, 5, 4, 10, 7)$. Note that, by construction, $i > j$ for every directed edge (v_i, v_j) of $D[\pi^*]$ since the element j of π^* is d-dominated by the element i.

Dmax-domination Relations: Let d-dom(j) be the set of all the elements of the permutation π^* which d-dominates the element j and dmax(j) be the element of the set d-dom(j) with maximum value; similarly, d-dom(v_j) is the set of all the nodes v_i of the graph $D[\pi^*]$ such that i d-dominates j in π^* and let dmax(v_j) = $v_{\text{dmax}(j)}$, $1 \leq j \leq n^*$. For example, in Figure 3.2 d-dom(5)=(6,8,11) and dmax(5)=11, and d-dom(7)=(8,10) and dmax(7)=10.

We say that the element i *dmax-dominates* j (node v_i dmax-dominates v_j, resp.) if i = dmax(j) (v_i = dmax(v_j), resp.). The ordered pair (i, j) of elements of π^* (the ordered pair (v_i, v_j) of nodes of $D[\pi^*]$, resp.) are in a dmax-domination relation if i dmax-dominates j (v_i dmax-dominates v_j, resp.).

By definition, the element i = dmax(j) of permutation π^*, i.e., i dmax-dominates j, is the rightmost element on the left of j in π^* which d-dominates j; equivalently, the node v_i = dmax(v_j) of graph $D[\pi^*]$ is the parent of the node v_j with the maximum label.

Dmax-tree $T_d[\pi^*]$: We construct the directed tree $T_d[\pi^*]$, which we call Dmax-tree, by exploiting the dmax-domination relation on the nodes of $D[\pi^*]$. The Dmax-tree $T_d[\pi^*]$ is simply constructed as follows:

(i) construct the D-dag $D[\pi^*]$;

(ii) delete all the directed edges (v_i, v_j) from $D[\pi^*]$ if v_i and v_j are not in dmax-domination relation, i.e., $v_i \neq$ dmax(v_j).

The Dmax-tree $T_d[\pi^*]$ of the permutation $\pi^* = (6, 3, 2, 9, 8, 1, 11, 5, 4, 10, 7)$ is shown in Figure 3.2. We point out that the construction of the Dmax-tree $T_d[\pi^*]$ can also be done directly from permutation π^* by finding the dmax-domination relation of each element of π^*; note that $s = v_{n^*+1}$ dominates all the elements of π^*.

Algorithm Encode_SiP.to.RPG-I

Given a self-inverting permutation π^* of length n^*, our proposed algorithm `Encode_W.to.SiP-I` works as follows: first, it computes the dmax-domination relation of each of the n elements of the self-inverting permutation π^* (Step 1), and then, it constructs a directed graph $F[\pi^*]$ on $n^* + 2$ nodes using the dmax-domination relation of the elements of the permutation π^* (Steps 2 and 3).

Now we present the encoding algorithm in detail.

Algorithm `Encode_SiP.to.RPG-I`

1. for each element $i \in \pi^*$, $1 \leq i \leq n^*$, do
- set $P(i) = m$, where $m = \text{dmax}(i)$, i.e., m is the element from d-dom(i) with the maximum value;

2. Construct a directed graph $F[\pi^*]$ on $n^* + 2$ vertices as follows:
- $V(F[\pi^*]) = \{s = u_{n^*+1}, u_{n^*}, \ldots, u_1, u_0 = t\}$;
- for $i = n^*, n^* - 1, \ldots, 0$ do
  add the edge (u_{i+1}, u_i) in $E(F[\pi^*])$;

3. for each vertex $u_i \in V(F[\pi^*])$, $1 \leq i \leq n^*$, do
- add the edge (u_i, u_m) in $E(F[\pi^*])$ where $m = P(i)$;

4. Return the graph $F[\pi^*]$.

Time and Space Complexity. The most time-consuming step of the algorithm is the computation of the dmax-domination relation on each element of π^* (Step 1). On the other hand, the construction of the reducible permutation flow-graph $F[\pi^*]$ on $n^* + 2$ nodes requires only the forward pointers (Step 2) which can be trivially computed, and the dmax-domination pointers (Step 3) which can be computed using the function $P()$.

Returning to Step 1, we observe that the element m is the max-indexed element on the left of the element i in the permutation π^* that is greater than i. Thus, the function $P()$ can be alternatively computed using the input permutation as follows:

(i) insert the value $n^* + 1$ into an initially empty stack S;

(ii) for each element $\pi_i^* \in \pi^*$, $i = 1, 2, \ldots, n^*$, do the following:
  while the element at the the top of S is less than π_i^* do
    remove from S the element at its top;
  $P(\pi_i^*)$ = element at the top of S;
  insert π_i^* in stack S;

For the correctness of this procedure, note that the contents of the stack S are in decreasing order from bottom to top; in fact, at the completion of the processing of element π_i^*, S contains (from top to bottom) the left-to-right maxima of the reverse subpermutation $(\pi_i^*, \pi_{i-1}^*, \ldots, \pi_1^*, n^* + 1)$. Additionally, it is important to observe that the value $n^* + 1$ at the bottom of the stack S is never removed.

The time to process element π_i^* in step (ii) is $O(1 + t_i)$ where t_i is the number of elements popped from the stack S while processing π_i^*. Since the number of pops from S does not exceed the number of pushes in S and since each element of the input permutation π^* is inserted exactly once in S, 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 3.2. *The algorithm* `Encode_SiP.to.RPG-I` *for encoding a self-inverting permutation π^* of length n^* as a reducible permutation flow-graph $F[\pi^*]$ requires $O(n^*)$ time and space.*

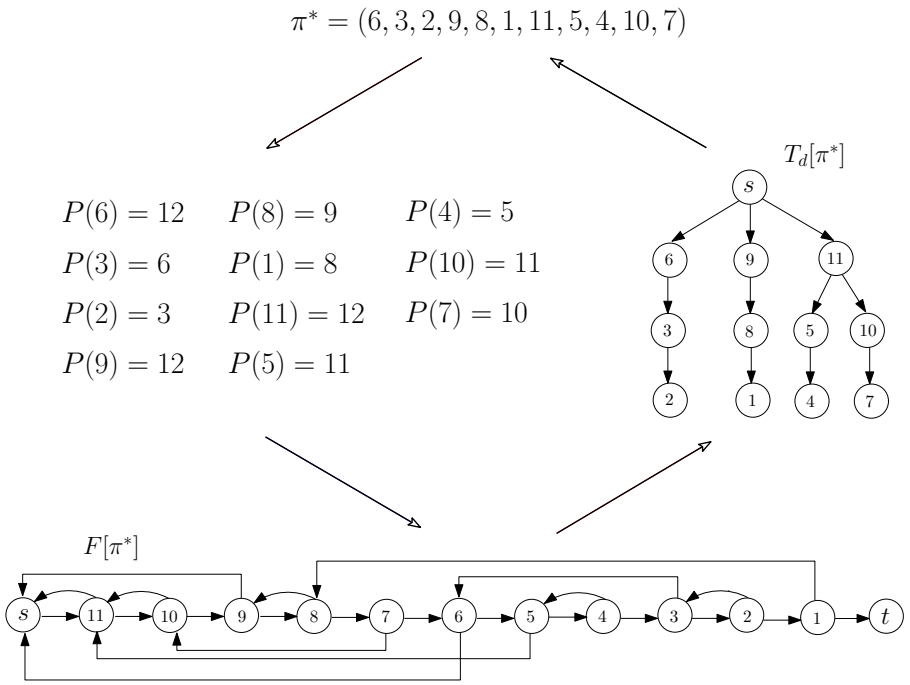


Figure 3.3: The main structures used or constructed by the algorithms `Encode_SiP.to.RPG-I` and `Decode_RPG.to.SiP-I`, that is, the self-inverting permutation π^*, the values of function $P()$, the reducible permutation graph $F[\pi^*]$, and the Dmax-tree $T_d[\pi^*]$.

Algorithm Decode_RPG.to.SiP-I

The algorithm `Encode_SiP.to.RPG-I` produces a 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 implementable 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-I`, which takes time and space linear in the size of the flow-graph $F[\pi^*]$, and is easily implementable: 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 constructed by algorithm `Encode_SiP.to.RPG-I`, and produces a self-inverting permutation π^* of length n^*; it works as follows:

Algorithm `Decode_RPG.to.SiP-I`

1. Delete the directed edges (u_{i+1}, u_i) from the edge set $E(F[\pi^*])$, $1 \leq i \leq n^*$, and the node $t = u_0$ from $V(F[\pi^*])$;

2. Flip all the remaining directed edges of the graph $F[\pi^*]$;
let $T_d[\pi^*]$ be the resulting tree with nodes $s, u_1, u_2, \ldots, u_{n^*}$;

3. Perform DFS-search on the tree $T_d[\pi^*]$ starting at node s by always proceeding to the minimum-labeled child node and compute the DFS discovery time $d[u]$ of each node u of $T_d[\pi^*]$;

4. Order the nodes $u_1, u_2, \ldots, u_{n^*}$ of the tree $T_d[\pi^*]$ by their DFS discovery time $d[]$ and let $\pi = (u'_1, u'_2, \ldots, u'_{n^*})$ be the resulting order, where $d[u'_i] < d[u'_j]$ for $i < j$, $1 \leq i, j \leq n^*$;

5. Return $\pi^* = \pi$.

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

Theorem 3.3. *Let $F[\pi^*]$ be a reducible permutation flow-graph of size $O(n^*)$ produced by the algorithm* `Encode_SiP.to.RPG-I`. *The algorithm* `Decode_RPG.to.SiP-I` *decodes the graph $F[\pi^*]$ in $O(n^*)$ time and space.*

3.5.2 Encode SiPs as Reducible Permutation Graphs - II

The proposed encoding algorithm, which we call `Encode_SiP.to.RPG-II`, takes as input a self-inverting permutation π^* of length $n^* = 2n + 1$ produced by Algorithm `Encode_W.to.SiP` and constructs a reducible permutation flow-graph $F[\pi^*]$ by using the properties of the decreasing subsequences of π^* [21].

Decreasing Subsequences of π^*: For the special case in which $w = 2^n - 1$ (that is, all the bits in w's binary representation are 1), the fact that both sub-permutations π_1^* and π_2^* are increasing (see, Eq. 2.1) implies that the self-inverting permutation π^* has $n + 1$ decreasing subsequences which are in order: $(n + 1,\ 1)$, $(n + 2,\ 2)$, ..., $(2n,\ n)$, and $(2n + 1)$.

Next, if $w \neq 2^n - 1$, the sub-permutation π_1^* (see, Eq. 2.2) has the following $A_\ell + 1$ decreasing subsequences in order:

- the a_1 subsequences $(n + 1)$, $(n + 2)$, ..., $(n + a_1)$ of length 1 each;
- the subsequence $(n + \Gamma_1 + 1,\ n + \Gamma_1,\ \ldots,\ n + a_1 + 2)$ of length b_1;
- the $a_2 - 1$ subsequences $(n + \Gamma_1 + 2)$, $(n + \Gamma_1 + 3)$, ..., $(n + \Gamma_1 + a_2)$ of length 1 each;
- for $i = 2, 3 \ldots, \ell - 1$,

  the subsequence $(n + \Gamma_i + 1,\ n + \Gamma_i,\ \ldots,\ n + \Gamma_{i-1} + a_i + 1)$ of length $b_i + 1$ and

  the $a_{i+1} - 1$ subsequences $(n + \Gamma_i + 2)$, $(n + \Gamma_i + 3)$, ..., $(n + \Gamma_i + a_{i+1})$ of length 1 each;
- the subsequence $(n + \Gamma_\ell,\ n + \Gamma_\ell - 1,\ \ldots,\ n + \Gamma_{\ell-1} + a_\ell + 1)$ of length b_ℓ.

In turn, the sub-permutation π_2^* (see, Eq. 2.3) has an equal number (i.e., $A_\ell + 1$) of decreasing subsequences. In order, they are:

- the a_1 subsequences (1), (2), ..., (A_1) of length 1 each;
- the subsequence $(n + A_1 + 1,\ n,\ n - 1,\ \ldots,\ n - B_1 + 2,\ A_1 + 1)$ of length $b_1 + 1$;

- the $a_2 - 1$ subsequences $(A_1 + 2)$, $(A_1 + 3)$, ..., (A_2) of length 1 each;
- for $i = 2, 3 \ldots, \ell - 1$,

  the subsequence $(n - B_{i-1} + 1,\ n - B_{i-1},\ \ldots,\ n - B_i + 2,\ A_i + 1)$ of length $b_i + 1$ and the $a_{i+1} - 1$ subsequences $(A_i + 2)$, $(A_i + 3)$, ..., (A_{i+1}) of length 1 each;
- the subsequence $(n - B_{\ell-1} + 1,\ n - B_{\ell-1},\ \ldots,\ n - B_\ell + 2)$ of length b_ℓ.

Then we can show the following result.

Lemma 3.1. *The decreasing subsequences of π^* can be computed by concatenating the corresponding decreasing subsequences of π_1^* and of π_2^*.*

Proof: It suffices to show that the last (smallest) element of the i-th decreasing subsequence of π_1^* is larger than the first (largest) element of the i-th decreasing subsequence of π_2^*. Indeed, this is true for the first $a_1 = A_1$ decreasing subsequences, since the (single) element of each of these subsequences of π_1^* is larger than n, whereas the (single) element of each of these subsequences of π_2^* is at most equal to $A_1 = a_1 \leq n$. For the $(a_1 + 1)$-st decreasing subsequence, we have that the smallest element of this subsequence of π_1^* is $n + a_1 + 2$ whereas the largest element of that of π_2^* is $n + A_1 + 1$; clearly, $n + a_1 + 2 = n + A_1 + 2 > n + A_1 + 1$. For the remaining decreasing subsequences, we observe that the elements of all these subsequences of permutation π_1^* are at least equal to $n + \Gamma_1 + 2 \geq n + 2$, whereas the elements of those of π_2^* are at most equal to $n - B_1 + 1 \leq n + 1$. ∎

Example 3.1 For the self-inverting permutation $\pi^* = (5, 6, 9, 8, 1, 2, 7, 4, 3)$ of Example 4.2, we have that $\pi_1^* = (5, 6, 9, 8)$ with decreasing subsequences (5), (6), and (9, 8), and $\pi_2^* = (1, 2, 7, 4, 3)$ with decreasing subsequences (1), (2), and (7, 4, 3). In turn, the decreasing subsequences of π^* are: (5, 1), (6, 2), and (9, 8, 7, 4, 3).

Algorithm Encode_SiP.to.RPG-II

Our `Encode_SiP.to.RPG-II` algorithm works as follows: (i) first, it computes the decreasing subsequences $S_1, S_2, \ldots, S_k$ of the permutation π^* and then (ii) it constructs a directed graph $F[\pi^*]$ on $n^* + 2$ nodes using the subsequences $S_1, S_2, \ldots, S_k$. The algorithm takes $O(n^*)$ time and requires $O(n^*)$ space.

Next, we present the proposed encoding algorithm in detail (see, Figure 3.4).

Algorithm `Encode_SiP.to.RPG-II`

1. Compute the decreasing subsequences S_1, S_2, ..., S_k of permutation π^*;

2. Construct a directed graph $F[\pi^*]$ on $n^* + 2$ vertices as follows:

- $V(F[\pi^*]) = \{s = u_{n^*+1}, u_{n^*}, \ldots, u_1, u_0 = t\}$;

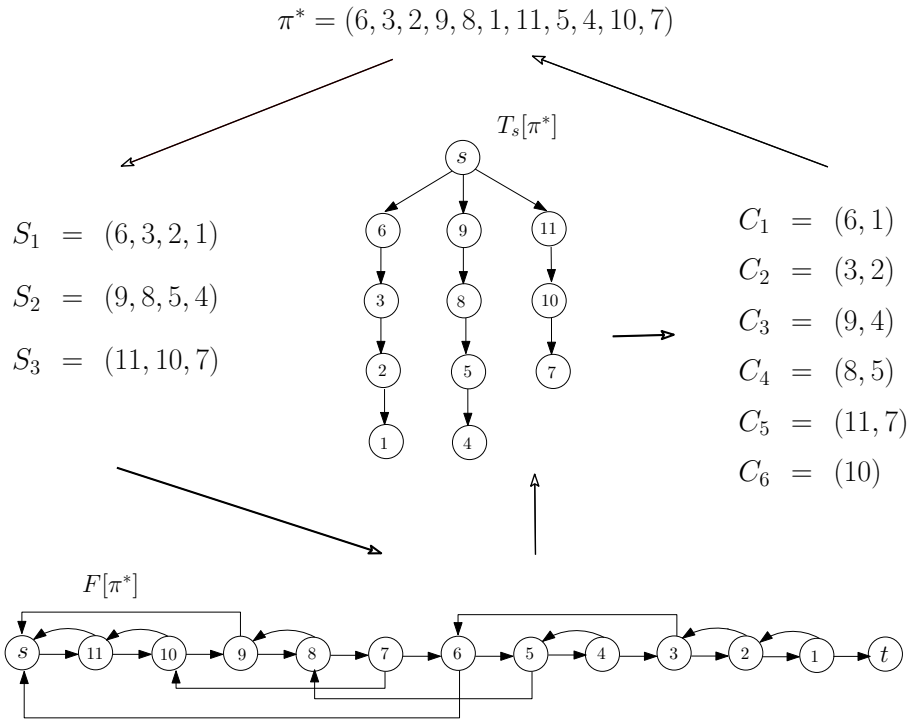


Figure 3.4: The main structures used or constructed by Algorithms `Encode_SiP.to.RPG-II` and `Decode_RPG.to.SiP-II`, i.e., the self-inverting permutation π^*, the decreasing subsequences of π^*, the graph $F[\pi^*]$, the tree $T_s[\pi^*]$, and the elements of π^* in pairs.

- for $i = n^*, n^* - 1, \ldots, 0$ do
  add the edge (u_{i+1}, u_i) in $E(F[\pi^*])$;

3. for each decreasing subsequence $S_i = (i_1, i_2, \ldots, i_t)$, $1 \leq i \leq k$, do
  add the edge (u_{i_1}, s) in $E(F[\pi^*])$;
  for $j = t, t-1, \ldots, 2$ do
    add the edge $(u_{i_j}, u_{i_{j-1}})$ in $E(F[\pi^*])$;

4. Return the graph $F[\pi^*]$.

Time and Space Complexity. The sequences $S_1, S_2, \ldots, S_k$ of π^* can be computed in $O(n^*)$ time and space: from π^*, we isolate subpermutation π_2^* (see, Eq. 2.1 and 2.3) from which we determine the a_is, the b_is, and ℓ, and from them, the A_is, B_is, and Γ_is; then, the sequences $S_1, S_2, \ldots, S_k$ can be computed based on the description of the decreasing subsequences of π_1^* and π_2^* given earlier in this section and Lemma 3.1. Furthermore, the construction of the graph $F[\pi^*]$ also takes $O(n^*)$ time and space. Thus, the following theorem holds.

Theorem 3.4. *The algorithm* `Encode_SiP.to.RPG-II` *for encoding a self-inverting permutation π^* of length n^* as a reducible permutation flow-graph $F[\pi^*]$ requires $O(n^*)$ time and space.*

Algorithm Decode_RPG.to.SiP-II

Having designed the efficient encoding algorithm `Encode_SiP.to.RPG-II`, we next present the decoding algorithm `Decode_RPG.to.SiP-II` which takes as input a flow-graph $F[\pi^*]$ and extracts the self-inverting permutation π^* from $F[\pi^*]$ (see, Figure 3.4); it works as follows:

Algorithm `Decode_RPG.to.SiP-II`

1. Delete the directed edges (u_{i+1}, u_i) from the edge set $E(F[\pi^*])$, $1 \leq i \leq n^*$, and the node $t = u_0$ from $V(F[\pi^*]) = \{s = u_{n^*+1}, u_{n^*}, \ldots, u_1, u_0 = t\}$;

2. Flip all the remaining directed edges of the graph $F[\pi^*]$; the resulting graph is a tree $T_s[\pi^*]$ rooted at $s = u_{n^*+1}$; let $v_{n^*+1}, v_{n^*}, \ldots, v_1$ be the corresponding nodes of $T_s[\pi^*]$;

3. While the root s of $T_s[\pi^*]$ has at least one child v_i do

   - find the leaf v_j of $T_s[\pi^*]$ which is reachable from node v_i;
   - set $P_m = (v_i, v_j)$ and delete both v_i and v_j from $T_s[\pi^*]$;

4. Initialize the permutation $\pi^* = (\pi^*_1, \pi^*_2, \ldots, \pi^*_{n^*})$ to the identity permutation $(1, 2, \ldots, n^*)$, and let P be the set of all pairs $P_1, P_2, \ldots, P_k$ computed at Step 3; then

   - for each pair $(v_i, v_j) \in P$, swap elements π^*_i and π^*_j in permutation π^*;

5. Return the self-inverting permutation π^*.

Time and Space Complexity. The size of the tree $T_s[\pi^*]$ is $O(n^*)$ since the input graph $F[\pi^*]$ constructed by algorithm `Encode_SiP.to.RPG-II` has $O(n^*)$ nodes. Based on the structure of the tree $T_s[\pi^*]$ we can compute the pairs $P_1, P_2, \ldots, P_k$ in $O(n^*)$ time using $O(n^*)$ space. Thus, we can obtain the following result.

Theorem 3.5. *Let $F[\pi^*]$ be a reducible permutation flow-graph of size $2n^* + 1$ produced by the algorithm* `Encode_SiP.to.RPG-II`*. The algorithm* `Decode_RPG.to.SiP-II` *decodes the flow-graph $F[\pi^*]$ in $O(n^*)$ time and space.*

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

In this section, we analyze the structures of the main component of our proposed codec system, that is, the reducible permutation graph $F[\pi^*]$ produced by the algorithms `Encode_SiP.to.RPG-I` and `-II`, and discuss their properties with respect to resilience to attacks.

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

3.6.1 Structural Properties

In graph-based watermarking environment, the watermark w is encoded by a codec algorithm into some special kind of graphs G; in such an environment, graph G is usually called watermark graph. In general, the watermark graph G should not differ from the graph data structures built by real programs. Important properties are the maximum outdegree of G which should not exceed two or three, and the existence of a unique root node so that all other nodes can be reached from it. Moreover, G should be resilient to attacks against edge and/or node modifications. Finally, G should be efficiently constructed.

Our watermark graph $F[\pi^*]$ and a corresponding codec system $(\texttt{encode}, \texttt{decode})_{F[\pi^*]}$ incorporate 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 at most two; that is, it has low max-outdegree, and thus it matches real program graphs.

- **High resiliency**: Since exactly one node of the reducible permutation graph $F[\pi^*]$ has outdegree 0, one other has outdegree 1, and the rest have outdegree 2, we can with high probability identify and correct single edge modifications, i.e., edge-flips, edge-additions, or edge-deletions. Thus, the graph $F[\pi^*]$ enables us to correct single edge changes.

- **Small size**: The size $|P_w| - |P|$ of the embedded watermark w is relatively small since the size of the corresponding watermark graph $F[\pi^*]$ is $O(n^*)$; in fact, $F[\pi^*]$'s size is $O(\log_2 w)$ because $n^* = 2n + 1$ and $n = \lceil \log_2 w \rceil$.

- **Efficient codecs**: The codec $(\texttt{encode}, \texttt{decode})_{F[\pi^*]}$ has low time and space complexity; indeed, we have showed that both the encoding algorithms `Encode_SiP.to.RPG-I` and `-II` and the decoding algorithms `Decode_RPG.to.SiP-I` and `-II` require $O(n^*)$ time and space, where n^* is the size of the input permutation π^* (see, Theorems 3.2 and 3.3 and Theorems 3.4 and 3.5).

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

3.6.2 Unique Hamiltonian Path

It is well-known that any acyclic digraph G has at most one Hamiltonian path (HP); G has one HP if there exists an ordering $(v_1, v_2, \ldots, v_n)$ of its n nodes such that in the subgraphs $G_0, G_1, \ldots, G_{n-1}$ the nodes $v_1, v_2, \ldots, v_n$, respectively, are the only nodes with indegree zero, where $G_0 = G$ and $G_i = G \backslash \{v_1, v_2, \ldots, v_i\}$, $1 \leq i \leq n-1$. Furthermore, it has been shown that any reducible flow-graph has at most one Hamiltonian path [30]. It is not difficult to see that the reducible permutation graphs $F[\pi^*]$ constructed by algorithms `Encode_SiP.to.RPG-I` and `-II` have a unique Hamiltonian path, denoted by $\mathrm{HP}(F[\pi^*])$; this is precisely the path $u_{n^*+1} u_{n^*} \cdots u_1 u_0$. Such a path can be found in time linear in the size of $F[\pi^*]$. The following algorithm, which we call `Unique_HP`, takes as input a graph $F[\pi^*]$ on $n^* + 2$ nodes and produces

its unique Hamiltonian path HP($F[\pi^*]$).

Algorithm `Unique_HP`

1. Find the node u_{n^*+1} of the graph $F[\pi^*]$ with outdegree 1;

2. Perform DFS-search on graph $F[\pi^*]$ starting at node u_{n^*+1} and compute the DFS discovery time $d[u]$ of each node u of $F[\pi^*]$;

3. Return HP($F[\pi^*]$) $= (u'_0, u'_1, \ldots, u'_{n^*+1})$ where $(u'_0, u'_1, \ldots, u'_{n^*+1})$ is the ordering of the nodes $u_{n^*+1}, u_{n^*}, \ldots, u_0$ of the graph $F[\pi^*]$ by increasing DFS discovery time $d[]$, i.e., $d[u'_i] < d[u'_j]$ for $i < j$, $0 \leq i, j \leq n^* + 1$.

Since the graph $F[\pi^*]$ contains $n^* + 2$ nodes and $2n^* + 1$ edges, both finding the node of $F[\pi^*]$ with outdegree 1 and performing DFS-search on $F[\pi^*]$ take $O(n^*)$ time and require $O(n^*)$ space. Moreover, ordering the nodes by their DFS discovery time can also be done in $O(n^*)$ time by bucket sorting. Thus, we have the following result.

Theorem 3.6. *Let $F[\pi^*]$ be a reducible permutation graph of size $O(n^*)$ constructed by algorithm* `Encode_SiP.to.RPG-I` *or* `-II`*. The algorithm* `Unique_HP` *correctly computes the unique Hamiltonian path of $F[\pi^*]$ in $O(n^*)$ time and space.*

3.7 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.

3.7.1 Node-label Modification

By construction, our reducible permutation graph $F[\pi^*]$ is a node-labeled graph on n^*+2 nodes, where n^* is the length of π^*. 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 $n^* + 1 - i$ is assigned to the ith body node u_{n^*+1-i}, $1 \leq i \leq n$.

A label modification attacker may perform swapping of the labels of two nodes of $F[\pi^*]$, altering the value of the label of a node, or even removing all the labels of the graph $F[\pi^*]$ resulting in a node-unlabeled graph. Since the extraction of the watermark w relies on the labels of the flow-graph $F[\pi^*]$ (see, algorithms `Decode_RPG.to.SiP-I` and `-II`), it follows that our codec system $(\texttt{encode}, \texttt{decode})_{\texttt{F}[\pi^*]}$ is susceptible to node-label modification attacks.

Therefore, it is important for us to also have a way to extract the watermark w efficiently from $F[\pi^*]$ without relying on its labels. Obtaining the correct labels can be easily done in $O(n^*)$ time and space thanks to the unique Hamiltonian path HP($F[\pi^*]$) since the nodes of $F[\pi^*]$) are encountered along HP($F[\pi^*]$) in decreasing order of their labels. Thus, we can recover from any change of the labels or even from complete deletion of them. Therefore, we have the following result.

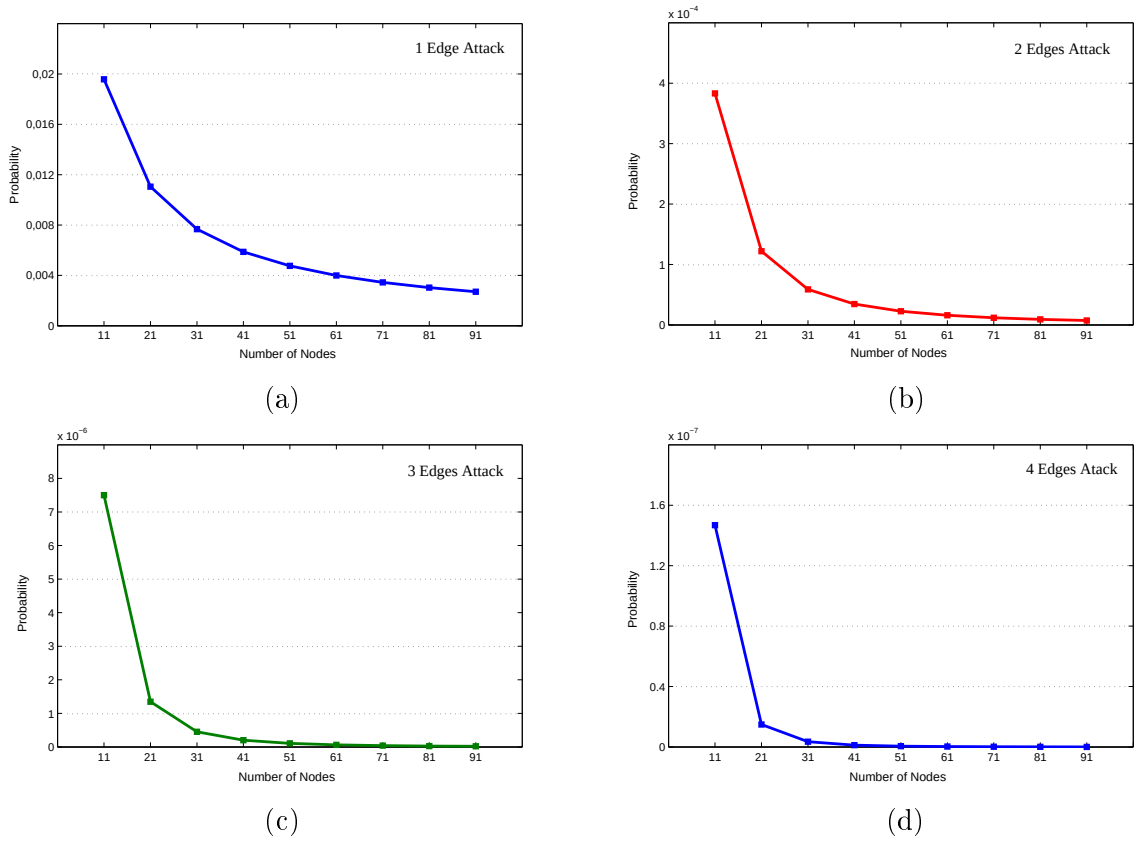


Figure 3.5: The probability for the RPG $F[\pi^*]$ to have the RPG property after a modification of (a) 1 edge, (b) 2 edges, (c) 3 edges, and (d) 4 edges. Note the different scaling of the four diagrams.

Lemma 3.2. *Let $F[\pi^*]$ be a reducible permutation graph of size $O(n^*)$ produced by either algorithm* `Encode_SiP.to.RPG-I` *or* `-II`*, and let $F'[\pi^*]$ be the graph resulting from $F[\pi^*]$ after having modified or deleted the node-labels of $F[\pi^*]$. Given $F'[\pi^*]$, the flow-graph $F[\pi^*]$ can be constructed in $O(n^*)$ time and space.*

3.7.2 Edge Modification

We next argue that we can decide, with high probability, whether the reducible permutation graph $F[\pi^*]$ produced by our codec system has suffered an attack on its edges.

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

Since the SiP properties of the permutation π^*, i.e., odd-length property, one-cycle property, bitonic property, and block property (see, Section 2.6), are incorporated in the structure of the reducible permutation graph $F[\pi^*]$, it follows that the graph $F'[\pi^*]$ resulting from $F[\pi^*]$ after any

edge-modification may be F-incorrect if at least one of the SiP properties does not hold. Indeed, if $F'[\pi^*]$ decodes a permutation $\pi'^* \neq \pi^*$, then the subsequence X (Y, resp.) may not be increasing and thus the bitonic property does not hold. In addition, if an attacker makes appropriate edge-modifications to $F[\pi^*]$ so that the resulting graph $F'[\pi^*]$ decodes a permutation π'^* which is still self-inverting, then the first block of the binary sequence B' may contain one or more 1s or the third block may be 0. On the other hand, due to the odd-length property any single node-modification in $F[\pi^*]$, i.e., node-addition or node-deletion, can be easily identified.

We also experimentally evaluated the resilience of the flow-graph $F[\pi^*]$, the main component of our system, in edge-modifications. To this end, we have produced reducible permutation graphs $F[\pi^*]$ on $n = 11, 21, 31, \ldots, 91$ nodes and computed the probability for the graph $F_i[\pi^*]$ to be F-incorrect, where $F_i[\pi^*]$ is the graph resulting from $F[\pi^*]$ after a modification of i edges, $1 \leq i \leq 4$.

In our experimental study the graphs $F_i[\pi^*]$, $1 \leq i \leq 4$, are produced in the following manner: we first choose an integer w uniformly at random from $[2^{n-1}, 2^n - 1]$, where $n = 5, 10, 15, \ldots, 45$, then generate a SiP π^* of length $n^* = 2n+1$, and finally encode the permutation π^* as a reducible permutation graph $F[\pi^*]$ using the algorithm `Encode_SiP.to.RPG-I`; we next randomly select i edges (u_x, u_y) and i nodes u_z, and then delete the edge (u_x, u_y) and add the edge (u_x, u_z), $1 \leq i \leq 4$.

The experimental results show that we can decide with high probability whether our reducible permutation graph $F[\pi^*]$ has suffered an attack on its edges. Figure 3.5 depicts the high-resilience structure of the graph $F[\pi^*]$.

3.8 Concluding Remarks

In this chapter we proposed an efficient and easily implementable codec system for encoding watermark numbers as graph structures. In particular, we proposed an efficient codec method for encoding a self-inverting permutation π^* as a reducible permutation graph $F[\pi^*]$; the proposed flow-graph $F[\pi^*]$ can be efficiently used for software watermarking since its structure mimics real codes.

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

In light of the two main data components of our codec system, i.e., the permutation π^* and the graph $F[\pi^*]$, 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 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, the evaluation of our codec algorithms and structures under other watermarking measurements in order to obtain detailed information about their practical behavior is a interesting problem for future study.

Chapter 4

Multiple Encoding of a Number into RPGs using Cographs

4.1 Introduction

4.2 Background Results

4.3 Multiple Encoding of a SiP into Cographs

4.4 Encoding Cographs as RPGs

4.5 Concluding Remarks

4.1 Introduction

In previous chapters, we proposed efficient and easily implemented codec systems for encoding watermark numbers as reducible permutation flow-graphs, extending thus the class of graph structures for software watermarking. More precisely, we have presented an efficient encoding algorithm which encodes a watermark number w as self-inverting permutation π^* and two encoding algorithms which encode the permutation π^* as a reducible permutation flow-graph $F[\pi^*]$: the former applies to any permutation π and relies on domination relations on the elements of π whereas the latter applies to a self-inverting permutation π^* produced in any way and relies on the decreasing subsequences of π^*. Again, we point out that the two main components of the proposed codec system, i.e., the self-inverting permutation π^* and the reducible permutation graph $F[\pi^*]$, incorporate important structural properties which make the encoding of permutation π^* into graph $F[\pi^*]$ robust and resilient to attacks (see, Chapters 2 and 3).

Contribution. In this chapter, we extend the class of graphs which can be efficiently used in a software watermarking codec system by proposing efficient encoding and decoding algorithms that embed a watermark value into cographs and efficiently extract the watermark from the graph structures. Moreover, we present a randomized encoding algorithm and show that the

same watermark number w can be efficiently encoded into several different reducible permutation graphs $F_1[\pi^*], F_2[\pi^*], \ldots, F_n[\pi^*]$, $n \geq 2$, all of equal size; see, Figure 4.1.

More precisely, we first present the randomized encoding algorithm `Encode_SiP.to.Cograph`, which takes as input a self-inverting permutation π^*, encoding a watermark number w (see, algorithm `Encode_W.to.SiP` in Chapter 3), and encodes the permutation π^* into a cograph $C[\pi^*]$. We also present its corresponding decoding algorithm `Decode_Cograph.to.SiP`; it takes as input a cograph $C[\pi^*]$, produced by our randomized encoding algorithm, and extracts the self-inverting permutation π^* from it. Thus, in light of our encoding algorithm `Encode_W.to.SiP`, which encodes an integer w as a self-inverting permutation π^*, we can appropriately use the proposed randomized algorithm to encode the same integer w into several cographs $C_1[\pi^*], C_2[\pi^*], \ldots, C_n[\pi^*]$, where $n \geq 2$.

Having presented our randomized encoding algorithm, we next present an efficient transformation of a cograph $C[\pi^*]$ into a reducible permutation graph $F[\pi^*]$. Indeed, we propose the encoding algorithm `Encode_Cograph.to.RPG` which embeds a cograph $C[\pi^*]$ into $F[\pi^*]$ by exploiting the structure and some important algorithmic properties of the cotree $T[\pi^*]$ of the cograph $C[\pi^*]$; recall that, a cograph admits a unique tree representation, up to isomorphism, called a cotree [74]. Thus, having such an encoding algorithm, we can encode a watermark number w into many RPGs $F_1[\pi^*], F_2[\pi^*], \ldots, F_n[\pi^*]$, $n \geq 2$.

Based on the above results, we can propose a codec system for graph-based software watermarking having better behavior with respect to code attacks since it can embed multiple copies of the same watermark number w into an application program; in general, a digital object can be made more resilient to attacks if multiple copies of the same watermark w are embedded into it. Moreover, the proposed codec system has low time complexity and can be easily implemented.

Road Map. The chapter is organized as follows: In Section 4.2 we give background results on self-inverting permutations, reducible permutation graphs and cographs. In Section 4.3 we present the randomize encoding algorithm `Encode_SiP.to.Cograph`, along with its corresponding decoding algorithm, which takes as input a self-inverting permutation π^*, encoding a watermark number w, and encodes the permutation π^* into a cograph $C[\pi^*]$. In Section 4.4 we present the algorithm `Encode_Cograph.to.RPG`, along with its corresponding decoding algorithm, which embeds a cograph into an RPG by exploiting the structure and some important algorithmic properties of its cotree. Finally, Section 4.5 concludes the chapter and gives futures research directions.

4.2 Background Results

In this section, we give some definitions that are key-objects in our algorithms for multiple encoding self-inverting permutations as complement reducible graphs (or, cographs) [52], and also algorithms for encoding cographs as reducible permutation graphs (or, RPGs).

Self-Inverting Permutations. In Chapter 2, we defined a permutation π over the set N_n as a sequence $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$ whereas a self-inverting permutation (or, involution) as a permutation that is its own inverse, i.e., $\pi_{\pi_i} = i$. By definition, all the cycles of a self-inverting

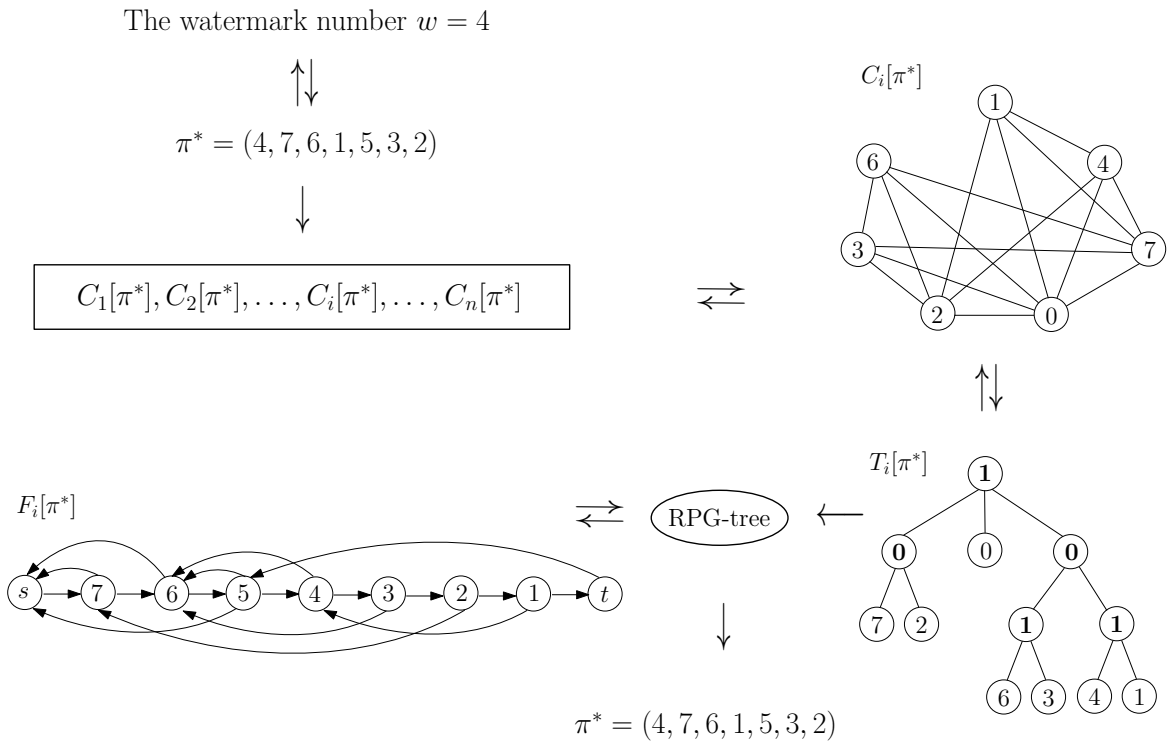


Figure 4.1: The main data components used by the algorithms of our codec system for multiple encoding the same watermark number $w = 4$ into several RPGs using Cographs.

permutation are of length 1 or 2.

Moreover, in the same chapter, we proposed the encoding algorithm `Encode_W.to.SiP` for encoding numbers as self-inverting permutations (or SiP, for short) along with the corresponding decoding algorithm `Decode_SiP.to.W`. The self-inverting permutation π^* produced by our encoding algorithm encompasses important structural properties, i.e., (i) odd-length property, (ii) one-cycle property, (iii) bitonic property, and (iv) block property, which makes our codec system resilient to attacks.

Reducible Permutation Graphs. In Chapter 3, we defined a flow-graph as a directed graph F with an initial node s from which all other nodes are reachable, and showed that a flow-graph is reducible when it does not have a strongly connected subgraph with two (or more) entries; see, Section 3.3.

In the same chapter, we proposed the reducible permutation graph $F[\pi^*]$ on $n^* + 2$ nodes consisting of three components:

(A) a header node,

(B) a footer node, and

(C) the body.

Recall that, the header is a root node with outdegree one from which all other nodes of the graph $F[\pi^*]$ are reachable, the footer is a node with outdegree zero that is reachable from all other nodes of $F[\pi^*]$, while the body consists of n^* nodes each with outdegree two; see,

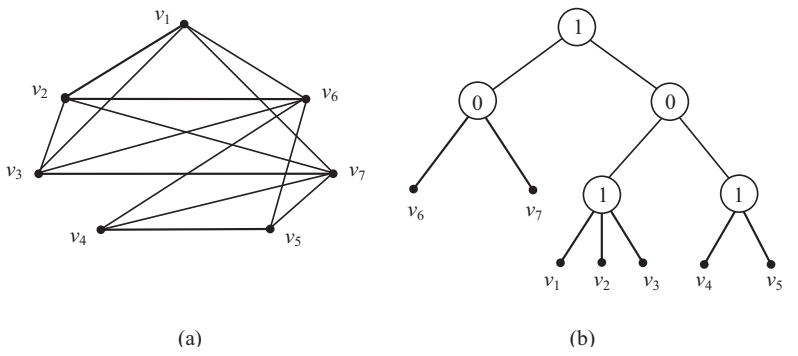


Figure 4.2: (a) A cograph on 7 vertices, and (b) the corresponding cotree.

Section 3.4.

4.2.1 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 [74]. 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 names. 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 [14, 52]. Furthermore, cographs are precisely the graphs which contain no induced subgraph isomorphic to a P_4 (i.e., a chordless path on four vertices).

Cographs were introduced in the early 1970s by Lerchs [74] 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 an 1-1 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 an 1-node; if however we relax this condition and allow the root to be an 0-node as well, then we obtain cotrees whose internal nodes all have at least two children, and whose root is an 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 [52]. The first linear-time recognition and cotree-construction algorithm was proposed by Corneil, Perl, and Stewart in 1985 [39]. Recently, Bretscher et. al [7] 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.

4.3 Multiple Encoding of a SiP into Cographs

In this section we present a randomized encoding algorithm, we call it `Encode_SiP.to.Cograph`, which takes as input a self-inverting permutation π^*, encoding a watermark number w, and encodes the permutation π^* into a cograph $C[\pi^*]$; we also present the corresponding decoding algorithm which we call `Decode_Cograph.to.SiP`. Thus, in light of our encoding algorithm `Encode_W.to.SiP` (see, Chapter 3), which encodes an integer w as a self-inverting permutation π^*, we can multiply use the proposed randomized algorithm to encode the same integer w into several cographs $C_1[\pi^*], C_2[\pi^*], \ldots, C_n[\pi^*]$, $n \geq 2$ [24, 27].

The main property of our randomized 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 causes a watermarking codec system to be resilient to attacks since it can embed multiple copies of the same watermark number w into a digital object. Moreover, such a codec system has low time complexity and can be easily implemented.

4.3.1 Algorithm Encode_SiP.to.Cograph

In this section, we present the `Encode_SiP.to.Cograph` algorithm which takes as input a self-inverting permutation π^* of length $2n+1$ produced by algorithm `Encode_W.to.SiP` (see, [28]), and constructs an arbitrary cograph $C_i[\pi^*]$ on $2n+1$ vertices by preserving the cycle relation of permutation π^*; recall that, by construction the self-inverting permutation π^* has length $2n+1$ and contains one 1-cycle (x, x) and n 2-cycles (x_1, y_1), (x_2, y_2), ..., (x_n, y_n).

Algorithm `Encode_SiP.to.Cograph`

1. Construct a graph H from the input permutation $\pi^* = (\pi_1, \pi_2, \ldots, \pi_n)$ such that:
   $V(H) = \{\pi_1, \pi_2, \ldots, \pi_n\}$;
   $E(H) = \{(\pi_i, \pi_j)$ is an edge if (π_i, π_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 H_i and H_j,
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 $C_1[\pi^*]$ of Figure 4.3: 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 $C_1[\pi^*]$ of Figure 4.3.

Construction of the cograph $C_2[\pi^*]$ of Figure 4.3: in a similar way, the algorithm constructs the graph $C_2[\pi^*]$ of Figure 4.3 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.

4.3.2 Algorithm Decode_Cograph.to.SiP

Next, we present a decoding algorithm for extracting a self-inverting permutation from a cograph. Our decoding 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`

1. Compute the cotree $T(G)$ of the input cograph $G = C[\pi^*]$;
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

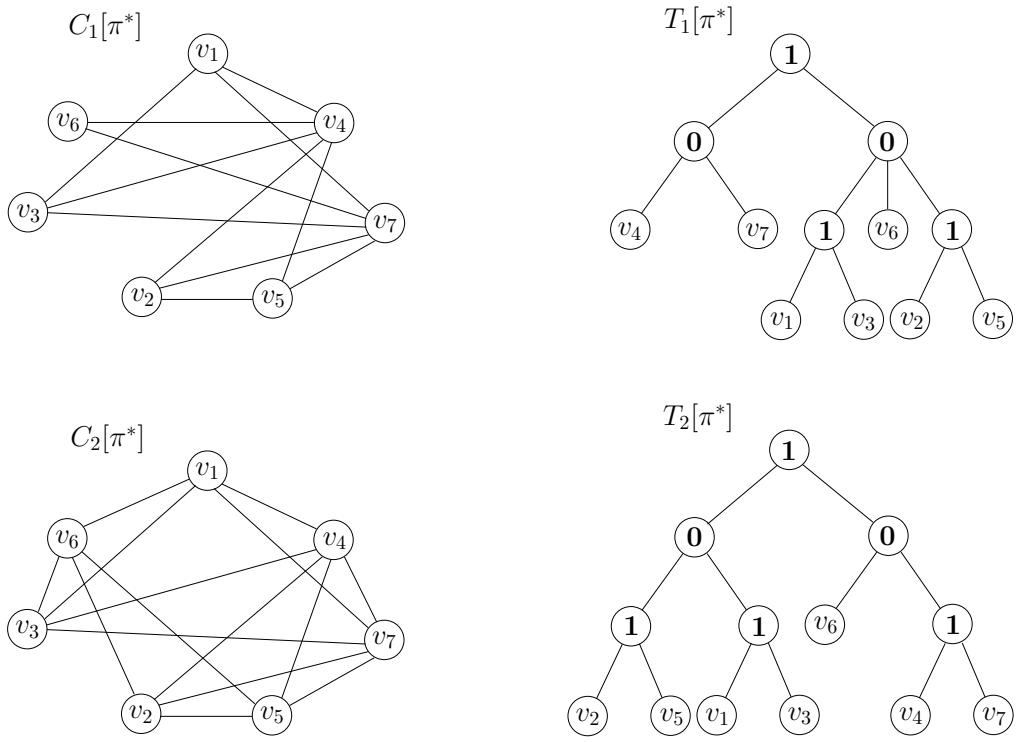


Figure 4.3: Two cographs $C_1[\pi^*]$ and $C_2[\pi^*]$ on 7 vertices which encode the same watermark number w, corresponding to permutation $\pi^* = (3, 5, 1, 7, 2, 6, 4)$, along with their corresponding cotrees.

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 π^*.

Decode Example: Let $C_1[\pi^*]$ and $C_2[\pi^*]$ be two cographs produced by the encoding algorithm `Encode_SiP.to.Cograph`. The decoding algorithm constructs first the corresponding cotrees $T_1[\pi^*]$ and $T_2[\pi^*]$, and then computes the pairs of nodes (v_4, v_7), (v_1, v_3) and (v_2, v_5) (see, Figure 4.3). 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 swap the 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.

4.4 Encoding Cographs as RPGs

Having presented in the previous section an encoding algorithm which embeds a watermark w number into several deferent cographs $C_1[\pi^*], C_2[\pi^*], \ldots, C_n[\pi^*]$, $n \geq 2$, let us next propose an algorithm which embeds a cograph into a reducible permutation graph (or, for short, RPG). To this end, we exploit the structure and some important algorithmic properties of the class of cographs; recall that, a cograph admits a unique tree representation, up to isomorphism, called a cotree [74]. Thus, having such an algorithm, we can encode a watermark number w into many RPGs $F_1[\pi^*], F_2[\pi^*], \ldots, F_n[\pi^*]$, $n \geq 2$.

4.4.1 Algorithm Encode_Cotree.to.RPG

We next propose the algorithm `Encode_Cotree.to.RPG` which takes as input the cotree $T[\pi^*]$ of a cograph $C[\pi^*]$, produced by our randomized algorithm `Encode_SiP.to.Cograph`, and constructs a reducible permutation graph $F[\pi^*]$ by using an efficient node elimination and subtree modification on $T[\pi^*]$. Note that, the cotree of a cograph can be constructed in linear time [14, 39].

Given a cograph $C[\pi^*]$ on n vertices, our encoding algorithm first construct its corresponding cotree $T[\pi^*]$ and then it works on two phases:

(I) it first uses a strategy based on node elimination and subtree modification to transform the cotree $T[\pi^*]$ into a binary tree $R[\pi^*]$, which we call RPG-tree, having the property that each node has value smaller than the value of its parent;

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

Next, we describe in detail the encoding algorithm `Encode_Cotree.to.RPG` (see, Figure 4.4 and Figure 4.5).

Algorithm `Encode_Cotree.to.RPG`

1. Construct the RPG-tree $R[\pi^*]$ from $T[\pi^*]$ (Phase I);

2. Construct the RPG $F[\pi^*]$ from $R[\pi^*]$ (Phase II);

3. Return the graph $F[\pi^*]$.

We next describe in detail the two phases of the encoding algorithm `Encode_Cotree.to.RPG`.

Phase I. Construction of the RPG-tree $R[\pi^*]$ from $T[\pi^*]$: We construct the RPG-tree $R[\pi^*]$ by eliminating the internal nodes of the cotree $T[\pi^*]$ and max-merging certain subtrees in an appropriate way, as follows:

I.1. Let r be the root of the cotree $T[\pi^*]$ and t be the internal node of $T[\pi^*]$ with only one leaf, say, v;

- Replace the label of the root r with the number $n+1$ and the label of each internal node with the number -1;
- Create a new node with value 0 and make it child of the node v;

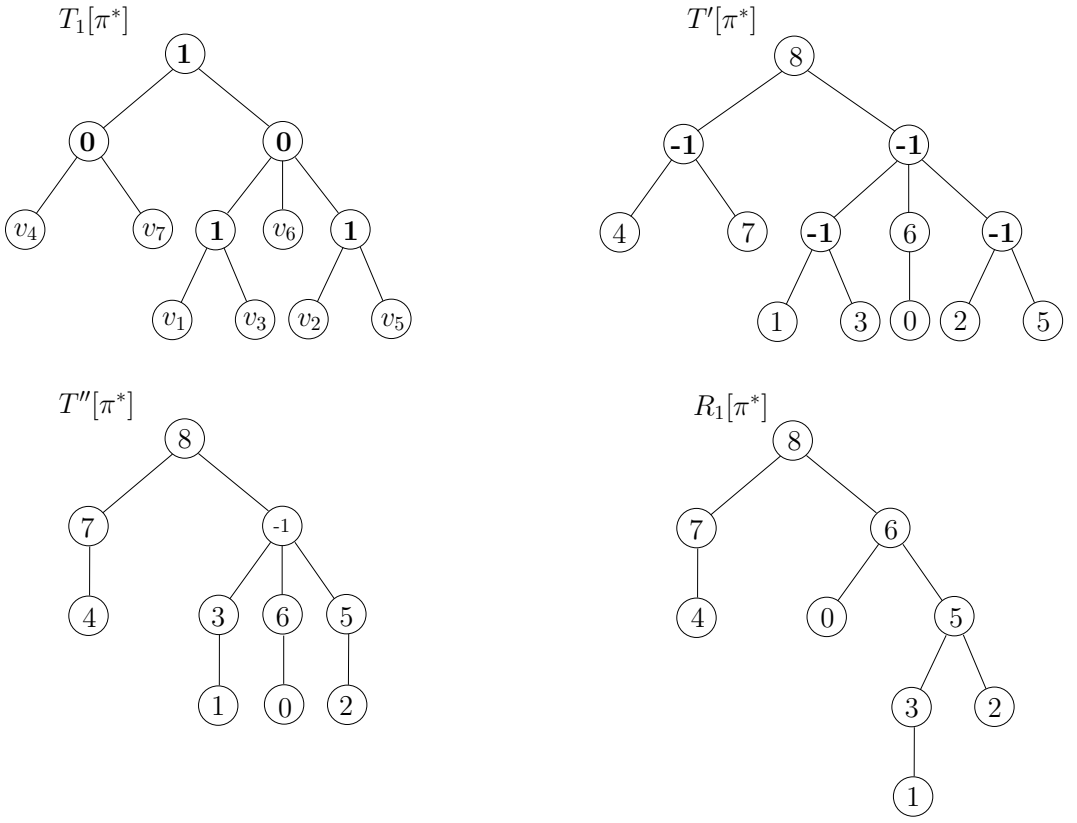


Figure 4.4: The cotree $T_1[\pi^*]$ and the resulting RPG-tree $R_1[\pi^*]$; trees $T'[\pi^*]$ and $T''[\pi^*]$ show the contraction process.

I.2. While the tree $T[\pi^*]$ contains internal nodes u with value -1, do the following:

- Find such an internal node u of maximum high and let $u_1, u_2, \ldots, u_k$ be its children, $k > 1$;
- Max-merge the subtrees $T(u_1), T(u_2), \ldots, T(u_k)$ and let $T(u_m)$ be the resulting subtree rooted at node u_m, $1 \leq m \leq k$;
- Make the root u_m of the subtree $T(u_m)$ child of the parent of u;
- Delete the node u from the tree $T[\pi^*]$;

where the function Max-merge works as follows:

- Order the subtrees $T(u_1), T(u_2), \ldots, T(u_k)$ according to their root values, and let $u_1 < u_2 < \ldots < u_k$;
- Make the root u_i of $T(u_i)$ child of the root u_{i+1} of $T(u_{i+1})$, for all i $(1 \leq i \leq k-1)$;

Phase II: Construction of the RPG $F[\pi^*]$ from $R[\pi^*]$: We construct the directed graph $F[\pi^*]$ by exploiting the child-parent relation of the nodes of the rpgtree $R[\pi^*]$, as follows:

II.1. For every node u_i of $R[\pi^*]$, $0 \leq i \leq n+1$, create a node v_i and add it to $V(F[\pi^*])$; that is, $V(F[\pi^*]) = \{s = v_{n+1}, v_n, v_{n-1}, \ldots, v_1, , v_0 = t\}$;

II.2. For every pair of nodes (v_i, v_{i-1}) of the set $V(F[\pi^*])$ add the directed edge (v_i, v_{i-1}) in $E(F[\pi^*])$, $1 \leq i \leq n+1$; we call it *list* pointer;

II.3. For every node u_i of $R[\pi^*]$ compute its parent node $p(v_i)$, $0 \leq i \leq n$, and add the directed edge $(u_i, p(u_i))$ in $E(F[\pi^*])$; we call it *tree* pointer.

Time and Space Complexity. It is easy to see that the most time-consuming step of the algorithm is that of max-merging the subtrees $T(u_1), T(u_2), \ldots, T(u_k)$ of the node u (see, function Max-merge); indeed, a sorting sequence S of $k > 1$ values must be computed. Since the values $u_1, u_2, \ldots, u_k$ are integers in the range $[0, n+1]$, the sequence S can be computed in $O(n)$ time and space [40]. On the other hand, the construction of the reducible permutation flow-graph $F[\pi^*]$ (Steps II.1 – II.3) requires only the list pointers, which can be trivially computed, and the tree pointers which can be computed using the parent function on $R[\pi^*]$. Thus, we obtain the following result:

Theorem 4.1. *Let $T[\pi^*]$ be the cotree of a cograph $G[\pi^*]$ on n vertices. The encoding algorithm* `Encode_Cotree.to.RPG` *encodes the cotree $T[\pi^*]$ into a reducible permutation graph $F[\pi^*]$ in $O(n)$ time and space.*

4.4.2 Algorithm Decode_RPG.to.SiP

Having designed the algorithm `Encode_Cotree.to.RPG` which encodes a self-inverting permutation π^* into a reducible permutation flow-graph $F[\pi^*]$, let us now propose 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 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 inorder traversal on trees.

The algorithm takes as input a reducible permutation 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 $E(F[\pi^*])$, $1 \leq i \leq n$;

2. Flip all the remaining directed edges of the graph $F[\pi^*]$; let $R[\pi^*]$ be the resulting tree and let $s = v_0, v_1, v_2, \ldots, v_n, v_{n+1} = t$ be the nodes of $R[\pi^*]$;

3. Make first the binary tree $R[\pi^*]$ rooted at node $s = v_0$ and ordered, and then perform inorder traversal on $R[\pi^*]$;

4. List the nodes $s = v_0, v_1, v_2, \ldots, v_{n+1}$ of the tree $R[\pi^*]$ by the order in which the nodes are visited; remove from the list the root node and let $v'_1, v'_2, \ldots, v'_{n+1}$ be the resulting list;

5. Compute the pairs $(v'_1, v'_2), (v'_3, v'_4), \ldots, (v'_n, v'_{n+1})$ and remove the pair (v'_i, v'_{i+1}) for which $v'_i = 0$; let n' be the remaining pairs;

6. Construct the identity permutation on $N_{2n'}$, and then compute a self-inverting permutation π using the pairs of step 5; return $\pi^* = \pi$.

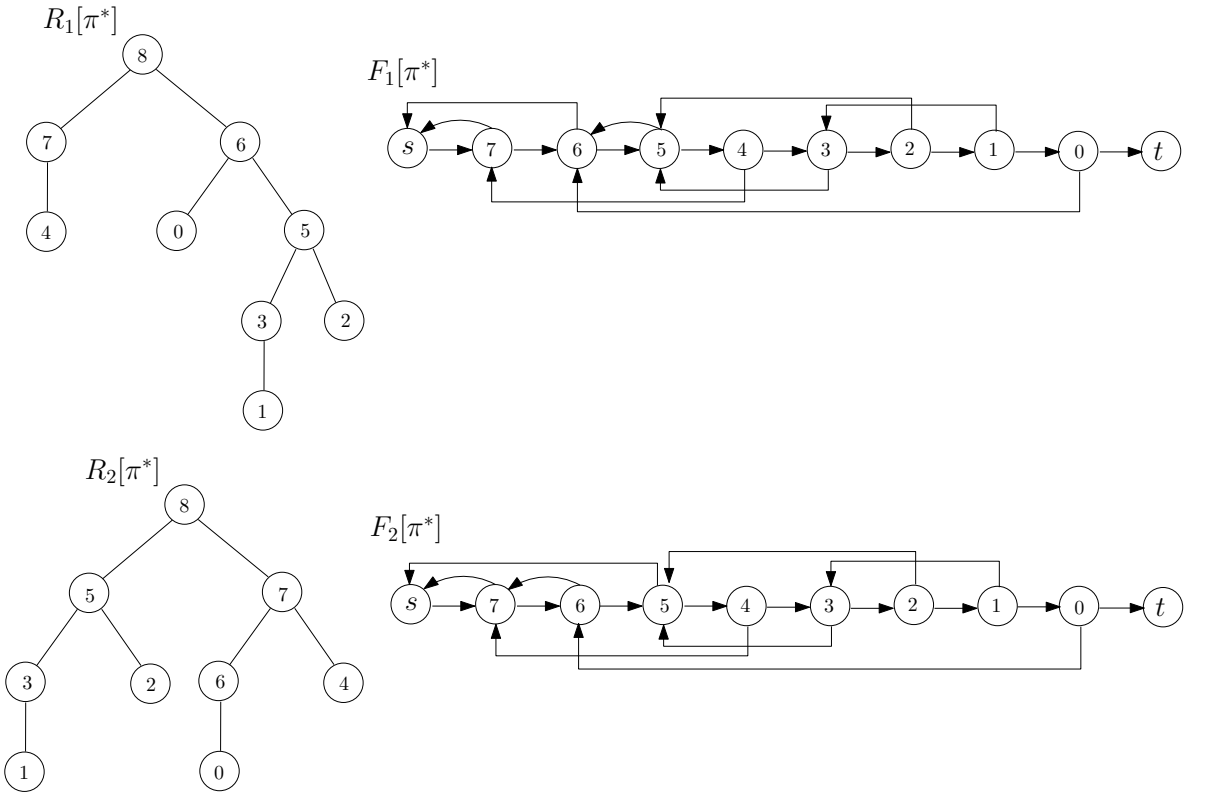


Figure 4.5: Two RPG-trees $R_1[\pi^*]$ and $R_2[\pi^*]$ and the corresponding reducible permutation graphs $F_1[\pi^*]$ and $F_2[\pi^*]$, respectively, produced by the algorithm `Encode_Cotree.to.RPG`.

Let us now describe the decoding process using the two RPGs $F_1[\pi^*]$ and $F_2[\pi^*]$ of Figure 4.5. Let $F_1[\pi^*]$ be the input of the algorithm `Decode_RPG.to.SiP`. It first computes the rooted ordered tree $R_1[\pi^*]$ and then computes the inorder sequence $I_1 = (4, 7, 8, 0, 6, 1, 3, 5, 2)$; it deletes the value 8 of the root resulting the sequence $I_1' = (4, 7, 0, 6, 1, 3, 5, 2)$; then the algorithm computes the pairs $C_1 = \{(4, 7), (0, 6), (1, 3), (5, 2)\}$ and removes the pair $(0, 6)$ from C_1. Then, it takes the identity permutation $\pi_I = (1, 2, 3, 4, 5, 6, 7)$ and using the pairs $(4, 7), (1, 3), (5, 2)$ as 2-cycles produces the SiP $\pi^* = (3, 5, 1, 7, 2, 6, 4)$.

If the algorithm takes as input the RPG $F_1[\pi^*]$ then $I_1 = (1, 3, 5, 2, 8, 0, 6, 7, 4)$ and thus $I_1' = (1, 3, 5, 2, 0, 6, 7, 4)$. It is easy to see that the 2-cycles in C_2 are $(1, 3), (5, 2), (7, 4)$, and thus the identity permutation π_I becomes $\pi^* = (3, 5, 1, 7, 2, 6, 4)$.

Time and Space Complexity. The size of the reducible permutation graph $F[\pi^*]$ constructed by the algorithm `Encode_Cotree.to.RPG` is $O(n)$, where n is the length of the permutation π^*, and thus the size of the resulting RPG-tree $R[\pi^*]$ is also $O(n)$. The inorder traversal on the tree $R[\pi^*]$ takes time linear in the size of $R[\pi^*]$ (see, [40]). Thus, the decoding algorithm is executed in $O(n)$ time using $O(n)$ space. Thus, the following theorem holds:

Theorem 4.2. *Let $T[\pi^*]$ be a cotree which encodes the self-inverting permutation π^* and let $F[\pi^*]$ be a reducible permutation flow-graph of size $O(n)$ produced by the encoding algorithm*

`Encode_Cotree.to.RPG`. *The algorithm* `Decode_RPG.to.SIP` *extracts the permutation* π^* *from the flow-graph* $F[\pi^*]$ *in* $O(n)$ *time and space.*

4.5 Concluding Remarks

In this chapter we proposed an efficient algorithm which encodes a self-inverting permutation π^* into several cographs $C_1[\pi^*], C_2[\pi^*], \ldots, C_n[\pi^*]$, $n \geq 2$, and an efficient transformation of a cograph into a reducible permutation graph $F[\pi^*]$. In light of our encoding algorithms which encode a watermark integer w as a self-inverting permutation π^* [28] and the permutation π^* into many different cographs, we conclude that we can efficiently encode the same watermark integer w into several reducible permutation graphs $F_1[\pi^*], F_2[\pi^*], \ldots, F_n[\pi^*]$, $n \geq 2$.

It is worth noting that this property causes a codec watermarking system resilient to attacks since we can embed multiple copies of the same watermark value w into an application program.

An interesting open question is whether the approach and techniques used in this chapter can help develop efficient codec algorithms and graph structures having "better" properties with respect to resilience, size, and/or time and space efficiency; we leave it as an open problem for future investigation.

Chapter 5

Software Watermarking

5.1 Introduction

5.2 Background Results

5.3 The Model WaterPRG

5.4 Implementation

5.5 Model Evaluation

5.6 Concluding Remarks

5.1 Introduction

The rapid growth of World Wide Web users, the ease of distributing fast and in the original form digital content through internet, as well as the lack of technical measures to assure the intellectual property right of owners, has led to an increment in copyright infringement. Digital watermarking is a technique for protecting the intellectual property of any digital content, i.e., software, image, audio, video, text, ect. The main idea of digital watermarking is the embedding of a unique identifier into the digital content through the introduction of errors not detectable by human perception [15, 29]. Although digital watermarking has made considerable progress and become a popular technique for copyright protection of multimedia information [15], research on software watermarking has recently received sufficient attention.

Software Watermarking. 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 [15].

The Software Watermarking problem can be described as the problem of embedding a structure w into a program P resulting the program P_w such that w can be reliably located and extracted from P_w even after P_w has been subjected to code transformations such as translation, optimization, and obfuscation [84]. 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:

- $\texttt{embed}(P, w, k) \longrightarrow P_w$
- $\texttt{extract}(P_w, k) \longrightarrow w$.

There are two main categories of software watermarking techniques namely *static* and *dynamic* [32]. Moreover, depending on the behavioral properties of the embedded watermark w, software watermarking techniques can also be divided into other categories namely *robust* and *fragile*, *visible* and *invisible*, *blind* and *informed*, *focus* and *spread spectrum*; further discussion on the above software watermarking classification issues can be found in [48, 85, 120].

Static and Dynamic Watermarking. A static watermark w is embedded 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.

(i) A data watermark stores watermark information as program data, and can be stored anywhere inside a program, such as in comments or in variables.

(ii) 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.

On the other hand, a dynamic watermark w is encoded in a data structure built at runtime (i.e., during program execution), perhaps only after receiving a particular input I_{key}; it might be retrieved from the watermarked program P_w by analyzing the data structures built when program P_w is running on input I_{key}. There are three kinds of dynamic watermarks: Easter eggs, execution trace watermarks, and dynamic data structure watermarks [32]. Further discussion of dynamic and/or static watermarking issues can be found in [48, 85, 120].

Graph-based Codecs and Attacks. Software watermarking involves embedding a unique identifier or, equivalently, a watermark value within a software to prove owner's authenticity and thus to prevent or discourage copyright infringement. Towards the embedding process, several graph theoretic watermarking algorithmic techniques (or, equivalently, models or systems) encode the watermark values as graph structures and embed them in application programs.

In general, such a graph-based software watermarking model mainly consists of two codec algorithms: an encoding algorithm which embeds a graph G which represents a watermark w into an application program P resulting thus the watermarked program P_w, i.e., $\texttt{embed}(P, G, k) \rightarrow P_w$, and a decoding algorithm which extracts the graph G from P_w, i.e., $\texttt{extract}(P_w, k) \rightarrow G$. We usually call the pair

- $(\texttt{embed}, \texttt{extract})_G$

as *graph codec model* and the embedding and extracting algorithms as *codec* or *watermarking algorithms*.

In a graph-based software watermarking environment we are interested in both finding a class of graphs $\mathcal{G}$ having appropriate graph properties, e.g., graphs in $\mathcal{G}$ should be contained nodes with small outdegree so that matching real program graphs, and designing efficient codec algorithms, e.g., both algorithms of $(\texttt{embed}, \texttt{extract})_G$ model should be computed in polynomial time.

Having designed a software watermarking algorithm, it is very important to evaluate it under various assessment criteria in order to gain information about its practical behavior [30]; the most valuable and broadly used criteria can be divided into two main categories: (i) performance criteria (e.g., data-rate, time and space overhead, part protection, stealth, credibility), and (ii) resilience criteria (e.g., resistance against obfuscation, optimization, language-transformation) [36, 84]. We mention that the performance criteria measure the behavior of the watermarked program P_w and the quality and effectiveness of the embedded watermark w, while the resilience criteria measure the robustness and resistance of the embedded watermark w against malicious user attacks.

From a graph-theoretical and practical point of view, we are interested in finding a class of graphs $\mathcal{G}$ having appropriate graph properties, e.g., graphs $G \in \mathcal{G}$ should contain nodes with small outdegree so that matching real program graphs, and developing software watermarking models $(\texttt{embed}, \texttt{extract})_G$ which meet both:

- High Performance: both programs, the original P and the watermarked P_w, have almost identical execution behavior, almost same size and similar codes; and
- High Resiliency: the algorithm `extract()` is insensitive to small changes of P_w caused by various attacks, that is, if $G \in \mathcal{G}$ represents the watermark w and $\texttt{extract}(P_w, k) \rightarrow w$ then $\texttt{extract}(P'_w, k) \rightarrow w$ with $P'_w \approx P_w$.

Related Work[1]**.** The most important software watermarking algorithms currently available in the literature are based on several techniques, among which the register allocation [101], spread-spectrum [106], opaque predicate [1], abstract interpretation [37], dynamic path techniques [35], code re-orderings [112]; see, also [29] for an exposition of the main results. It is worth noting that many algorithmic techniques on software watermarking have also received patent protection [34, 48, 103, 114].

In 1996, Davidson and Myhrvold [48] presented the first patented static software watermarking algorithm. The preliminary concepts of software watermarking also appeared in the paper [51] and the patents [85, 105]. Collberg et al. [32, 33] presented detailed definitions for software watermarking, while Zhang et al. [131] and Zhu et al. [130] gave brief surveys of software watermarking research; see, Collberg and Nagra [29] for an exposition of the main results.

The algorithm of Davidson and Myhrvold [48] embeds the watermark into a program by reordering the basic blocks of a control flow-graph; note that a static watermark is stored inside programs' code in a certain format and it does not change during the programs' execution. Based on this idea, Venkatesan, Vazirani and Sinha [120] proposed the first graph-based software watermarking algorithm which embeds the watermark by extending a method's control flow-graph through the insertion of a directed subgraph; it is also a static algorithm called `VVS` or

[1]Some references of this part have also appeared in Chapter 3: Encode Watermark Numbers as Self-inverting Permutations, and in Chapter 4: Encoding SiPs as Reducible Permutation Graphs.

GTW. In [120], the construction of the directed watermark graph G is not discussed. Collberg et al. [31] 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 (or, for short, RPG) [30], which is a reducible control flow-graph with a 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 [32]; it embeds the watermark through a graph structure which is built on a heap at runtime.

Several software watermarking algorithms have been appeared in the literature that encode watermarks as graph structures [30, 31, 48, 120]. Recently, Chroni and Nikolopoulos extended the class of software watermarking codec algorithms and graph structures by proposing efficient and easily implemented algorithms for encoding numbers as reducible permutation flow-graphs (RPG) through the use of self-inverting permutations (or, for short, SiP). More precisely, they have presented an efficient method for encoding first an integer w as a self-inverting permutation π^* and then encoding π^* as a reducible permutation flow-graph $F[\pi^*]$ [28]; see, also [23]. The watermark graph $F[\pi^*]$ incorporates properties capable to mimic real code, that is, it does not differ from the graph data structures built by real programs. Moreover, the structural properties of $F[\pi^*]$ cause it resilient to edge, node and label modification attacks; see, Chapter 3 (see, also [19, 26, 20]).

Contribution. In this chapter, we present a dynamic watermarking model, which we call WaterRPG, for embedding the watermark graph $F[\pi^*]$ into an application program P resulting thus the watermarked program P^*; throughout the thesis, we denote by P^* the program P_w watermarked by our codec model.

The main idea behind the proposed watermarking model is a systematic modification of appropriate function calls of the program P, through the use of control statements and opaque predicates, so that the execution of the watermarked program P^* with a specific input gives a dynamic call-graph from which the watermark graph $F[\pi^*]$ can be easily constructed. More precisely, for a specific input I_{key} of a given program P, our model takes the dynamic call-graph $G(P, I_{key})$ of P and the watermarked graph $F[\pi^*]$, and produces the watermarked program P^* so that the following key property holds:

- The dynamic call-graph $G(P^*, I_{key})$ of P^* with input I_{key} is isomorphic to the watermark graph $F[\pi^*]$.

Within this idea the program P^* is produced by only altering appropriate calls of specific functions of the input program P and manipulating the execution flow of P^* by including these altered function calls into control statements using opaque predicates. In the resulting watermarked program P^*, the control statements are executed following specific and well-defined execution rules and offer high functionality of P^*. Indeed, our model achieves low time and space overhead and ensures correctness, that is,

- $T(P, I) \approx T(P^*, I)$, $S(P, I) \approx S(P^*, I)$, and $O(P, I) = O(P^*, I)$, for every input I

where $T()$, $S()$, and $O()$ are the execution time, the heap space, and the output of P or P^* with input I. Note that, performance and correctness are the two most important properties of any software watermarking model.

Models' Properties	WaterRPG's Properties
static - dynamic	dynamic (execution trace)
robust - fragile	robust
visible - invisible	invisible
blind - informed	blind
focus - spread spectrum	spread spectrum

Table 5.1: General properties of watermarking models and the properties of our WarerRPG model.

We have implemented our watermarking model WaterRPG on Java application programs downloaded from a free non commercial game database, and evaluated its performance under various and commonly used watermarking evaluation criteria. In particular, we selected a number of Java application programs and watermarked them using two main approaches: (i) the straightforward or naive approach, and (ii) the stealthy approach. The naive approach watermarks a given program P using the well-defined call patterns of our model, while the stealthy approach watermarks P using structural and programming properties of the code.

The evaluation results show the efficient functionality of all the Java programs P^* watermarked under both the naive and stealthy cases. The experiments also show that the watermarking approaches supported by our model can help develop efficient watermarked Java programs with respect to time and space overhead, credibility, stealthiness, and other watermarking metrics. Moreover, our WaterPRG model incorporates properties which cause it resilient to several watermark and code attacks.

Table 5.1 summarizes the most important general properties, in complementary or opposite pairs, of a software watermarking model and shows the properties of our WarerRPG model. Throughout the paper, for a given program P we shall denote by P^* the watermarked program produced by our model WaterRPG.

Road Map. The paper is organized as follows: In Section 5.2 we establish the notation and related terminology, and present background results. In Section 5.3 we present our dynamic watermarking model WaterRPG; we first describe its structural and operational components and then the embedding algorithm `Embed_RPG.to.CODE` and the extracting algorithm `Extract_CODE.to.RPG`. In Section 5.4 we implement our watermarking model in real Java application programs and show two main watermarking approaches supported by the WaterRpg model, namely naive and stealthy. In Section 5.5 we evaluate our model under several software watermarking assessment criteria, while in Section 5.6 we summarize our work and propose possible future extensions.

5.2 Background Results

In this section, we present background results and key objects that are used in the design of our watermarking model WaterRPG. In particular, we briefly present the main results of our previous work concerning the process of encoding numbers as graph structures namely reducible permutation graphs (or, for short, RPG); we denote such a graph as $F[\pi^*]$. We also briefly

discuss properties of dynamic call-graphs which are used as key-objects in our watermarking model for embedding the graph $F[\pi^*]$ into an application program.

5.2.1 Encode Numbers as RPGs

In Chapter 2, we introduced the notion of *bitonic permutations* and we presented two algorithms, namely `Encode_W.to.SiP` and `Decode_SiP.to.W`, for encoding an integer w into a self-inverting permutation π^* and extracting it from π^*. We have actually proved the following results.

Theorem 5.1. *Let w be an integer and let $b_1 b_2 \cdots b_n$ be the binary representation of w. The algorithm* `Encode_W.to.SiP` *encodes the number w in a self-inverting permutation π^* of length $2n+1$ in $O(n)$ time and space.*

Theorem 5.2. *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.*

In Chapter 3, we have presented an efficient and easily implemented algorithm for encoding numbers as reducible permutation flow-graphs through the use of self-inverting permutations; see, also [23]. In particular, we have proposed two such encoding algorithms: the algorithm `Encode_SiP.to.RPG-I` applies to any permutation π and relies on domination relations on the elements of π whereas the algorithm `Encode_SiP.to.RPG-II` applies to a self-inverting permutation π^* produced in any way and relies on the decreasing subsequences of π^*. Our results are summarized in the following theorems.

Theorem 5.3. *Let π^* be a self-inverting permutation over the set N_n. The permutation π^* can be encoded into a reducible permutation graph $F[\pi^*]$ in $O(n)$ time and space using algorithm* `Encode_SiP.to.RPG-I` *or* `-II`.

Theorem 5.4. *Let $F[\pi^*]$ be a reducible permutation graph of order $O(n)$ produced by the encoding algorithm* `Encode_SiP.to.RPG-I` *or* `-II`*. The permutation π^* can be correctly extracted from $F[\pi^*]$ in $O(n)$ time and space using algorithm* `Decode_RPG.to.SiP-I` *or* `-II`.

The reducible permutation graph $F[\pi^*]$ of the self-inverting permutation π^* is directed with a descending ordering on its nodes $s = u_{n+1}, u_n, \ldots, u_1, u_0 = t$. Hereafter, we shall call the edge (u_i, u_j) of graph $F[\pi^*]$ *forward* if $i > j$ and *backward* otherwise.

5.2.2 Call-graphs

A *call-graph* is a directed graph that represents calling relationships between program units in a computer program. Specifically, the nodes $f_1, f_2, \ldots, f_n$ of a call-graph represent functions, procedures, classes, or similar program units and each edge (f_i, f_j) indicates that f_i calls f_j; function f_i is called *caller* while f_j is called *callee*.

Call-graphs can be divided in two main classes of graphs, namely *static* and *dynamic*. A static call-graph is the structure describing those invocations that could be made from one program unit to another in any possible execution of the program [128]. The static call-graph can be determined from the program source code; we mention that, its construction is a time consuming process specifically in the case of large scale software [57].

A dynamic call-graph G is a directed graph that includes invocations of caller–callee pairs over an execution of the program P. Such a graph can be considered as an instance of the corresponding static call-graph for a specific input sequence I. The call-graph G is a data structure that is used by dynamic optimizers for analyzing and optimizing the whole-program's behavior; such a graph can be extracted by a profiler. It is fair to mention that the construction of a dynamic call-graph G of a program P is not a time consuming process even if P is a large scale software.

Throughout the paper we denote a dynamic call-graph G of the program P over the input I as $G(P, I)$. Figure 5.1(a) depicts the structure of the dynamic call-graph $G(P, I_{key})$ of an application program P with input I_{key}.

5.3 The Dynamic Watermarking Model WaterRPG

Having encoded a watermark number w as reducible permutation graph $F[\pi^*]$, let us now present our dynamic watermarking model WaterRPG; we first demonstrate its structural and operational components and, then, we describe the embedding and extracting watermarking algorithms.

5.3.1 Operational Framework

The main idea behind the proposed watermarking model is a systematic modification of appropriate function calls of the program P so that the execution of the resulting watermarked program P^* with a specific input I_{key} gives a dynamic call-graph $G(P^*, I_{key})$ from which the watermark graph $F[\pi^*]$ can be easily constructed.

More precisely, the main operations performed by the WaterRPG model can be outlined as follows: for a specific input I_{key} of the original program P, it takes the dynamic call-graph $G(P, I_{key})$ and the graph $F[\pi^*]$, and produces the watermarked program P^* so that its dynamic call-graph $G(P^*, I_{key})$ with input I_{key} is isomorphic to the watermark graph $F[\pi^*]$. The call-graphs $G(P, I_{key})$ and $G(P^*, I_{key})$ dictate the execution flow of the original program P and the watermarked program P^*, respectively. Thus, since the call-graph $G(P, I_{key})$ is not isomorphic to $G(P^*, I_{key})$ in general, the model controls the flow of selected function calls of P^* so that $O(P, I) = O(P^*, I)$ for every input I, where $O(P, I)$ (resp. $O(P^*, I)$) is the output of the program P (resp. P^*) with input I. In this framework, the program P^* is produced by only altering appropriate calls of specific functions of the input program P.

Figure 5.1 shows the dynamic call-graph $G(P, I_{key})$ of an application program P, the reducible permutation graph $F[\pi^*]$ which encodes the number $w = 4$ and the dynamic call-graph $G(P^*, I_{key})$ of the watermarked program P^*.

5.3.2 Model Components

Our watermarking model uses two main categories of components namely *structural* components and *operational* components. The first category includes the dynamic call-graph $G(P, I_{key})$ of the input program P, the watermark graph $F[\pi^*]$, and the dynamic call-graph $G(P^*, I_{key})$ of the watermarked program P^*, while the second category includes call patterns, control statements

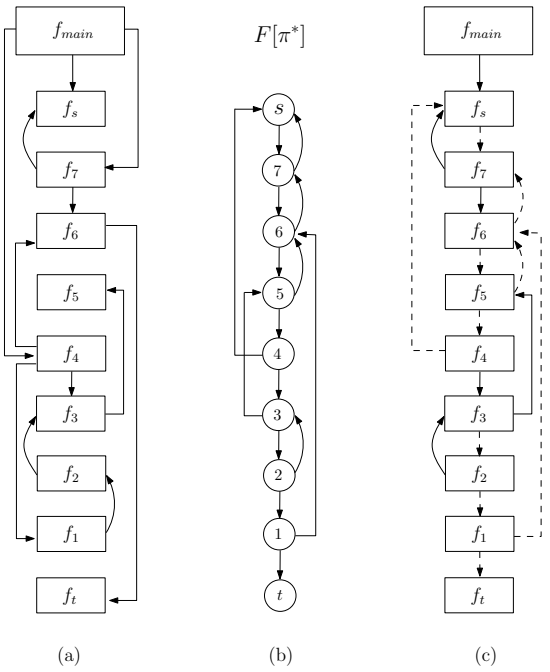


Figure 5.1: (a) The dynamic call-graph $G(P, I_{key})$ of an application program P. (b) The reducible permutation graph $F[\pi^*]$. (c) The dynamic call-graph $G(P^*, I_{key})$ of the watermarked program P^*.

and execution rules which are components related to the process of embedding the graph $F[\pi^*]$ into application program P.

We next describe the construction and main properties of the dynamic call-graph $G(P^*, I_{key})$, two call patterns based on which we correspond edges of the call-graph $G(P^*, I_{key})$ to function calls, and specific variables and statements which control the execution of real and water functions.

(I) The Dynamic Call-graph $\mathbf{G(P^*, I_{key})}$

Let $F[\pi^*]$ be a watermark graph (or, equivalently, water-graph) on $n+2$ nodes and $G(P, I_{key})$ be the dynamic call-graph of a program P on $n+3$ nodes $f_{main}, f_s, f_1, \ldots, f_n, f_t$ taken after running the program P with the input I_{key}. In general, the selection of the input I_{key} is such that it produces the call-graph $G(P, I_{key})$ having structure as "close" as possible to the structure of $F[\pi^*]$. We assign the $n+2$ nodes $f_s = f_{n+1}, f_n, \ldots, f_1, f_0 = f_t$ of the call-graph $G(P, I_{key})$ to $n+2$ nodes $s = u_{n+1}, u_n, \ldots, u_1, u_0 = t$ of $F[\pi^*]$ into 1-1 correspondence; the main function f_{main} do not correspond to any node of $F[\pi^*]$.

Let (u_i, u_j) be an edge in graph $F[\pi^*]$ and let (f_i, f_j) be an edge in call-graph $G(P, I_{key})$. We say that the edge (f_i, f_j) corresponds to edge (u_i, u_j) iff the node f_i corresponds to u_i and the node f_j corresponds to u_j, $0 \leq i, j \leq n+1$. Moreover, if (u_i, u_j) is a forward (resp. backward) edge in the graph $F[\pi^*]$ we say that the corresponding edge (f_i, f_j) in graph $G(P, I_{key})$ is a forward (resp. backward) edge.

The dynamic call-graph $G(P^*, I_{key})$ is constructed as follows:

- $V(G(P^*, I_{key})) = V(G(P, I_{key}))$, i.e., it has the same nodes as the call-graph $G(P, I_{key})$;

- $E(G(P^*, I_{key})) = E(F[\pi^*])$, i.e., (f_i, f_j) is an edge in $E(G(P^*, I_{key}))$ iff the corresponding (u_i, u_j) is an edge in $F[\pi^*]$.

The edges of the call-graph $G(P^*, I_{key})$ are divided into two categories namely *real* and *water* edges; note that, the real (resp. water) edges correspond to real (resp. water) function calls. An edge (f_i, f_j) of the call-graph $G(P, I_{key})$ is characterized as either

- *real edge* if (f_i, f_j) is an edge in $G(P, I_{key})$, or
- *water edge* if (f_i, f_j) is not an edge in $G(P, I_{key})$.

Figure 5.1(c) shows the dynamic call-graph $G(P^*, I_{key})$ along with its real edges (solid arrows) and water edges (dashed arrows).

(II) Call Patterns

In the implementation phase, we modify the source code of program P using specific function call patterns which we describe below.

Let P be an application program, $G(P, I_{key})$ be the dynamic call-graph of the program P with input I_{key}, and $F[\pi^*]$ be a watermark-graph which we have to embed into P. According to our watermarking model, the embedding process relies mainly on altering the execution-flow of appropriate function calls of P such that the execution of the resulting program P^* with the input I_{key} produces a call-graph $G(P^*, I_{key})$ which, after removing the node f_{main}, is isomorphic to watermark-graph $F[\pi^*]$.

Let (f_i, f_j) be an edge of call-graph $G(P^*, I_{key})$ or, equivalently, an edge which we want to appear in $G(P^*, I_{key})$. Since $G(P^*, I_{key})$ has two types of edges it follows that (f_i, f_j) is either real or water edge. Based on the type of (f_i, f_j), we do the following:

- if (f_i, f_j) is a water edge we add the statement `call`(f_j) in the function f_i, while
- if (f_i, f_j) is a real edge we add no call statement since the statement `call`(f_j) exists in f_i.

Based on whether (f_i, f_j) is either a forward or a backward edge we add specific statements in functions f_i and f_j according to the following two call patterns namely *forward* and *backward* call patterns:

(a) if (f_i, f_j) is a forward edge we add the statement $x = x + h()$ in function f_i before the call-site or, equivalently, call-point of the function f_j, and the statement $x = x + c()$ in the function f_j, while

(b) if (f_i, f_j) is a backward edge we add the statement $x = x + g()$ in function f_i before the call-site of the function f_j, and the statement $x = x + c()$ in the function f_j,

where x is a variable and $h()$, $g()$ and $c()$ are functions. Figure 5.3(a) depicts the forward call pattern or, for short, *f-call*, while Figure 5.3(b) depicts the backward call pattern or, for short, *b-call*.

Recall that the direct edge (f_i, f_j) of a call-graph represents a function call operation where f_i is the caller function and f_j the callee function; in other words, it means that in function f_i

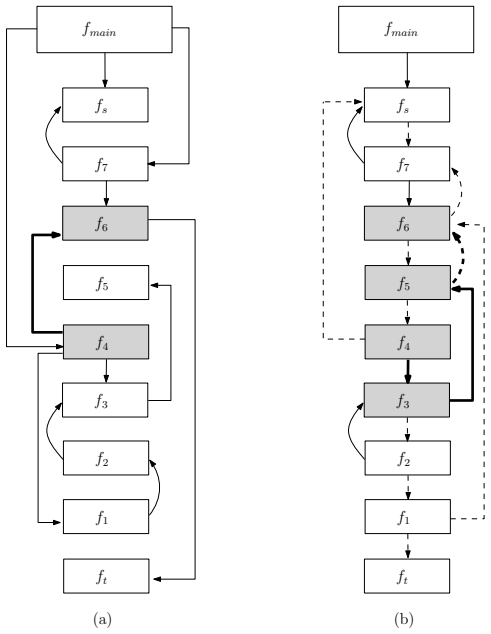


Figure 5.2: (a) The real-call (f_4, f_6) in the call-graph $G(P, I_{key})$ of a program P; bold arrow. (b) The corresponding path-call (f_4, f_3, f_5, f_6) in the call-graph $G(P^*, I_{key})$ of the watermarked program P^*; bold arrows.

there exists the statement `call`(f_j). Hereafter, in this case we shall say that (f_i, f_j) is a direct call.

In a call-graph of an application program we usually meet sequences of calls of the form $(f_i, f_{k_1}, f_{k_2}, \ldots, f_{k_m}, f_j)$. For simplicity we set $f_i = f_{k_0}$ and $f_j = f_{k_{m+1}}$ and suppose that each of these calls (f_{k_0}, f_{k_1}), (f_{k_1}, f_{k_2}), ..., (f_{k_m}, f_{m+1}) is either forward or backward. We extend the notion of the direct call (f_i, f_j) to indirect call $(f_i \rightarrow f_j)$; an indirect call consists of a path of functions $(f_i, f_{k_1}, \ldots, f_j)$ of length $\ell \geq 2$. Using the f-call and b-call patterns, we next define the path call pattern or, for short, *p-call* as follows:

(c) if $(f_{k_i}, f_{k_{i+1}})$ and $(f_{k_{i+1}}, f_{k_{i+2}})$ are two consecutive calls of a call sequence, we apply an f-call or a b-call in $(f_{k_{i+1}}, f_{k_{i+2}})$ by first adding either the statement $x = x + h()$ or $x = x + g()$ in function $f_{k_{i+1}}$ after the call-point of statement $x = x + c()$, and then adding the statement $x = x + c()$ in $f_{k_{i+2}}$, $0 \leq i \leq m - 1$.

Figure 5.3 shows the structures of the patterns f-call and b-call of the direct call (f_i, f_j), and the structure of the pattern p-call of an indirect call $(f_i \rightarrow f_j)$.

(III) Control Statements and Variables

In any watermarking model both the original program P and the watermarked program P^* have to operate identically. Thus, since the call-graphs $G(P, I_{key})$ and $G(P^*, I_{key})$ dictate the execution flow of programs P and P^*, respectively, and the call-graph $G(P, I_{key})$ is not isomorphic to $G(P^*, I_{key})$, we have to control the flow of selected function calls of program P^* so that $O(P, I) = O(P^*, I)$ for every input I.

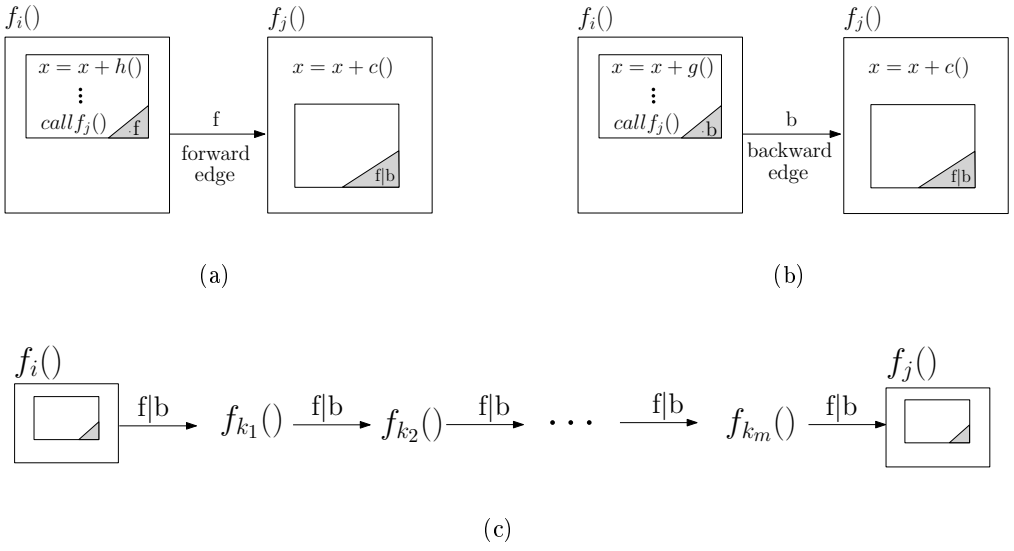


Figure 5.3: (a) The forward call pattern *f-call*; (b) The backward call pattern *b-call*; (c) The path call pattern *p-call*. The boxes with marked corners are the f|b-blocks.

To do this, we exploit the values of specific variables in a function f_i by using them in some selected or added control statements as part of opaque predicates. More precisely, we use the variable x of the f-call and b-call patterns and include it in a specific control statement s causing thus an "appropriate execution flow" of the functions of the call-graph $G(P^*, I_{key})$; with the term "appropriate execution flow" we mean that the execution flow of the functions of the call-graph $G(P^*, I_{key})$ is such that $O(P, I) = O(P^*, I)$ for every input I. Hereafter, we call *cf-statement* the control statement s and *cf-variable* the variable x. In this point, we also define the *f-block* and *b-block* to be specific parts of the code which contain (i) cf-statements, (ii) cf-variables, and (iii) water-forward or water-backward function calls. We denote by f|b-block either an f-block or a b-block; in Figure 5.3, the f|b-blocks are shown by boxes with marked corners.

We next describe the mechanism which ensures an appropriate execution flow of the functions of $G(P^*, I_{key})$ through the altering of the execution flow of the functions of the program P by modifying or adding some specific control statements. In fact, what the mechanism actually does is to modify the conditions or expressions of these control statements by adding opaque predicates.

Definition 5.1. A predicate Q is opaque at a program point p, if at point p the outcome of Q is known at embedding time. If Q always evaluates to *true* we write Q_p^T, for *false* we write Q_p^F, and if Q sometimes evaluates to *true* and sometimes to *false* we write $Q_p^?$.

Let (f_i, f_j) be a direct call in our program P^* or, equivalently, an edge in the call-graph $G(P^*, I_{key})$; it is either real-forward, real-backward, water-forward, or water-backward edge. In any case, the proposed mechanism uses the value of the cf-variable x and makes the following operations:

Program P	Program P^*
`function` $f_i()$ ... `if` (*condition)* ... `statements`; ...	`function` $f_i()$ ... `if` (*condition* & $Q_p^?$*)* ... `x = x + h();` ... **call** $f_j()$; ... `statements`; ...

Figure 5.4: An example of cf-statement modification via opaque predicates in the case where (f_i, f_j) is a water-forward function call.

Program P	Program P^*
`function` $f_i()$ ... `if` (*condition)* ... **call** $f_j()$; ... `statements`; ...	`function` $f_i()$ ... `if` (*condition* & $Q_p^?$*)* ... `x = x + g();` ... **call** $f_j()$; ... `statements`; ...

Figure 5.5: An example of cf-statement modification via opaque predicates in the case where (f_i, f_j) is a real-backward function call.

- in function f_i: create a control statement (if, switch, for, while, etc), add an opaque predicate $Q_p^?$ containing the cf-variable x, and insert it at a point p before the statement $x = x + h()$ or $x = x + g()$; we could also select a control statement at a point p, if there exists, consider it as cf-statement and include the opaque predicate $Q_p^?$ in its condition part.

- in function f_j: create a control statement as in function f_i, if such a statement does not exist, and insert it at a point p before the statement $x = x + c()$; if such a statement exists, we only add a new opaque predicate $Q_p^?$ in the condition of that statement. The main body of f_j is included in a block of a cf-statement the execution of which is depending upon the behavior (i.e., true or false) of the opaque predicate $Q_p^?$.

Note that, the above operations form specific parts of code of functions f_i and f_j, namely f|b-blocks, i.e., either f-blocks or b-blocks; see, Figure 5.3.

Call (f_i, f_j) of Program P^*	

`function` $f_i()$
...
`if` (*condition* & $Q_p^?$)
...
`x = x + h();`
...
call $f_j()$;
...
`statements;`
...
	`function` $f_j()$
	...
	`if` (*condition* & $Q_p^?$)
	...
	`x = x + c();`
	...
	`if` (*condition* & $Q_p^?$)
	...
	`statements;`
	...

Figure 5.6: An example of cf-statement modification via opaque predicates of the function f_j in the case where (f_i, f_j) is a water-forward function call.

Figure 5.4 shows an example of the modification of the condition part of an `if` cf-statement via an opaque predicate; since (f_i, f_j) is a water-forward function call, the statement `call`(f_j) does not exist in function f_i, and thus we add it in f_i, while the cf-statement is the $x = x + h()$. On the other hand, Figure 5.5 shows an example in case where (f_i, f_j) is a real-backward function call. In this case, the statement `call`(f_j) does exist in f_i while the cf-statement is the $x = x+g()$. Figure 5.6 shows an example of the modification of the function f_j in the case where (f_i, f_j) is a water-forward function call.

Remark 5.1. Based on the structural properties of the watermark graph $F[\pi^*]$ and call-graph $G(P^*, I_{key})$ we can easily prove the following lemma.

Lemma 5.1. *Let $G(P, I_{key})$ and $G(P^*, I_{key})$ be the call-graphs of programs P and P^*, respectively, on input I_{key}, and let (f_i, f_j) be an edge in call-graph $G(P, I_{key})$. Then, there always exists an edge (f_i, f_j) or a path $(f_i, f_{k_1}, f_{k_2}, \ldots, f_{k_m}, f_j)$ in call-graph $G(P^*, I_{key})$.*

Remark 5.2. In our implementation, in the case where (f_i, f_j) is an edge in $G(P, I_{key})$ and (f_i, f_j) is not an edge in $G(P^*, I_{key})$ we have to compute a path $(f_i, f_{k_1}, \ldots, f_j)$ of function calls in $G(P^*, I_{key})$. Such a path is a shortest path from f_4 to f_6 in the graph $G(P^*, I_{key})$; it may consist of all types of edges, that is, real-forward or real-backward and water-forward or water-backward edges. Figure 5.2(a) shows the edge (f_4, f_6) in $G(P, I_{key})$ which is not an edge in $G(P^*, I_{key})$, while Figure 5.2(b) shows its corresponding shortest path from f_i to f_j, that is, the path (f_4, f_3, f_5, f_6); note that, (f_4, f_3) is a real-forward edge, (f_3, f_5) is a real-backward edge, and (f_5, f_6) is a water-backward edge.

(VI) Execution Rules

We present the rules based on which we control the execution flow of the functions of P^* such that $O(P, I) = O(P^*, I)$ for every input I. In fact, we show in all the cases how the value of $Q_p^?$ dictates the execution flow of functions of $G(P^*, I_{key})$.

Let (f_i, f_j) be a direct call in program P^* or, equivalently, an edge in the call-graph $G(P^*, I_{key})$.

We distinguish the following cases:

- (f_i, f_j) is **real-forward** or **real-backward**: in this case we modify the functions f_i and f_j as follows:
  - Function f_i: the opaque predicate $Q_p^?$ in the cf-statement before the cf-value $x = x + h()$ or $x = x + g()$ and the `call`(f_j) is evaluated to true, that is, Q_p^T.
  - Function f_j: the opaque predicate $Q_p^?$ in the cf-statement before the cf-value $x = x+c()$ is evaluated to true, that is, Q_p^T, while the $Q_p^?$ for the cf-statement which controls the statements of the main body of the function f_j is also evaluated to true, that is, Q_p^T.

- (f_i, f_j) is **water-forward** or **water-backward**: in this case we modify the functions f_i and f_j as follows:
  - Function f_i: the opaque predicate $Q_p^?$ in the cf-statement before the cf-value $x = x + h()$ or $x = x + g()$ and the `call`(f_j) is evaluated to true, that is, Q_p^T.
  - Function f_j: the opaque predicate $Q_p^?$ in the cf-statement before the cf-value $x = x+c()$ is evaluated to true, that is, Q_p^T, while the $Q_p^?$ for the cf-statement which controls the statements of the main body of the function f_j is evaluated to false, that is, Q_p^F.

Recall that a predicate Q is opaque at a program point p, if at point p the outcome of Q is known at the embedding time.

Remark 5.3. During the execution of the function f_i of the program P^* only one opaque predicate $Q_p^?$ of the cf-statements is evaluated to true with respect to the current value of the cf-variable x.

5.3.3 Embedding an RPG into a Code

Let us now present our model's algorithm which efficiently watermarks an application program P by embedding the reducible permutation graph $F[\pi^*]$ into P. The proposed embedding algorithm, which we call `Embed_RPG.to.CODE`, is described below.

Algorithm `Embed_RPG.to.CODE`

1. Take as input the source code of the program P, select an input I_{key}, and construct the call-graph $G(P, I_{key})$; let $S = (f_{main}, f_s = f_{n+1}, f_n, \ldots, f_1, f_0 = f_t)$ be the execution sequence of the functions of call-graph $G(P, I_{key})$, that is, f_i appears before f_j in S if f_i is executed before f_j with input I_{key}, and let $s = u_{n+1}, u_n, \ldots, u_1, u_0 = t$ be the $n + 2$ nodes of the watermark graph $F[\pi^*]$;

2. Remove the node f_{main} from the call-graph $G(P, I_{key})$ and assign an exact pairing (i.e., 1-1 correspondence) of the $n + 2$ nodes of graph $G(P, I_{key})$ or, equivalently, of the $n + 2$ functions $f_s = f_{n+1}, f_n, \ldots, f_1, f_0 = f_t$ to the nodes of $F[\pi^*]$;

3. Construct the graph $G(P^*, I_{key})$ as follows:

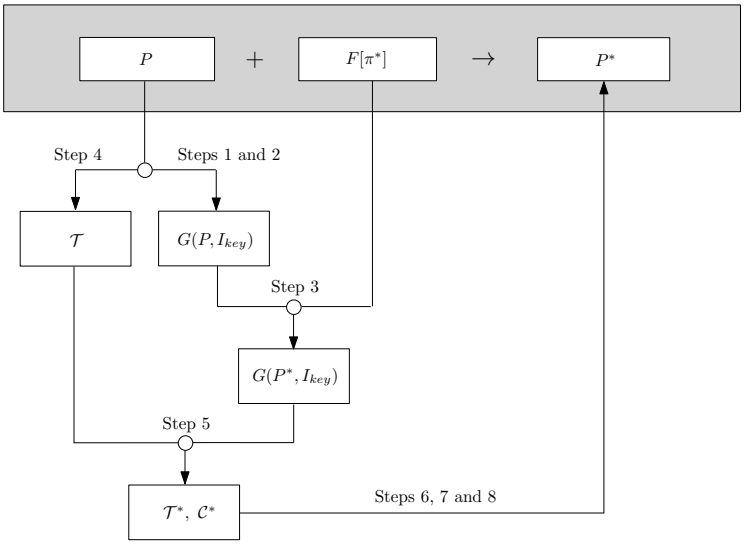


Figure 5.7: A block diagram of the main operations of the embedding algorithm.

3.1. $V(G(P^*, I_{key})) = V(G(P, I_{key}))$, i.e., $G(P^*, I_{key})$ has the same nodes as the call-graph $G(P, I_{key})$;

3.2. $E(G(P^*, I_{key})) = E(F[\pi^*])$, i.e., (f_i, f_j) is an edge in $E(G(P^*, I_{key}))$ iff the corresponding (u_i, u_j) is an edge in graph $F[\pi^*]$;

4. Create a call-table $\mathcal{T}$ of size m which contains all the m function calls (f_i, f_j) as they appear in the execution trace of program P with input I_{key};

5. Create the tables $\mathcal{T}^*$ and $\mathcal{C}^*$, both of size m^*, as follows:

5.1. For each function call (f_i, f_j) of table $\mathcal{T}$ do the following:

- if (f_i, f_j) is a function call of $G(P^*, I_{key})$ then insert (f_i, f_j) in table $\mathcal{T}^*$ and its characterization in table $\mathcal{C}^*$; in this step, (f_i, f_j) is characterized as either real-forward or real-backward;
- if (f_i, f_j) is not a function call in graph $G(P^*, I_{key})$ then:
  - compute the shortest path $(f_i, f_{k_1}, f_{k_2}, \ldots, f_{k_\ell}, f_j)$ from node f_i to node f_j in $G(P^*, I_{key})$,
  - insert the calls (f_i, f_{k_1}), (f_{k_1}, f_{k_2}), ..., (f_{k_ℓ}, f_j) in table $\mathcal{T}^*$, in that order, and their characterizations in table $\mathcal{C}^*$; in this step, a call is characterized as either real-forward, real-backward, water-forward, or water-backward (see, Subsection 5.3.2), and
  - mark the first and last function calls of the shortest path, i.e., (f_i, f_{k_1}) and (f_{k_ℓ}, f_j), as *first* and *last*, respectively;

5.2. For each function call (f_i, f_j) of graph $G(P^*, I_{key})$ check whether it appears in table $\mathcal{T}^*$; if not, do the following:

- find the first appearance of a function call of type (f_o, f_i) in table $\mathcal{T}^*$, where f_o is any function;

- insert the function call (f_i, f_j) in table $\mathcal{T}^*$ after the call (f_o, f_i) and its characterization in the corresponding row in table $\mathcal{C}^*$;

6. Take each function call (f_i, f_j) of the table $\mathcal{T}^*$ and modify the functions f_i and f_j of program P as follows:

6.1. Add/replace call statements and locate appropriate call points in function f_i as follows:

- if (f_i, f_j) is a real function call then find the statement $\texttt{call}(f_j)$ in function f_i and locate its call-point;
- if (f_i, f_j) is a water function call then:
  - if it is the first function call of a short path, then find the *last* function call of that path in table $\mathcal{T}^*$, say, (f_{k_ℓ}, f_{last}), replace the statement $\texttt{call}(f_{last})$ in function f_i with the statement $\texttt{call}(f_j)$, and locate its call-point;
  - otherwise, if $\texttt{call}(f_j)$ does not exist in f-block of function f_i, add the statement $\texttt{call}(f_j)$ in f|b-block and locate its call-point;

6.2. Insert either statement $x = x + h()$ or $x = x + g()$ before the statement $\texttt{call}(f_j)$, if (f_i, f_j) is characterized either as forward or backward, respectively;

6.3. Include both statements $x = x+h()$ or $x = x+g()$ and $\texttt{call}(f_j)$ in a control statement and evaluate it as true or false using the *cf-variable* x;

6.4. Include all the statements of the b-block of function f_i in a control statement and evaluate it using the *cf-variable* x;

6.5. Add the statement $x = x + c()$ before the f|b-block of function f_j;

7. For each function call (f_{out}, f_j) of program P s.t. $f_{out} \notin G(P^*, I_{key})$ and $f_j \in G(P^*, I_{key})$ do the following:

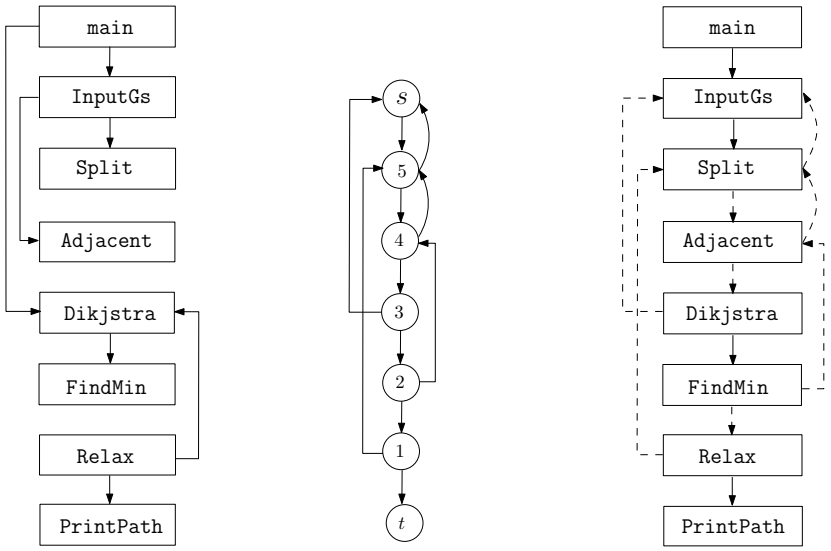
7.1. Find the first appearance of a function call of type (f_o, f_j) in table $\mathcal{T}^*$, such that f_o is any function and f_j is either a real function call or the last function of a shortest path $(f_i, f_{k_1}, f_{k_2}, \ldots, f_{k_\ell}, f_j)$, i.e., $(f_o, f_j) = (f_{k_\ell}, f_j)$;

7.2. Take the value of the *cf-variable* x of function f_{k_ℓ}, say, "value", and insert the statement $x =$ "*value*" in function f_{out} before the call-point of the statement $\texttt{call}(f_j)$;

8. Return the source code of the modified program P which is the watermarked program P^*.

Remark 5.4. In Step 5 of the embedding algorithm, the edges (f_i, f_j) are included into the table $\mathcal{T}$ in a specific order. This order is determined by the order they appeared in the execution trace of program P with input I_{key}, i.e., if the function call (f_i, f_j) appears before (f_k, f_ℓ) in the execution trace of P, then the edge (f_i, f_j) appears before the edge (f_k, f_ℓ) in table $\mathcal{T}$.

Remark 5.5. Let (f_i, f_j) be an edge which is handled in Step 6 of the embedding algorithm and let the statement $\texttt{call}(f_j)$ appear more that once in function f_i. We point out that in this case we insert both the cf-variable and cf-statement before the call-site of each statement $\texttt{call}(f_j)$ in function f_i.



(a) (b) (c)

Figure 5.8: (a) The dynamic call-graph $G(\texttt{Shortest\_Path}, I_{key})$. (b) The reducible permutation graph $F[\pi^*]$ which encodes the watermark $w = 2$, where $\pi^* = (3, 5, 1, 4, 2)$. (c) The dynamic call-graph $G(\texttt{Shortest\_Path}^*, I_{key})$.

The Algorithm by an Example. In order to illustrate the working of the embedding algorithm `Embed_RPG.to.CODE`, we present a simple example and show the main operations (i.e., function calls) and the values of the main variables during the algorithm's execution.

In our example, we chose the original program P to be one that computes the shortest paths in a weighted graph G with non-negative edge-values; it takes as input a graph G and a node s and computes the shortest paths from s to every other node $v \in V(G) - \{s\}$. The program P, which we call `Shortest_Path`, consists of 8 functions (i.e., 7 functions plus the main) and have the property that its dynamic call-graph $G(\texttt{Shortest\_Path}, I_{key})$ is the same for every input I_{key}; see, its dynamic call-graph in Figure 5.8(a). Moreover, we chose to embed the watermark number $w = 2$ into the source code of program `Shortest_Path`. The reducible permutation graph $F[\pi^*]$ which encodes the watermark $w = 2$ consists of 7 nodes and is depicted in Figure 5.8(b). We point out that, according to our model's rules, the number $w = 2$ is encoded by the SiP $\pi^* = (3, 5, 1, 4, 2)$ and it can be successfully embedded into `Shortest_Path` since our program consists of 8 functions and the graph $F[\pi^*]$ contains 7 nodes. The dynamic call-graph $G(\texttt{Shortest\_Path}^*, I_{key})$ of our watermarked program is presented in Figure 5.8(c). Observe that, $G(\texttt{Shortest\_Path}^*, I_{key})$ is isomorphic to the watermark graph $F[(3, 5, 1, 4, 2)]$.

In Figure 5.9 we show the call-tables $\mathcal{T}$ and $\mathcal{T}^*$ of the original and watermarked programs `Shortest_Path` and `Shortest_Path`*, respectively, the edge characterization of each function call (f_i, f_j) of the call-table $\mathcal{T}^*$ (see, Table $\mathcal{C}^*$), and the increment of the value of the cf-variable x in each case. More precisely, in Table $\mathcal{C}^*$ each (f_i, f_j) is characterized as either rf (real-forward), rb (real-backward), wf (water-forward), or wb (water-backward), while in the case where a function call (f_i, f_j) is replaced by a path of calls (shortest path) we characterize as *first* (resp. *last*) the

Call-table $\mathcal{T}$	Call-table $\mathcal{T}^*$	Table $\mathcal{C}^*$	Values of the *cf-variable*
`main` → `InputGs`	`main` → `InputGs`	*rf*	x+=3 (3) → x+=1 (4)
`InputGs` → `Split`	`InputGs` → `Split`	*rf*	x+=3 (7) → x+=1 (8)
`InputGs` → `Split`	`Split` → `InputGs`	*wb*	x+=2 (10) → ... (11)
`InputGs` → `Adjacent`	`InputGs` → `Split`	*rf*	x+=3 (14) → ... (15)
`main` → `Dikjstra`	`InputGs` → `Split`	*wf* – first	x+=3 (18) → ... (19)
`Dikjstra` → `FindMin`	`Split` → `Adjacent`	*wf* – last	x+=3 (22) → ... (23)
`Relax` → `Dikjstra`	`Adjacent` → `Split`	*wb*	x+=2 (25) → ... (26)
`Relax` → `PrintPath`	`main` → `InputGs`	*wf* – first	x+=3 (29) → ... (30)
	`InputGs` → `Split`	*wf*	x+=3 (33) → ... (34)
	`Split` → `Adjacent`	*wf*	x+=3 (37) → ... (38)
	`Adjacent` → `Dikjstra`	*wf* – last	x+=3 (41) → ... (42)
	`Dikjstra` → `InputG`	*wb*	x+=2 (44) → ... (45)
	`Dikjstra` → `FindMin`	*rf*	x+=3 (48) → ... (49)
	`FindMin` → `Adjacent`	*wb*	x+=2 (51) → ... (52)
	`Dikjstra` → `FindMin`	*wf* – first	x+=3 (55) → ... (56)
	`FindMin` → `Relax`	*wf* – last	x+=3 (59) → ... (60)
	`Relax` → `Split`	*wb*	x+=2 (62) → ... (63)
	`Relax` → `PrintPath`	*rf*	x+=3 (66) → x+=1 (67)

Figure 5.9: The call-tables $\mathcal{T}$ and $\mathcal{T}^*$ of the programs `Shortest_Path` and `Shortest_Path`*, respectively, the edge characterization table $\mathcal{C}^*$, and the values of the cf-variable.

first (resp. last) function call of that path. The fourth table of Figure 5.9 shows the values (in parentheses) of the cf-variable x in both functions f_i and f_j of the program `Shortest_Path`*. Recall that according to the f-call and b-call patterns (see, Section 5.3.2), if (f_i, f_j) is a forward edge we add the statement $x = x + h()$ in function f_i, while if (f_i, f_j) is a backward edge we add the statement $x = x + g()$ in function f_i; in both cases we unconditionally add the statement $x = x + c()$ in function f_j.

In our example, we initialize the cf-variable $x = 0$ and consider for simplicity reasons constant values for the functions $h()$, $g()$, and $c()$, that is, $h() = 3$, $g() = 2$, and $c() = 1$. Based on the above, we take the first function call (`main`, `InputGs`) of Table $\mathcal{T}^*$ and, since it is characterized as rf in Table $\mathcal{C}^*$, we increase by $h()$ the value of the cf-variable x in function `main` before the call site of `InputGs`, i.e., we set $x = x + h()$ and thus $x = 3$. In the callee function `InputGs` we always increase by $c()$ the value of the variable x, i.e., we set $x = x + c()$ and thus $x = 4$. We observe that, the function call (`Split`, `InputGs`) of Table $\mathcal{T}^*$ is characterized as wb (water-backward) and, thus, we increase by $g() = 2$ the value of the variable x in function `Split`, again before the call site of `InputGs`, i.e., x becomes equal to 10 because in the previous function call x's value was 8 (see, last table of Figure 5.9). Note that, by construction the shortest paths of function calls do not intersect.

5.3.4 Extracting the RPG from the Code

We next present our WaterRPG model's algorithm for extracting the graph $F[\pi^*]$ from the program P^* watermarked by the embedding algorithm `Embed_RPG.to.CODE`. The proposed extracting algorithm works as follows:

Algorithm `Extract_CODE.to.RPG`

1. Take as input the program P^* watermarked by the embedding algorithm `Embed_RPG.to.CODE` and run it with input I_{key};
2. Construct the call-table $\mathcal{T}$ using the execution trace of the program P^* with input I_{key};
3. Construct the dynamic call-graph $G(P^*, I_{key})$ using the call-table $\mathcal{T}$ as follows:
   - 3.1. take all the function calls (f_i, f_j) of table $\mathcal{T}$ and add both functions f_i and f_j in the set V; note that, V has $n+2$ elements since $\mathcal{T}$ contains $n+2$ different functions;
   - 3.2. take all the function calls (f_i, f_j) of table $\mathcal{T}$ and add the selected pairs in the set E; note that, E contains $2n+1$ elements;
   - 3.3. assign the set V to $V(G(P^*, I_{key}))$ and the set E to $E(G(P^*, I_{key}))$;
4. Remove the node f_{main} from the graph $G(P^*, I_{key})$; the resulting graph is a reducible permutation graph isomorphic to $F[\pi^*]$ (see, algorithm `Embed_RPG.to.CODE`);
5. Compute the unique Hamiltonian path $HP = (f_0, f_1, f_2, \ldots, f_n, f_{n+1})$ of $G(P^*, I_{key})$;
6. Relabel the nodes of the graph $G(P^*, I_{key})$ according to their order in the HP as follows: $f_0 = u_{n+1}$, $f_1 = u_n$, $f_2 = u_{n-1}$, $\ldots$, $f_n = u_1$, $f_{n+1} = u_0$; the resulting graph $G(P^*, I_{key})$ has a unique Hamiltonian path $HP = (u_{n+1}, u_n, u_{n-1}, \ldots, u_1, u_0)$ and thus $G(P^*, I_{key}) = F[\pi^*]$;
7. Return the reducible permutation graph $F[\pi^*]$.

Remark 5.6. In Step 5 of the extracting algorithm, we compute the unique Hamiltonian path of the graph $F[P^*]$. Indeed, it has been shown that the reducible permutation graph $F[\pi^*]$ has always a unique Hamiltonian path, denoted by $\mathrm{HP}(F[\pi^*])$, and this Hamiltonian path can be found in $O(n)$ time, where n is the number of nodes of $F[\pi^*]$ [23]. Since $F[\pi^*]$ is isomorphic to $G'(P^*, I_{key})$ we can compute the unique Hamiltonian path HP of the graph $F[P^*]$ within the same time complexity.

5.4 Implementation

In this section we present in detail the watermarking process performed by our WaterRPG model on a Java application program P. We show the implementation of our watermarking process using a real program with market-name `Laser` which we have downloaded from the website `www.java-gaming.org` containing various and different in characteristics game application programs.

Let $f_s = f_{n+1}, f_n, \ldots, f_1, f_0 = f_t$ be the functions of the dynamic call-graph $G(P, I_{key})$, where $P =$ `Laser`. Recall that the functions $f_s = f_{n+1}, f_n, \ldots, f_1, f_0 = f_t$ are into 1-1 correspondence with the nodes $s = u_{n+1}, u_n, \ldots, u_1, u_0 = t$ of the reducible permutation graph $F[\pi^*]$ which encodes the watermark number w.

We focus on the function $f_i =$ `up()` of the program `Laser`; in our implementation, the important part of the Java code of $f_i =$ `up()` is the following:

```
public void up{
    if (b[cx + 1][cy − 1 − 1].bgr() ...){
            hlth − −;
    }
    b[cx + 1][cy].bgr(black);
    ⋮
```

We first show the straightforward case of the watermarking process on function $f_i =$ `up()` and, then, we proceed with advanced cases. In all cases our model uses the cf-variable x which increases its value by $h() = 3$, $g() = 2$, and $c() = 1$; see, Call Patterns in Section 3.2.

Before we proceed to watermark the function $f_i =$ `up()`, we divide the callee functions of f_i into the following three categories:

$\mathcal{A}_{callee}$: contains the callee functions f_j^1 and f_j^2 of f_i which correspond to forward node u_j^1 and backward node u_j^2 of graph $F[\pi^*]$, respectively; that is, both f_j^1 and f_j^2 are functions of the dynamic call-graph $G(P, I_{key})$.

$\mathcal{B}_{callee}$: contains the callee functions f_j^* of f_i which are executed with the input I_{key} except of f_j^1 and f_j^2; that is, f_j^* is a function of the dynamic call-graph $G(P, I_{key})$.

$\mathcal{C}_{callee}$: contains all the callee functions f_j^{**} of f_i which are not executed with the input I_{key}.

Naive-case Implementation

Let u_i be the node of graph $F[\pi^*]$ which corresponds to $f_i =$ `up()`, and let u_j^1 and u_j^2 be the two nodes of $F[\pi^*]$ such that (u_i, u_j^1) and (u_i, u_j^2) are the forward and backward outgoing edges of node u_i, respectively. Let f_j^1 and f_j^2 be the two functions of $G(P, I_{key})$ which correspond to nodes u_j^1 and u_j^2, respectively; in our implementation, $f_j^1 =$ `down()` and $f_j^2 =$ `health()`.

We next describe in a step-by-step manner the modifications we make in function $f_i =$ `up()` according to the watermarking rules of our WaterRPG model. The watermarking process of the naive-case implementation consists of the following phases:

(I) We first include the body of the function f_i into a control statement holding opaque predicates of the cf-variable x. In our naive-case implementation, we use the statement `if-then-else` and add opaque predicates of the form `x == value`; see, statement `if (x == 271 && down == false) {...}` of Figure 5.11.

Then, we handle the functions f_j^1 and f_j^2 of categories $\mathcal{A}_{callee}$; in particular, we locate the call-points of all the statements `call`(f_j^1) and `call`(f_j^2) in f_i, if any, and we do the following:

- We form an f-block, in the case where f_i contains f_j^1, by adding the statement $x = x + h()$ in a call-point before that of $\texttt{call}(f_j^1)$ and including both $x = x + h()$ and $\texttt{call}(f_j^1)$ into a control statement with opaque predicates using the cf-variable x; in our implementation, $f_j^1 = \texttt{down}()$ and $h() = 3$.
- We similarly form a b-block, in the case where f_i contains f_j^2, by adding the statement $x = x + g()$ instead of $x = x + h()$ as before; in our implementation, $f_j^2 = \texttt{health}()$ and $g() = 2$.

In the case where the function f_i does not contain $\texttt{call}(f_j^1)$ or $\texttt{call}(f_j^2)$, we locate a call-point before that of the control statement `if-then-else` and we do the following:

- If f_i does not contain $\texttt{call}(f_j^1)$, we add the statements $x = x+h()$ and $\texttt{call}(f_j^1)$ in this order and, then, we include both $x = x + h()$ and $\texttt{call}(f_j^1)$ into a control statement with conditions consisting of opaque predicates using the cf-variable x; recall that $h() = 3$; see, statement `if (x == 271) && down == true) {...}` of Figure 5.11.
- If f_i does not contain $\texttt{call}(f_j^2)$, we add the statements $x = x + g()$ and $\texttt{call}(f_j^2)$ in this order; we also include both statements into a control statement as before; see, statement `if (x == 268) {...}` of Figure 5.11.

(II) In this phase, we locate a point in the beginning of the callee function f_j^1 (resp. f_j^2) of function f_i, add the statement $x = x + c()$ in this point and include $x = x + c()$ into a control statement with conditions consisting of opaque predicates using the cf-variable x; in our implementation $c() = 1$; see, statement `if (x == 267) {...}` of Figure 5.11.

(III) We next handle all the functions f_j^* of category $\mathcal{B}_{callee}$, that is, the callee functions of f_i that are functions of the call-graph $G(P, I_{key})$ except of f_j^1 and f_j^2. For every direct call (f_i, f_j^*) we compute the sequence $(f_i, f_{k_1}, \ldots, f_j^*)$ which corresponds to the shortest path $(u_i, u_{k_1}, \ldots, u_j^*)$ from u_i to u_j^* in graph $F[\pi^*]$; then, we remove the statement $\texttt{call}(f_j^*)$ from f_i and add either the statements $x = x + h()$ and $\texttt{call}(f_j^1)$ if (u_i, u_{k_1}) is a forward edge or the statements $x = x + g()$ and $\texttt{call}(f_j^1)$ if (u_i, u_{k_1}) is a backward edge in $F[\pi^*]$; in any case, we include the added statements into a control statement with conditions consisting of opaque predicates using the cf-variable x.

(IV) In the last phase we handle all the functions f_{out} of program P which call functions that correspond to nodes of graph $F[\pi^*]$, i.e., f_{out} is not a function of the call-graph $G(P, I_{key})$ and calls a function f_j of $f_s = f_{n+1}, f_n, \ldots, f_1, f_0 = f_t$. We find the first appearance of a function call of type (f_{out}, f_j) in table $\mathcal{T}^*$ such that f_j is either a real function call or the last function of a shortest path $(f_i, f_{k_1}, f_{k_2}, \ldots, f_{k_\ell}, f_j)$ (see, Step 7 of embedding algorithm), take the value of the cf-variable x of function f_{k_ℓ}, say, "value", and insert the statement $x =$ "$value$" in function f_{out} before the call-point of the statement $\texttt{call}(f_j)$.

All the functions f_j^{**} of category $\mathcal{C}_{callee}$ are ignored during the process of watermarking the function $f_i = \texttt{up}()$ since they are not executed with the input I_{key}.

Remark 5.7. In order to avoid a large increment of the value of cf-variable x, in the implementation Phase (I), if a call function is included into a loop structure we could apply the call patterns as follows: (i) we add the assignment of the cf-variable (i.e., $x = x + h()$ or $x = x + g()$

The Naive-case Implementation

```
public void up{
  if(x == 267){
        x = x + 1;
  }
  if(x == 268){
        x = x + 2;
        health();
  }
  if(x == 271&&down == true){
        x = x + 3;
        down();
  }
  if(x == 271&&down == false){
        if(b[cx + 1][cy − 1 − 1]...){
              hlth − −;
        }
        b[cx + 1][cy].bgr(black);
        ⋮
```

Figure 5.10: The function `up`() of the original program `Laser` watermarked with the naive approach; the functions `down`() and `health`() are both water functions and belong to category $\mathcal{B}_{callee}$, i.e., both are functions of $G(\texttt{Laser}, I_{key})$.

assignments) outside this loop, and (ii) we add the statement x=*value_before_loop* inside and before the end of the loop, where *value_before_loop* is the value of x before the execution of this loop.

Stealthy-case Implementation

We next show properties and modification rules of the model's call patterns based on which we can stealthily watermark a Java application program P. The main modification cases, which we call stealthy cases, supported by the WaterRPG model are the following:

(S.1) *Making nested patterns*: We can merge f-statements and b-statements in any way; for example, we can include the control b-statement `if` (`x` == `268`) {...} inside the f-statement `if` (`x` == `271` && `down` == `true`) {...} after the statement `call`(f_j^1) = `up`(); we appropriately change their opaque predicates; see, Figure 5.11.

(S.2) *Adding multiple water-calls*: Since water-calls do not affect the functionality of the program, we can add multiple water-calls in the body of the function f_i = `up`(). Our aim is to increase the complexity of the source code making thus difficult for an attacker to understand it, the more the complexity the more the extend of the code.

Two Stealthy-case Implementations

```
public void up{
  x = x + 1;
  if(x == 268&&down == true){
        x = x + 3;
        down();
        if(x == 272){
            x = x + 2;
            health();
        }
  }
  else{
    if(b[cx + 1][cy − 1 − 1]...
        &&x == 268){
            hlth − −;
    }
    b[cx + 1][cy].bgr(black);
    ⋮
```

```
public void up{
  x = x + 1;
  if(x == 268&&down == true){
        x = x + 3;
        down();
  }
  else{
    if(b[cx + 1][cy − 1 − 1]...
        &&x == 268){
            hlth − −;
            x = x + 2;
            health();
    }
    b[cx + 1][cy].bgr(black);
    ⋮
```

Figure 5.11: The function `up`() of the original program `Laser` watermarked with a stealthy approach; the functions `down`() and `health`() are both water functions and belong to category $\mathcal{B}_{callee}$, i.e., both are functions of $G(\texttt{Laser}, I_{key})$.

(S.3) *Removing control statements*: We can remove the control statement that includes the statement $x = x + c()$ of a function $f_i = \texttt{up}()$ (Phase III); we can do that in the case where f_i is called by a function of category $\mathcal{C}_{callee}$; note that, functions of category $\mathcal{C}_{callee}$ do not modify the value of the cf-variable x.

(S.4) *Constructing complex opaque predicates*: We can construct more complex opaque predicates thus making the control flow of a program more difficult for an attacker to analyze it. In Phase I, we added opaque predicates of the form ($\texttt{x} == \texttt{value}_1$ || $\texttt{x} == \texttt{value}_2$ || ... || $\texttt{x} == \texttt{value}_\texttt{m}$), whereas in the stealthy case we evaluate the cf-variable in a range of values ($\texttt{x} <= \texttt{value}_\texttt{i}$ && $\texttt{x} >= \texttt{value}_\texttt{j}$) by adding logical and relational operators.

(S.5) *Merging control statements*: We can merge control statements that we added in program P^* with program's original control statements by appropriately merging their corresponding logical expressions.

(S.6) *Assigning complex expressions*: In the naive case the incremental functions of statements $x = x + h()$ and $x = x + g()$ have constant values $h() = 3$ and $g() = 2$, respectively. We can easily use any complex function for $h()$ and $g()$ in order to systematically increase the cf-variable x.

(S.7) *Using more cf-variables*: We can use more that one cf-variable to control the flow of the watermarked program P^*. We built relationships between the cf-variables in order to be used interchangeably throughout the execution phase. We establish thresholds that determine the use of different cf-variables.

In Figure 5.11 we present two stealthy-case implementations of the function `up`() of the original program `Laser` (left and right code). The functions `down`() and `health`() are both water functions and belong to category $\mathcal{B}_{callee}$, that is, both are functions of $G(\texttt{Laser}, I_{key})$.

5.5 Model Evaluation

Having designed a static or dynamic software watermarking model, it is very important to evaluate it under various criteria in order to gain information about its practical behavior. Several criteria have been appeared in the literature and used for evaluating the properties of a proposed watermarking model and showing its strong and weak implementation points [31]. It is a common belief that a good watermarking model must have at least the following characteristics [30]:

- a software watermarking model should not adversely affect the size and execution time of the program P;
- the ratio of the number of bits of the whole program P to the number of bits encoded by the watermark w should be high;
- a model must be resilient against a reasonable set of malicious watermarking attacks;
- both host program P and watermarked program P^* should have similar statistical properties.

In our work, we use various criteria which mainly aim to evaluate our WaterRPG model's performance and its resiliency. More precisely, we propose a set of evaluation criteria consisting of two main categories:

(I) Performance Criteria

(II) Resilience Criteria

The Performance criteria (or, P-criteria) concern the behavior of the resulting watermarked program P^* and the quality and effectiveness of the embedded watermark w, while the Resilience criteria (or, R-criteria) concern the robustness and resistance of the embedded watermark w against malicious user attacks.

For our evaluation process, we implemented the WaterRPG model on Java application programs and experimentally evaluated it under several P-criteria and R-criteria. More precisely, we selected a number of Java application programs downloaded from the free non commercial game database website `www.java-gaming.org` and watermarked them using the two watermarking approaches supported by our WaterRPG model, i.e., the Naive approach, and the Stealthy approach. The selected Java programs are almost of the same size and are watermarked by embedding watermarks of three different sizes; we use watermarking graphs $F[\pi^*]$ having number of nodes $n = 11$, $n = 13$, and $n = 15$.

All the experiments were performed on a computer with dual-core 2.0 GHZ processors, 3.0 GB of main memory under Linux operating system using Java version 1.6.0.26 of the SDK (Software Development Kit).

5.5.1 Performance

The performance criteria (or, P-criteria) mostly focus only on how much a watermarking model modifies the code of a program P. As these criteria range in satisfactory levels, both programs P and P^* have almost identical execution behavior and similar codes, and thus the code associated with the watermark w is very likely to pass unnoticed by the attacker's eyes; in our classification, the P-criteria are divided into the following two main categories:

- **Data-rate**: The data-rate criterion measures the ratio $|w|/|P|$, where $|w|$ is the size of the embedded watermark w and $|P|$ is the size of the original program P. A model should have a high data-rate so that it can embed a large message.
- **Embedding overhead**: The additional execution time and space caused by embedding the watermark w into program P, that is,

  (i) time overhead, and

  (ii) space overhead: (ii.a) disk space usage, and (ii.b) heap space usage.
- **Part protection**: The part protection criterion evaluates how well the watermark is distributed or spread throughout the entire code of P. This is an important performance property of program P^* because it decreases the probability that the watermark will be altered or destroyed when small changes are made to program P^*.
- **Credibility**: The credibility criterion evaluates how much detectable the watermark is. The embedded watermark should be easily extracted from P^* and the detector (i.e., the extracting algorithm) should minimize the probability to generate false positives and false negative results.
- **Stealth**: A watermarked program P^* has the stealth property if the embedded watermark should exhibit the same properties as the code of P or data around it and thus it should be difficult to detect. In other words, P^* should have characteristics that are not different from a typical program so that an attacker can not use these characteristics to locate and attack the watermark.

Let us now discuss on the performance of the WaterRPG model and let us first focus on the data-rate of our model.

Data-rate. This criterion essentially depends on the size of the watermark w or, in our model, of the size of the embedding watermark graph $F[\pi^*]$. We consider that the size of the watermark graph is the number of vertices that it contains. In order to measure the data-rate ratio $|w|/|P|$, we compute the size of the original program P by counting the number of functions it has, since in our model we assign an exact pairing of the nodes of $F[\pi^*]$ to the functions of P. We claim that our model has high data rate for large programs since in such programs we are able to

encode a watermark graph less than or equal to programs' size. According to our model for encoding a number as reducible permutation graph a relatively large graph encodes a large set of different integer numbers.

Embedding overhead. In order to evaluate the embedding overhead of our WaterRPG model we choose the parameters (i) execution time, (ii.a) disk usage, and (ii.b) heap space usage. We measure these parameters on the selected Java application programs P and the corresponding watermarked programs P^* under both the naive and stealthy approaches. In the evaluation process, each program is executed "n" times with different inputs. The run-time of each tested program is computed by taking the difference of the start-value and the end-value of the Java method `System.currentTimeMillis()`.

The execution time overhead is proportional to the size of the watermarking graph $F[\pi^*]$. The experimental results in Table 5.2 indicate that for a graph $F[\pi^*]$ on $n = 11$, $n = 13$ and $n = 15$ nodes the execution time of the naive watermarking causes a slight increase of 5.25%, 7.65% and 11.07%, respectively, while the corresponding increments for the stealthy case are even smaller.

The disk storage requirements of program P^* compared to P increases as the number of nodes of the graph $F[\pi^*]$ increases. Applying the stealthy approach a noteworthy amount of storage memory is saved because many of the control statements and opaque predicates that were not necessary to maintain proper functionality of the program P^* removed safely from the code. Table 5.3 illustrates the percentage increment of disk demand for P_N^* and P_S^*, as well as the improvement caused by the stealth approach in comparison to the naive. The experimental results show that our WareRpg watermarking model has a similar performance for the heap space usage; see, Table 5.4. The results for all the evaluating parameters are also depicted in a graphical form in Figure 5.12.

Towards the evaluation of the space overhead of our watermarking method we compute the total amount of the bytecode instructions added to watermarked program P^*. In particular, we compute the percentage of the increment resulted by adding control statements, functions calls and variable assignments to the program P. To this end, we count the bytecode instructions of watermarked programs P_N^* and P_S^* that belong to four main categories: (i) Control statements, (ii) Invocations, (iii) Assignments, and (iv) Rest instructions; see, Table 5.5. Note that the category (iv) contains all the bytecode instructions that remain unchanged after the watermarking process. Indicatively, in Table 5.6 we show the number of some bytecode instructions of the tested Java application programs P.

Part protection. The idea behind the property of part protection is to split the watermark into pieces and then broadly spread it across the application program P. The splitted watermark w has a better chance to survive if an attack modification on some w's parts does not affect the recognition process. The more the part protection is increased, the more likely a watermark remains unchanged after a possible theft or modification of a portion of the whole watermarked code.

In our case, we do not split the watermark in order to encoded it into code, and thus our model has a low part protection. However, an attack modification on a part of our watermark code may cause incorrectness of P^* unless the attacker goes through all the parts of the watermark code

Table 5.2: Execution Time (msec)

Nodes in $F[\pi^*]$	$P \rightarrow P_N^*$	$P \rightarrow P_S^*$	$P_N^* \rightarrow P_S^*$
11	+5.25%	+3.82%	-1.37%
13	+7.65%	+5.99%	-1.56%
15	+11.07%	+9.19%	-1.72%

Table 5.3: Disk Usage (Kb)

Nodes in $F[\pi^*]$	$P \rightarrow P_N^*$	$P \rightarrow P_S^*$	$P_N^* \rightarrow P_S^*$
11	+20.98%	+16.71%	-3.65%
13	+26.35%	+18.81%	-6.34%
15	+30.10%	+21.76%	-6.85%

Table 5.4: Heap Space Usage (Mb)

Nodes in $F[\pi^*]$	$P \rightarrow P_N^*$	$P \rightarrow P_S^*$	$P_N^* \rightarrow P_S^*$
11	+7.69%	+4.61%	-2.94%
13	+10.76%	+6.15%	-4.34%
15	+15.38%	+9.23%	-5.63%

and makes appropriate modifications. On the other hand, since we can encode the same number w into more than one reducible permutation graphs $F[\pi^*]$ (see, [24]), our model could obtain higher part protection by encoding multiple water-graphs $F[\pi^*]$ using different input sequences which produce different dynamic call graphs protecting thus larger code area.

Credibility. The credibility of a watermarking model is dependent on how detectable the watermark is. Concerning our model, we should point out that the rates of false positive and false negative outcomes are noticeable low in the case where the program P^* has not being attacked. In addition, even the watermarked code undergoes attacks, the execution sequence S of the functions of P^* is very hardly distorted. Nevertheless, an attacker could make such a distortion possible but then he has to replace the sequence of function calls S with a sequence S' in order to produce exactly the same output keeping the functionality and correctness of the remaining code; it is very time consuming for an attacker to find an S' and test it under various inputs.

Stealth. Some common attacks against watermarking systems begin by identifying the code composing the watermark. To resist such attacks, watermarking should be stealthy: the watermark code embedded to a program P should be locally indistinguishable from the rest code of P so that it is hidden from malicious users.

The code embedded to program P by our watermarking model WaterRPG is not highly unusual since our model modifies the existing source code of P by only altering its control flow in order to produce, during the execution of P^* with the secret input I_{key}, a dynamic call graph

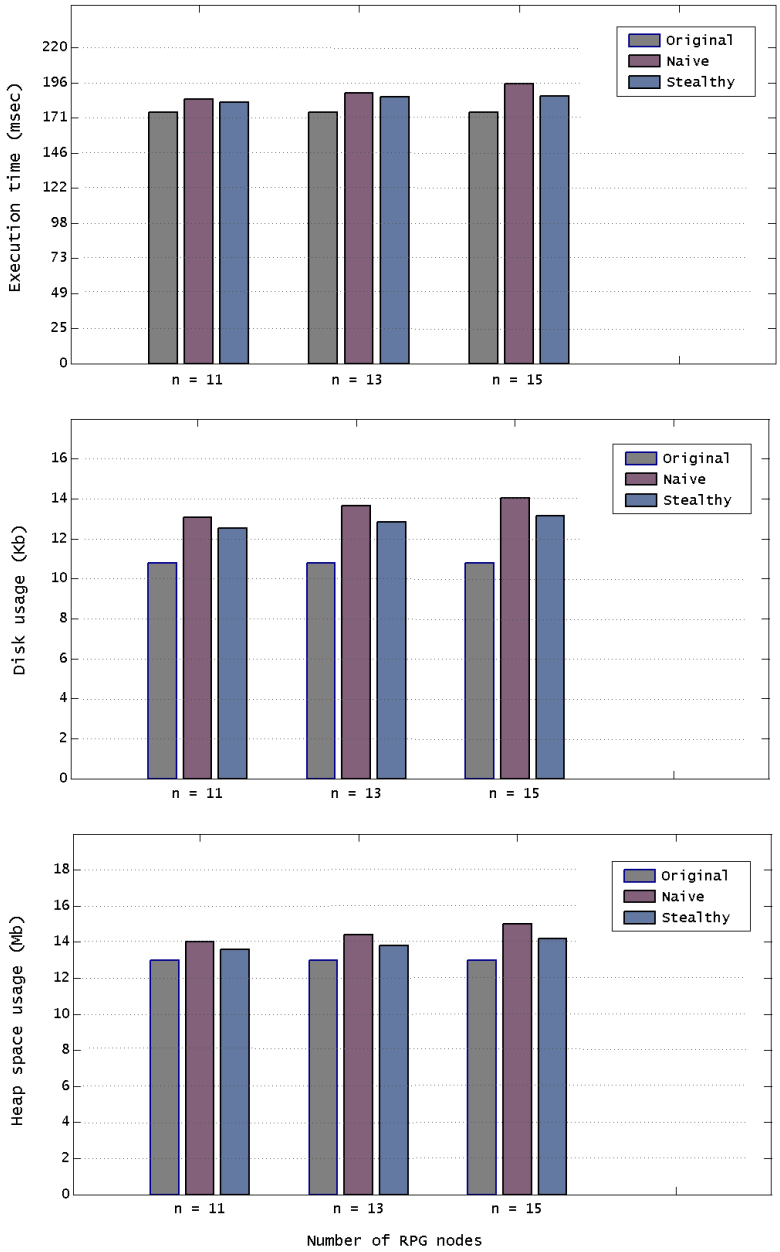


Figure 5.12: Graphical representation of the results for parameters (i) Execution time, (ii.a) Disk usage and (ii.b) Heap space usage of the original program P, and the corresponding watermarked program under both the Naive P_N^* and Stealthy P_S^* approaches.

isomorphic to the watermark graph $F[\pi^*]$. More precisely, our model does not add any dead or dummy code but only encodes the graph $F[\pi^*]$ using three groups of bytecode instructions:

(i) call functions,

(ii) control statements, and

(iii) variable assignments.

Most of these instructions are already used in the original source code P and thus the embedded watermark code is quite difficult to be located in the watermarked program P^*. The experimental

Table 5.5: Three Group of Bytecode Instructions

Bytecode	P	P_N^*	P_S^*
Control Statements	519.4	42.0%	25.0%
Invocations	188.3	10.6%	10.6%
Assignments	1346.7	45.5%	32.4%
Rest Instructions	941.6	0%	0%

Table 5.6: Indicative Bytecode Instructions of each Group

Bytecode	P	P_N^*	P_S^*
Control Statements			
`if_icmpne`	19,2	78,1	57,6
`ifne`	3,1	4,5	4,5
`goto`	43,5	45,3	45,3
Invocations			
`invokevirtual`	188,2	208,4	208,4
Assignments			
`iconst_1`	186,2	202,7	202,7
`getstatic`	368,6	614,4	529,5
`iadd`	84,9	132,8	132,8
`alaod_0`	136,7	156,2	156,2
Rest Instructions			
`dup`	33,6	33,6	33,6
`ldc`	19,7	19,7	19,7

results indicate that there is an increment from 10.6% to 45.5% (resp. from 10.6% to 32.4%) of instructions of these three groups in the naive (resp. stealthy) implementation. Table 5.5 shows the number of bytecode instructions, on average, of each of the three instruction groups of the tested programs P and the increments of these instructions in both naive P_N^* and stealthy P_S^* implementation cases, while Table 5.6 depicts the number of some indicative bytecode instructions of the three instruction groups.

5.5.2 Resilience

The resilience criteria (or, R-criteria) mainly focus on how the embedded watermark resists against attacks made by malicious users. These attacks are either targeted-attacks on the code composing the watermark w or widespread-attacks on the whole code of program P^*. According to the type of attacks, the R-criteria are divided into the following two main categories:

- **Watermark resilience**: The watermark resilience (or, water-resilience) criteria measure the resistance of the watermark w against attacks on its own code; we call these attacks targeted-attacks or water-attacks. In this case the attacker first detects the watermark w,

that is, the code of program P^* associated with the watermark w, and then makes specific operations on that code in order to

- remove (e.g., by subtracting part of the watermark, or even the whole watermark),
- destroy (e.g., by applying semantics preserving transformations so that w can be undetectable, i.e., the detector can not find the watermark), or even
- alter the watermark w (e.g., by changing the structure of the embedded watermark w so that it causes the extracting algorithm to produce a different watermark w').

The attacker can also add a new watermark w' into P^* without modifying the existing w in order to confuse the detector.

- **Code resilience**: The code-resilience criteria measure the resistance of the watermark w against attacks made on the whole code of the watermarked program P^*; we call these attacks widespread-attacks or code-attacks. In the case the attacker fails to detect the code of program P^* associated with the watermark w, and thus he makes attacks in the whole code aiming in this way to maximize the distortion of possible watermarking protections; in our classification, the code-resilience criteria include:
  - obfuscation (e.g., by transferring a reducible flow graph to non-reducible),
  - optimization (e.g., by removing information for debugging with an automated tool, such as ProGuard),
  - de-compilation (e.g., by using a malicious tool, such as Java-Decompiler), and
  - language-transformation (e.g., by converting a watermarked program P^* from C++ to Java).

As in the case performance criteria, we next discuss on the resilience of our WaterRPG model focusing first on the water-resilience criteria.

Watermark resilience. The water-attacks take place when the code of the watermark w, the graph $F[\pi^*]$ in our model, is known to the attacker. In this case, he makes attacks on the structure of watermark graph $F[\pi^*]$ in order to destroy it or even to remove $F[\pi^*]$ from program P^*.

Our model embeds the watermark graph $F[\pi^*]$ into an application program P by using opaque predicates in specific control statements in order to manipulate the flow of selected function calls of the watermarked program P^* so that it reserves an appropriate execution, that is, $O(P, I) = O(P^*, I)$ for every input I. In general, it is hard for an attacker to deduce an opaque predicate at run time. Specifically, the usage of opaque predicates in our model enables us to dictate the execution flow of function calls and also makes the programs' control flow difficult for an attacker to analyze it either statically or dynamically. In fact, our model creates dependencies on the data between the original program P and the watermark $F[\pi^*]$ making thus the watermark graph $F[\pi^*]$ operational part of the program P. This causes our model resilient to water-attacks.

Indeed, if the attacker makes a modification in a value of a cf-variable x in a call-site p (e.g., he increases x by a constant c), then he has to properly modify all the values of all the cf-variables

in every call-site of the execution flow after p (e.g., by adding the same constant c) so that P^* still remains functional, but then the watermark $F[\pi^*]$ also remains unchanged. Moreover, if the attacker tries to remove any of the *cf-statements* then the program P^* is not longer functional since a "gap" is occurred on the execution flow of P^* (e.g., the program executes a part of a code which in same cases produces incorrect results). What is more, every *cf-statement* produces different results, so our model withstands common subexpression elimination.

Additionally, if the attacker modifies either a real-call or a water-call then the program P^* is no longer operational. Indeed, let the attacker modifies a water-call `call`(f_j) in function f_i which appears in a path (f_a, ..., f_k, f_i, f_j, ..., f_b). Then, he has to make several modifications on function calls, among which the deletion of `call`(f_j) from function f_i and the replacement of `call`(f_i) with `call`(f_j) in function f_k, in order to remain the program P^* functional. Such modification have very low probability to destroy the structure of the embedded watermark $F[\pi^*]$ without breakdown the functionality of program P^*.

It is worth noting that any code modification which destroys the structure of $F[\pi^*]$ can be detected with high probability by our model WaterRPG due to error-correcting properties of the watermark graph $F[\pi^*]$. Finally, we should point out that our model does not add any dead or dummy code in the watermarked program P^* and also it does not use any mark during the embedding process in order to locate and extract the watermark from P^*.

Code-resilience. Resiliency against code-attacks refers to the ability of a watermarking model to recognize a watermark even after the program P^* has been attacked or subjected to code transformations such as translation, optimization and obfuscation [84]. The code-attacks take place when the attacker fails to detect the code of program P^* associated with our watermark $F[\pi^*]$. In this case, the attacker broadly applies code transformation attacks in the whole code of P^* in order to reduce the ability of the model's recognizer to extract the watermark.

Roughly speaking, the goal of an obfuscation attack is to make a model's recognizer to hardly extract the watermark from P^*. Our model watermarks an application program P in such a way that it withstands several obfuscation attacks among which layout-obfuscation and data-obfuscation attacks. Indeed, the experiments showed that the water-graph $F[\pi^*]$ can be efficiently extracted even the code of program P^* has been subjected to control-obfuscation attacks such as expression reordering or loop reordering; some of our experiments were performed with an automated tool (e.g., ProGuard). It is fair to mention that our model does not properly operate on some other control obfuscation attacks such as aggregation including inline functions and outline functions; for example, if an attacker split a function or merge functions of P^* associated with the code of the watermark $F[\pi^*]$, he actually makes a targeted attack both on vertices and edges of watermark graph $F[\pi^*]$ destroying thus the structure of $F[\pi^*]$.

As far as the optimization attacks are concerned we point out that they are mainly applied by a compiler or interpreter into the executable program altering the generated traces of program P^*; note that our WaterRpg model encodes execution trace watermarks. The embedded watermark $F[\pi^*]$ withstands on such attacks since any removal of function calls, register reallocation, or information for debugging does not affect the structural properties of the embedding graph $F[\pi^*]$. Again, the experimental study showed that our WaterRPG withstands on a relatively large subset of optimization attacks including semantic-preserving transformations, shrinking, and also it withstands on the most time consuming operation namely language transformation

(i.e., the attacker rewrites the whole code of P in another language).

A watermarking model must also be resilient against a reasonable set of de-compilation attacks. Thus, in our experimental study, we also included the evaluation of our watermarking model WaterRPG against de-compilation attacks. More precisely, we tested our programs with a revere engineering tool (e.g., Java Decompiler) and gure out that in all the cases that WaterRPG successfully extracts the watermarking graph $F[\ ]$ from the watermarked programs P_N and P_S; indeed, in all the cases the dynamic call-graph $G(P\ ; I_{key})$ taken by the input I_{key} were isomorphic to graph $F[\ ]$.

5.6 Concluding Remarks

Through the evaluation of WaterRPG, we showed that our model has zero false positive and false negative rates in the case where the watermarked code has not been attacked. Indeed, it is true because the execution of the watermarked program P with the secret input sequence always builds a call graph $G(P\ ; I_{key})$ which is isomorphic with the water-graph $F[\ ]$.

The execution time and space overhead varies depending on the size of the embedded watermark; in fact, the overhead increases linearly in the size of the water-graph $F[\ ]$. It is worth noting that the data-rate is directly correlated with the number of functions used or, equivalently, with the size of the water-graph. In the case where the code (in bits) of the original program P is large enough, our model has high data-rate and extremely low embedding overhead. We point out that the number of nodes of the water-graph $F[\ ]$ a ects the number of functions we use for embedding. Thus, it is possible to use fewer functions which would result in a graph $F[\ ]$ with fewer nodes; note that, the graph $F[\ ]$ on $n = 2k + 1$ nodes can encode a watermarking integer w in the range $[0; 2^{k\ 1}\ \ 1]$; see, authors' work [23, 28].

Furthermore, in our model the code which is associated with the watermark is composed both by new code and host code; this enable us to obtain high stealth watermarked programs P . Moreover, since the watermark code has become an indispensable piece of the functionality of program P , a malicious user would need to fully understand the operations of P in order to intervene changing possible execution ows. On the other hand, the extraction of our watermark takes into account and uses the traces of all the functions that are assigned to the nodes of the water-graph $F[\ ]$ which, in turn, means that if a subset of these functions is intercepted then the watermark can not be extracted; unfortunately, this implies a poor part protection of our watermarked program P .

Finally, the experimental results show the high functionality of all the Java programs P watermarked under both the naive and stealthy cases, and also their low time complexity. The experiments also show that the watermarking approaches supported by our model can help develop e cient watermarked Java programs with respect to various and broadly used performance and resilience watermarking criteria.

Closing, we note that in light of our dynamic watermarking model WaterRPG it would be very interesting to compare it with other dynamic, or even static, already proposed software watermarking models [31, 35, 86, 95, 106]; we leave it as a direction for future work.

Recently, a significant number of watermarking techniques have been proposed in the literature in order to create robust and imperceptible audio watermarks. Initial research on audio watermarking dates back to the mid-nineties where Bender et al. [8] presented data hiding techniques for audio signals; the first techniques were directly inspired from previous research on image watermarking. A broad range of audio watermarking techniques goes from simple least significant bit (LSB) scheme to the various spread spectrum methods and can be classified according to the domain where the watermarking takes place in frequency, time, and compressed domain [3, 43, 61, 108].

Attacks. An efficient watermarking technique should resist to any process specifically intended to thwart the watermark's purpose, i.e., to protect the intellectual property of the digital object. Generally digital image and audio watermarking has certain requirements, the most important being invisibility and robustness. Invisibility is the state of a watermark that cannot be seen, while robustness means that the embedded watermark cannot be removed by an attack, that is, an intentional or unintentional digital data change.

In general, attacks against embedded data (i.e., image and audio watermarks) can included various combinations of several image processing techniques, or other techniques for overwriting or removing the embedded information. Attacks can be classified into the following four different types [110]:

- removal and interference attacks,
- geometric attacks,
- cryptographic attacks, and
- protocol attacks.

Lossy compression, quantization, collusion, denoising, remodulation, averaging, and noise storm are some examples of removal and interference attacks. Geometric attacks do not actually remove the watermark, but manipulate the watermarked object in such a way that the detector cannot find the watermark data. This type of attack includes affine transformations such as rotation, translation, and scaling. The aim of cryptographic attacks is to crack the security methods in watermarking schemes and thus find a way to remove the embedded watermark or to embed misleading watermarks. Protocol attacks add the attacker's own watermark signal introducing thus ambiguities on the true ownership of data.

Contribution. In this chapter, we present efficient and easily implemented techniques for watermarking images and audio signals that we are interested in uploading in the web and making them public online; this way web users are enabled to claim the ownership of their images and audio signals.

We first present our primary work on image watermarking in the spatial domain and we next expand our idea by moving from the spatial domain to the image's frequency domain. What is important for our idea is the fact that it suggests a way in which an integer number can be represented with a two dimensional representation (or, for short, 2D-representation). Thus, since images are two dimensional objects that representation can be efficiently marked on them resulting the watermarked images. In a similar way, such a 2D-representation can be extracted for a watermarked image and converted back to the integer w.

Having designed an efficient method for encoding integers as self-inverting permutations, we propose an efficient algorithm for encoding a self-inverting permutation π^* into an image I by first mapping the elements of π^* into an $n^* \times n^*$ matrix A^* and then using the information stored in A^* to mark specific areas of image I in the frequency domain resulting the watermarked image I_w. We also propose an efficient algorithm for extracting the embedded self-inverting permutation π^* from the watermarked image I_w by locating the positions of the marks in I_w; it enables us to reconstruct the 2D-representation of the self-inverting permutation π^*.

It is worth noting that although digital watermarking has made considerable progress and became a popular technique for copyright protection of multimedia information [15], our work proposes something new. We first point out that our watermarking method incorporates such properties which allow us to successfully extract the watermark w from the image I_w even if the input image has been compressed with a lossy method. In addition, our embedding method can transform a watermark from a numerical form into a two dimensional (2D) representation and, since images are 2D structures, it can efficiently embed the 2D-representation of the watermark by marking the high frequency bands of specific areas of an image. The key idea behind our extracting method is that it does not actually extract the embedded information instead it locates the marked areas reconstructing the watermark's numerical value.

We have evaluated the embedding and extracting algorithms by testing them on various and different in characteristics images that were initially in JPEG format and we had positive results as the watermark was successfully extracted even if the image was converted back into JPEG format with various compression ratios. What is more, the method is open to extensions as the same method might be used with a different marking procedure such as the one we used in our previous work. Note that, all the algorithms have been developed and tested in MATLAB [54].

Based on the idea behind our image watermarking technique, where the integer watermark number w is represented as a two dimensional array, we present an audio signal watermarking technique. Since audio is one dimensional signal, the 2D-representation can be efficiently reconstructed in the 1D space and be used to mark an audio signal.

What is important of the proposed audio watermarking technique is the fact that it suggests a way in which an integer number w can be represented as an one-dimensional array (or, equivalently, 1D-representation). More precisely, our proposed algorithm embeds a self-inverting permutation π^* over the set N_n into an audio signal S by first mapping the elements of π^* into an $n \times n$ matrix B^* and then, based on the information stored in B^*, marking specific areas of audio S in the frequency domain resulting thus the watermarked audio S_w. An efficient algorithm extracts the embedded self-inverting permutation π^* from the watermarked audio S_w by locating the positions of the marks in S_w; it enables us to reconstruct the 1D-representation of π^* and, then, obtain the watermark w.

Finally, we would like to point out that the primary purpose of our approach is not to fill a gap of the existing audio watermarking methods by proposing a new embedding technique, but to expand the idea used on our previous work and show that it can be efficiently applied for audio watermarking depicting thus the high versatility of the whole concept.

Road Map. The chapter is organized as follows: In Section 6.2 we establish the notation and related terminology, and we present background results. In Section 6.3 we describe our primary work on image watermarking in the spatial domain and give the `Embed-SiP.to.Image-S`

and `Extract-SiP.from.Image-S` algorithms. In Section 6.4 we present our codec algorithms, `Embed-SiP.to.Image-F` and `Extract-SiP.from.Image-F`, for watermarking images in the frequency domain. In Section 6.5 we expand our idea on image watermarking by applying it in audio watermarking and present our audio watermarking algorithms, i.e., the embedding algorithm `Embed-SiP.to.Audio` and the extracting algorithm `Extract-SiP.from.Audio`. Finally, in Section 6.6 we conclude the chapter and discuss possible future extensions.

6.2 Background Results

In this section we present basic tools which are used by our image and audio watermarking algorithms. Since images are two dimensional objects we present a transformation of a watermark from a numerical form to a 2D form (i.e., 2D-representation) through the exploitation of self-inverting permutation properties. The 2D-representation can be efficiently marked on images resulting thus the watermarked images. Similarly, since audio signal is an one dimensional object we present a transformation of a watermark from a numerical form to a 1D form (i.e., 1D-representation).

In Chapter 2, we defined a permutation π over the set N_n as a sequence $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$ whereas a self-inverting permutation (or, involution) as a permutation that is its own inverse, i.e., $\pi_{\pi_i} = i$. By definition, all the cycles of a self-inverting permutation are of length 1 or 2.

Moreover, in the same chapter, we proposed the encoding algorithm `Encode_W.to.SiP` for encoding numbers as self-inverting permutations (or SiP, for short) along with the corresponding decoding algorithm `Decode_SiP.to.W`; recall that, the algorithms of our codec system run in $O(n)$ time, where n is the length of the binary representation of the integer w [28].

6.2.1 2D Representation of SiP

Given a permutation π over the set N_n, we first define a two-dimensional representation (2D-representation) of the permutation π that is useful for studying properties which help us to define, later, a more suitable representation of π for efficient use in our watermarking system.

In this representation, the elements of the permutation $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$ are mapped in specific cells of an $n \times n$ matrix A as follows:

- number $\pi_i \longrightarrow$ entry $A(\pi_i^{-1}, \pi_i)$

or, equivalently,

- the cell at row i and column π_i is labeled by the number π_i, for each $i = 1, 2, \ldots, n$.

Figure 6.1(a) depicts the 2D-representation of the permutation $\pi = (5, 6, 9, 8, 1, 2, 7, 4, 3)$ over the set N_9; the permutation π is self-inverting.

By definition, there is one label in each row and in each column, so each cell in the matrix A corresponds to a unique pair of labels; see, [107] for a long bibliography on permutation representations and also in [23] for a DAG representation.

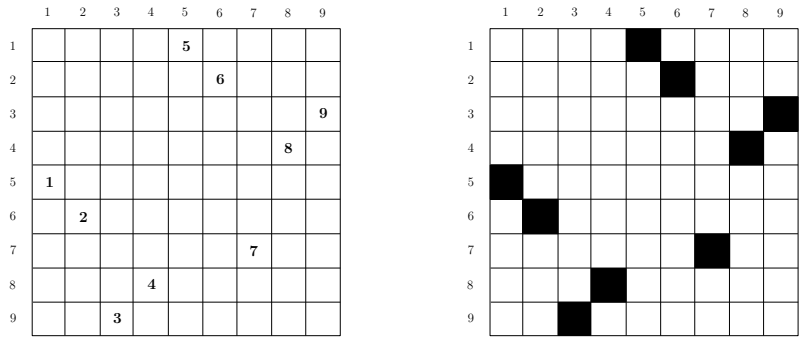


Figure 6.1: (a) A 2D representation of the self-inverting permutation $\pi = (5, 6, 9, 8, 1, 2, 7, 4, 3)$; (b) A 2DM representation of permutation π.

Based on the previous 2D-representation of a permutation, we next propose a two-dimensional marked representation (2DM-representation) of a permutation which is an efficient tool for watermarking images.

In our 2DM-representation, a permutation π over the set N_n is represented by an $n \times n$ matrix A^* as follows:

- the cell at row i and column π_i is marked by a specific symbol, for each $i = 1, 2, \ldots, n$.

Figure 6.1(b) shows the 2DM-representation of the permutation π. Note that, as in the 2D-representation, there is also one symbol in each row and in each column of the matrix A^*.

We next present an algorithm which extracts the permutation π from its 2D-representation matrix. More precisely, let π be a permutation over N_n and let A^* be the 2DM-representation matrix of π (see, Figure 6.1); given the matrix A^*, we can easily extract π from A^* in linear time (in the size of matrix A^*) by the following algorithm:

Algorithm `Extract_π_from_2DM`

1. For each row i of matrix A^*, $1 \leq i \leq n$, do:
find the marked cell and let j be its column;
set $\pi_i \leftarrow j$;

2. Return the permutation π.

Remark 6.1. It is easy to see that the resulting permutation π, after the execution of Step 1, can be taken by reading the matrix A^* from top row to bottom row and write down the positions of its marked cells. Since the permutation π is a self-inverting permutation, its 2D matrix A has the following property:

- $A(i, j) = j$ if $\pi_i = j$, and
- $A(i, j) = 0$ otherwise, $1 \leq i, j \leq n$.

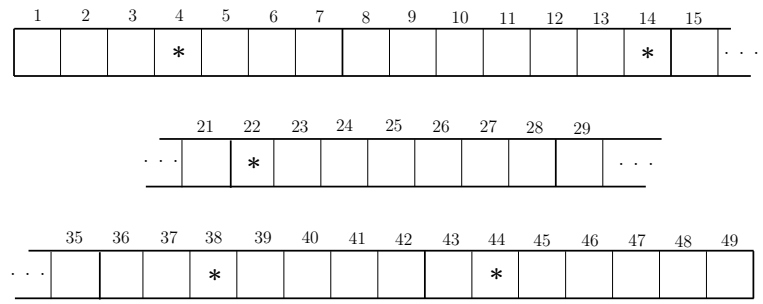


Figure 6.2: The 1DM representations of the self-inverting permutation $\pi = (4, 7, 6, 1, 5, 3, 2)$.

Thus, the corresponding matrix A^* is symmetric:

- $A^*(i, j) = A^*(j, i) =$ "mark" if $\pi_i = j$, and
- $A^*(i, j) = A^*(j, i) = 0$ otherwise, $1 \leq i, j \leq n$.

Based on this property, it is also easy to see that the resulting permutation π can be also taken by reading the matrix A^* from left column to right column and write down the positions of its marked cells.

6.2.2 1D Representation of SiP

We next present an one-dimensional representation (1D-representation) of the permutation π, over the set N_n, which we will use as a mark to embed it into audio signal.

In our 1D-representation, the elements of the permutation π are mapped in specific cells of an array B of size n^2 as follows:

$$\text{number} \quad \pi_i \quad \longrightarrow \quad \text{entry} \quad B((\pi_{\pi_i}^{-1} - 1)n + \pi_i)$$

or, equivalently, the cell at the position $(i-1)n + \pi_i$ is labeled by the number π_i, for each $i = 1, 2, \ldots, n$.

We next describe the 1DM-representation acquired in a similar manner as the 2DM-representation. In our 1DM-representation, a permutation π over the set N_n is represented by an n^2 array B^* as follows:

- the cell at the position $(i-1)n + \pi_i$ is marked by a specific symbol, for each $i = 1, 2, \ldots, n$;

where, in our implementation, the used symbol is again the asterisk character "*". Figure 6.2 shows the 1DM-representation of the same permutation $\pi = (4, 7, 6, 1, 5, 3, 2)$.

Hereafter, we shall denote by π^* a self-inverting permutation and by n^* the number of elements of π^*.

6.2.3 Color Images

A digital image is a numeric representation of a 2-dimensional image; it has a finite set of values, called *picture elements* or *pixels*, that represent the brightness of a given color at any specific point in the image [53].

A digital image contains a fixed number of rows and columns of pixels which are usually stored in computer memory as a two-dimensional matrix I of numeric values; in our system the numeric values are integers from 0 to 255. When we say that an image has a *resolution* of $N \times M$ we mean that its two-dimensional matrix I contains N rows and M columns and the value of each entry $I(i, j)$, i.e., the value of each pixel, is an integer k_0 (grayscale image), or a triple of integers (k_1, k_2, k_3) (color image), $0 \leq k_0, k_1, k_2, k_3 \leq 255$.

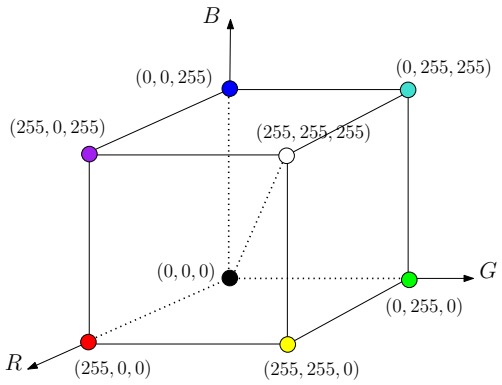


Figure 6.3: The range of colors represented on the Cartesian 3-dimensional system.

There are several models used for representing color. In our system, we use the RGB model; it is an additive color model in which *red*, *green*, and *blue* light is added together in various ways to reproduce a broad array of colors. The name of the model comes from the initials of the three additive primary colors, Red, Green, and Blue [53, 97].

The range of colors can be represented on the Cartesian 3-dimensional system as a cube with the following characteristics:

- on the x-axis (R-axis) we have the brightness of the **red** color,
- on the y-axis (G-axis) we have the brightness of the **green** color, and
- on the z-axis (B-axis) we have the brightness of the **blue** color.

Figure 6.3 shows the 3D topology of the colors. For example, the white color (255, 255, 255) is located in the front upper right point of the color cube.

In our system, since a color is a triple of integers (x, y, z), a digital image I of resolution $N \times M$ (i.e., it contains N rows and M columns) is stored in a three-dimensional matrix Img of size $N \times M \times 3$ as follows:

- if the pixel $I(i, j)$ of the image I has (x, y, z) color, then $Img(i, j, 1) = x$, $Img(i, j, 2) = y$, and $Img(i, j, 3) = z$.

For example, let (240, 29, 35) be the color of the upper left pixel of an image I, i.e., $I(1, 1) = (240, 29, 35)$. Then, in our system $Img(1, 1, 1) = 240$, $Img(1, 1, 2) = 29$, and $Img(1, 1, 3) = 35$.

6.2.4 The Discrete Fourier Transform

The Discrete Fourier Transform (DFT) is used to decompose an image into its sine and cosine components. The output of the transformation represents the image in the frequency domain, while the input image is the spatial domain equivalent. In the image's fourier representation, each point represents a particular frequency contained in the image's spatial domain.

If $f(x, y)$ is an image of size $N \times M$ we use the following formula for the Discrete Fourier Transform:

$$F(u, v) = \sum_{x=0}^{N-1} \sum_{y=0}^{M-1} f(x, y) e^{-j2\pi(\frac{ux}{N} + \frac{vy}{M})} \tag{6.1}$$

for $u = 0, 1, \ldots, N-1$ and $v = 0, 1, \ldots, M-1$.

In a similar manner, if we have the transform $F(u, v)$ i.e the image's fourier representation we can use the Inverse Fourier Transform to get back the image $f(x, y)$ using the following formula:

$$f(x, y) = \frac{1}{NM} \sum_{u=0}^{N-1} \sum_{v=0}^{M-1} F(u, v) e^{j2\pi(\frac{ux}{N} + \frac{vy}{M})} \tag{6.2}$$

for $x = 0, 1, \ldots, N-1$ and $y = 0, 1, \ldots, M-1$.

Typically, in our method, we are interested in the magnitudes of DFT coefficients. The magnitude $|F(u, v)|$ of the Fourier transform at a point is how much frequency content there is and is calculated by Equation (6.1) [53].

6.3 Image Watermarking in the Spatial Domain

Having proposed an efficient method for encoding integers as self-inverting permutations using the algorithm `Embed_W.to.SiP` (see, Section 2), and the 2DM-representation of self-inverting permutations (see, Section 6.2), we next describe the two main algorithms of our image watermarking system; the encoding algorithm `Embed_SiP.to.Image-S` which encodes a self-inverting permutation π^*, corresponding to watermark w, into an image I resulting the watermarked image I_w and the decoding algorithm `Extract_SiP.from.Image-S` which extracts the permutation π^* from the image I_w.

6.3.1 Embed Watermark into Image - S

We next describe the algorithm `Embed_SiP.to.Image-S` of our codec system which embeds a self-inverting permutation π^* into an image I [25]; recall that, in our codec system we encode the watermark w as a self-inverting permutation π^* over the set N_{n^*}, where $n^* = 2n + 1$ and n is the length of the binary representation of the integer w [28]; see, Subsection 2.4.1.

The algorithm takes as input a SIP π^* and an image I, in which the user wants to embed the watermark $w = \pi^*$, and produces the watermarked image I_w; it works as follows:

Algorithm `Embed_SiP.to.Image-S`

1. The algorithm first computes the 2DM-representation of the permutation π^*, that is, it computes the $n^* \times n^*$ array A (see, Subsection 6.2.1); the entry (i, π_i^*) of the array A contains the symbol "*", $1 \leq i \leq n^*$;

2. Next, the algorithm computes the size $N \times M$ of the input image I and do the following: if N is an even number it removes the pixels from the bottom row of I and reduces N by 1, while if M is an even number it removes the pixels from the right column of I and reduces M by 1. The resulting image has size $N^* \times M^*$, where N^* and M^* are both odd numbers;

3. Let n^* be the size of the SiP π^* and let $N^* \leq M^*$. Now the algorithm takes the input image I and places on it an imaginary grid $\mathcal{G}$, covering almost all the image I, having

$$n^* \times n^* \text{ grid-cells } C_{ij}(I)$$

each $C_{ij}(I)$ of size

$$\lfloor N^*/n^* \rfloor \times \lfloor N^*/n^* \rfloor$$

where, $1 \leq i, j \leq n^*$.

It places the imaginary grid $\mathcal{G}$ on I as follows: it first locates the central pixel P_{cent}^0 of the image I, which is at position $(\lfloor N^*/2 \rfloor + 1, \lfloor M^*/2 \rfloor + 1)$, then locates the central pixel p_{ii}^0 of the central grid-cell $C_{ii}(I)$, where $i = \lfloor n^*/2 \rfloor + 1$, and places the grid $\mathcal{G}$ on image I such that both P_{cent}^0 and p_{ii}^0 have the same position in I;

4. Then it scans the image and goes to each grid-cell $C_{ij}(I)$ (there are always $n^* \times n^*$ grid-cells in any image) and locates the central pixel p_{ij}^0 of the grid-cell $C_{ij}(I)$ and also the four pixels p_{ij}^1, p_{ij}^2, p_{ij}^3, and p_{ij}^4 around it, $1 \leq i, j \leq n^*$; hereafter, we shall call these four pixels *cross* pixels.

Then, it computes the difference between the brightness of the central pixel p_{ij}^0 and the average brightness of the twelve pixels around it, that is, the pixels $p_{ij}^{\ell 1}$, $p_{ij}^{\ell 2}$, and $p_{ij}^{\ell 3}$ $(\ell = 1, 2, 3, 4)$, and stores this value in the variable $\text{dif}(p_{ij}^0)$ (see, Figure 6.4).

Finally, it computes the maximum absolute value of all $n^* \times n^*$ differences $\text{dif}(p_{ij}^0)$, $1 \leq i, j \leq n^*$, and stores it in the variable $\text{Maxdif}(I)$;

5. The algorithm goes again to each central pixel p_{ij}^0 of each grid-cell C_{ij} and if the corresponding entry $A(i, j)$ contains the symbol "*", then it increases

- the brightness k_{ij}^0 of the central pixel p_{ij}^0, and
- the brightness k_{ij}^1, k_{ij}^2, k_{ij}^3, and k_{ij}^4 of its cross pixels.

Actually, it first increases the central pixel p_{ij}^0 by the value e_{ij}^0 so that it surpasses the image's maximum difference $\text{Maxdif}(I)$ by a constant c; that is,

- $k_{ij}^0 + e_{ij}^0 = \text{Maxdif}(I) + c$

and, then, it sets the brightness of the four cross pixels p_{ij}^1, p_{ij}^2, p_{ij}^3, and p_{ij}^4 equal to k_{ij}^0.

In our system we use $c = 5$, and thus the brightness k_{ij}^0 of the central pixel of each grid-cell C_{ij} is increased by e_{ij}^0, where,

$$e_{ij}^0 = \text{Maxdif}(I) - k_{ij}^0 + 5 \tag{6.3}$$

where, $1 \leq i, j \leq n^*$.

Let I_w be the resulting image after increasing the brightness of the n^* central and the corresponding cross pixels, with respect to π, of the image I. Hereafter, we call the n^* central pixels of I as *2DM-pixels*; recall that, p_{ij}^0 is a 2DM-pixel if $A(i, \pi_i) =$ "*", or, equivalently, the cell (i, π_i) of the matrix A is marked;

6. The algorithm returns the watermarked image I_w.

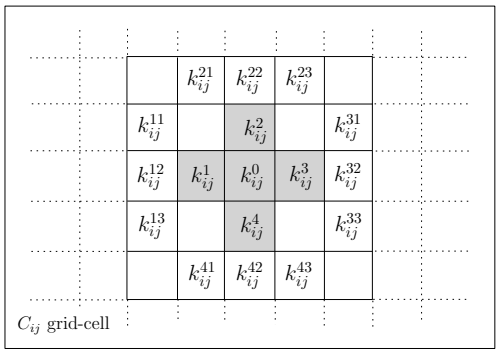


Figure 6.4: The brightness k_{ij}^{ℓ} of the central and cross pixels p_{ij}^{ℓ} of the grid-cell $C_{ij}(I)$, $0 \leq \ell \leq 4$, and the brightness $k_{ij}^{\ell m}$ of the cycle-cross pixels $p_{ij}^{\ell m}$, $1 \leq \ell \leq 4$ and $m = 1, 2, 3$.

Having described our encoding algorithm which embeds a permutation π into an image I, let us now show the efficiency of our algorithm by computing its time and space complexity.

Time and Space. We shall compute the complexity of each step of the algorithm; suppose that the input image I has $N \times M$ size (i.e., pixels).

It is easy to see that Step 1 requires $n^* \times n^*$ (asymptotic) time and space, since the length of the permutation π and the size of the array A are n^* and $n^* \times n^*$, respectively.

In Step 2 the algorithm computes the values of the two dimensions of the image I, and thus this computations takes $N + M$ time.

In Step 3 the algorithm places on I the imaginary grid $\mathcal{G}$ having $n^* \times n^*$ grid-cells $C_{ij}(I)$ each of size $\lfloor N^*/n^* \rfloor \times \lfloor N^*/n^* \rfloor$, where $1 \leq i, j \leq n^*$, and thus it covers $(\lfloor N^*/n^* \rfloor \times \lfloor N^*/n^* \rfloor) \times (n^* \times n^*)$ pixels of the image I. The location of the $n^* \times n^*$ central pixels p_{ij}^0 can be done in $n^* \times n^*$ time, where $1 \leq i, j \leq n^*$ and $n^* < N^*$. Thus, the step takes $(\lfloor N^*/n^* \rfloor n^*)^2$ time.

In Step 4 it computes the difference between the brightness of the central pixel p_{ij}^0 and the average brightness of the twelve cycle-cross pixels $p_{ij}^{\ell m}$ around it, $\ell = 1, 2, 3, 4$ and $m = 1, 2, 3$. This difference, denoted by $\text{dif}(p_{ij}^0)$, is computed as follows:

(a) (b)

Figure 6.5: (a) The original image I; (b) The watermarked image I_w.

$$\mathrm{dif}(p_{ij}^0) = \left| k_{ij}^0 - \frac{\sum_{\ell=1}^{4} \sum_{m=1}^{3} k_{ij}^{\ell m}}{12} \right| \tag{6.4}$$

where,

$$k_{ij}^{\ell m} = \frac{x_{ij}^{\ell m} + y_{ij}^{\ell m} + z_{ij}^{\ell m}}{3}. \tag{6.5}$$

Recall that, the values $x_{ij}^{\ell m}$, $y_{ij}^{\ell m}$, and $z_{ij}^{\ell m}$ compose the brightness $k_{ij}^{\ell m}$ of the pixel $p_{ij}^{\ell m}$ in the RGB model (see, Subsection 6.2.3). Thus, the $n^* \times n^*$ differences $\mathrm{dif}(p_{ij}^0)$ can be computed in $n^* \times n^*$ time and require $n^* \times n^*$ space (i.e., an array of $n^* \times n^*$ size).

Finally, in this step the algorithm computes the maximum absolute value $\mathrm{Maxdif}(I)$ of all $n^* \times n^*$ differences $\mathrm{dif}(p_{ij}^0)$, that is,

$$\mathrm{Maxdif}(I) = max\{\mathrm{dif}(p_{ij}^0) | 1 \leq i, j \leq n^*\} \tag{6.6}$$

which obviously takes also $n^* \times n^*$ time.

The only operation performed in Step 5 is the increment of the brightness k_{ij}^0 of each central pixel and the brightness k_{ij}^1, k_{ij}^2, k_{ij}^3, and k_{ij}^4 of its four cross pixels by the value e_{ij}^0 (see, Equation 6.8); it obviously takes $n^* \times n^*$ time since there are $n^* \times n^*$ such central pixels.

Based on the above step-by-step analysis of our encoding algorithm `Embed_SiP.to.Image-S` we conclude that it runs (asymptotically) in order $N \times n^*$ time and requires $n^* \times n^*$ space, where N is the smallest dimension of the input image I and n^* is the size of the SiP.

Remark 6.2. The values x_{ij}^{ℓ}, y_{ij}^{ℓ}, and z_{ij}^{ℓ} which compose the brightness k_{ij}^{ℓ} of the pixel p_{ij}^{ℓ} are stored in the array Img at the entries $(i', j', 1)$, $(i', j', 2)$, and $(i', j', 3)$, respectively (see, Subsection 6.2.3). Note that, (i', j') is the position of pixel p_{ij}^{ℓ} in image I, while (i, j) is the position of pixel p_{ij}^{ℓ} in the $n^* \times n^*$ grid.

6.3.2 Extract Watermark from Image - S

Next we describe our algorithm which is responsible for extracting the watermark $w = \pi^*$ form image I_w. In particular, the algorithm, which we call `Extract_SiP.from.Image-S`, takes as input a watermarked image I_w and returns the SiP π^* which corresponds to integer watermark w; the steps of the algorithm are the following:

Algorithm `Extract_SiP.from.Image-S`

1. The algorithm places again the same imaginary $n^* \times n^*$ grid on image I_w and locates the central pixel p^0_{ij} of each grid-cell $C_{ij}(I)$, $1 \leq i, j \leq n^*$; there are $n^* \times n^*$ central pixels in total. Then, it finds the n^* central pixels $p^0_1, p^0_2, \ldots, p^0_{n^*}$, among the $n^* \times n^*$, with the maximum brightness using a known sorting algorithm [40];

2. In this step, the algorithm takes the n^* grid-cell $C_1, C_2, \ldots, C_{n^*}$ of the image I_w which correspond to n^* central pixels $p^0_1, p^0_2, \ldots, p^0_{n^*}$, and compute an $n^* \times n^*$ matrix A^* as follows:

- Initially, set $A^*(i, j) \leftarrow 0$, $1 \leq i, j \leq n^*$;
- For each grid-cell C_m, $1 \leq m \leq n^*$, do:

  if (i, j) is the position of the grid-cell C_m in the grid $\mathcal{G}$
  then set $A^*(i, j) \leftarrow$ "$*$";

It is easy to see that, the $n^* \times n^*$ matrix A^* is exactly the 2DM-representation of the self-inverting permutation π^* embedded in image I_w by the algorithm `Embed_SiP.to.Image-S`.

Then, the permutation π^* can be extracted from the matrix A^* using the algorithm Extract_π_`from_2DM`; see, Subsection 6.2.1;

3. Finally, the algorithm returns the self-inverting permutation π^*.

Let us next compute the time and space efficiency of the proposed decoding algorithm `Extract_SiP.from.Image-S` by computing the complexity of each step separately.

Time and Space. Again, we suppose, as we did with the encoding algorithm `Embed_SiP.to.Image-S`, that the input image I_w has $N \times M$ size (i.e., it consists of $N \times M$ pixels) and $N \leq M$.

In Step 1 the algorithm places on I_w an imaginary $n^* \times n^*$ grid, as the embedding algorithm do on image I, and thus the values of the two dimensions of the image I_w must be known; this computations takes $N + M$ time. Then, the location of the $n^* \times n^*$ central pixels p^0_{ij} can be done in $n^* \times n^*$ time, $1 \leq i, j \leq n^*$, while the finding of the n^* central pixels, among the $n^* \times n^*$, with the maximum brightness can be done in $(n^* \times n^*) \times \log(n^* \times n^*)$ time; note that, it is well known that fastest sorting algorithm on an input of size n takes $n \log n$ time [40].

In Step 2 the algorithm takes the n^* grid-cell $C_1, C_2, \ldots, C_{n^*}$ of the image I_w which correspond to the n^* central pixels $p^0_1, p^0_2, \ldots, p^0_{n^*}$, and compute an $n^* \times n^*$ matrix A^*. It is easy to see that this computation can be done in $n^* \times n^*$ time. It is also easy to see that the permutation π^* can be extracted from A^*, using the algorithm Extract_π_`from_2DM`, within the same time. Thus, Step 2 requires $n^* \times n^*$ time. Obviously, Step 3 takes constant time.

Based on the above complexity analysis we conclude that the proposed decoding algorithm `Extract_SiP.from.Image-S` extracts the watermark SiP π^* from the image I_w in $N+M+(n^* \times n^*) \times \log(n^* \times n^*)$ time; it requires $n^* \times n^*$ space.

6.3.3 Performance

Our image watermarking system mainly consists of four algorithms, each of which is responsible for a particular codec operation:

- `Encode_W.to.SiP`: algorithm for encoding an integer w into a self-inverting permutation π^*;
- `Decode_SiP.to.W`: algorithm for decoding w from π^*;
- `Embed_SiP.to.Image-S`: algorithm for embedding a self-inverting permutation π^* into an image I;
- `Extract_SiP.from.Image-S`: algorithm for extracting π^* form the watermarked image I_w;

The two algorithms that are considered the basic ones for our system are those responsible for embedding a SiP into an image and extracting the SiP from it.

We next discuss some issues concerning the performance of our image watermarking system. In particular, we mainly focus on the embedding algorithm `Embed_SiP.to.Image-S` and the efficiency of watermarking image I_w produced by this algorithm, and also on important properties of the $n^* \times n^*$ matrix A^* which stores the 2DM-representation of a SiP. Finally we show the time and space performance of our system by computing the complexity of their algorithms.

It is worth noting that our system incorporates such properties which allow us to successfully extract the watermark w from the image I_w even if the input image I_w of algorithm `Extract_SiP.from.Image-S` has been compressed with a lossy method and/or rotated.

We have evaluated the embedding and extracting algorithms by testing them on various and different in characteristics images that were initially in JPEG format and we had positive results as the watermark was successfully extracted at every case even if the image was converted back into JPEG format. What is more, the method is open to extensions as the same method might be used with a different marking procedure part of the `Embed_SiP.to.Image-S` algorithm.

All the system's algorithms have been initially developed and tested in MATLAB [54] and then redeveloped and also tested in JAVA.

Compression. The experimental results show that the watermark w is "well hidden" in the image I_w. It is because we mark the image by changing the difference between the brightness of the 2DM-pixels p_{ij}^0 of the $n^* \times n^*$ imaginary grid and its 12 neighborhood pixels around it, that is, the pixels $p_{ij}^{\ell 1}$, $p_{ij}^{\ell 2}$, and $p_{ij}^{\ell 3}$, for $\ell = 1, 2, 3, 4$ (see, Figure 6.4 and also Step 2 of the embedding algorithm `Encode_SiP.to.Image-S`); recall that, we also set the brightness of the four cross pixels of each 2DM-pixel p_{ij}^0, that is, the pixels p_{ij}^1, p_{ij}^2, p_{ij}^3, and p_{ij}^4, to be equal to the brightness of the 2DM-pixel p_{ij}^0.

Note that, we change the brightness of the 2DM-pixels by increasing them so that they surpass the image's maximum difference Maxdif(I) by a constant c, where in our implementation $c = 5$. We add five because if we compress the image the values of the pixels may slightly change, and

we want our watermark to be robust. We also believe that this technique despite being simple, it is efficient because the brightness of each of the n^* marked central pixels does not have a great difference from the brightness of the 12 neighborhood pixels and thus the modified central pixel, along with the cross pixels, does not change something significantly in the image.

Rotation. The watermarked images produced by our embedding method have a property worth to be referenced. And this is certain characteristics noticed at the 2DM-representation of the image's watermarks which in our system are self-inverting permutations. Sometimes an image might show an indeterminate depiction, such as a night sky or an aerial view. These types of images might be rotated changing the coordinates of the watermark's marks making invalid the watermark that we are about to extract. Also it is about an indeterminate depiction which does not allow someone to tell which is the right angle of the image.

Thanks to our embedding method's properties this problem can be overcome. It has to do with the coordinates of the marks of a 2DM-representation of a self-inverting permutation found on image I_w. Those coordinates allow us to turn the image into the initial angle and then extract the watermark successfully.

The 2DM-representation of a self-inverting permutation (see, Section 6.2) has the following properties:

(i) The main diagonal of the $n^* \times n^*$ symmetric matrix A^* have always one and only one marked cell, and

(ii) this marked cell are always in the entries (i, i) of A^*, where $i = \lceil \frac{n^*}{2} \rceil + 1, \lceil \frac{n^*}{2} \rceil + 2, \ldots, n^*$.

If the main diagonal of matrix A^* has no marked cell then we rotate the image by 90 degrees. Additionally, if the marked cell of the main diagonal is in entry (i, i) with $i \leq \lceil \frac{n^*}{2} \rceil$, then we turn the image by 180 degrees and thus we end up at the initial image from which we are able to extract the right watermark.

Time and Space Performance. As far as the time and space performance of our codec system is concerned, we should mention that it is asymptotically linear in the size (i.e., number $N \times M$ of pixels) of the input images.

More precisely, the embedding algorithm takes $(N \times n^*) + (n^* \times n^*)$ time which is less than the size $N \times M$ of the input image I. Recall that, in our implementation: (i) $N \leq M$, and (ii) the length of the watermark is n^* and thus we always have $n^* \times n^*$ grid-cells.

The extracting algorithm is also very fast since it also operates mainly on the $n^* \times n^*$ grid-cells of the input image I_w. The most time consuming step of the algorithm is that of sorting the $n^* \times n^*$ cental pixels of the image in order to find the n^* pixels with the max brightness; it takes $n^* \times n^* \times \log(n^* \times n^*)$ time.

Finally, it is fair for the time performance of our system to take into consideration the time needed for converting the image I that the system takes as input from the initial format to raw raster format; note that, the system usually uses compressed images as input. It is obvious that the time needed for converting the image I into a *raw raster format* depends on the type of the image selected. The most common types of images would be the JPEG as digital cameras store images of this type and also nearly every image on the WWW (world wide web) is in JPEG format. The compression to a JPEG requires the usage of the DCT (discrete cosine transform);

the DCT is similar to a Fourier transform and it is of order n^2, but it is also possible to do the same thing by doing something similar to the FFT (fast fourier transform) which is of order $n \log n$. Note that the same techniques applies for the JIF images which are also popular in the web [2, 41].

Summarizing, the total time performance of our codec system, neglecting the conversion of the input image I into raw raster format, is $N \times n^*$ for embedding the watermark w into I and $N + M + (n^* \times n^*) \times \log(n^* \times n^*)$ for extracting w from the watermarked image I_w. Moreover, the extra space needed by our codec system is linear in the size of the input image, i.e., it uses only some extra auxiliary variables and an auxiliary matrix for the 2DM-representation of the self-inverting permutation.

6.4 Image Watermarking in the Frequency Domain

In this work we present an efficient and easily implemented technique for watermarking images that we are interested in uploading in the web and making them public online; this way web users are enabled to claim the ownership of their images.

What is important for our idea is the fact that it suggests a way in which an integer number can be represented with a two dimensional representation (or, for short, 2D-representation). Thus, since images are two dimensional objects that representation can be efficiently marked on them resulting the watermarked images. In a similar way, such a 2D-representation can be extracted for a watermarked image and converted back to the integer w.

Having designed an efficient method for encoding integers as self-inverting permutations, we propose an efficient algorithm for encoding a self-inverting permutation π^* into an image I by first mapping the elements of π^* into an $n^* \times n^*$ matrix A^* and then using the information stored in matrix A^* to mark specific areas of image I in the frequency domain resulting the watermarked image I_w. We also propose an efficient algorithm for extracting the embedded self-inverting permutation π^* from the watermarked image I_w by locating the positions of the marks in image I_w; it enables us to recontract the 2D-representation of the self-inverting permutation π^*.

It is worth noting that although digital watermarking has made considerable progress and became a popular technique for copyright protection of multimedia information [15], our work proposes something new. We first point out that our watermarking method incorporates such properties which allow us to successfully extract the watermark w from the image I_w even if the input image has been compressed with a lossy method, scaled and/or rotated. In addition, our embedding method can transform a watermark from a numerical form into a two dimensional (2D) representation and, since images are 2D structures, it can efficiently embed the 2D representation of the watermark by marking the high frequency bands of specific areas of an image. The key idea behind our extracting method is that it does not actually extract the embedded information instead it locates the marked areas reconstructing the watermark's numerical value.

We have evaluated the embedding and extracting algorithms by testing them on various and different in characteristics images that were initially in JPEG format and we had positive results as the watermark was successfully extracted even if the image was converted back into JPEG format with various JPEG compression ratios. We had also positive results on Gaussian noise

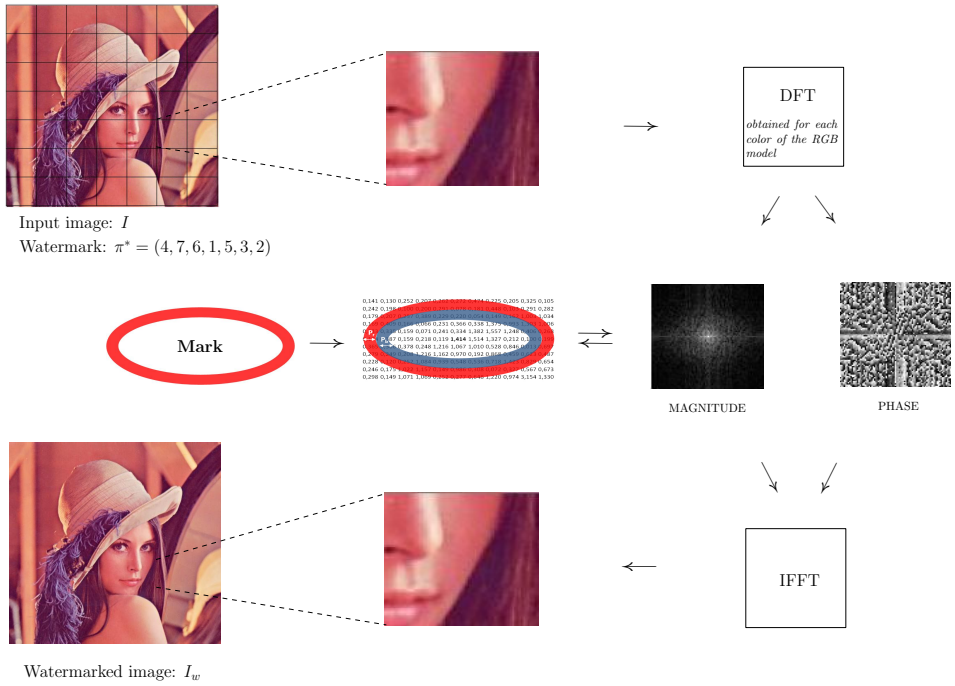


Figure 6.6: The embedding process.

addition and geometric transformation attacks. All the algorithms have been developed and tested in MATLAB environment [54].

We next describe codec algorithms that efficiently encode and decode a watermark into the image's frequency domain [53, 75, 109].

6.4.1 Embed Watermark into Image - F

We next describe the embedding algorithm of our proposed technique which encodes a self-inverting permutation π^* into a digital image I [16, 18, 22]. Recall that, the permutation π^* is obtained over the set N_{n^*}, where $n^* = 2n + 1$ and n is the length of the binary representation of an integer w which actually is the image's watermark [28].

The watermark w, or equivalently the self-inverting permutation π^*, is inserted in the frequency domain of specific areas of the image I. More precisely, we mark the DFT's magnitude of an image's area using two ellipsoidal annuli, denoted hereafter as "Red" and "Blue" (see, Figure 6.6). The ellipsoidal annuli are specified by the following parameters:

- P_r, the width of the "Red" ellipsoidal annulus,
- P_b, the width of the "Blue" ellipsoidal annulus,
- R_1 and R_2, the radiuses of the "Red" ellipsoidal annulus on y-axis and x-axis, respectively.

The algorithm takes as input a SiP π^* and an image I, and returns the watermarked image I_w; it consists of the following steps.

Algorithm `Embed_SiP.to.Image-F`

1. Compute first the 2DM-representation of the permutation π^*, i.e., construct an array A^* of size $n^* \times n^*$ such that the entry $A^*(i, \pi_i^*)$ contains the symbol "*", $1 \leq i \leq n^*$;

2. Next, compute the size of the input image I, say, $N \times M$, and cover the image I with an imaginary grid C with $n^* \times n^*$ grid-cells C_{ij} of size $\lfloor \frac{N}{n^*} \rfloor \times \lfloor \frac{M}{n^*} \rfloor$, $1 \leq i, j \leq n^*$;

3. For each grid-cell C_{ij}, compute the Discrete Fourier Transform (DFT) using the Fast Fourier Transform (FFT) algorithm, resulting in a $n^* \times n^*$ grid of DFT cells F_{ij}, $1 \leq i, j \leq n^*$;

4. For each DFT cell F_{ij}, compute its magnitude M_{ij} and phase P_{ij} matrices which are both of size $\lfloor \frac{N}{n^*} \rfloor \times \lfloor \frac{M}{n^*} \rfloor$, $1 \leq i, j \leq n^*$;

5. Then, the algorithm takes each of the $n^* \times n^*$ magnitude matrices M_{ij}, $1 \leq i, j \leq n^*$, and places two imaginary ellipsoidal annuli, denoted as "Red" and "Blue", in the matrix M_{ij} (see, Figure 6.6). In our implementation,

   - the "Red" is the outer ellipsoidal annulus while the "Blue" is the inner one. Both are concentric at the center of the M_{ij} magnitude matrix and have widths P_r and P_b, respectively;
   - the radiuses of the "Red" ellipsoidal annulus are R_1 (y-axis) and R_2 (x-axis), while the "Blue" ellipsoidal annulus radiuses are computed in accordance to the "Red" ellipsoidal annulus and have values $(R_1 - P_r)$ and $(R_2 - P_r)$, respectively;
   - the inner perimeter of the "Red" ellipsoidal annulus coincides to the outer perimeter of the "Blue" ellipsoidal annulus;
   - the values of the widths of the two ellipsoidal annuli are $P_r = 2$ and $P_b = 2$, while the values of their radiuses are $R_1 = \lfloor \frac{N}{2n^*} \rfloor$ and $R_2 = \lfloor \frac{M}{2n^*} \rfloor$.

   The areas covered by the "Red" and the "Blue" ellipsoidal annuli determine two groups of magnitude values on M_{ij} (see, Figure 6.6);

6. For each magnitude matrix M_{ij}, $1 \leq i, j \leq n^*$, compute the average of the values that are in the areas covered by the "Red" and the "Blue" ellipsoidal annuli; let $AvgR_{ij}$ be the average of the magnitude values belonging to the "Red" ellipsoidal annulus and $AvgB_{ij}$ be the one of the "Blue" ellipsoidal annulus;

7. For each magnitude matrix M_{ij}, $1 \leq i, j \leq n^*$, compute first the variable D_{ij} as follows:

   - $D_{ij} = |AvgB_{ij} - AvgR_{ij}|$, if $AvgB_{ij} \leq AvgR_{ij}$
   - $D_{ij} = 0$, otherwise.

   Then, for each row i of the matrix M_{ij}, $1 \leq i, j \leq n^*$, compute the maximum value of the variables $D_{i1}, D_{i2}, \ldots, D_{in^*}$ in row i; let $MaxD_i$ be the max value;

8. For each cell (i, j) of the 2DM-representation matrix A^* of the permutation π^* such that $A^*_{ij} =$ " $*$ " (i.e., marked cell), mark the corresponding grid-cell C_{ij}, $1 \leq i, j \leq n^*$; the marking is performed by increasing all the values in magnitude matrix M_{ij} covered by the "Red" ellipsoidal annulus by the value

$$AvgB_{ij} - AvgR_{ij} + MaxD_i + c, \tag{6.7}$$

where $c = c_{opt}$. The additive value of c_{opt} is calculated by the function f (see, Subsection 6.4.3) which returns the minimum possible value of c that enables successful extracting;

9. Reconstruct the DFT of the corresponding modified magnitude matrices M_{ij}, using the trigonometric form formula [53], and then perform the Inverse Fast Fourier Transform (IFFT) for each marked cell C_{ij}, $1 \leq i, j \leq n^*$, in order to obtain the image I_w;

10. Return the watermarked image I_w.

In Figure 6.6 we demonstrate the main operations performed by our embedding algorithm. In particular, we show the marking process of the grid-cell C_{55} of the Lena image; in this example, we embed in the Lena image the watermark number w which corresponds to SiP $(4, 7, 6, 1, 5, 3, 2)$.

6.4.2 Extract Watermark from Image - F

In this section we describe the decoding algorithm of our proposed technique. The algorithm extracts a self-inverting permutation π^* from a watermarked digital image I_w, which can be later represented as an integer w.

The self-inverting permutation π^* is obtained from the frequency domain of specific areas of the watermarked image I_w. More precisely, using the same two "Red" and "Blue" ellipsoidal annuli, we detect certain areas of the watermarked image I_w that are marked by our embedding algorithm and these marked areas enable us to obtain the 2D-representation of the permutation π^*. The extracting algorithm works as follows:

Algorithm `Extract_SiP.from.Image-F`

1. Take the input watermarked image I_w and compute its $N \times M$ size. Then, cover I_w with the same imaginary grid C, as described in the embedding method, having $n^* \times n^*$ grid-cells C_{ij} of size $\lfloor \frac{N}{n^*} \rfloor \times \lfloor \frac{M}{n^*} \rfloor$;

2. Then, again for each grid-cell C_{ij}, $1 \leq i, j \leq n^*$, using the Fast Fourier Transform (FFT) get the Discrete Fourier Transform (DFT) resulting a $n^* \times n^*$ grid of DFT cells;

3. For each DFT cell, compute its magnitude matrix M_{ij} and phase matrix P_{ij} which are both of size $\lfloor \frac{N}{n^*} \rfloor \times \lfloor \frac{M}{n^*} \rfloor$;

4. For each magnitude matrix M_{ij}, place the same imaginary "Red" and "Blue" ellipsoidal annuli, as described in the embedding method, and compute as before the average values

that coincide in the area covered by the "Red" and the "Blue" ellipsoidal annuli; let $AvgR_{ij}$ and $AvgB_{ij}$ be these values;

5. For each row i of C_{ij}, $1 \leq i \leq n^*$, search for the j_{th} column where $AvgB_{ij} - AvgR_{ij}$ is minimized and set $\pi_i^* = j$, $1 \leq j \leq n^*$;

6. Return the self-inverting permutation π^*.

Having presented the embedding and extracting algorithms, let us next comment on the function f which returns the additive value $c = c_{opt}$ (see, Step 8 of the embedding algorithm).

6.4.3 Function f

In our watermarking model, the embedding algorithm amplifies the marks in the "Red" ellipsoidal annulus by adding the output of the function f. What exactly f does is returning the optimal value that allows the extracting algorithm under the current requirements, such as JPEG compression, noise addition, to still be able to extract the watermark from the image.

The function f takes as an input the characteristics of the image and the parameters R_1, R_2, P_b, and P_r of our proposed watermark model (see, Step 5 of embedding algorithm and Figure 6.6), and returns the minimum possible c_{opt} that added as c to the values of the "Red" ellipsoidal annulus enables extracting (see, Step 8 of the embedding algorithm). More precisely, the function f initially takes the interval $[0, c_{max}]$, where c_{max} is a relatively great value such that if c_{max} is taken as c for marking the "Red" ellipsoidal annulus it allows extracting, and computes the c_{opt} in $[0, c_{max}]$.

Note that, c_{max} allows extracting but because of being great damages the quality of the image (see, Figure 6.7). We mentioned relatively great because it depends on the characteristics of each image. For a specific image it is useless to use a c_{max} greater than a specific value, we only need a value that definitely enables the extracting algorithm to successfully extract the watermark.

We next describe the computation of the value c_{opt} returned by f; note that, the parameters P_b and P_r of our implementation are fixed with the values 2 and 2, respectively. The main steps of this computation are the following:

(i) Check if the extracting algorithm for $c = 0$ validly obtains the watermark $\pi^* = w$ from the image I_w; if yes, then the function f returns $c_{opt} = 0$;

(ii) If not, that means, $c = 0$ doesn't allow extracting; then, the function f uses binary search on $[0, c_{max}]$ and computes the interval $[c_1, c_2]$ such that:

- $c = c_1$ doesn't allow extracting,
- $c = c_2$ do allow extracting, and
- $|c_1 - c_2| < 0.2$;

(iii) The function f returns $c_{opt} = c_2$.

As mentioned before, the function f returns the optimal value c_{opt}. Recall that, optimal means that it is the smallest possible value which enables extracting $\pi^* = w$ from the image I_w. It is important to be the smallest one as that minimizes the additive information to the image and, thus, assures minimum drop to the image quality.

6.4.4 Experimental Evaluation

In this section we present the experimental results of the proposed watermarking codec algorithms which we have implemented using the general-purpose mathematical software package Matlab (version 7.7.0) [54]. We tested our codec algorithms on various 24-bit digital color images of various sizes (from 200×130 up to 4600×3700) and quality characteristics. Many of the images in our image repository where taken from a web image gallery [99] and enriched by some other images different in characteristics.

In this work we used JPEG images due to their great importance on the web, since they are small in size, while storing full color information (24 bit/pixel), and can be easily and efficiently transmitted. Moreover, robustness to lossy compression is an important issue when dealing with image authentication. It should be observed that the design goal of lossy compression systems is opposed to that of watermark embedding systems. The Human Visual System (HVS) attempts to identify and discard perceptually insignificant information of the image, whereas the goal of the watermarking system is to embed the watermark information without altering the visual perception of the image [132].

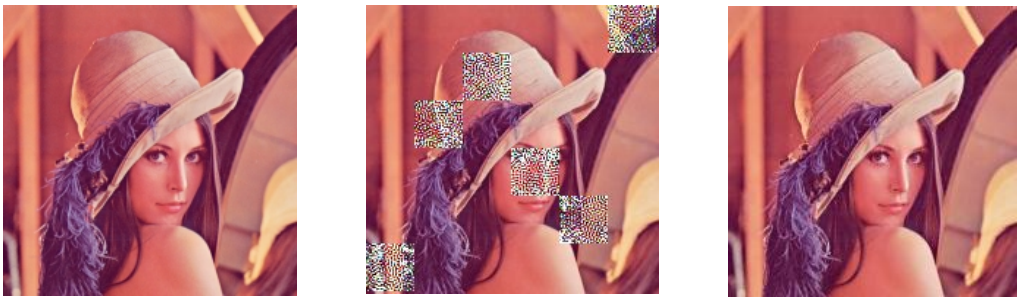
In order to evaluate the quality of the watermarked image obtained from our watermarking method we used two objective image quality assessment metrics, namely the Peak Signal to Noise Ratio (PSNR) and the Structural Similarity Index Metric (SSIM). Our aim was to prove that the watermarked image is closely related to the original (image fidelity [43]), because watermarking should not introduce visible distortions in the original image as that would reduce images' commercial value.

The PSNR metric is the ratio between the reference signal and the distortion signal, i.e., watermark, in an image given in decibels (dB). It is well known that, PSNR is most commonly used as a measure of quality of reconstruction of lossy compression codecs (e.g., for image compression). The higher the PSNR value the closer the distorted image is to the original or the better the watermark conceals. It is a popular metric due to its simplicity, although it is well known that this distortion metric is not absolutely correlated with human vision. The SSIM image quality metric, developed by [122], is considered to be correlated with the quality perception of the HVS [62]. The highest value of SSIM is 1, and it is achieved when the original I and watermarked image I_w are identical.

6.4.5 Performance

Initially, we had to choose the appropriate values for the parameters of the quality function f. In our implementation we set both of the parameters P_r and P_b equal to 2 (see, Step 5 of Algorithm `Extract_SiP.from.Image-F`). Recall that, the value 2 is a relatively small value which allows us to modify a satisfactory number of pixels in order to embed the watermark and successfully extract it, without affecting images' quality. Note that, for great in size images, a smaller width reduces the strength of the watermark. There isn't a distance between the two ellipsoidal annuli as that enables the algorithm to apply a small additive information to the values of the "Red" annulus. The two ellipsoidal annuli are inscribed to the rectangle magnitude matrix, as we want to mark images' cells on the high frequency bands.

We mark the high frequencies by increasing their values using mainly the additive parameter $c = c_{opt}$ because alterations in the high frequencies are less detectable by human eye [70]. What



original $c = c_{opt}$ $c = c_{max}$

Figure 6.7: The original image of Lena and its two watermarked images with $c = c_{max}$ and $c = c_{opt}$; the watermark corresponds to SiP (6,3,2,4,5,1).

is more, in high frequencies most images contain less information.

The quality function f returns the factor c, which has the minimum value c_{opt} that allows the extracting algorithm to successfully extract the watermark. In fact, this value c_{opt} (see, Formula 6.8) is the main additive information embedded into the image. Depending on the images and the amount of compression, we need to increase the watermark strength by increasing the factor c. The value of c increases as the quality factor of JPEG compression decreases. It is obvious that the embedding algorithm is image dependent. It is worth noting that, the c_{opt} values are small for images of relatively small size while these values increase as we move to images of greater size.

To demonstrate the differences on watermarked image quality, with respect to the values of the additive factor c, we watermarked the original image lena.jpg and we embedded a watermark with $c = c_{max}$ and $c = c_{opt}$, where $c_{max} >> c_{opt}$ (see, Figure 6.7); in the watermarked image in the middle we used $c = c_{max}$ for illustrative purposes.

6.4.6 Attack Issues

In this section we present the experimental results of our watermarking method under several attacks. In fact, we test the robustness of our method after applying the following attacks:

(A) JPEG Compression

(B) Gaussian Noise

(C) Geometric Transformations

Recall that, for the evaluation process we use the PSNR and SSIM metrics.

(A) JPEG Compression

The quality factor (or, for short, Q-factor) is a number that determines the degree of loss in the compression process when saving an image. In general, JPEG recommends a quality factor of

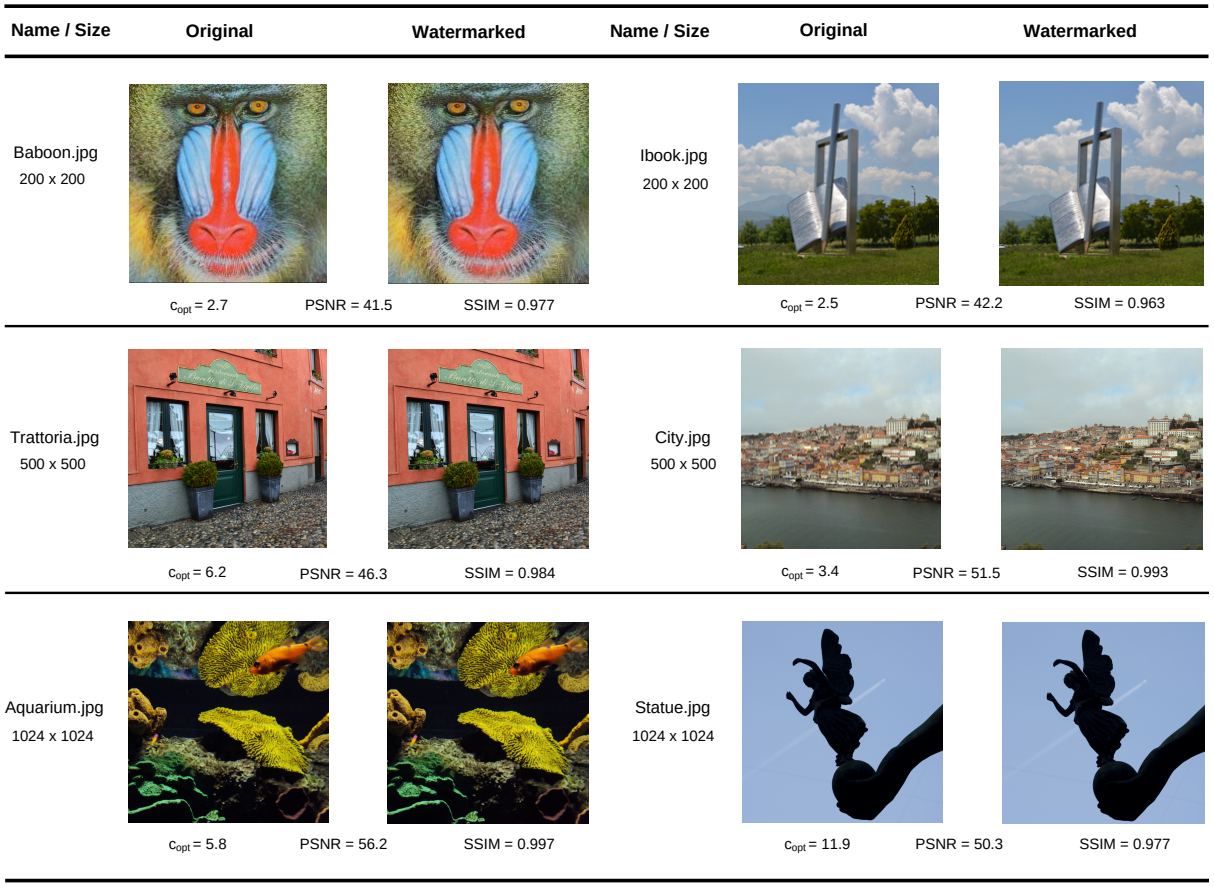


Figure 6.8: Some original images and their corresponding watermarked ones; for each image, its size and its c_{opt}, and PSNR and SSIM values are also shown, for $Q = 55$.

	PSNR				SSIM			
Filename	Q = 85	Q = 75	Q = 65	Q = 55	Q = 85	Q = 75	Q = 65	Q = 55
Lena.jpg	54.04	50.10	46.82	44.86	0.997	0.993	0.986	0.981
Baboon.jpg	49.19	46.17	42.48	41.53	0.995	0.989	0.980	0.977
Trattoria.jpg	67.79	60.59	53.50	46.36	0.999	0.999	0.996	0.984
Aquarium.jpg	65.19	61.20	58.26	56.18	0.999	0.999	0.998	0.997
Ibook.jpg	51.47	47.78	44.76	42.21	0.994	0.987	0.976	0.963
City.jpg	57.20	52.86	48.63	51.54	0.998	0.995	0.987	0.993
Statue.jpg	63.58	58.40	54.90	50.30	0.998	0.995	0.990	0.977

Table 6.1: The PSNR and SSIM values of the original and watermarked images, for compression of qualities $Q = 85$, $Q = 75$, $Q = 65$, and $Q = 55$.

75–95 for visually indistinguishable quality difference, and a quality factor of 50–75 for merely acceptable quality. We compressed the images with Matlab using `imwrite` with different JPEG quality factors; we present results for $Q = 85$, $Q = 75$, $Q = 65$, and $Q = 55$.

original $\sigma^2 = 0.01$ $\sigma^2 = 0.001$ $\sigma^2 = 0.0001$

Figure 6.9: The original image of Lena and its watermarked images with $\sigma^2 = 0.01$, $\sigma^2 = 0.001$ and $\sigma^2 = 0.0001$.

Our watermarked images have excellent PSNR and SSIM values. In Figure 6.8 we present six images of different sizes, along with their corresponding PSNR and SSIM values. Typical values for the PSNR in lossy image compression are between 40 and 70 dB, where higher is better. In our experiments, the PSNR values of 90% of the watermarked images were greater than 40 dB. The SSIM values are almost equal to 1, which means that the watermarked image is quite similar to the original one, which explains the method's high fidelity.

In Table 6.1 we demonstrate the PSNR and SSIM values of some images that are used in this work. We observe that these values are decreasing on smaller quality factors. Also, as the additive value $c = c_{opt}$ increases for each quality factor, the quality decreases. Moreover, the additive value c that embeds robust marks for qualities $Q = 85$, $Q = 75$ and $Q = 65$, does not result in a significant image distortion as the tables suggest.

(B) Gaussian Noise

We test the robustness of our watermarking model by adding Gaussian noise in the images with $mean = 0$ and different variances σ^2, that is, we use $\sigma^2 = 0.01$, $\sigma^2 = 0.001$ and $\sigma^2 = 0.0001$. Figure 6.9 illustrates the original image of Lena and the watermarked images with Gaussian noise of these three variance values. We have to mention that the watermark can be extracted

Filename	$\sigma^2 = 0.01$	$\sigma^2 = 0.001$	$\sigma^2 = 0.0001$
Lena.jpg	24.94	34.75	44.62
Baboon.jpg	24.89	34.79	44.65
Trattoria.jpg	25.04	34.83	44.73
Aquarium.jpg	25.97	35.27	44.81
Ibook.jpg	25.01	34.79	44.62
City.jpg	24.89	34.76	44.70
Statue.jpg	25.37	35.12	44.92

Table 6.2: The PSNR values of the original and watermarked images, for Gaussian noise with variance values $\sigma^2 = 0.01$, $\sigma^2 = 0.001$, and $\sigma^2 = 0.0001$.

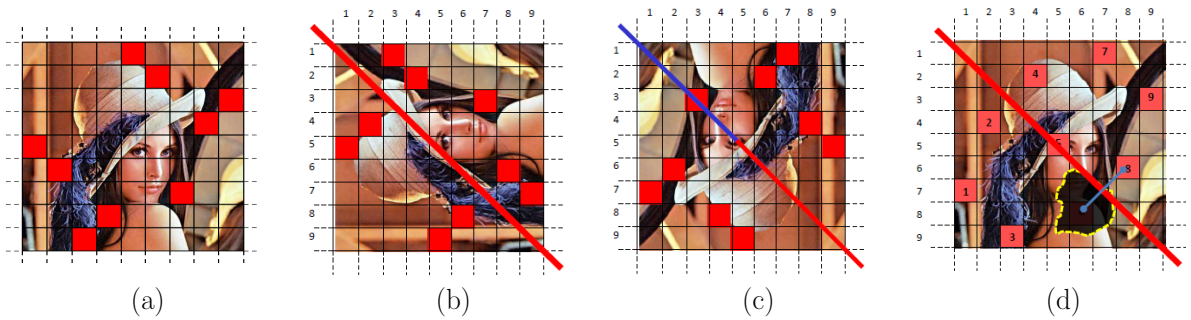


Figure 6.10: (a) Watermarked image of Lena, (b) 90 degrees angled image, (c) 180 degrees angled image, and (d) cropped image.

successfully from the attacked image.

Table 6.2 presents the PSNR values of the original image and the watermarked image with Gaussian noise. As Table 6.2 and Figure 6.9 indicate, although Gaussian noise with $\sigma^2 = 0.01$ introduces significant perceptual distortion in images, watermark remains imperceptible.

(C) Geometric Transformations

The robustness of the proposed model against geometric attacks was evaluated by applying common geometric attacks, which included rotation, cropping, and scaling.

C.1. Rotation Attacks

It is possible to detect whether the watermarked image has been subjected to rotations, thanks to the following two properties of the 2DM-representation of self-inverting permutations.

Due to the fact that the 2DM-representation that has been used to mark the image is the result of a self-inverting permutation the sequence of the marked cells on the image is not random but there are the two properties that can be used to determine the angle of a watermarked image in respect with the original one and that has to do with the position of the marked cell in the main diagonal of the grid. The two properties are the following:

- The main diagonal of the $n^* \times n^*$ symmetric matrix A^* has always one and only one marked cell, and

- The marked cell on the diagonal is always in the entry (i, i) of A^* where: $i = \lceil \frac{n*}{2} \rceil + 1, \lceil \frac{n*}{2} \rceil + 2, \ldots, n^*$.

In case the watermarked image has been subject to 90 degree rotation as demonstrated in Figure 6.10(b) you may notice that the main diagonal has not any cells marked which means that we are dealing with a non valid watermark as there should have been exactly one marked cell.

The second case is when the watermarked image has been subject to 180 degree rotation as demonstrated in Figure 6.10(c). In this case, beginning with the first property someone may notice that the main diagonal has one and only one marked cell meaning that the first property is satisfied confirming that the image has not been subjected to 90 degree rotation.

The diagonal marked cell is situated in the grid's position (i, i); in our image $i = 3$ and thus $i < \lceil \frac{n^*}{2} \rceil$ since $n^* = 9$. It is against the second property meaning that the watermarked image has been subjected to 180 degree rotation.

C.2. Cropping

Once again thanks to the fact that the 2DM-representation a SiP has the symmetric property, i.e., the $n^* \times n^*$ matrix A^* is symmetric, our algorithm successfully extracts the watermark even marked parts of the watermarked image have been lost. This loss can be the result of cropping procedures to certain areas of the image. Recall that, this property is a consequence of the fact that at a self-inverting permutation, each element has its own inverse.

In Figure 6.10(d) the removed marked part of the image can be recovered since the matrix A^* of the SiP π^* is symmetric; for example, because of the fact that $A^*(4, 8)$ is marked, $A^*(8, 4)$ is marked as well. Based on this property, we conclude that the lost marked cell is $A^*(8, 4)$ and then we correctly extract the embedded watermark.

C.3. Scaling

In the case where a watermarked image has underwent significant scaling then extracting a watermark may be unsuccessful. In our model if an image has been scaled by a known ratio then each cell of the imaginary grid has underwent exactly the same scaling meaning that the magnitude cell has now a different size as well. Due to this fact, the width of the annuli will be incorrect making it impossible to calculate the appropriate difference between them. A solution to this would be to use different sized annuli in order to calculate the valid difference between them so that to spot marked areas.

The idea is simple, considering that we know the scaling ratio that the image has underwent we apply the same ratio calculating the new width for the two annuli. So if for example we used $P_b = 2$ and $P_r = 2$ for the width of the "Blue" and the "Red" annulus respectively when we performed the embedding procedure and the image has underwent 50% scaling then in order to extract the embedded watermark from the image we have to use $P_b^* = 1$ and $P_r^* = 1$.

In order to calculate the difference between the same frequency bands, in the second case where the magnitude cell has 50% of the initial size, we use annuli that have 50% less width in comparison with the ones originally embedded.

6.5 Audio Watermarking

In this section we present an algorithm for encoding a self-inverting permutation π^* into an audio signal S by marking specific time segments of S in the frequency domain resulting thus the watermarked audio signal S_w. We also present a decoding algorithm which extracts the embedded permutation π^* from S_w by locating the positions of the marks in S_w [17].

6.5.1 Embed Watermark into Audio

The embedding algorithm of our proposed technique encodes a self-inverting permutation π^* into a digital audio signal S. Recall that, the permutation π^* is obtained over the set N_{n^*}, where $n^* = 2n + 1$ and n is the length of the binary representation of an integer w which actually is

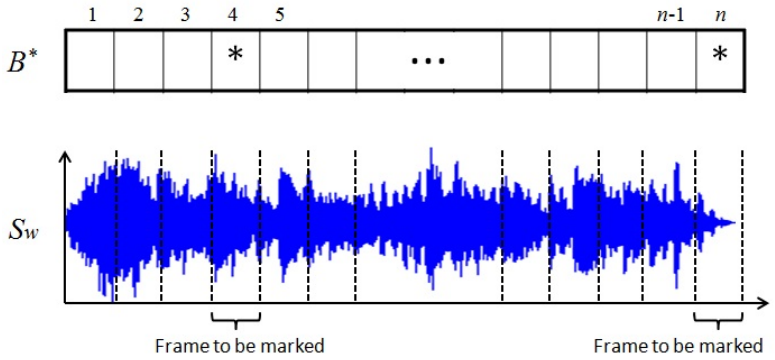


Figure 6.11: Segmentation of the S's signal into specific frames according to 1DM-representation of the permutation π^*.

the audio's watermark [28].

The main idea of embedding. The watermark w, or equivalently the corresponding self-inverting permutation π^*, is imperceptibly inserted in the frequency domain of specific frames on the audio track signals S; see, Figure 6.11. More precisely, we mark certain frames getting the DFT and do alterations at the magnitude values of high frequencies for each audio frame to be marked; see, Figure 6.12. This is achieved by choosing two groups of magnitude values specified with two segments of the magnitude vector namely "Red" and "Blue" and the alterations are actually on their difference; see, Figure 6.13. In our implementation we use fixed segments' widths and distances from the center of symmetry of the DFT's magnitude vector. The added value is specified by the maximum value in the defined area.

The embedding algorithm. Our embedding algorithm takes as input a SiP π^* and an audio signal S and returns the watermarked audio signal S_w; it performs the following main processes:

(i) construct the 1DM-representation of the watermark number w;

(ii) transform the input audio signal S and acquire the frequency representation of it;

(iii) modify signals' frequency representation according to the 1DM-representation of the signal S;

(iv) returns the watermarked audio signal S_w;

We describe below in detail the embedding algorithm in steps.

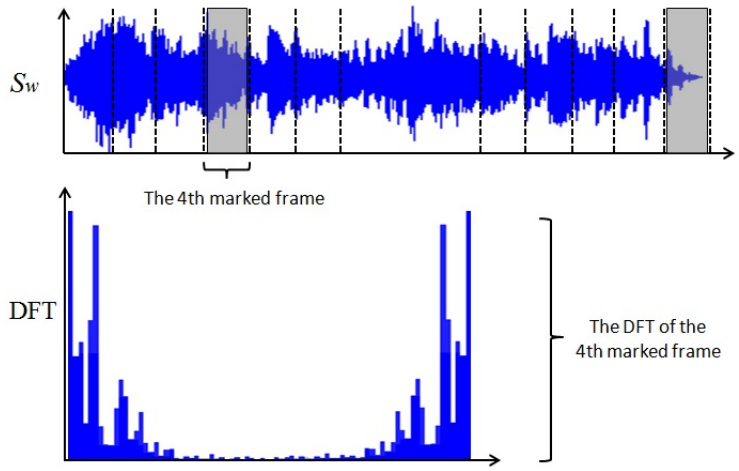


Figure 6.12: The DFT representation of a marked frame.

Algorithm `Embed_SiP.to.Audio`

1. Compute first the 1DM-representation of the permutation π^*, i.e., construct the array B^* of size $n = n^* \times n^*$; recall that the entry $B^*((i-1)n^* + \pi_i^*)$ contains the symbol "*", $1 \leq i \leq n$;

2. Segment the audio signal S into n non-overlapping frames f_i of size $f_i[a,b] = \lfloor \frac{N-1}{n} \rfloor$, $1 \leq i \leq n$, where N is the length of the audio signal;

3. For each frame f_i, compute the Discrete Fourier Transform (DFT) using the Fast Fourier Transform (FFT) algorithm, resulting in n DFT frames F_i of size $F_i[a,b] = \lfloor \frac{N-1}{n} \rfloor$, $1 \leq i \leq n$, that is, $F_i = \texttt{FFT}(f_i)$;

4. For each DFT frame F_i, compute its magnitude M_i and phase P_i vectors (or, arrays) which are both of size $M_i[a,b] = P_i[a,b] = \lfloor \frac{N-1}{n} \rfloor$, $1 \leq i \leq n$;

5. Then, the algorithm takes each of the n magnitude vectors M_i and determines two segments in M_i, $1 \leq i \leq n$, denoted as "Red" and "Blue" (see, Figure 6.13). In our implementation,

   - each "Red" segment $[x_r, y_r]$ has length ℓ_r (even), where $x_r = \lfloor \frac{N-1}{2n} \rfloor - \frac{\ell_r}{2}$ and $y_r = \lfloor \frac{N-1}{2n} \rfloor + \frac{\ell_r}{2}$;
   - each "Blue" segment $[x_b, y_b]$ has length ℓ_b (even), where $x_b = x_r - \frac{\ell_b}{2}$ and $y_b = y_r + \frac{\ell_b}{2}$

   The "Red" and the "Blue" segments determine two groups of magnitude values on M_i; the Red_Values and the Blue_Values (see, Figure 6.13);

6. For each magnitude vector M_i, $1 \leq i \leq n$, compute the average value $AvgR_i$ of the Red_Values and the average value $AvgB_i$ of the Blue_Values of M_i;

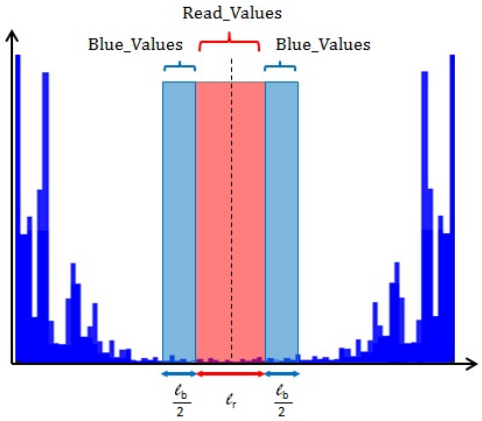


Figure 6.13: The "Red" and "Blue" segments on DFT.

7. For each magnitude vector M_i, $1 \leq i \leq n$, compute first the variable D_i as follows:

- $D_i = |AvgB_i - AvgR_i|$, if $AvgB_i \geq AvgR_i$
- $D_i = 0$, otherwise;

8. Partition the n values D_1, D_2, ..., D_n into n^* sets $E_1, E_2, \ldots, E_{n^*}$, each of size n^* (recall that $n = n^* \times n^*$); let $\{D_{i1}, D_{i2}, \ldots, D_{in^*}\}$ be the elements of the i-th set E_i, $1 \leq i \leq n^*$. Then, compute the values

- $MaxD_1$, $MaxD_2$, ..., $MaxD_{n^*}$

where $MaxD_i$ is the maximum value of the i-th set $E_i = \{D_{i1}, D_{i2}, \ldots, D_{in^*}\}$, $1 \leq i \leq n^*$;

9. For each marked cell $B^*(i)$ of the 1DM-representation matrix B^* of the permutation π^* (i.e., the call which contains the symbol "*", mark the corresponding frame F_i, $1 \leq i \leq n$; the marking is performed by increasing all the Red_Values in M_i by the value

$$AvgB_i - AvgR_i + MaxD_k + c, \tag{6.8}$$

where $k = \lceil \frac{i}{n^*} \rceil$ and $c = c_{opt}$. The additive value of c_{opt} is a predefined value which enables successful extracting;

10. Reconstruct the DFT of the corresponding modified magnitude vector M_i, using the trigonometric form formula [53], and then perform the Inverse Fast Fourier Transform (IFFT) for each frame F_i, $1 \leq i \leq n$, in order to obtain the audio signal S_w;

11. Return the watermarked audio signal S_w.

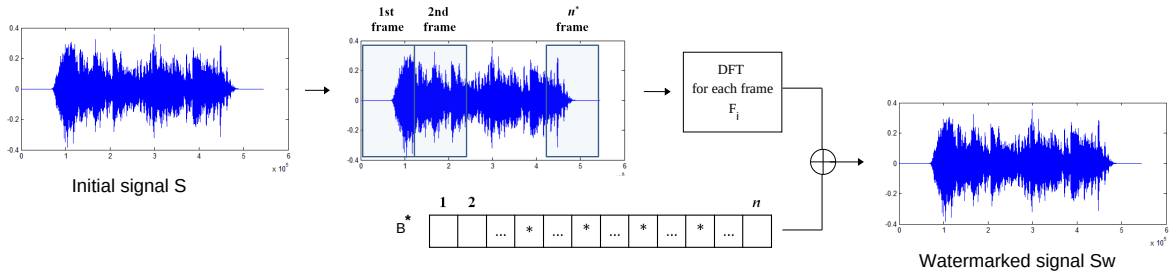


Figure 6.14: The encoding process of audio signal watermarking.

Note that concerning the placement of the "Red" and "Blue" segments, their position can vary according to the frequency band in which we want to mark a frame. At the above illustration we mark it in the high frequencies but there can be a different approach. Specifically, we can mark instead lower frequencies and that is performed by moving the segments from the center to the right and left edges of the magnitude array of the Discrete Fourier Transform (DFT) representation.

6.5.2 Extract Watermark from Audio

In this section we describe the decoding algorithm of our proposed technique. The algorithm extracts the SiP π^* from a watermarked digital audio signal S_w, which can be later represented as an integer w.

The main idea of extracting. The main idea behind the extracting algorithm is that the self-inverting permutation π^* is obtained from the frequency domain of specific frames of the watermarked audio signal S_w. More precisely, using the same two "Red" and "Blue" segments, we detect certain areas of the watermarked audio signal S_w so that the difference between the average values of the "Red" segment have the maximum positive difference over the average values of the "Blue" segments. In this way we can detect marked frames that enable us to obtain the 1DM-representation of the permutation π^*.

The extracting algorithm. We next describe the extracting algorithm which consists of the following steps.

Algorithm `Extract_SiP.from.Audio`

1. Take the input watermarked audio S_w and compute its size N. Then, segment S_w into n non-overlapping frames f_i of size $f_i[a,b] = \lfloor \frac{N-1}{n} \rfloor$, $1 \leq i \leq n$;

2. Then, using the Fast Fourier Transform (FFT), get the Discrete Fourier Transform (DFT) for each frame f_i, resulting in n DFT frames F_i, $1 \leq i \leq n$;

3. For each DFT frame F_i, compute its magnitude M_i and phase P_i vectors, which are both of size $M_i[a,b]=P_i[a,b]=\lfloor \frac{N-1}{n} \rfloor$, $1 \leq i \leq n$;

4. For each magnitude vector M_i, compute the average values $AvgR_i$ and $AvgB_i$ of the Red_Values and Blue_Values of M_i, respectively, as described in the embedding algorithm;

5. Partition the n vectors M_i, $1 \leq i \leq n$, into n^* sets $L_1, L_2, \ldots, L_{n^*}$, each of size n^*; let $\{M_{i1}, M_{i2}, \ldots, M_{in^*}\}$ be the elements of the i-th set L_i and let $AvgR_{ij}$ and $AvgB_{ij}$ be the average values of the Red_Values and Blue_Values, respectively, of the vector M_{ij}, $1 \leq i, j \leq n^*$;

6. For each set $L_i = \{M_{i1}, M_{i2}, \ldots, M_{in^*}\}$ find the k_{th} vector M_{ij} such that $AvgB_{ik} - AvgR_{ik}$ is minimum and set $\pi_i^* = k$, $1 \leq k \leq n^*$;

7. Return the self-inverting permutation π^*.

Having presented the embedding and extracting algorithms, we next briefly comment on the purpose of the additive value $c = c_{opt}$ (see, Step 9 of the embedding algorithm). Similar to image watermarking, we add at the corresponding embedding marking step the additive value c_{opt} which by getting greater increases the robustness of the marks; in our audio watermarking case, we just used a very small value for it.

6.5.3 Experimental Results

This section summarizes the experimental results of the proposed audio watermarking codec algorithms; we implemented our algorithms and carried out tests using the general-purpose mathematical software package Matlab (Version 7.7.0) [66].

Testing of our embedding and extracting algorithms has been made by the use of various 16-bit digital audio tracks in *wav* format with 44.1 KHz sampling frequency. Concerning the audio samples used, they where relatively short abstracts with different characteristics. For instance there were tracks containing speech which have many silent segments as well as music track samples and tracks with extreme features such as low and high frequency sounds. Many of the audio tracks that we used for testing were acquired from a web audio repository called wavsource and enriched by some other audio tracks from various sources. It is well known in the field of watermarking that there are three main characteristics to take into account describing and evaluating a digital watermarking system: *Fidelity*, *Robustness*, and *Capacity* [43].

Concerning our watermarking system, it seems to be of high fidelity as watermarked tracks were not distinguished over the original ones and the results using the PSNR metric were interestingly positive. Concerning the marking procedure of our implementation, we set both lengths ℓ_r and ℓ_b of the "Red" and "Blue" segments respectively, equal to 20% of half the length of magnitude vector as it is mirrored (see, Section 6.5.1). Recall that, the value 20% is a relatively small percentage which allows us to modify the audio track segments in a satisfactory level in order to detect the watermark and successfully extract it without affecting audio tracks' initial quality. Moreover, we choose to alter higher frequencies and thus the two segments are at the center of the magnitude vector. This is because high frequencies are less perceptible according to the human auditory system. What is more, at high frequencies audio tracks contain less information which means that information is less likely to be lost due to post alterations.

Filename	PSNR
bach.wav	67.2
clarinet.wav	67.9
castanets.wav	68.2
elvis_riverside.wav	75.3
family_man.wav	73.8
high10sec.wav	58.8
low10sec.wav	64.5

Table 6.3: The PSNR values of the watermarked audio signals.

Fidelity. In order to evaluate the watermarked audio track quality obtained from our proposed watermarking method we used the Peak Signal to Noise Ratio (PSNR) metric. Our aim was to prove that the watermarked audio track is closely related to the original track proving the high fidelity attribute of our system. This is something vital as watermarking should not introduce audible distortions in the original audio track, as that would certainly reduce its commercial value.

Giving a short introduction to the PSNR metric, we should mention that it is defined as the ratio between the reference (or, original) signal and the distorted (or, watermarked) signal of an audio track and it is given in decibels (dB). It is well known that PSNR is most commonly used as a measure of quality of reconstruction of lossy compression codecs (e.g., for image or audio compression methods). The higher the PSNR value the closer the distorted signal is to the original or the better the watermark conceals. We mentions that PSNR is a popular metric due to its simplicity.

For an initial audio signal S of size N and its watermarked equivalent signal S_w, PSNR is defined by the formula:

$$\mathrm{PSNR}(S, S_w) = 10 \log_{10} \frac{N_{max}^2}{MSE}, \tag{6.9}$$

where N_{max} is the maximum signal value that exists in the original audio track and MSE is the Mean Square Error which is given by the following formula:

$$\mathrm{MSE}(S, S_w) = \frac{1}{N} \sum_{i=0}^{N-1} (S(i) - S_w(i))^2. \tag{6.10}$$

Comparing the original audio tracks with the watermarked ones, we immediately get to notice that they depict excellent fidelity according to the PSNR values that we have obtained. In every case PSNR is over 50 dB which proves that fact that there is a striking similarity between the original and the watermarked signal of an audio track.

In Table 6.3 you can see the performance of our method as we demonstrate the PSNR values of some audio tracks that we used in this work. Each of them was sampled at 41.1 KHz and the duration was of about 10 sec. Additionally, each one has much different characteristics. More specifically, the audio tracks

- `bach.wav`, `clarinet.wav` and `castanets.wav`

where a concert, a clarinet and castanets solo respectively. The audio track

- `elvis_riverside.wav`

combines human voice with music, while the

- `family_man.wav`

contains only speech which means that it also has periods of silence. Lastly the audio tracks

- `high10sec.wav` and `low10sec.wav`

are some extreme cases of high and low frequency sounds.

Robustness. The watermarked signals were subjected to distortions or common signal attacks in order to evaluate the robustness of our audio watermarking algorithms. We tested the performance of each audio track under white noise addition, cropping, resampling, requantization and MP3 compression. Below we describe in more details each one of the five different attacks that we applied in our experiments.

(a) Gaussian Noise. A white gaussian noise of SNR 20 dB was added to the original audio signal.

(b) Cropping. A 10% of the beginning of the watermarked audio signal was cropped and subsequently replaced by zeros.

(c) Resampling. The watermarked signal, originally sampled at 44.1 KHz, is resampled at 22.05 KHz, and then restored back by sampling again at 44.1 KHz

(d) Requantization. The 24-bit watermarked audio signal is re-quantized down to 16 bits/sample and then back to 24 bits/sample.

(e) MP3 compression. The watermarked audio signal is compressed using a bit rate of 128 Kb/s and then decompressed back to the WAV format.

Since the watermark that we embed in our audio signal is a permutation, i.e. a vector over the set N_n $(n > 1)$, we test after each attack the similarity of the extracted watermark with the original one using the Hamming distance [63].

The Hamming distance $d(x, y)$ between two vectors x and y is the number of coefficients in which they differ [63]. The Hamming distance equals to zero, i.e., $d(x, y) = 0$, if x and y agree in all coordinates; it happens if and only if $x = y$. In our case the Hamming distance is computed between the watermark $w = \pi^*$ that we embedded into the audio track and the watermark π^*_{ext} that we extract from the audio. If $d(\pi^*, \pi^*_{ext}) = 0$ the watermark $w = \pi^*$ successfully extracts from the attacked audio signal. Additionally, it is worth noting that if $d(\pi^*, \pi^*_{ext})$ is relatively small, then the watermark π^* can be reconstructed with high probability by exploiting the self-inverting properties of the permutation π^*.

In Table 6.4 we demonstrate similarity results between the watermark that we embedded into the audio track and the watermark that we extracted after various signal processing attacks.

Filename	Gaussian Noise	Cropping	Resampling	Requantization	MP3 Compres.
bach.wav	0	1	0	0	3.2
clarinet.wav	0	1	0	0	2.4
castanets.wav	0	1	0	0	5.2
elvis_riverside.wav	0	1	0	0	2.8
family_man.wav	0	1	0	0	0.2
high10sec.wav	0	1	0	0	3.0
low10sec.wav	0	1	0	0	2.0

Table 6.4: The Hamming distance of the watermark w and the extracted watermark w^* after common signal attacks.

As the experimental results show, our audio watermarking algorithm is robust against additive gaussian noise of SNR 20 dB, cropping, resampling and requantization. Evaluating our method's robustness over lossy compression we tested it using the MP3 encoding format with a bit rate of 128 Kb/s. In order to optimize the results as high frequency information is mostly lost using MP3 we made the appropriate adjustments concerning the width of the segments to be marked as well as the additive value $c = c_{opt}$ (see, embedding algorithm `Embed_SiP.to.Audio`). For most cases the results were positive as despite not being able in every case to successfully extract all the elements of the watermark, using the properties of self-inverting permutations recovery of the initial watermark can be successfully operated.

Closing the robustness evaluation of our method, we should point out that a drawback of our method is actually when we want to watermark an audio track with extreme high frequencies; it is something that could be encountered on future work.

Capacity. The capacity of our audio watermark method has been computed by measuring the percentage of the watermarked parts of an audio track over the length of the entire audio track. Our method partitions the audio track into $n^* \times n^*$ frames, where n^* is the length of the permutation π^*, and marks only one frame of a set of n^* frames; recall that our embedding method groups the frames into n^* sets each containing n^* frames (see, embedding algorithm `Embed_SiP.to.Audio`). That means, a total n^* over $n^* \times n^*$ frames are marked, so the ratio of the watermarked part over the entire length of the audio track is $\frac{n^*}{(n^* \times n^*)}$. Thus, our audio watermarking method has $\frac{1}{n^*}$ capacity.

6.6 Concluding Remarks

In this chapter we proposed watermarking models for embedding invisible watermarks into digital images and audio signals. We presented methods for embedding invisible watermarks into images and their intention is to prove the authenticity of an image. The watermarks are given in numerical form, transformed into self-inverting permutations, and embedded into an image by partially marking the image in the frequency domain; more precisely, thanks to 2D-representation of self-inverting permutations, we locate specific areas of the image and modify their magnitude of high frequency bands by adding the least possible information ensuring robustness and im-

perceptiveness.

We experimentally tested our embedding and extracting algorithms on color JPEG images with various and different characteristics; we obtained positive results as the watermarks were invisible, they didn't affect the images' quality and they were extractable despite the JPEG compression. In addition, the experimental results show an improvement in comparison to the initial obtained results on spatial domain and they also depict the validity of our proposed codec algorithms.

It is worth noting that the proposed algorithms on image watermarking in frequency domain are robust against cropping or rotation attacks since the watermarks are in SiP form, meaning that they determine the embedding positions in specific image areas. Thus, if a part is being cropped or the image is rotated, SiP's symmetry property may allow us to reconstruct the watermark. Furthermore, our codec algorithms can also be modified in the future to get robust against scaling attacks. That can be achieved by selecting multiple widths concerning the ellipsoidal annuli depending on the size of the input image.

Finally, we should point out that the study of our quality function f remains a problem for further investigation; indeed, f could incorporate learning algorithms [102] so that to be able to return the c_{opt} accurately and in a very short computational time.

Additionally, in this chapter we presented an audio watermarking technique which efficiently and invisibly embeds information, i.e., watermarks, into an audio digital signal. Our technique is based on the same main idea of image watermarking technique expanding thus the digital objects that can be efficiently watermarked through the use of self-inverting permutations.

We experimentally tested our embedding and extracting algorithms on WAV audio signals. Our testing procedure includes the phases of embedding a numerical watermark $w = \pi^*$ into several audio signals S, storing the watermarked audio S_w in WAV format, and extracting the watermark $w = \pi^*$ from the audio S_w. We obtained positive results as the watermarks were invisible, they didn't affect the audio's quality and they were extractable.

The performance evaluation of our audio watermarking technique on several other attacks remains a problem for further investigation.

Chapter 7

Text Watermarking

7.1 Introduction

7.2 Background Results

7.3 Watermarking PDF Documents

7.4 Concluding Remarks

7.1 Introduction

Information age has altered the way people communicate by breaking the barriers imposed on communications by time, distance, and location and has undoubtedly impact not only humans activities but also global industry and economy. Communication has been greatly affected by the constant and rapid evolution of many technologies such as fiber optic, cellular and satellite technology, networking, digital transmission and compression as well as advanced computers, and improved human-computer interaction. The aforementioned technologies allow the rapid transmission, and store, of great amounts of information.

The digital era has already had extensive impacts on business, commerce, education, services, and social life. The concepts of e-government, e-learning, e-commerce, e-business, e-publishing, refer/outline peoples' interaction in the digital world. In this world, people everyday, interact by exchanging e-mails, instant messages, video, audio, images, and digital documents. Part of the information transmitted is an increasing amount of sensitive information, such as personal data, medical and financial records, business information, government data, legal documents. Another part of information available in the web is used to promote ones' work or product.

Electronic document, is an extensively used medium traveling over the internet for information exchange and due to the ease of copying and distributing they are susceptible to threats like illegal copying, redistribution of copyrighted documents, and plagiarism. Subsequently, it has become more important to protect the electronic documents from any malicious user while existing in the digital world. Copyright protection of digital contents is such a need of time which

cannot be overlooked. In past, various methods like encryption, steganography and watermarking has been used to solve these problems. However, digital watermarking is the better solution for copyright protection than encryption and steganography. Digital watermarking methods are efficient enough to identify the original copyright owner of the contents.

Recall that there are many reasons why you would want to use watermarks in digital documents: as a copying deterrent, as a means of identifying the source of a printed document, as a means of determining whether a document has been altered, etc.

Attacks. Any action that a user can perform on a text that can affect the watermark, or its usefulness, is called attack. In [134] existing attacks on text watermarking can be classified into three main categories:

- watermark attacks,
- geometric attacks, and
- system attacks.

In a watermark attack, the adversary aims to detect and destroy the watermark, without necessarily decoding the original message. In contrast to watermark attacks, geometrical attacks are blind attacks on watermarked text documents. The process of these attacks requires neither the algorithmic knowledge of the watermarking technique nor the watermarking key, geometrical attacks intend not to remove the embedded watermark itself, but to prevent it from serving its intended purpose through altering format or content of the watermarked text documents. This type of attack includes reformatting, reproducing, sentences swapping, paragraphs shuffling, the addition/deletion of words, sentences and paragraphs. System attacks use signal processing tools such as principal component analysis, independent component analysis, clustering, vector quantization.

Related Work. Text watermarking is the area of research that has emerged after the development of Internet and communication technologies; we mention that the first reported effort on marking documents dates back to 1993 [87].

Generally, we can classify the previous work on digital text watermarking in the following categories:

- image based approach,
- syntactic approach,
- semantic approach, and
- structural approach.

In image based approach, watermark is embedded in text image. Brassil, et al. were the first to propose a few text watermarking methods utilizing text image [9, 10]; they also developed document watermarking schemes based on line shifts, word shifts as well as slight modifications to the characters [11]. Maxemchuk, et al. [87, 88, 89] analyzed the performance of these methods, while later Low, et al. [80, 81] further analyzed their efficiency. Huang and Yan [64] proposed a text watermarking method based on an average inter-word distance in each line. The distances are adjusted according to the sine-wave of a specific phase and frequency. Feature and pixel level

algorithms were also developed which mark the documents by modifying the stroke features such as width or serif [6].

In syntactic approach, the syntactic structure of text has been used to embed watermark. Atallah, et al. [5] proposed several methods of natural language watermarking, which opened up a brand-new and challenging research direction for text watermarking. Meral et al. performed morpho-syntactic alterations to the text to watermark it [90]. They also provided an overview of available syntactic tools for text watermarking [91].

In semantic approach, semantics of text are used to embed the watermark in text. Atallah et al. were the first to propose the semantic watermarking schemes [5]. Later, the synonym substitution method was proposed, in which watermark was embedded by replacing certain words with their synonyms [118]. Sun, et al. [111] proposed noun-verb based technique for text watermarking which used nouns and verbs parsed by semantic networks. Topkara, et al. proposed an algorithm of the text watermarking by using typos, acronyms and abbreviation in the text to embed the watermark [119]. Algorithms were developed to watermark the text using the linguistic approach of presuppositions [92] in which the discourse structure, meaning, and representations are observed and utilized to embed watermark bits. The text pruning and the grafting algorithms were also developed in the past. Another algorithm based on text meaning representation (TMR) strings has also been proposed [82].

The structural approach is the most recent approach used for copyright protection of text documents. In this approach, text is not altered, rather it is used to logically embed watermark in it. A text watermarking algorithm for copyright protection of text using occurrences of double letters (aa-zz) in text has recently been proposed [67]. Recently, a significant number of techniques have been proposed in the literature which use Portable Document Format (PDF) files as cover media in order to hide data [12, 13, 76, 77, 78, 79, 133].

Contribution. In this chapter, in order to provide to web users copyright protection of their digital documents, we present easily implemented techniques for watermarking PDF documents. Our aim is to extent the digital objects that the proposed representations of a self-inverting permutation, i.e. the 1D-representation, the 2D-representation, and the RPG-representation, can be efficiently applied to; note that, RPG-representation means the encoding of permutation π^* as a reducible permutation graph $F[\pi^*]$.

We first propose an image-based technique for marking the PDF document T by exploiting the 1D-representation of a permutation (or, for short, SiP) π^*, which we presented in Subsection 6.2.2. The embedding of a mark is performed by increasing the distance (or, space) between two consecutive words in a paragraph of document T. The extraction algorithm operates in a reverse manner.

Since pages of PDF documents T are two dimensional objects, we propose an algorithm for encoding a self-inverting permutation π^* into a document T by first mapping the elements of π^* into an $n^* \times n^*$ matrix A^* and then using the information stored in A^* to mark invisibly specific areas of PDF document T resulting thus the watermarked PDF document T^*. We also propose an efficient algorithm for extracting the embedded self-inverting permutation π^* from the watermarked PDF document T^* by locating the positions of the marks in T^*; it enables us to recontract the 2D-representation of the self-inverting permutation π^*.

Finally, we describe a watermarking algorithm for embedding a self-inverting permutation

into the document structure of a PDF file T, by exploiting the graph representation of π^* proposed in this thesis and the structure of a PDF document T described in this chapter. More precisely, in light of the two embedding algorithms `Encode_SiP.to.RPG-I` and `-II`, we present an algorithm for embedding a reducible permutation graph $F[\pi^*]$ into a PDF document T. The main idea behind the proposed embedding algorithm is a systematic addition of appropriate object-references in the input PDF document T, through the use of entries of type $\backslash$`kye`$(\cdot)$, so that the graph $F[\pi^*]$ can be easily constructed from the page tree PT(T^*) of the resulting watermarked document T^*.

Road Map. The chapter is organized as follows: In Section 7.2 we establish the notation and related terminology, and we present background results. In Section 7.3 based on the three different representations of self-inverting permutation (SiP), i.e. the two dimensional (2D-representation), the one dimensional (1D-representation), and the the RPG-representation (the encoding of permutation π^* as a reducible permutation graph $F[\pi^*]$), we present the algorithms `Embed_SiP.to.PDF-I`, `Embed_SiP.to.PDF-II`, and `Embed_RPG.to.PDF`, along with the corresponding extracting algorithms, for embedding a watermark number (or, equivalently, a self-inverting permutation π^* or a reducible permutation graph $F[\pi^*]$) into a PDF document file. Finally, in Section 7.4 we conclude the chapter and discuss possible future extensions.

7.2 Background Results

In this section, we give some definitions and the theoretical background we use towards the watermarking of Portable Document Format (PDF) documents. We first briefly present the different representations of a self-inverting permutation (SiP), and then we present the structure of PDF documents.

1D representation of SiP. In Chapter 6, we presented the one-dimensional representation (1D-representation) of a self-inverting permutation (SiP) and the one dimensional marked representation of SiP (1DM-representation). We later showed how to embed a SiP, represented in 1D space, into an audio signal for watermarking. In our 1D-representation, the elements of the permutation π are mapped in specific cells of an array B of size n^2 as follows:

- number $\pi_i \longrightarrow$ entry $B((\pi_{\pi_i}^{-1} - 1)n + \pi_i)$

or, equivalently, the cell at the position $(i-1)n + \pi_i$ is labeled by the number π_i, for each $i = 1, 2, \ldots, n$.

In our 1DM-representation, a permutation π over the set N_n is represented by an n^2 array B^* by marking the cell at the position $(i-1)n + \pi_i$ by a specific symbol, for each $i = 1, 2, \ldots, n$, where, in our implementation, the used symbol is again the asterisk character "*".

2D representation of SiP. In Chapter 6, we presented the two-dimensional representation of a SiP (2D-representation) and the two dimensional marked representation of SiP (2DM-representation), which we later used to watermark an image. We defined the 2D-representation

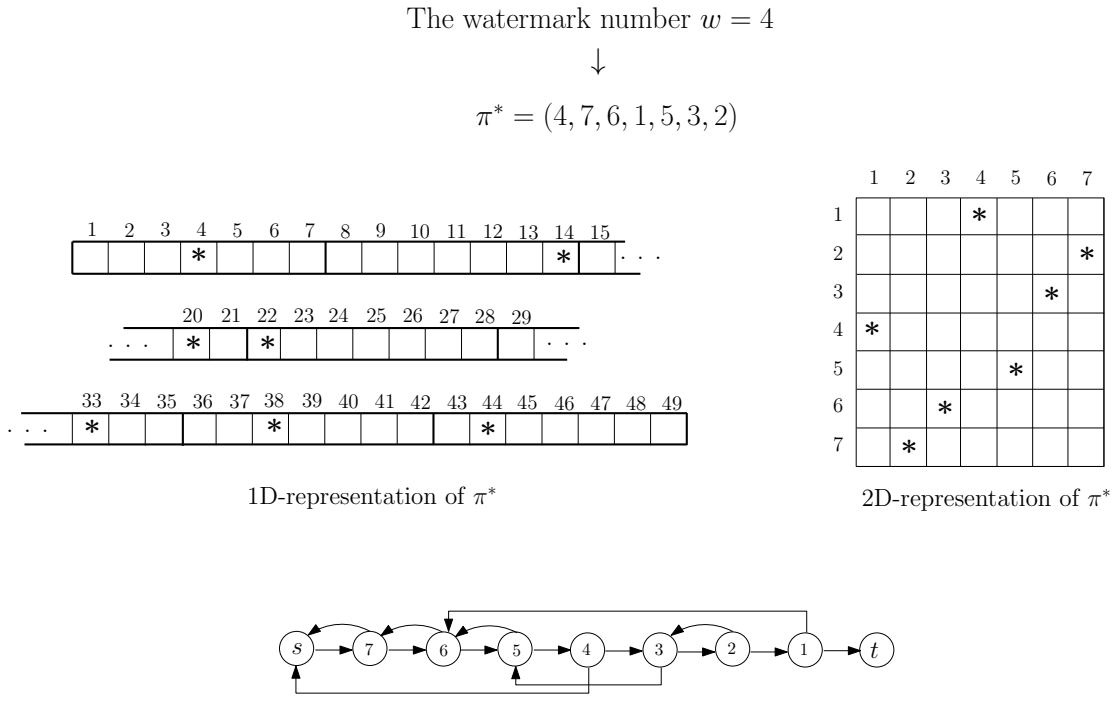


Figure 7.1: Three different representations of permutation $\pi^* = (4, 7, 6, 1, 5, 3, 2)$.

of a SiP as the representation where the elements of the permutation $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$ are mapped in specific cells of an $n \times n$ matrix A as follows:

- number $\pi_i \longrightarrow$ entry $A(\pi_i^{-1}, \pi_i)$

or, equivalently,

- the cell at row i and column π_i is labeled by the number π_i, for each $i = 1, 2, \ldots, n$.

In 2DM-representation the cell at row i and column π_i of matrix A is marked by a specific symbol, for each $i = 1, 2, \ldots, n$.

In Chapter 7, we also presented algorithms for embedding the 2D-dimensional representation of SiP in an image. Recall that the matrix A^* incorporates important structural properties which, in image watermarking, make it possible to detect geometric transformations on the watermarked image. The properties of the matrix A^* are the following:

- The matrix A^* is symmetric.
- The main diagonal of the symmetric matrix A^* has always one and only one marked cell.
- The marked cell on the diagonal is always in the entry (i, i) of A^* where: $i = \lceil \frac{n^*}{2} \rceil + 1, \lceil \frac{n^*}{2} \rceil + 2, \ldots, n^*$.

In Chapter 3, we have also presented an efficient and easily implemented algorithm for encoding numbers as reducible permutation graphs through the use of self-inverting permutations. In

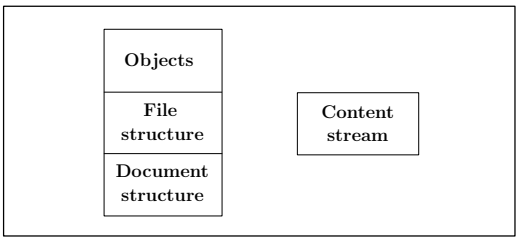


Figure 7.2: Components of a PDF file.

particular, we have proposed two such encoding algorithms: the algorithm `Encode_SiP.to.RPG-I` applies to any permutation π and relies on domination relations on the elements of π whereas the algorithm `Encode_SiP.to.RPG-II` applies to a self-inverting permutation π^* produced in any way and relies on the decreasing subsequences of π^*. Figure 7.1 summarizes by an example the representations of the permutation $\pi^* = (4, 7, 6, 1, 5, 3, 2)$ over the set N_7.

7.2.1 Structure of a PDF Documents

The Portable Document Format (PDF) [4] is an open standard (defined in ISO 32000) which facilitates device and platform independent capture and representation of rich information such as text, multimedia and graphics, into a single medium. Thus the PDF format enables viewing and printing of a rich document, independent of either application software or hardware. In this section we present a structural analysis of a PDF file, by giving its basic components. Figure 7.2 shows the main components of a PDF file are. which we briefly present below.

Object. An object is the basic element in PDF files, in which eight kinds of objects, namely Boolean Object, Numeric Object, String Object, Name Object, Array Object, Null Object, Dictionary and Stream Object are sustained. Objects may be labeled so that they can be referred to by other objects. A labeled object is called an indirect object.

File structure. The PDF file structure determines how objects are stored in a PDF file, how they are accessed, and how they are updated. The file structure (see, Figure 7.6) includes the following:

- an one-line header identifying the version of the PDF specification to which the file conforms,
- a body containing the objects that make up the document contained in the file,
- a cross-reference table containing information about the indirect objects in the file, and
- a trailer giving the location of the cross-reference table and of certain special objects within the body of the file.

Figure 7.6 shows an example of a PDF file and its internal file structure.

Document structure. The PDF document structure specifies how the basic object types are used to represent components of a PDF document: pages, fonts, annotations, and so forth. The document structure of PDF file is organized in the shape of an object tree topped by

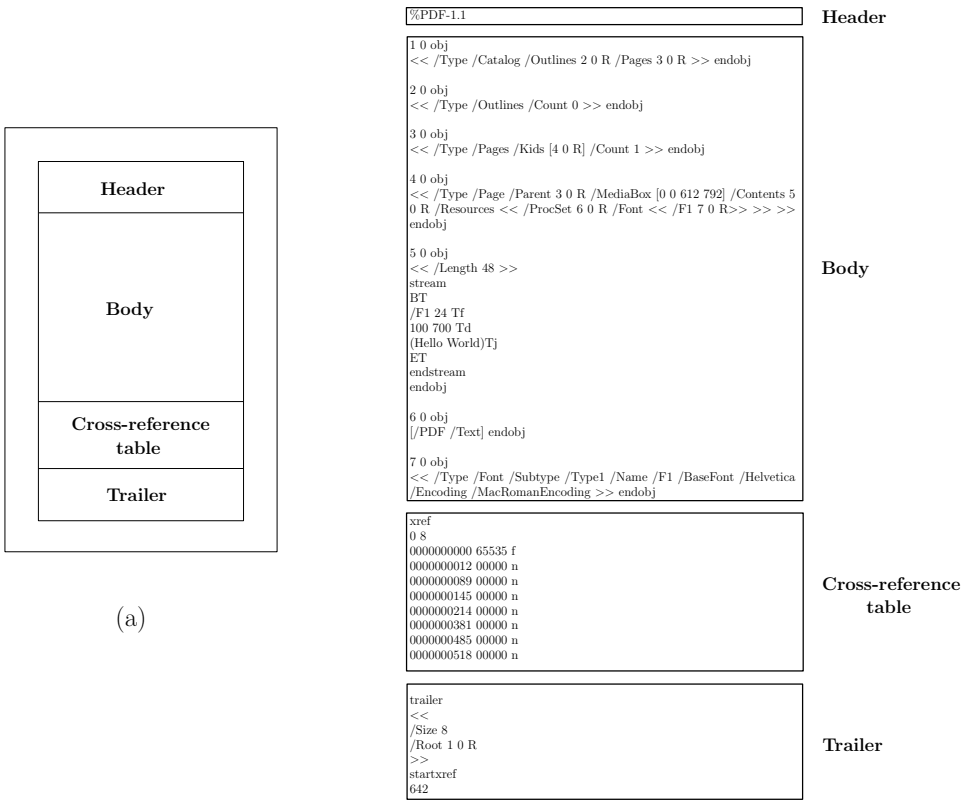


Figure 7.3: (a) The structure of a PDF file; (b) The code of a PDF file containing, in object `50obj`, the text "Hello World".

Catalog, Page tree, Outline hierarchy and Article thread included. The Outline hierarchy is the bookmarker of PDF, and Page tree includes page and Pages which in turn includes the total page number and each page marker. Page, the main body of PDF file, is the most important object which involves the typeface applied, the text, pictures, page size and so on. The organization of other objects is analogous to Page tree. With the object tree topped by Catalog, any object in PDF file can be visited. Figure 7.4 illustrates the structure of the object hierarchy.

7.3 Watermarking PDF Documents

In this section we describe embedding algorithms for encoding a self-inverting permutation π^* into a digital document T. More specifically, we embed the permutation π^* into a PDF document by exploiting (i) the one-dimensional representation of π^*, (ii) the two-dimensional representation of a π^*, and (iii) the encoding of π^* as a reducible permutation graph $F^*[\pi^*]$.

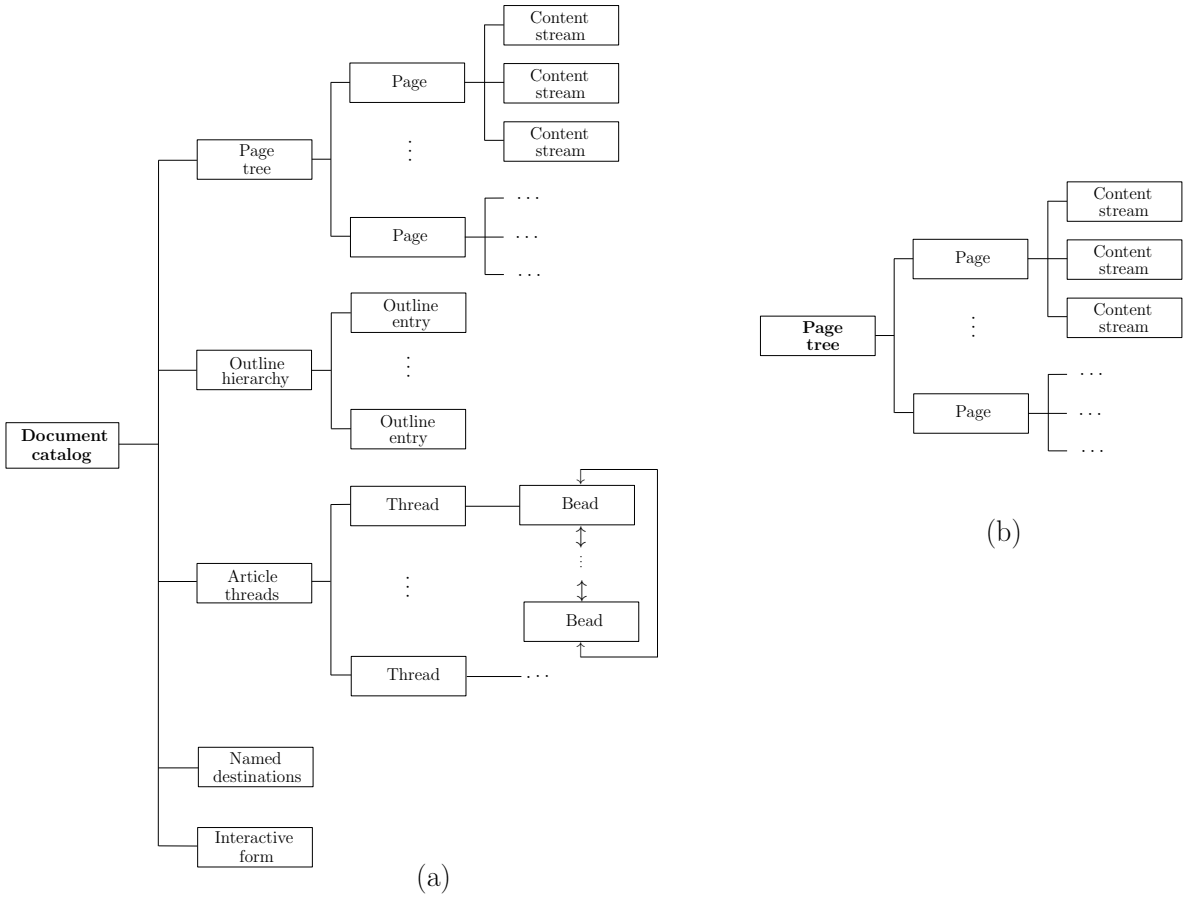


Figure 7.4: (a) The main structural components of a PDF file; (b) The document structure of PDF file.

7.3.1 Embed Watermark into PDF - I

We first design an embedding algorithm for watermarking a PDF document by exploiting the 1D-representation of a permutation π^*. The marking is performed by increasing the space between two consecutive words in a paragraph of T.

Let B^* be the 1D array of size $n = n^* \times n^*$ which represents the permutation π^* of length n^*, and let (w_1, s_1), (w_2, s_2), ..., (w_n, s_n) be n pairs of type "word-space" of a paragraph *par* of the input PDF document; recall that the entry $B^*((i-1)n^* + \pi_i^*)$ contains the symbol "*", $1 \leq i \leq n^*$. The algorithm increases by a small value "c" the i-th space of the pair $(\text{word}_i, \text{space}_i)$ if $B^*((i-1)n^* + \pi_i^*) =$ "$*$". We next give a high-level description, with respect to DPF modification, of our proposed embedding algorithm.

Algorithm `Embed_SiP.to.PDF-I`

1. Compute the 1DM representation of the permutation π^*, i.e., construct the array B^* of size $n = n^* \times n^*$ where the $(i-1)n^* + \pi_i^*$ entry of B^* contains the symbol "*", $1 \leq i \leq n^*$;

2. Select an appropriate paragraph *par* on a page P of PDF document T to embed the self-inverting permutation π^*;

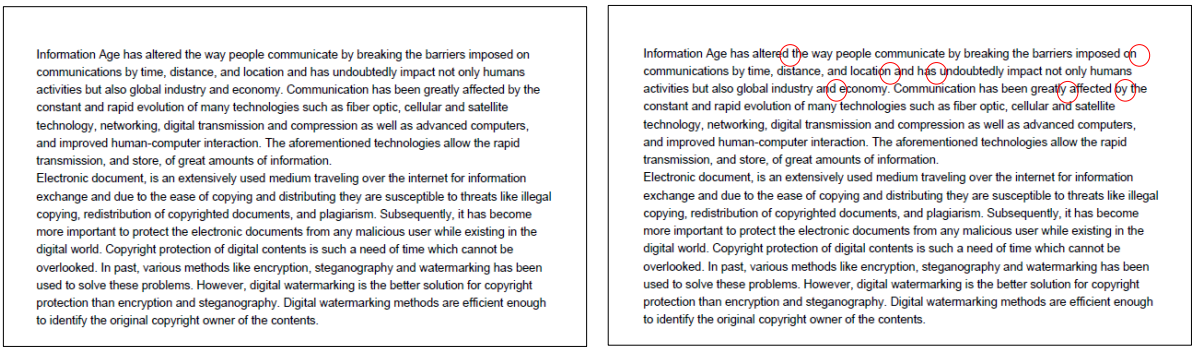


Figure 7.5: The initial PDF document T and watermarked PDF document T_w using the 1D representation of permutation $\pi^* = (4, 7, 6, 1, 5, 3, 2)$.

3. Partition the paragraph *par* into n pairs $(w_1, s_1), (w_2, s_2), \ldots, (w_n, s_n)$, where w_i and s_i are the i-th word and space, respectively, in selected paragraph *par*, $1 \leq i \leq n$;

4. For each pair (w_i, s_i) s.t. $B^*((i-1)n^* + \pi_i^*) =$ "$*$", increases the space s_i or, equivalently, distance $d(w_i, w_{i+1})$ between words w_i and w_{i+1}, by a relative small value c, $1 \leq i \leq n$;

5. Return the watermarked PDF document T_w.

Extraction. The extraction algorithm, which we call `Extract_PDF.from.SiP-I`, operates in a usual manner: it takes as input the watermarked PDF document T_w, locate the paragraph *par*, and compute the permutation π^* by finding the positions of the words w_i such that:

- $d(w_i, w_{i+1}) > d(w_{i-1}, w_i)$, or
- $d(w_i, w_{i+1}) > d(w_{i+1}, w_{i+2})$

where, $d(w_i, w_j)$ is the distance between words w_i and w_j in a paragraph *par* of T_w, $1 \leq i \leq n$; note that, an appropriate paragraph *par* contains more that n words.

7.3.2 Embed Watermark into PDF - II

In this section we describe a different approach of embedding algorithm a self-inverting permutation π^* into a digital document T, by exploiting the two-dimensional representation of permutation π^*.

The main idea behind the embedding algorithm, which we call `Embed_SiP.to.PDF-II`, is similar of that of algorithm `Embed_SiP.to.Image-F` (see, Section 6.4). The most important of this idea is the fact that it suggests a way in which the permutation π^* can be represented with a 2D-representation and since pages of a PDF documents T are two dimensional objects that representation can be efficiently marked on them resulting the watermarked PDF document T_w; in a similar way as in our image watermarking approach, such a 2D-representation can be

Figure 7.6: The initial PDF document T and watermarked PDF document T_w using the 2D representation of permutation $\pi^* = (4, 7, 6, 1, 5, 3, 2)$.

efficiently extracted for a watermarked PDF document T_w and converted back to the self-inverting permutation π^*.

Let A^* be the 2D matrix of size $n^* \times n^*$ which represents the permutation π^* of length n^*. The marking of the input PDF document T is performed by selecting an appropriate page P of T and setting n^* objects (e.g., characters, symbols, images) in a specific positions on page P, $1 \leq i \leq n^*$. In fact, we set an object O_i in position with (x'_i, y'_i) coordinates on page P if $A^*(x_i, y_i) =$ " $*$ ", where $1 \leq x_i, y_i \leq n^*$ and $0 \leq x'_i, y'_i \leq size(P)$; note that, $(0, 0)$ is the lower-left point (or, equivalently, the bottom-left corner) of page P.

The algorithm takes as input a SiP π^* and a PDF document T, and returns the watermarked document T_w; it consists of the following steps.

Algorithm `Embed_SiP.to.PDF-II`

1. Compute the 2DM representation of the self-inverting permutation π^*, i.e., construct an array A^* of size $n^* \times n^*$ s.t. the entry $A^*(i, \pi^*_i)$ contains the symbol "*", $1 \leq i \leq n^*$;

2. Select an appropriate page P to embed the permutation π^* and compute the size $size(P)$ of the page P, say, $N \times M$;

3. Segment the PDF page P into $n^* \times n^*$ grid-cells C_{ij} of size $\lfloor \frac{N}{n^*} \rfloor \times \lfloor \frac{M}{n^*} \rfloor$, $1 \leq i, j \leq n^*$;

4. For each grid-cell C_{ij} s.t. $A^*(i, j) =$ " $*$ ", mark the cell C_{ij} by setting a symbol, with an appropriate calor, in any position inside C_{ij} of P, $1 \leq i, j \leq n^*$, resulting thus the marked document T_w;

5. Return the watermarked PDF document T_w.

Extraction. The algorithm which extracts the permutation π^* from the watermarked PDF T_w operates in a similar way as the corresponding extraction algorithm for images: it takes the input watermarked image I_w, locate the marked page P, computes its $N \times M$ size, and segments P into $n^* \times n^*$ grid-cells C_{ij} of size $\lfloor \frac{N}{n^*} \rfloor \times \lfloor \frac{M}{n^*} \rfloor$; then, it computes the permutation π^* by finding the coordinates (x_i, y_i) of the n^* symbols in the page P, $1 \leq i \leq n^*$; it is called `Extract_PDF.from.SiP-II`.

7.3.3 Embed an RPG into a PDF

In this section we describe a watermarking algorithm for embedding a self-inverting permutation π^* into a PDF document T, by exploiting the graph representation of π^* proposed in this thesis and the structure of a PDF document T described in this chapter.

Indeed, in Chapter 3 we have presented two algorithms, namely `Encode_SiP.to.RPG-I` and `Encode_SiP.to.RPG-II`, for encoding self-inverting permutations π^* as reducible permutation graphs $F[\pi^*]$ (see, Section 3.5), while in this chapter we have described the document structure $\mathrm{DS}(T)$ of a PDF document T (see, Subsection 7.2.1); note that, the document structure of a PDF file always contains a node, namely `Document-catalog`, and a page tree $\mathrm{PT}(T)$ rooted at node `Page-tree`, denoted by `root(pt)`; see, Figure 7.4(b).

In light of the two embedding algorithms `Encode_SiP.to.RPG-I` and `-II`, we next present an algorithm for embedding a reducible permutation graph $F[\pi^*]$ into a PDF document T. The main idea behind the proposed embedding algorithm is a systematic addition of appropriate object-references in selected nodes of the page-tree $\mathrm{PT}(T)$ of the document structure $\mathrm{DS}(T)$, through the use of entries of type /`Kye`$(\cdot)$, so that the graph $F[\pi^*]$ can be easily constructed from the page-tree $\mathrm{PT}(T^*)$ of the resulting watermarked document T^*.

Let $F[\pi^*]$ be a reducible permutation graph produced by one of our two embedding algorithms (i.e. `Encode_SiP.to.RPG-I` or `-II`), and let $u_{n+1}, u_n, \ldots, u_1, u_0$ be the nodes of the graph $F[\pi^*]$; note that, $F[\pi^*]$ does not contain the back-edge (u_0, u_{n+1}). In order to simplify the extraction process, the graph $F[\pi^*]$ which is embedded into a PDF document T contains one extra back-edge, i.e., the edge (u_0, u_{n+1}); see, Step 4 of the embedding algorithm.

The proposed algorithm, which we call `Encode_RPG.to.PDF`, for embedding a reducible permutation graph $F[\pi^*]$ into a PDF document T is described below.

Algorithm `Encode_RPG.to.PDF`

1. Compute the document structure $\mathrm{DS}(T)$ of the input PDF document T and locate its page-tree $\mathrm{PT}(T)$; let `node(dc)` be the document catalog node of structure $\mathrm{DS}(T)$ and `root(pt)` be the root node of the page tree $\mathrm{PT}(T)$; see, Figure 7.4(b);

2. Compute a path $O(T) = (v_{n+1}, v_n, \ldots, v_1, v_0)$ on $n+2$ nodes (i.e., objects) of the page-tree $\mathrm{PT}(T)$ s.t. $v_{n+1} = root(pt)$, and set $s = v_{n+1}$ and $t = v_0$;

3. Assign an exact pairing (i.e., 1-1 correspondence) of the $n+2$ nodes of path $O(T)$ to the nodes $u_{n+1}, u_n, \ldots, u_1, u_0$ of the watermark graph $F[\pi^*]$;

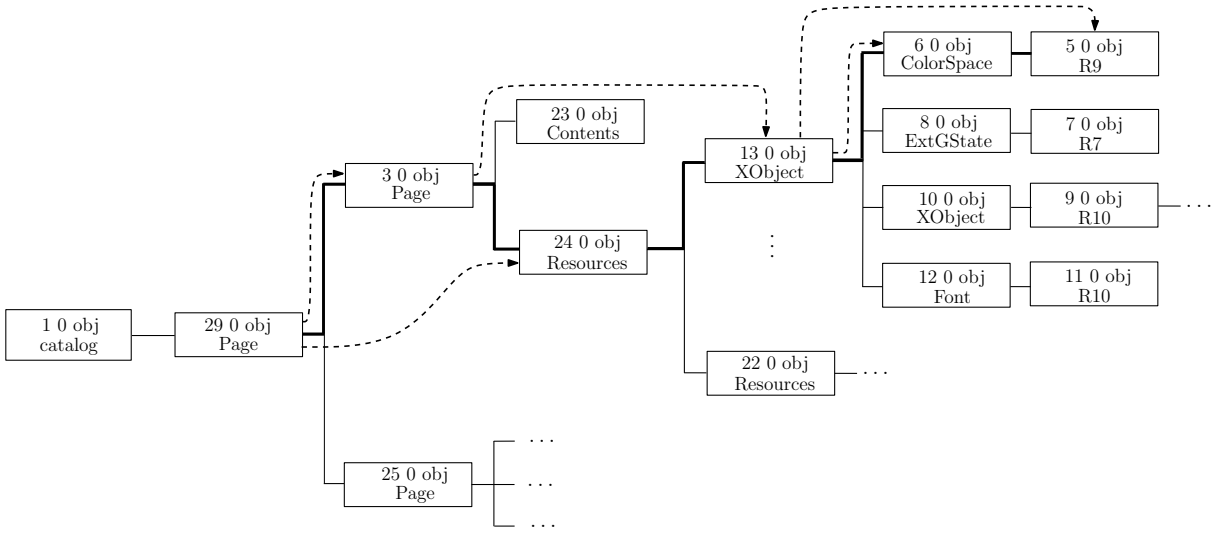


Figure 7.7: The watermarked $DS(T^*)$ which encodes the RPG of $\pi^* = (4, 5, 3, 1, 2)$.

4. For each back-edge (u_i, u_j) of the graph $F[\pi^*]$ (i.e., $u_j > u_i$), add the forward-edge (v_j, v_i) in page-tree PT(T) by adding in object [$\mathtt{v_j}$ 0 obj] an entry of type /Key($\mathtt{v_i}$ 0 R); add in object [$\mathtt{v_{n+1}}$ 0 obj] an entry of type /Key($\mathtt{v_0}$ 0 R);

5. Return the modified PDF document T which is the watermarked document T^*.

Let us briefly discuss the way we add forward-edge in the page-tree PT(T); recall that, in Step 4 of the previous algorithm Encode_RPG.to.PDF we add the forward-edge (v_j, v_i) in page-tree PT(T) by adding in object [$\mathtt{v_j}$ 0 obj] an entry of type /Key($\mathtt{v_i}$ 0 R). The entry /Key($\mathtt{v_i}$ 0 R) may be of various types; note that, /Key($\cdot$) is used as parameter in our algorithm's description.

In our implementation, for the forward-edge (v_j, v_i) such that the object [$\mathtt{v_j}$ 0 obj] is not the rood-node root(pt) of the page-tree PT(T), we always chose the entry /Key($\mathtt{v_i}$ 0 R) which we add in object [$\mathtt{v_j}$ 0 obj] to be of the same type of object [$\mathtt{v_i}$ 0 obj]. In the case where v_j = root(pt), we chose the entry /Key($\mathtt{v_i}$ 0 R) to be of type /Kids($\cdot$).

For example, in Figure 7.7 we have added forward-edges from object [29 0 obj] to object [3 0 obj], from object [29 0 obj] to object [24 0 obj], from object [3 0 obj] to object [13 0 obj], etc. Thus, in our implementation we have added in the rood-node object [29 0 obj] the entries /Kids(3 0 R) and /Kids(24 0 R), in object [3 0 obj] the entry /XObject(13 0 R), while in object [13 0 obj] the entries /ColorSpace(6 0 R) and /R9(5 0 R).

Remark 7.1. Let T be a PDF file and let PT(T) be a page-tree of the document structure DS(T). A node of the page-tree PT(T) may contain several entries /Kye($\cdot$) of various types. We mention that, some types are required for the entries in specific nodes of PT(T); for example, the required entries in the rood-node root(pt) of the page-tree PT(T) are the following four: /Type($\cdot$), /Parent($\cdot$), /Kids($\cdot$), and /Count($\cdot$).

Extraction. We next describe the corresponding extraction algorithm which extracts the graph

$F[\pi^*]$ from the PDF document T^* watermarked by the embedding algorithm `Encode_RPG.to.PDF`; the extraction algorithm, which we call `Extract_RPG.from.PDF`, works as follows:

- Take first as input the PDF document T^* watermarked by the embedding algorithm `Encode_RPG.to.PDF`, compute the document structure $\mathrm{DS}(T^*)$ of T^*, and locate its page tree $\mathrm{PT}(T^*)$; then, find in object $root(pt)$, where $root(pt)$ is the root of the tree $\mathrm{PT}(T^*)$, the entry `/Kids(`v_k `0 R)` s.t. v_k is not a child of $root(pt)$, and set $v_{n+1} = root(pt)$ and $v_0 = v_k$;

- Compute the path $O(T) = (v_{n+1}, v_n, \ldots, v_1, v_0)$ of $\mathrm{PT}(T^*)$, from node $root(pt)$ to v_0, and assign an exact pairing (i.e., 1-1 correspondence) of the $n+2$ nodes of path $O(T)$ to the nodes $u_{n+1}, u_n, \ldots, u_1, u_0$ of a graph $F[\pi^*]$; initially, $E(F[\pi^*]) = \emptyset$;

- Add edges (u_{i+1}, u_i) in $F[\pi^*]$ for $i = n, n-1, \ldots, 0$, and the edge (u_i, u_j) iff (v_i, v_j) is a forward edge in the page tree $\mathrm{PT}(T^*)$;

- Delete the edge (u_{n+1}, u_0) from the graph $F[\pi^*]$;

- Return the graph $F[\pi^*]$;

It is easy to see that, by construction the returned graph $F[\pi^*]$ is a reducible permutation graph produced by either algorithm `Encode_SiP.to.RPG-I` or algorithm `Encode_SiP.to.RPG-II`. Thus, $F[\pi^*]$ has the following property: the structure which results after deleting

(i) all the forward edges (u_{i+1}, u_i) of $F[\pi^*]$, $0 \leq i \leq n$, and

(ii) the node u_0

is either the tree $T_d[\pi^*]$ or tree $T_s[\pi^*]$ produced during the execution of either the decoding algorithm `Decode_RPG.to.SiP-I` or algorithm `Decode_RPG.to.SiP-II`, respectively (see, Figures 3.3 and 3.4). Thus, we can efficiently extract the self-inverting permutation π^* embedded into a PDF document T by algorithm `Encode_RPG.to.PDF`.

7.4 Concluding Remarks

In this chapter we presented embedded algorithms, along with their corresponding extraction algorithms, for embedding watermark numbers w into PDF documents T using three different representations of a self-inverting permutation π^*, namely 1D-representation, 2D-representation, and RPG-representation; note that, RPG-representation means the encoding of permutation π^* as a reducible permutation graph $F^*[\pi^*]$.

The main features of our algorithms, i.e., the way they mark a PDF document T or, equivalently, the way they embed a self-inverting permutation π^* into document T, are summarized as follows:

- In the first algorithm `Embed_SiP.to.PDF-I` the marking of a PDF document T is performed by increasing the distance (or, space) between two consecutive words in a paragraph of T.

- The main idea behind the second algorithm `Embed_SiP.to.PDF-II` based on the fact that π^* has a 2D-representation and, since pages of PDF documents T are two dimensional objects, it can be efficiently used to mark specific positions on a page of T resulting thus the watermarked PDF document T^*.

- The third graph-based embedding algorithm `Encode_RPG.to.PDF` uses a different approach: it exploits the structure of a PDF document T and embeds the graph $F[\pi^*]$ into T by adding appropriate object-references in the document T, through the use of entries of type /`Kids`(`k 0 R`), so that the graph $F[\pi^*]$ can be easily constructed from the page tree $\mathrm{PT}(T^*)$ of the resulting watermarked document T^*.

In light of our graph-based embedding algorithm `Encode_RPG.to.PDF` it would be very interesting to investigate the possibility of altering other components of the document structure of a PDF file in order to embed the graph $F[\pi^*]$; we leave it as a direction for future work.

Moreover, an interesting open question is whether the approach and techniques used in this chapter can help develop efficient encoding algorithms having "better" properties with respect text attacks; we leave it as an open problem for future investigation.

Chapter 8

Conclusions and Future Work

8.1 Encoding Numbers as SiPs and RPGs

8.2 Software Watermarking

8.3 Image and Audio Watermarking

8.4 Text Watermarking

8.1 Encoding Numbers as SiPs and RPGs

In Chapter 2, 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 of w. We also presented the corresponding decoding algorithm; it takes as input a self-inverting permutation π^* produced by the encoding algorithm and returns the encoding 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 encoding approach enable us to encode any integer w as self-inverting permutation π^* of any length $n^* \geq 3$; indeed, π^* can be constructed over the set N_{n^*}, where $n^* = 2\lceil \log w \rceil + 1$.

In Chapter 3, we proposed an efficient and easily implementable codec system for encoding watermark numbers as graph structures. In particular, we proposed an efficient codec method for encoding a self-inverting permutation π^* as a reducible permutation graph $F[\pi^*]$; the proposed flow-graph $F[\pi^*]$ can be efficiently used for software watermarking since its structure mimics real codes.

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

In light of the two main data components of our codec system, i.e., the permutation π^* and the graph $F[\pi^*]$, 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 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, the evaluation of our codec algorithms and structures under other watermarking measurements in order to obtain detailed information about their practical behavior is an interesting problem for future study.

In Chapter 4, we proposed an efficient algorithm which encodes a self-inverting permutation π^* into several cographs $C_1[\pi^*], C_2[\pi^*], \ldots, C_n[\pi^*]$, $n \geq 2$, and an efficient transformation of a cograph into a reducible permutation graph $F[\pi^*]$. In light of our encoding algorithms which encode a watermark integer w as a self-inverting permutation π^* [28] and the permutation π^* into many different cographs, we conclude that we can efficiently encode the same watermark integer w into several reducible permutation graphs $F_1[\pi^*], F_2[\pi^*], \ldots, F_n[\pi^*]$, $n \geq 2$.

It is worth noting that this property causes a codec watermarking system resilient to attacks since we can embed multiple copies of the same watermark value w into an application program.

An interesting open question is whether the approach and techniques used in this chapter can help develop efficient codec algorithms and graph structures having "better" properties with respect to resilience, size, and/or time and space efficiency; we leave it as an open problem for future investigation.

8.2 Software Watermarking

In Chapter 5, through the evaluation of WaterRPG, we showed that our model has zero false positive and false negative rates in the case where the watermarked code has not been attacked. Indeed, it is true because the execution of the watermarked program P^* with the secret input sequence always builds a call graph $G(P^*, I_{key})$ which is isomorphic with the water-graph $F[\pi^*]$.

The execution time and space overhead varies depending on the size of the embedded watermark; in fact, the overhead increases linearly in the size of the water-graph $F[\pi^*]$. It is worth noting that the data-rate is directly correlated with the number of functions used or, equivalently, with the size of the water-graph. In the case where the code (in bits) of the original program P is large enough, our model has high data-rate and extremely low embedding overhead. We point out that the number of nodes of the water-graph $F[\pi^*]$ affects the number of functions we use for embedding. Thus, it is possible to use fewer functions which would result in a graph $F[\pi^*]$ with fewer nodes; note that, the graph $F[\pi^*]$ on $n = 2k + 1$ nodes can encode a watermarking integer w in the range $[0, 2^{k-1} - 1]$; see, authors' work [28, 23].

Furthermore, in our model the code which is associated with the watermark is composed both by new code and host code; this enable us to obtain high stealth watermarked programs P^*. Moreover, since the watermark code has become an indispensable piece of the functionality of program P^*, a malicious user would need to fully understand the operations of P^* in order to intervene changing possible execution flows. On the other hand, the extraction of our watermark takes into account and uses the traces of all the functions that are assigned to the nodes of the

water-graph $F[\pi^*]$ which, in turn, means that if a subset of these functions is intercepted then the watermark can not be extracted; unfortunately, this implies a poor part protection of our watermarked program P^*.

Finally, the experimental results show the high functionality of all the Java programs P^* watermarked under both the naive and stealthy cases, and also their low time complexity. The experiments also show that the watermarking approaches supported by our model can help develop efficient watermarked Java programs with respect to various and broadly used performance and resilience watermarking criteria.

Closing, we note that in light of our dynamic watermarking model WaterRPG it would be very interesting to compare it with other dynamic, or even static, already proposed software watermarking models [35, 31, 86, 95, 106]; we leave it as a direction for future work.

8.3 Image and Audio Watermarking

In Chapter 6, we proposed watermarking models for embedding invisible watermarks into digital images and audio signals. We presented methods for embedding invisible watermarks into images and their intention is to prove the authenticity of an image. The watermarks are given in numerical form, transformed into self-inverting permutations, and embedded into an image by partially marking the image in the frequency domain; more precisely, thanks to 2D representation of self-inverting permutations, we locate specific areas of the image and modify their magnitude of high frequency bands by adding the least possible information ensuring robustness and imperceptiveness.

We experimentally tested our embedding and extracting algorithms on color JPEG images with various and different characteristics; we obtained positive results as the watermarks were invisible, they didn't affect the images' quality and they were extractable despite the JPEG compression. In addition, the experimental results show an improvement in comparison to the initial obtained results on spatial domain and they also depict the validity of our proposed codec algorithms.

It is worth noting that the proposed algorithms on image watermarking in frequency domain are robust against cropping or rotation attacks since the watermarks are in SiP form, meaning that they determine the embedding positions in specific image areas. Thus, if a part is being cropped or the image is rotated, SiP's symmetry property may allow us to reconstruct the watermark. Furthermore, our codec algorithms can also be modified in the future to get robust against scaling attacks. That can be achieved by selecting multiple widths concerning the ellipsoidal annuli depending on the size of the input image.

Finally, we should point out that the study of our quality function f remains a problem for further investigation; indeed, f could incorporate learning algorithms [102] so that to be able to return the c_{opt} accurately and in a very short computational time.

Additionally, in this chapter we presented an audio watermarking technique which efficiently and invisibly embeds information, i.e., watermarks, into an audio digital signal. Our technique is based on the same main idea of image watermarking technique expanding thus the digital objects that can be efficiently watermarked through the use of self-inverting permutations.

We experimentally tested our embedding and extracting algorithms on WAV audio signals.

Our testing procedure includes the phases of embedding a numerical watermark $w = \pi^*$ into several audio signals S, storing the watermarked audio S_w in WAV format, and extracting the watermark $w = \pi^*$ from the audio S_w. We obtained positive results as the watermarks were invisible, they didn't affect the audio's quality and they were extractable.

The performance evaluation of our audio watermarking technique on several other attacks remains a problem for further investigation.

8.4 Text Watermarking

In Chapter 7, we presented embedded algorithms, along with their corresponding extraction algorithms, for embedding watermark numbers w into PDF documents T using three different representations of a self-inverting permutation π^*, namely 1D-representation, 2D-representation, and RPG-representation; note that, RPG-representation means the encoding of permutation π^* as a reducible permutation graph $F[\pi^*]$.

The main features of our algorithms, i.e., the way they mark a PDF document T or, equivalently, the way they embed a self-inverting permutation π^* into document T, are summarized as follows:

- In the first algorithm `Embed_SiP.to.PDF-I` the marking of a PDF document T is performed by increasing the distance (or, space) between two consecutive words in a paragraph of T.
- The main idea behind the second algorithm `Embed_SiP.to.PDF-II` based on the fact that π^* has a 2D-representation and, since pages of a PDF documents T are two dimensional objects, it can be efficiently used to mark specific positions on a page of T resulting thus the watermarked PDF document T^*.
- The third graph-based embedding algorithm `Encode_RPG.to.PDF` uses a different approach: it exploits the structure of a PDF document T and embeds the graph $F[\pi^*]$ into T by adding appropriate object-references in the document T, through the use of statements of type `/Kids(k 0 R)`, so that the graph $F[\pi^*]$ can be easily constructed from the page tree $\mathrm{PT}(T^*)$ of the resulting watermarked document T^*.

In light of our graph-based embedding algorithm `Encode_RPG.to.PDF` it would be very interesting to investigate the possibility of altering other components of the document structure of a PDF file in order to embed the graph $F[\pi^*]$; we leave it as a direction for future work.

Moreover, an interesting open question is whether the approach and techniques used in this chapter can help develop efficient encoding algorithms having "better" properties with respect text attacks; we leave it as an open problem for future investigation.

Bibliography

[1] G. Arboit. A method for watermarking Java programs via opaque predicates. Proc. of the 5th Int'l Conference on Electronic Commerce Research (ICECR-5), 2002.

[2] N. Ahmed, T. Natarajan and K.R. Rao. Discrete cosine transform. IEEE Transactions on Computers (C-23), 90–93, 1974.

[3] M.A. Alsalami, and M.M. Al-Akaidi. Digital audio watermarking: survey. De Montfort University, 1–14, 2003.

[4] Adobe Systems Incorporated. Adobe Portable document format Version 1.7, http://www.adobe.com, Nov. 2006.

[5] M.J. Atallah, V. Raskin, C.F. Hempelmann, M. Karahan, R. Sion, U. Topkara, and K.E. Triezenberg. Natural language watermarking and tamperproofing. LNCS 5, Springer, 196–212, 2003.

[6] T. Amano, and D. Misaki. A feature calibration method for watermarking of document images. Proc. of the 5th In'l Conference on Document Analysis and Recognition (ICDAR'99), IEEE, 91–94, 1999.

[7] A. Bretscher, D. Corneil, M. Habib, and C. Paul. A simple linear time LexBFS cograph recognition algorithm. SIAM J. Discrete Math. 22, 1277–1296, 2008.

[8] W. Bender, D. Gruhl, and N. Morimoto. Techniques for data hiding. Proc. of the IBM systems journal 35(3-4), 313–336, 1996.

[9] J.T. Brassil, S. Low, N.F. Maxemchuk, and L.O. Gorman. Electronic Marking and Identification Techniques to Discourage Document Copying. IEEE Journal on Selected Areas in Communications 13(8), 1495–1504, 1995.

[10] J.T. Brassil, S. Low, N.F. Maxemchuk, L.O. Gorman. Hiding information in document images. Proc. of the 29th Annual Conference on Information Sciences and Systems, Johns Hopkins University, 482–489, 1995.

[11] J.T. Brassil, S. Low, and N.F. Maxemchuk. Copyright protection for the electronic distribution of text documents. Proc. of the IEEE 87(7), 1181–1196, 1999.

[12] G.S. Bindra. Invisible communication through Portable Document File (PDF) format. Proc. of the 7th Int'l Conference on Intelligent Information Hiding and Multimedia Signal Processing (IIH-MSP), 173–176, 2011.

[13] G.S. Bindra. Masquerading as a trustworthy entity through Portable Document File (PDF) format. Proc. of the 2011 IEEE Int'l Conference on PASSAT, and IEEE Int'l Conference on Social Com., Boston, USA, 2011.

[14] A. Brandstädt, V.B. Le, and J. Spinrad. Graph classes: a survey. SIAM Monographs on Discrete Mathematics and Applications 3, 1999.

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

[16] M. Chroni, A. Fylakis, and S.D. Nikolopoulos. Watermarking Digital Images in the Frequency Domain: Performance and Attack Issues, LNBIP 189, Springer, 68–84, 2014.

[17] M. Chroni, A. Fylakis, and S.D. Nikolopoulos. From image to audio watermarking using self-inverting permutations. Proc. of the 10th Int'l Conference on Web Information Systems and Technologies (WEBIST'14), SciTePress, 177–184, 2014.

[18] M. Chroni, A. Fylakis, and S.D. Nikolopoulos. Watermarking images in the frequency domain by exploiting self-inverting permutations. Journal of Information Security 4(2), 80–91, 2013.

[19] I. Chionis, M. Chroni, and S.D. Nikolopoulos. A dynamic watermarking model for embedding reducible permutation graphs into software. Proc. of the 10th Int'l Conference on Security and Cryptography (SECRYPT'13), SciTePress, 74–85, 2013.

[20] I. Chionis, M. Chroni, and S.D. Nikolopoulos. Evaluating the WaterRpg software watermarking model on Java application programs. Proc. of the 17th Panhellenic Conference on Informatics (PCI'13), ACM, 144–151, 2013.

[21] M. Chroni and S.D. Nikolopoulos. Design and evaluation of a graph codec system for software watermarking. Proc. of the 2nd Int'l Conference on Data Management Technologies and Applications (DATA'13), SciTePress, 277–284, 2013.

[22] M. Chroni, A. Fylakis, and S.D. Nikolopoulos. Watermarking images in the frequency domain by exploiting self-inverting permutations. Proc. of the 9th Int'l Conference on Web Information Systems and Technologies (WEBIST'13), SciTePress, 45–54, 2013, (Best Student Paper Award).

[23] M. Chroni and S.D. Nikolopoulos. An efficient graph codec system for software watermarking. Proc. of the 36th Int'l Conference on Computers, Software, and Applications (COMPSAC'12), Workshop STPSA'12, IEEE, 595–600, 2012.

[24] M. Chroni and S.D. Nikolopoulos, Multiple encoding of a watermark number into reducible permutation graphs using cotrees. Proc. of the 13th Int'l Conference on Computer Systems and Technologies (CompSysTech'12), ACM, 118–125, 2012.

[25] M. Chroni, A. Fylakis, and S.D. Nikolopoulos. Watermarking images using 2D representations of self-inverting permutations. Proc. of the 8th Int'l Conference on Web Information Systems and Technologies (WEBIST'12), SciTePress, 380–385, 2012.

[26] M. Chroni and S.D. Nikolopoulos. An embedding graph-based model for software watermarking. Proc. of the 8th Int'l Conference on Intelligent Information Hiding and Multimedia Signal Processing (IIH-MSP'12), IEEE, 261–264, 2012.

[27] M. Chroni and S.D. Nikolopoulos. Encoding watermark numbers as cographs using self-inverting permutations. Proc. of the 12th Int'l Conference on Computer Systems and Technologies (CompSysTech'11), ACM ICPS 578, 142–148, 2011.

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

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

[30] C. Collberg, S. Kobourov, E. Carter, and C. Thomborson. Error-correcting graphs for software watermarking. Proc. of the 29th Workshop on Graph-Theoretic Concepts in Computer Science (WG'03), LNCS 2880, 156–167, 2003.

[31] 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.

[32] C. Collberg and C. Thomborson. Software watermarking: models and dynamic embeddings. Proc. of the 26th ACM SIGPLAN-SIGACT on Principles of Programming Languages (POPL'99), 311–324, 1999.

[33] 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.

[34] C.S. Collberg, C.D. Thomborson,J.J. Horning, W.O. Silbert, L.R. Matheson, A.K. Wright, and S.S. Owicki. Software watermarking techniques. US Patent, 0214188, 2011.

[35] C. Collberg, E. Carter, S. Debray, A. Huntwork, J. Kececioglu, C. Linn and M. Stepp. Dynamic path-based software watermarking. Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM SIGPLAN 39, 107–118, 2004.

[36] C. Collberg, and C. Thomborson. Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection. IEEE Trans. Software Engineering 28, 735–746, 2000.

[37] 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.

[38] D. Curran, N. Hurley and M. Cinneide. Securing Java through software satermarking, Proc. Proc. of the Int'l Conference on Principles and Practice of Programming in Java (PPPJ'03), 145–148, 2003.

[39] D.G. Corneil, Y. Perl, and L.K. Stewart. A linear recognition algorithm for cographs. SIAM J. Comput. **14**, 926–984, 1985.

[40] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. Introduction to Algorithms. MIT Press (2nd ed.), 2001.

[41] J.W. Cooley and J.W. Tukey. An algorithm for the machine calculation of complex fourier series. Mathematics of Computation C-23, 297–301, 1965.

[42] L. Chun-Shien, H. Shih-Kun, S. Chwen-Jye, M.L. Hong-Yuan. Cocktail watermarking for digital image protection. IEEE Trans. on Multimedia 4, 209–224, 2000.

[43] I.J. Cox, M.L. Miller, J.A. Bloom, J. Fridrich, and T. Kalker. Digital Watermarking and Steganography. Morgan Kaufmann (2nd ed.), 2008.

[44] I.J. Cox, J. Kilian, T. Leighton, and T. Shamoon. Secure Spread Spectrum Watermarking for Multimedia. IEEE Transactions on Image Processing 6, 1673–1687, 1997.

[45] I.J. Cox, G. Doërr, and T. Furon. Watermarking is not cryptography. Proc. of the Digital Watermarking, Springer, 1–15, 2006.

[46] S. Craver, N. Memon, B.L. Yeo, and M. Yeung. Resolving rightful ownerships with invisible watermarking techniques: limitations, attacks, and implications. IEEE Journal on Selected Areas in Communications (Special issue on Copyright and Privacy Protection), 16(4), 573–586, 1998.

[47] B. Chen, and G.W. Wornell. Quantization Index Modulation: A Class of Provably Good Methods for Digital Watermarking and Information Embedding. IEEE Transactions on Information Theory 47(4), 1423–1443, 2001.

[48] 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.

[49] J. C. Davis. Intellectual property in cyberspace - what technological / legislative tools are necessary for building a sturdy global information infrastructure? Proc. of the Int'l Symposium on Technology and Society, IEEE, 66–74, 1997.

[50] I.P. Goulden, and D.M. Jackson. Combinatorial Enumeration. New York, Wiley, 1983.

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

[52] M.C. Golumbic. Algorithmic Graph Theory and Perfect Graphs. Academic Press, New York (1980). Annals of Discrete Math. 57 (2nd ed.), Elsevier 2004.

[53] R.C. Gonzalez and R.E. Woods. Digital Image Processing. Prentice-Hall, 2007.

[54] R.C. Gonzalez, R.E. Woods, and S.L. Eddins. Digital Image Processing using Matlab. Prentice-Hall, 2003.

[55] S. Garfinkel. Web Security, Privacy and Commerce. O'Reilly (2nd ed.), 2001.

[56] R. Raysman, E.A.Pisacreta, and K.A. Adler. Intellectual Property Licensing: Forms and Analysis.Law Journal Press, 1999.

[57] S. Graham and P. Kessler and M. Mckusick. Gprof: A call graph execution profiler. ACM SIGPLAN Notices 17(6), 120–126, 1982.

[58] F. Harary, and E. Palmer. Graphical Enumeration. New York, Academic Press, 1973.

[59] M.S. Hecht and J.D. Ullman. Flow graph reducibility. Proc. of the 4th Annual ACM Symposium on Theory of Computing, ACM, 238–250, 1972.

[60] M.S. Hecht and J.D. Ullman. Characterizations of reducible flow graphs. Journal of the ACM 21(3), 367–375, 1974.

[61] F. Hartung, and M. Kutter. Multimedia watermarking techniques. Proc. of the IEEE 87(70), 1079–1107, 1999.

[62] A. Hore and D. Ziou. Image Quality Metrics: PSNR vs. SSIM. Proc. of the 20th Int'l Conference on Pattern Recognition, 2366–2369, 2010.

[63] R.W. Hamming. Error detecting and error correcting codes. Bell System Technical Journal 29(2), 147–160, 1950.

[64] D. Huang, and H. Yan. Interword distance changes represented by sine waves for watermarking text images. IEEE Trans. Circuits and Systems for Video Technology 11(12), 1237–1245, 2001.

[65] E.F. Hembrooke. Identification of sound and like signals. US Patent 3.004.104, Google Patents, 1961.

[66] V.K. Ingle and J.G. Proakis. Digital Signal Processing using Matlab. Cengage Learning (3rd ed.), 2010.

[67] Z. Jalil, and A.M. Mirza. An invisible text watermarking algorithm using image watermark. Proc. of the Innovations in Computing Sciences and Software Engineering, 147–152, 2010.

[68] K. Jain, M. Raghavan, and S. K. Jha. Study of the linkages between innovation and intellectual property. Proc. of the PICMET 2009, 1945–1953, 2009.

[69] D.E. Knuth. Fundamental Algorithms. The Art of Computer Programming: Vol.1, Addison-Wesley (3rd ed.), Reading, MA, USA, 1997.

[70] M. Kaur, S. Jindal, and S. Behal. A Study of Digital Image Watermarking. Journal of Research in Engineering and Applied Sciences 2, 126–136, 2012.

[71] N. Komatsu, and T. Hideyoshi. A proposal on digital watermark in document image communication and its application to realizing a signature. Electronics and Communications in Japan (Part I: Communications) 73(5), 22–33, 1990.

[72] B.S. Ko, R. Nishimura, and Y. Suzuki. Time-Spread Echo Method for Digital Audio Watermarking. IEEE Transactions on Multimedia 7(2), 212-221, 2005.

[73] M. Kutter, F. Jordan, and F. Bossen. Digital Watermarking of Color Images using Amplitude Modulation. Journal of Electronic Imaging 7(2), 326–332, 1998.

[74] H. Lerchs. On cliques and kernels. Department of Computer Science, University of Toronto, March 1971.

[75] V. Licks and R. Hordan. On Digital Image Watermarking Robust to Geometric Transformations. Proc. of the IEEE Int'l Conference on Image Proceesing 3, 690–693, 2000.

[76] H. Liu, L. Li, , J. Li, and J. Huang. Three novel algorithms for hiding data in pdf files based on incremental updates. In Digital Forensics and Watermarking, Springer Berlin Heidelberg, 167–180, 2012.

[77] X. Liu, Q. Zhang, C. Tang, J. Zhao, and J. Liu. A Steganographic Algorithm for Hiding Data in PDF Files Based on Equivalent Transformation. Int'l Symposiums on Information Processing (ISIP), 417–421, 2008.

[78] Y. Liu, X. Sun, and G. Luo. A novel information hiding algorithm based on structure of PDF document. Computer Engineering 32(17), 230–232, 2006.

[79] I.S. Lee, and W.H. Tsai. A new approach to covert communication via PDF files. Signal Processing 90(2), 557–565, 2010.

[80] S.H. Low, N.F. Maxemchuk, and A.M. Lapone. Document identification for copyright protection using centroid detection. IEEE Transactions on Communications 46(3), 372–381, 1998.

[81] S.H. Low, and N.F. Maxemchuk. Capacity of text marking channel. IEEE Signal Processing Letters 7(12), 345–347, 2000.

[82] P. Lu, Z. Lu, Z. Zhou, and J. Gu. An optimized natural language watermarking algorithm based on TMR. Proc. of the 9th Int'l Conference for Young Computer Scientists, 1459–1463, 2008.

[83] M. A. Lemley. Intellectual property, and free riding. Texas Law Review 83, 1031, 2005.

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

[85] S.A. Moskowitz and M. Cooperman. Method for stegacipher protection of computer code. US Patent 5.745.569, 1996.

[86] A. Monden, H. Iida, K. Matsumoto, K. Inoue and K. Torii. A practical method for watermarking Java programs, Proc. of the 24th Computer Software and Applications Conference (COMPSAC'00), 191–197, 2000.

[87] N.F. Maxemchuk, and S. Low. Marking text documents. Proc. of the IEEE Int'l Conference on Image Processing, Washington DC, 13–16, 1997.

[88] N.F. Maxemchuk, ans S.H. Low. Performance comparison of two text marking methods. IEEE Journal of Selected Areas in Communications 16(4), 561–572, 1998.

[89] N.F. Maxemchuk. Electronic document distribution. AT&T Technical Journal 73(5), 73–80, 1994.

[90] H.M. Meral, and B. Sankur, A. Özsoy, T. Güngör, and E. Sevinç. Natural language watermarking via morphosyntactic alterations. Computer Speech and Language 23(1), 107–125, 2009.

[91] H.M. Meral,, E. Sevinç, E. Ünkar, B. Sankur, A. Özsoy, and T. Güngör. Syntactic tools for text watermarking, Proc. of the 19th SPIE Electronic Imaging Conference on Security, Steganography, and Watermarking of Multimedia Contents, San Jose, CA, 2007.

[92] B. Macq, and O. Vybornova. A method of text watermarking using presuppositions. In Electronic Imaging 2007, Society for Optics and Photonics, 65051R–65051R–10, 2007.

[93] J. Nagra, C. Thomborson, and C. Collberg. A functional taxonomy for software watermarking. Australian Computer Science Communications 24(1), 177–186, 2002.

[94] S.D. Nikolopoulos. Coloring permutation graphs in parallel. Discrete Applied Mathematics 120, 165–195, 2002.

[95] J. Nagra, and C. Thomborson. Threading software watermarks. Proc. of the 6th Int'l Workshop on Information Hiding (IH'04), LNCS 3200, 208–223, 2004.

[96] J.J.K. O'Ruanaidh, W.J. Dowling, F.M. Boland. Watermarking digital images for copyright protection. Proc. of the Vision, Image and Signal Processing 143, IEEE, 250–256, 1996.

[97] D. Pascale. A Review of RGB Color Spaces ...from xyY to R'G'B'. The BabelColor Company, 2003.

[98] V.M. Potdar, S. Han, and E. Chang. A survey of digital image watermarking techniques. Proc. of the IEEE Third Int'l Conference on Industrial Informatics (INDIN), 709–716, 2005.

[99] F. Petitcolas. Image Database for Watermarking. Retrive from Petitcolas' personal web site (http://www.petitcolas.net/fabien/watermarking/), Retrieved September, 2012.

[100] I. Pitas. A Method of Signature Casting on Digital Images. Proc. of the IEEE Int'l Conference on Image Processing, ICIP-1996, 215–218, 1996.

[101] 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.

[102] S. Russell and P. Norvig. Artificial Intelligence: A Modern Approach (3rd ed.), Prentice-Hall, 2010.

[103] T.F. Rodriguez, B.T. MacIntosh, and A.E. Gustafson. Software watermarking. US Patent 20100095376, 2010.

[104] C. Rey, and J.L. Dugelay. A survey of watermarking algorithms for image authentication. EURASIP Journal on Applied Signal Processing 6, 613-621, 2002.

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

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

[107] R. Sedgewick, and P. Flajolet. An Introduction to the Analysis of Algorithms. Addison-Wesley, 1996.

[108] S. Sharma, J. Rajpurohit, and S. Dhankar. Survey on different level of audio watermarking techniques. Proc. of the Int'l Journal of Comput. Applications 49(10), 41–48, 2012.

[109] V. Solachidis and I. Pitas. Circularly Symmetric Watermark Embedding in 2D DFT Domain. IEEE Transactions on Image Processing 10(11), 1741–1753, 2001.

[110] C. Song, S. Sudirman, M. Merabti, and D. Llewellyn-Jones. Analysis of digital image watermark attacks. In Consumer Communications and Networking Conference (CCNC), 1–5, 2010.

[111] X. Sun, and A.J. Asiimwe. Noun-verb based technique of text watermarking using recursive decent semantic net parsers. LNCS 3612, 958–961, 2005.

[112] B.K. Sharma, R.P. Agarwal, and R. Singh. An efficient software watermark by equation reordering and FDOS. Proc. of the Int'l Conference on Soft Computing for Problem Solving (SocProS 2011, 131, 735–745, 2011.

[113] H. Tamada, M. Nakamura, A. Monden, and K. Matsumoto. Design and evaluation of birthmarks for detecting theft of Java programs. Proc. of the Int'l Conference on Software Engineering (IASTED SE'04), 569–575, 2004.

[114] W.G. Horne, U. Maheshwari, R.E. Tarjan, J.J. Horning, W.O. Silbert, L.R. Matheson, A.K. Wright, and S.S. Owicki. Systems and methods for watermarking software and other media. US Patent 8.140.850, 2012.

[115] R.G. Van Schyndel, A.Z. Tirkel, and C.F. Osborne. A digital watermark. Proc. of the Image Processing (ICIP-94), 86–90, 1994.

[116] A.Z. Tirkel, G.A. Rankin, R.M. Van Schyndel, W.J. Ho, N.R.A. Mee, and C.F. Osborne. Electronic watermark. Digital Image Computing, Technology and Applications (DICTA'93), 666–673, 1993.

[117] A.Z. Tirkel, R.G. Schyndel, and C. Osborne. A Two-Dimensional Watermark. Proc. of the DICTA 95(7), 5–8, 1995.

[118] U. Topkara, M. Topkara, and M.J. Atallah. The hiding virtues of ambiguity: Quantifiably resilient watermarking of natural language text through synonym substitutions, Proc. of the ACM Multimedia and Security Conference, 2006.

[119] M. Topkara, U. Topraka, and M.J. Atallah. Information hiding through errors: A confusing ap- proach. Proc. of the SPIE Int'l Conference on Security, Steganography, and Watermarking of Multimedia Contents, San Jose, CA, 2007.

[120] R. Venkatesan, V. Vazirani, and S. Sinha. A graph theoretic approach to software watermarking. Proc. of the 4th Int'l Workshop on Information Hiding (IH'01), LNCS 2137, 157–168, 2001.

[121] F.H. Wang, J.S. Pan, and L.C. Jain. Innovations in Digital Watermarking Techniques. Springer, 2009.

[122] Z. Wang, A.C. Bovic, H.R. Sheikh, and E.P. Simoncelli. Image Quaity Assessment: From Error Visibility to Structural Similarity. IEEE Transactions on Image Processing 13(4), 600–612, 2004.

[123] R.B. Wolfgang, and E.J. Delp. A watermark for digital images. Proc. of the IEEE Int'l Conference on Image Processing, ICIP-1996, 219–222, 1996.

[124] X.Y. Wang, and H. Zhao. A Novel Synchronization Invariant Audio Watermarking Scheme Based on DWT and DCT. IEEE Transactions on Signal Processing 54(12), 4835-4840, 2006.

[125] Y. Xiong, and Z.X. Ming. Covert Communication Audio Watermarking Algorithm Based on LSB. Proc. of the In'l Conference on Communication Technology, ICCT-2006, 1–4, 2006.

[126] L. Xie, J. Zhang, and H. He. Robust Audio Watermarking Scheme Based on Non-uniform Discrete Fourier Transform. Proc. of the IEEE Int'l Conference on Engineering of Intelligent Systems, 1–5, 2006.

[127] S. Yang, W. Tan, Y. Chen, and W. Ma. Quantization-Based Digital Audio Watermarking in Discrete Fourier Transform Domain. Journal of Multimedia 5(2), 151–158, 2010.

[128] T. Xie, and D. Notkin. An Empirical Study of Java Dynamic Call Graph Extractors. University of Washington CSE, Technical Report 02-12-03, 2002.

[129] J. Zhu, Y. Liu, and K. Yin. A Novel Planar IPPCT Tree Structure and Characteristics Analysis. Journal of Software 5(3), 344–351, 2010.

[130] 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.

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

[132] J.M. Zain. Strict Authentication Watermarking with JPEG Compression (SAW-JPEG) for Medical Images for medical images. European Journal of Scientific Research 42(2), 250–256, 2010.

[133] S. Zhong, X. Cheng, and T. Chen. Data hiding in a kind of PDF texts for secret communication. Int'l Journal of Network Security 4(1), 17–26, 2007.

[134] X. Zhou, W. Zhao, Z. Wang, and L.Pan. Security Theory and Attack Anlysis for Text watermarking. Proc. of the Int'l Conference on E-Business and Information System Security (EBISS), 1–6, 2009.

Author's Publications

1. M. Chroni, A. Fylakis, and S.D. Nikolopoulos, "Watermarking Digital Images in the Frequency Domain: Performance and Attack Issues", LNBIP 189, Springer, 68–84, 2014.

2. M. Chroni, A. Fylakis, and S.D. Nikolopoulos, "From image to audio watermarking using self-inverting permutations", Proceedings of the 10th Int'l Conference on Web Information Systems and Technologies (WEBIST'14), SciTePress, 177–184, 2014.

3. M. Chroni, A. Fylakis, and S.D. Nikolopoulos, "Watermarking images in the frequency domain by exploiting self-inverting permutations", Journal of Information Security 4(2), 80–91, 2013.

4. I. Chionis, M. Chroni, and S.D. Nikolopoulos, "A dynamic watermarking model for embedding reducible permutation graphs into software", Proceedings of the 10th Int'l Conference on Security and Cryptography (SECRYPT'13), SciTePress, 74–85, 2013.

5. M. Chroni and S.D. Nikolopoulos, "Design and evaluation of a graph codec system for software watermarking", Proceedings of the 2nd Int'l Conference on Data Management Technologies and Applications (DATA'13), SciTePress, 277–284, 2013.

6. M. Chroni, A. Fylakis, and S.D. Nikolopoulos, "Watermarking images in the frequency domain by exploiting self-inverting permutations", Proceedings of the 9th Int'l Conference on Web Information Systems and Technologies (WEBIST'13), SciTePress, 45–54, 2013, (Best Student Paper Award).

7. I. Chionis, M. Chroni, and S.D. Nikolopoulos, "Evaluating the WaterRpg software watermarking model on Java application programs", Proceedings of the 17th Panhellenic Conference on Informatics (PCI'13), ACM, 144–151, 2013.

8. M. Chroni and S.D. Nikolopoulos, "Multiple encoding of a watermark number into reducible permutation graphs using cotrees", Proceedings of the 13th Int'l Conference on Computer Systems and Technologies (CompSysTech'12), ACM, 118–125, 2012.

9. M. Chroni and S.D. Nikolopoulos, "An efficient graph codec system for software watermarking", Proceedings of the 36th Int'l Conference on Computers, Software, and Applications (COMPSAC'12), Workshop STPSA'12, IEEE, 595–600, 2012.

Author's Publications (cont.)

10. M. Chroni, A. Fylakis, and S.D. Nikolopoulos,"A watermarking system for teaching intellectual property rights: implementation and performance", Proceedings of the 11th Int'l Conference on Information Technology Based Higher Education and Training (ITHET'12), IEEE, 1–8, 2012.

11. M. Chroni and S.D. Nikolopoulos, "An embedding graph-based model for software watermarking", Proceedings of the 8th Int'l Conference on Intelligent Information Hiding and Multimedia Signal Processing (IIH-MSP'12), IEEE, 261–264, 2012.

12. M. Chroni, A. Fylakis, and S.D. Nikolopoulos, "Watermarking images using 2D representations of self-inverting permutations", Proceedings of the 8th Int'l Conference on Web Information Systems and Technologies (WEBIST'12), SciTePress, 380–385, 2012.

13. M. Chroni, A. Fylakis, and S.D. Nikolopoulos, "A watermarking system for teaching students to respect intellectual property rights", Proceedings of th 4th Int'l Conference on Computer Supported Education (CSEDU'12), SciTePress, 336–339, 2012.

14. M. Chroni and S.D. Nikolopoulos, "Encoding watermark numbers as cographs using self-inverting permutations", Proceedings of the 12th Int'l Conference on Computer Systems and Technologies (CompSysTech'11), ACM, 142–148, 2011, (Best Paper Award).

15. M. Chroni and S.D. Nikolopoulos, "Encoding watermark integers as self-inverting permutations," Proceedings of the 11th Int' l Conference on Computer Systems and Technologies (CompSysTech'10), ACM, 125–130, 2010.

Short CV

Maria G. Chroni was born in Ioannina, Greece, on 4th of December, 1982. She holds a BSc degree (2004) and an MSc degree (2007) in Computer Science both from the Department of Computer Science, University of Ioannina. She also holds a BEdu degree (2013) in Early Childhood Education from the Department of Early Childhood Education of the University of Ioannina. She defended her PhD on 19th of December, 2014.

Her research interests are in design and analysis of algorithms, algorithmic graph theory, and information hiding (in this area, she has proposed several algorithmic techniques for watermarking software, image, audio, and text). Her research interests are extended in applications of Information and Communication Technology (ICT) in Education. She has participated in the research project Pythogoras-II and the European projects TRICE (Teaching, Research, Innovation in Computing Education) and FETCH (Future Education and Training in Computing: How to support learning at anytime anywhere). She is a researcher at the Algorithms Engineering Lab (AlgoLab) of the Department of Computer Science and Engineering, University of Ioannina (2009-now). She is a member of IEEE (2012-now) and Student Branch Chair of IEEE at University of Ioannina (2012-14).