Graph-based Algorithmic Techniques for Watermarking using Self-inverting Permutations and Bitonic Sequences

A Thesis

submitted to the designated
by the General Assembly of Special Composition
of the Department of Computer Science and Engineering
Examination Committee

by

Anna Mpanti

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WITH SPECIALIZATION
IN COMPUTER SCIENCE THEORY

University of Ioannina
March 2017

Examining Committee:

- **Σταύρος Δ. Νικολόπουλος**, Καθηγητής, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων (Επιβλέπων)
- **Λουκάς Γεωργιάδης**, Αναπλ. Καθηγητής, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων
- **Λεωνίδας Παληός**, Καθηγητής, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων

Acknowledgements

Αρχικά θα ήθελα να ευχαριστήσω την οικογένεια μου, για την αμέριστη υποστήριξη που μου παρείχαν όλα τα χρόνια των σπουδών μου, χωρίς την οποία δεν θα μπορούσα να είχα επιτύχει τον στόχο μου. Μέσα από την καρδιά μου, ευχαριστώ τους γονείς μου, Ναπολέων και Ασημούλα, και ιδαίτερα την αδερφή μου, Χριστίνα η οποία αποτέλεσαι πρότυπο για την ζωή και την καριέρα μου και ένας σπουδαίος σύμβουλος.

Επιπρόσθετα, θα ήθελα να ευχαριστήσω τον πνευματικό μου πατέρα, τον καθηγητή κύριο Σταύρο Δ. Νικολόπουλο για την αμέριστη εμπιστοσύνη που μου επέδειξε στα πρώτα βήματα της ερευνητικής μου πορείας καθώς και για την καθοδίγηση του, αποτελώντας αρωγός στην προσπάθεια περάτωσης και επιτυχίας της Μεταπτυχιακής Εργασίας μου. Ακόμη, θα ήθελα να ευχαριστήσω τον αναπληρωτή καθηγητή κύριο Λουκά Γεωργιάδη και τον καθηγητή κύριο Λεωνίδα Παληό για την συμμετοχή τους στην τριμελή επιτροπή καθώς επίσης και για τις πολύτιμές, εύστοχες παρατηρήσεις τους που συνέβαλαν στην βελτίωση αυτής της εργασίας.

Τέλος, ευχαριστώ όλους τους φίλους μου, τους συμφοιτητές μου και τους ανθρώπους που στάθηκαν δίπλα μου και πίστεψαν σε εμένα.

Table of Contents

List of Figures

List of Tables

Abstract

Anna Mpanti, M.Sc. in Computer Science, Department of Computer Science and Engineering, University of Ioannina, Greece, March 2017.
Graph-based Algorithmic Techniques for Watermarking using Self-inverting Permutations and Bitonic Sequences.
Advisor: Stavros D. Nikolopoulos, Professor.

Over the last 25 years, digital or multimedia watermarking has become a popular technique for protecting the intellectual property of any digital content such as image, audio, video or software data. Software watermarking has received considerable attention and was adopted by the software development community as a technique to prevent or discourage software piracy and copyright infringement. A wide range of software watermarking techniques has been proposed among which the graph-based methods that encode watermark numbers as graphs whose structure resembles that of real program graphs.

Following up on recently proposed methods for encoding watermark numbers w as reducible permutation flow-graphs $F[\pi^*]$ through the use of self-inverting permutations π^*, in this thesis, we extend the types of flow-graphs available for software watermarking by proposing two different reducible permutation flow-graphs, namely, $F_s[\pi^*]$ and $F_t[\pi^*]$. These flow-graphs incorporate important properties which are derived from specific properties of the bitonic subsequences composing the self-inverting permutation π^*. We show that a self-inverting permutation π^* can be efficiently encoded into either $F_s[\pi^*]$ or $F_t[\pi^*]$ and also efficiently decoded from theses graph structures.

The proposed flow-graphs $F_s[\pi^*]$ and $F_t[\pi^*]$ enrich the repository of graphs which can encode the same watermark number w and, thus, enable us to embed multiple copies of the same watermark w into an application program P. Moreover, the enrichment of that repository with new flow-graphs increases our ability to select

a graph structure more similar to the structure of a given application program P thereby enhancing the resilience of our codec system to attacks.

Finally, we compare the proposed watermarking algorithms with two previously proposed codec watermarking algorithms and present similarities and differences with respect to their structures and complexity. In addition, we compute the probabilities of edge and label modifications of our flow-graphs $F_s[\pi^*]$ and $F_t[\pi^*]$ in order to consider the resilience of our watermark systems.

Chapter 1

Introduction

1.1 Thesis's Scope

Internet technology is the ability of the Internet to transmit information and data through different servers and systems. It has lead to a wealth of information available to anyone who is able to access the Internet. In addition, internet technology has changed, and will continue to change, the way that the world does business and how people interact in daily life [1]. This frequent use of the internet means that measures taken for internet security are indispensable since the web is not risk-free [2, 3]. However, the spread of computing and the Internet have made it difficult to apply traditional intellectual property laws. Despite popular belief, just because it's easy to distribute information using the Internet does not mean that it's right to do so, and thus such data may end up on a user who falsely claims ownership.

Software piracy used to describe the act of illegally using, copying or distributing software without ownership or legal rights. The copyright holder is typically the work's creator, or a publisher or other business to whom copyright has been assigned.

Copyright holders routinely invoke legal and technological measures to prevent and penalize copyright infringement. But how can someone claim a software program's ownership? Watermarks are a solution to this problem, if he previously embedded one in software program.

From network topologies and online social networks to simple software programs, many of today's most sensitive datasets are captured in large graphs [4]. Current tools can provide limited node or edge privacy, but require modifications to the graph that significantly reduce its utility. That is what this thesis suggests, techniques according to which invisible watermarks are embedded into a digital object which may be used to verify its authenticity or the identity of its owners.

The purpose of this work is that intellectual property protection and proper use are some of the greatest concerns over internet users today. The term intellectual property (IP) refers to a creation of a mind for which a set of exclusive rights are recognized. 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, musical works, drawings, paintings, photographs, sculptures, and architectural designs. The objective of recognizing intellectual property is to encourage innovation. This is because people will not have the incentive to create if they are not legally protected in order to get the social value that they deserve from their creations. Of course the world's evolution and economic growth depend on creations and inventions and that makes intellectual property such an important and vital aspect.

From 1996, there is the Digital Millennium Copyright Act (DMCA) which is a United States copyright law that implements two treaties of the World Intellectual Property Organization (WIPO). It criminalizes production and dissemination of technology, devices, or services intended to circumvent measures (commonly known as digital rights management or DRM) that control access to copyrighted works. It also criminalizes the act of circumventing an access control, whether or not there is actual infringement of copyright itself. In addition, the DMCA heightens the penalties for copyright infringement on the Internet [5].

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.

The thing is though, how someone can be able to claim and thus enabled to protect his intellectual property. Watermarking and in our case Software Watermarking is a solution. The problem of Software Watermarking is inserting some data into a program, thereby protecting it against intellectual property theft [6].

Since the late 1990s, there has been an explosion in the number of digital watermarking techniques developed mostly for the rights protection of multimedia contents (See Chapter 2). However, it is worth mentioning that every single method requires attention because we can not discriminate a specific one as the best. Every case has its ideal solution and the same rule applies for Software Watermarking.

What this thesis's technique suggests, is an efficient and robust methods for software watermarking technique based on graph properties. The important fact for this idea, is that it suggests a way in which an integer number w can be represented by a watermark graph, which, in term, can be embedded into a software program. In a similar way, such a watermark graph can be extracted from a watermarked program and converted back to the integer w.

Concerning the mentioned interim state, this work suggests an efficient algorithm for encoding a self-inverting permutation π^* into a flow graph $F[\pi^*]$. This is done by partition of self-inverting permutation in bitonic subsequences. This is done by partition of self-inverting permutation in bitonic subsequences, the edge and vertex sets are created from them and after that a watermark graph is constructed.

This work also suggests an efficient algorithm for extracting the embedded self-inverting permutation π^* from the watermark graph. That enables us to reconstruct the subsequences of the self-inverting permutation π^* and thus extract the embedded watermark w.

As mentioned in a previous paragraph, the suggested watermarking technique has properties that make it robust to multiple transformations which are further analyzed at Chapter 4 dedicated to the evaluation of our watermarking method and the comparison of two proposed algorithms, too.

The key idea behind the proposed watermarking model is that an integer watermark w, is embedded into a software program P through the use of self-inverting

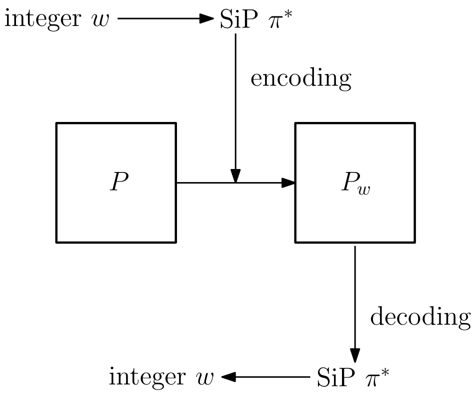

Figure 1.1: An illustration of the main idea behind the watermarking technique.

permutations π^* resulting the watermarked image P_w (See, Figure 1.1).

1.2 The Notion of Watermarking

Nowadays, we can find watermarks in nearly every official document, like banknote. In this case, a watermark is an identifying image or pattern in paper that appears as various shades of lightness/darkness when viewed by transmitted light (or when viewed by reflected light, atop a dark background), caused by thickness or density variations in the paper. The origin of the water part of a watermark can be found back when a watermark was something that only existed in paper, at that time the watermark was created by changing the thickness of the paper and thereby creating a shadow/lightness in the watermarked paper. This was done while the paper was still wet/watery and therefore the mark created by this process is called a watermark. Their purpose is to carry information about the object in which they are hidden and they are designed in such a way so that to be as difficult as possible to be reproduced by counterfeiting methods.

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. [7]. It is a process in which the information which verifies the owner is embedded into the digital object; 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.

We may also notice that some watermarks may be directly visible with the human eye, while others are hidden from view during normal use and only become visible under special viewing processes such as specific viewing angle or specific lighting conditions i.e. perceptible and imperceptible watermarks.

Note that when most researchers refer to watermarks, specifically to digital watermarks, they consider the imperceptibility as a defining characteristic. Digital watermarking is the act of hiding a message related to a digital signal (i.e. an image, song, video) within the signal itself. It is a concept closely related to steganography, in that they both hide a message inside a digital signal. However, what separates them is their goal. Watermarking tries to hide a message related to the actual content of the digital signal, while in steganography the digital signal has no relation to the message, and it is merely used as a cover to hide its existence.

Watermark is typically used to identify ownership of the copyright of such signal. More specifically, digital watermarking is a popular technique for copyright protection of a digital object or, in general, multimedia information [8, 9]; the idea is simple: a unique marker, which is called watermark, is embedded into a digital object which may be used to verify its authenticity or the identity of its owners [10, 11]. 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 addition, 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.

A watermarking system consists of two components:

- watermark encoder or embedder, and
- watermark decoder/extractor or detector.

In a similar manner we can tell that these two components are two processes, the embedding and extraction process. The first takes as an input the cover work and the value of the watermark w and returns as an output the watermarked work. Whereas the extracting process takes as an input the watermarked work and returns the value of the detected wa termark w as an output.

In this thesis, we are interested in watermarking where the cover work is a software program; this is called software watermarking. Software watermarking is a branch of digital watermarking which can be briefly described as the process of hiding digital information in a program. More details about software watermarking problem are about to follow in Chapter 2.

Furthermore, similar to traditional watermarking, digital watermarks can only be perceptible under specific conditions such as, after using special extracting algorithms [12]. There has been an explosion in the number of digital watermarking techniques, day to day. If a digital watermark distorts the carrier signal in a way that it becomes perceivable, it is of no use. In digital watermarking, the signal may be audio, picture, video, text, 3D models, etc. A signal can carry several different watermarks at the same time. Unlike metadata which is added to the carrier signal, a digital watermark does not change the size of the carrier signal meaning that the digital watermark do not add additional payload to the object.

In the software watermarking process, the digital information, i.e., the watermark, is hidden in program graph. The watermark is embedded into program's graph through structures which resemble that of real program graphs. Its aim is to prevent or discourage software piracy and copyright infringement.

1.3 Contriburion

Recently, Chroni and Nikolopoulos have presented two codec 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^* [13], and several codec algorithms for encoding π^* into several reducible permutation flow-graphs $F_i[\pi^*]$ $(i > 1)$ [14, 15]. Thus, they have created a wide repository of graph-structures, namely flow-graphs, whose structures resemble that of real program graphs.

In this thesis, we extend the types of flow-graphs which can efficiently encode a self-inverting permutation π^* by proposing two different reducible permutation flow-graphs $F_s[\pi^*]$ and $F_t[\pi^*]$ having properties which are derived from specific properties of bitonic subsequences $b_1^*, b_2^*, \cdots, b_k^*$ composing the self-inverting permutation π^*. We show relations between the elements of such a bitonic subsequence b_i^* and their indices in π^* and prove properties concerning the first, last, max and min elements of

π^*. We also show that the first bitonic subsequence b_1^* of a self-inverting permutation π^* of length n^* has always length $\lceil n_1^*/2 \rceil$ and structure $(\lceil n_1^*/2 \rceil, \ldots, \pi^*_{max}, \ldots, 1)$, where π^*_{max} is the max element of π^*.

Taking advantage of these properties, we construct two different reducible permutation flow-graphs $F_s[\pi^*]$ and $F_t[\pi^*]$ which can encode the same self-inverting permutation π^* and, thus, the same watermark number w. By construction, the indegree of the first node $s = u_{n^*+1}$ of the flow-graph $F_s[\pi^*]$ is equal to the number of bitonic subsequences $b_1^*, b_2^*, \cdots, b_k^*$ of π^*, while the indegree of the first node of the graph $F_t[\pi^*]$ is much smaller than k. This property causes $F_t[\pi^*]$ more appropriate, in same cases, since it does not contain an extreme characteristic thereby enhancing the resilience of graph-structure to attacks.

In particular, we propose the algorithm `Encode_SiP.to.RPG-Bitonic-S` which exploits the bitonic sequences b_i^* of π^* and their properties, and the algorithm `Encode SiP.to.RPG-Bitonic-T` which exploits the id-bitonic subsequences (i.e., a sequence is called id-bitonic if either monotonically increases and then monotonically decreases) and the indegree of header node s in $F_t[\pi^*]$ is decreased, 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, where $N = n^* + 2$ and n^* is the length of π^*. The corresponding decoding algorithm `Decode_RPG.to.SiP-Bitonic-S` extracts the self-inverting permutation π^* from the reducible permutation graph $F_s[\pi^*]$ by construction first an undirected graph $H[\pi^*]$ and then applying BFS-search on each of the connected components of $H[\pi^*]$. The decoding algorithm `Decode_RPG.to.SiP-Bitonic-T` also extracts π^* from $F_t[\pi^*]$ by converting $F_t[\pi^*]$ into an undirected graph $H[\pi^*]$. It applies BFS-search on the connected components of $H[\pi^*]$ and construct the new set R', which has the nodes u_i with $outdeg(u_i) \geq 2$. 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 $O(N)$, since $N = 2n^* + 1 = 2n + 3$.

It is worth noting that our codec $(\texttt{encode}, \texttt{decode})_{F[\pi^*]}$ system incorporates several important properties which characterize it as an efficient and easily implemented software watermarking component. In particular, the reducible permutation flow-graphs $F_s[\pi^*]$ and $F_t[\pi^*]$ do not differ from the graph data structures built by real programs since its maximum outdegree does not exceed two and it has a unique root node so the program can reach other nodes from the root node. The function

`Decode_RPG.to.SIP-Bitonic-S` or `-2` is high insensitive to edge and node modifications of $F_s[\pi^*]$ and $F_t[\pi^*]$. Moreover, the self-inverting permutation π^* captures important structural properties, due to the bitonic property used in the construction of π^*, which make our codec system resilient to attacks.

The flow-graphs $F_s[\pi^*]$ and $F_t[\pi^*]$ enrich the repository of graphs which can encode the same watermark number w and, thus, enable us to embed several copies of the same watermark w into an application program P. Moreover, it increases our ability to select a graph structure more similar to the structure of a given application program P thereby enhancing the resilience of our codec system to attacks.

1.4 Road Map

This thesis is strictured as follows.

Chapter 1 induces the basic concept of the watermarking process and briefly present the construction of this thesis by focusing on the main ideas.

Chapter 2 includes the theory behind software watermarking and it makes reference to the respective definitions and properties. It analyzes how the watermarking techniques are categorized and it describes its most important techniques for each category developed so far. In addition, it describes how watermark embeds into a software program and includes our contribution of this work.

Chapter 3 describes the main idea behind the proposed software watermarking algorithms by exploiting self-inverting permutations using an efficient transformation of a watermark from an integer. Finally, it shows properties of method's components.

Chapter 4 presents two codec algorithms for encoding a self-inverting permutation π^* into two different reducible permutation flow-graphs $F_s[\pi^*]$ and $F_t[\pi^*]$, incorporating important properties which are derived from specific properties of the bitonic subsequences composing the self-inverting permutation π^*. Also, it describes the similarities and differences between two codec algorithms.

Chapter 5 shows that properties of method's components help prevent edge and/or node modifications attacks.

Chapter 6 compares the proposed watermarking algorithms with two previously proposed codec watermarking algorithms based on their structures, edges and node modification, space and time complexity.

Finally, Chapter 7 concludes the thesis and discusses possible future extensions.

Chapter 2

Software Watermarking

2.1 Preliminaries

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. The process of watermarking involves embedding secret messages into a cover message. The software watermarking problem can be described as the problem of embeding a structure w into a program P such that w can be reliably located and extracted from P even after P has been subjected to code transformations such as translation, optimization and obfuscation [16, 17]. 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:

- $embed(P, w, k) \rightarrow P$ and
- $extract(P_w, k) \rightarrow w$.

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 [18].

A static watermark is stored inside program code in a certain format, and it does not change during the program execution. According to the representation of watermark information, there are two types of static watermarks: data watermarks and code watermarks.

(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.

On the other hand, a dynamic watermark w is built during program execution, perhaps only after a particular sequence of input I_{key}. It might be retrieved from the watermarked program P_w by analyzing the data structures built when watermarked program P_w is running on input I_{key}. In other cases, tracing the program execution may be required. There are three types of dynamic watermarks, Easter Egg, data structure and execution trace watermarks.

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., $\texttt{embed}(P, G, k) \to P_w$, and
- a decoding algorithm which extracts the graph G from P_w, i.e., $\texttt{extract}(P_w, k) \to G$.

We usually call the pair

$$(\mathtt{embed}, \mathtt{extract})_G$$

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 G having appropriate graph properties, e.g., graphs in G should be contained nodes with small outdegree so that matching real program graphs, and designing efficient codec algorithms, e.g., both algorithms of $(\mathtt{embed}, \mathtt{extract})_G$ model should be computed in polynomial time.

When referring to watermarks, we should always mention certain characteristics which are the key features defining each watermarking technique. Those characteristics describing a technique have mostly to do with the imperceptiveness and the resistance of the watermarks embedded using it. Furthermore, each watermarking technique also comes with an evaluation. That is measuring its behavior presenting certain characteristics. So, we have also certain evaluating techniques. Some basic ones, are the following:

- **Effectiveness**: We consider a watermark as effective when the extracting algorithm is able to successfully extract it. So, we define as embedding effectiveness the probability that the extracting algorithm successfully extracts the embedded watermark without loosing any information.

- **Invisibility**: Invisible watermarks, are considered those that are hidden under normal use and can only appear when extracted from an authorized user using a special software.

- **Robust Watermarks**: The target in most cases is to design a robust software watermarking technique. Piracy attacks should not affect the embedded watermark. Robustness is measured according to how the watermarked withstands attacks and transformations such as, edge and node modification etc.

- **Fragile Watermarks**: Fragile watermarks, are also known as tamper-proof watermarks. This kind of watermarks unlike robust watermarks is destroyed by data manipulation. They are designed in such a way in order to be destroyed by any form of copying or encoding other that bit-by-bit digital copy. Their absence indicates that a copy of the digital object has been made as slight errors occurred during copy destroyed the watermark. For example a fragile watermark designed for authentication purposes declares by its absence that the object is not original anymore as it has been through processing applications.

- **Security**: A watermark should be secret and thus be undetectable by an unauthorized users and only detectable by authorized ones. This requirement is regarded as a the security factor. Generally speaking, the security of a watermark refers to its ability to resist hostile attacks.
- **Watermark keys**: It should also be mentioned that there is a specific category of watermarks that make use of keys. That means that input at the encoder are not just the watermark and the digital object but also a key which is also needed at the decoder as an input in order to successfully extract the watermark.

2.2 Attacks

Watermarked software may be subject to attacks that have the objective of locating, distorting, or removing the watermark. The quality of a watermarking system is correlated with the degree to which the watermarked software is resistant to attacks. Most approaches to software watermarking have concentrated on resisting attacks by preserving program transformations, including compilation, optimization, obfuscation and decompilation. Such transformations do not change the behavior of a program, but they do change the form of the program.

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; 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). 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 $\texttt{extract}(P_w, k) \to w$ then $\texttt{extract}(P'_w, k) \to w$ with $P'_w \approx P$.

On the other side, 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.

The main approaches to defending against such attacks are to either make the watermark an inherent part of the programs behavior, or to make the watermark be represented by data structures that are created at run-time [19]. A goal of a good watermarking system should be to make an attacker to unable to locate the code that builds the graphs and to make it difficult to remove from rest of the program all dependencies on the graphs, hence also difficult to remove or distort the code that builds the two graphs.

2.3 Techniques for Software Watermarking

Since the late 1990s, there has been an explosion in the number of digital watermarking techniques among which time-series, biological sequences, graph-structured data, spatial data, spatiotemporal data, data-streams and others [6]. Recently, software watermarking has received considerable attention and many researchers have developed several codec algorithms mostly for watermarks that are encoded as graph-structures [20]. The patent by Davidson and Myhrvold [21] presented the first published software watermarking algorithm. The preliminary concepts of software watermarking also appeared in paper [22] and patents [23, 8]. Collberg et al. [18, 24] presented detailed definitions for software watermarking, while Zhang et al. [25] and Zhu et al. [26] gave brief surveys of software watermarking research; see, Collberg and Nagra [10] for an exposition of the main results.

The algorithm of Davidson and Myhrvold [21] embeds the watermark into a program by reordering the basic blocks of a control ow-graph; note that a static watermark is stored inside programs'code in a certain format and it does not change during the programs'execution. The first dynamic watermarking algorithm CT was proposed by Collberg and Thomborson [18]; 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 [27, 28, 21, 29]. 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

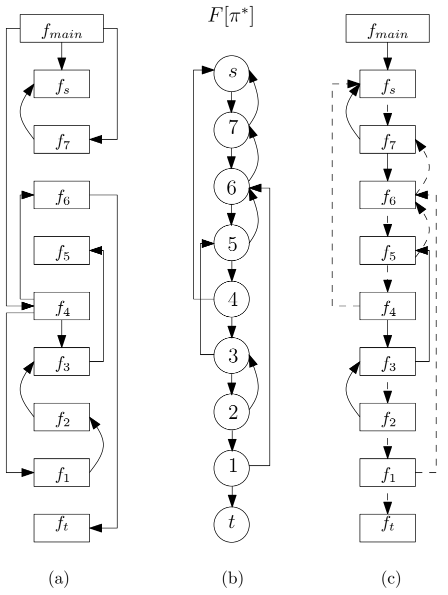

Figure 2.1: (a) The dynamic call-graph $G(P, I_{key})$ of an application program P. (b) The reducible permutation graph $F[\pi^*]$. (c) The dynamic call-graph $G(P^*, I_{key})$ of the watermarked program P^*.

an effcient method for encoding first an integer w as a self-inverting permutation π^* and then encoding π^* as a reducible permutation flow-graph $F[\pi^*]$ [13]; see, also [16]. 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.

Chroni and Nikolopoulos [15] and Chionis et al. [30] have proposed call-graphs as key-objects in their watermarking model for embedding the graph $F[\pi^*]$ into an application program discuss properties of dynamic call-graphs. A *call-graph* is a directed graph that represents calling relationships between program units in a computer program. Specifically, the nodes $f_1, f_2, \dots f_n$ 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 [31]. The static call-graph can be determined from the program source code; they mention that, its construction is a time consuming process specifically in the case of large scale software [32].

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.

They denote a dynamic call-graph G of the program P over the input I as $G(P, I)$. Figure 2.1 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^* along with its real edges (solid arrows) and water edges (dashed arrows).

Based on this idea and watermarking scheme proposed by these authors, Bento et al. [33] introduced a linear-time algorithm which succeeds in retrieving deterministically the n-bit identifiers encoded by such graphs (with $n > 2$) even if $k \leq 2$ edges are missing. In addition, they proved that $k \leq 5$ general edge modifications (removals/insertions) can always be detected in polynomial time. Both bounds are tight. Finally, their results reinforce the interest in regarding Chroni and Nikolopoulos's scheme as a possible software watermarking solution for numerous application.

A line of research that is more related to the graph watermarking problem we study is anonymization and de-anonymization for social networks. Eppstein et al. propose an approach [20] to graph watermarking that is necessarily related to the problem of graph isomorphism and its approximation. It is based on characterizing the feasibility of graph watermarking in terms of keygen, marking, and identification functions defined over graph families with known distributions. They demonstrated the strength of this approach with exemplary watermarking schemes for two random graph models, the classic Erdos-Renyi model and a random power-law graph model, both of which are used to model real-world networks. They studied these two random graph models and showed that watermarking in these models could be achieved in such a way that no adversary could remove the watermark with high probability and still have a graph that is "close" to his original graph. Also, they provided an exemplary implementation that works effectively for marks consisting only of edge flips.

Chapter 3

Our Watermarking Scheme

3.1 Components for Graph-based Watermarking

We consider finite graphs with no multiple edges. For a graph G, we denote by $V(G)$ and $E(G)$ the vertex (or, node) set and edge set of G, respectively. The subgraph of a graph G induced by a set $S \subseteq V(G)$ is denoted by $G[S]$. The *neighborhood* $N(x)$ of a vertex u of the graph G is the set of all the vertices of G which are adjacent to u. The *degree* of a vertex u in the graph G, denoted $deg(u)$, is the number of edges incident on node u; for a node u of a directed graph G, the number of head-endpoints of the directed edges adjacent to u is called the indegree of the node u, denoted $indeg(u)$, and the number of tail-endpoints is its outdegree, denoted $outdeg(u)$. The parent of a node x of a rooted tree T is denoted by $p(x)$.

The main components, which are used by the algorithms of our codec system, are illustrated in Figure 3.1 and we describe them in detail.

3.1.1 Bitonic Permutation

We think of a permutation π of length n as a permutation over the elements contained at a set $N_n = \{1, 2, ..., n\}$. That permutation π is represented as a sequence as follows:

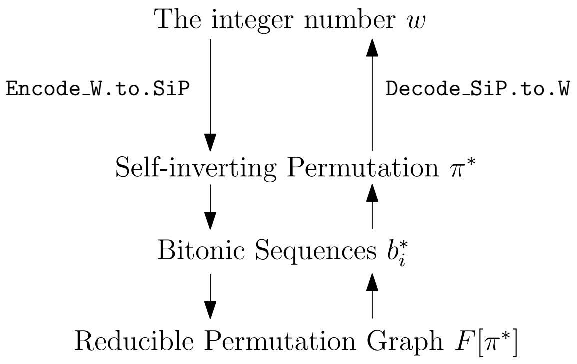
The integer number w
Encode_W.to.SiP
Decode_SiP.to.W
Self-inverting Permutation π^*
Bitonic Sequences b_i^*
Reducible Permutation Graph $F[\pi^*]$

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

$\pi = (\pi_1, \pi_2, ..., \pi_n)$, for example, the permutation $\pi = (4, 3, 6, 1, 5, 2)$ has $\pi_1 = 4$, $\pi_2 = 3$, $\pi_3 = 6$ etc. Having seen how we represent a permutation, note also that represented with π_i is the element of the permutation on the position with index i and with π_i^{-1} the index of the element i, in our example, $\pi_4^{-1} = 1$, $\pi_3^{-1} = 2$, $\pi_6^{-1} = 3$ etc. The length of a permutation π is the number of elements in π.

More formally, if π is a permutation of numbers $1, 2, \ldots, n$, then the graph $G[\pi] = (V, E)$ is defined as follows:

$$V = \{1, 2, \ldots, n\}$$

and

$$ij \in E \Leftrightarrow (i - j)(\pi_i^{-1} - \pi_j^{-1}) < 0.$$

An undirected graph G is called a *permutation* graph if there exists a permutation π such that $G \cong G[\pi]$.

A cycle of π is a sequence $c = (\pi_{i_1}, \pi_{i_2}, \ldots, \pi_{i_p})$ such that $\pi_{i_1}^{-1} = \pi_{i_2}, \pi_{i_2}^{-1} = \pi_{i_3}, \ldots, \pi_{i_p}^{-1} = \pi_{i_1})$. Throughout the thesis, a cycle of length k is referred to as a k-cycle.

A subsequence of a permutation $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$ is a sequence $b = (\pi_{i_1}, \pi_{i_2}, \ldots, \pi_{i_n})$ such that $i_1 < i_2 < \ldots < i_k$.

A sequence $b = (b_1, b_2, \ldots, b_n)$ is called bitonic if either monotonically increases and then monotonically decreases, or else monotonically decreases and then monotonically increases.

In this work, we consider only bitonic sequences that monotonically increases and then monotonically decreases, i.e., the minimum element of such a sequence b is either the first b_1 or the last b_n element of b; for example, $b = (5, 6, 8, 9, 1)$ is such a bitonic sequence. The maximum element of a bitonic sequence b, which we call *top* element of b, is denoted as $top(b)$. Obviously, b is an increasing sequence if $top(b) = b_n$, while b is a decreasing sequence if $top(b) = b_1$.

Definition 3.1 Let $b = (b_1, b_2, \ldots, b_n)$ be a bitonic sequence of length n. According to the index of the element $top(b)$, the sequence b is called:

- i-bitonic or *increasing bitonic* if $top(b) = b_n$;
- d-bitonic or *decreasing bitonic* if $top(b) = b_1$;
- id-bitonic or *full-bitonic* if $b_1 < top(b)$ and $top(b) > b_n$.

In terms of the above definition, $b_1 = (2, 7)$ is an i-bitonic or increasing bitonic sequence, $b_2 = (4, 3)$ is a d-bitonic or decreasing bitonic sequence, while $b_3 = (5, 6, 8, 9, 1)$ is an id-bitonic or full-bitonic sequence.

3.1.2 Self-inverting Permutations (SiPs)

We next define the main component of our codec system, namely, the self-inverting permutation (SiP). In mathematics, the notion of permutation relates to the act of arranging all the members of a set into a sequence or order. Self-inverting parmutations are a subclass belonging to the class of the permutations. Permutations may be represented in many ways, where the most straightforward is simply a rearrangement of the elements of the set $N_n = \{1, 2, \ldots, n\}$. For example, $\pi = (5, 6, 8, 9, 1, 2, 7, 3, 4)$ is a permutation of the elements of the set N_9; hereafter, we shall say that π is a permutation over the set N_9.

So, which is the extra property which distinguishes the subclass of the self-inverting permutations, or for short hereafter, SiP from the class of the permutations? The answer is at the following definition.

Definition 3.2 Let $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$ be a permutation over the set N_n, $n > 1$. The inverse of the permutation π is the permutation $q = (q_1, q_2, \ldots, q_n)$ with $q_{\pi_i} = \pi_{q_i} = i$. A *self-inverting permutation* (or, for short, SiP) is a permutation that is its own inverse: $\pi_{\pi_i} = i$.

By definition, a permutation is a SiP (Self-inverting Permutation) if and only if all its cycles are of length 1 or 2; for example, the permutation $\pi = (5, 6, 8, 9, 1, 2, 7, 3, 4)$ is a SiP with cycles: $(1, 5)$, $(2, 6)$, $(3, 8)$, $(4, 9)$, and (7).

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

3.1.3 Encode Watermark Numbers as SiPs

In this section, we present the way of encoding an integer w into a self-inverting permutation π^* and extracting it from π^*, which we need to construct our watermarking system.

In [16], Chroni and Nikolopoulos have proposed algorithms for such a system which efficiently encode an integer w into a self-inverting permutation π^* and efficiently decode it. First, they have presented the encoding algorithm Encode_W.to.SIP, which takes as input an integer w, computes its binary representation, and then produces a self-inverting permutation π^*. Also, they have proposed an extraction algorithm Decode_SIP.to.W, which takes as input a self-inverting permutation π^* produced by Algorithm Encode_W.to.SiP and returns its corresponding integer w. The key-idea behind its algorithms is mainly based on mathematical objects, namely, bitonic permutations.

Following, is the step-by-step description below the two codec algorithms (encode and decode) that correspond integer numbers into self-inverting permutations (SiPs).

Algorithm Encode_W.to.SiP (Chroni and Nikolopoulos [16])

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

2. Construct the binary number $B' = 00 \ldots 0$□B□1 of length $2n + 1$, and then the binary sequence $B^* = (b_1, b_2, \ldots, b_{n'})$ of $flip(B')$;

3. Construct the sequence $X = (x_1, x_2, \ldots, x_k)$ of the 0's positions and the sequence $Y = (y_1, y_2, \ldots, y_m)$ of the 1's positions in B^* from left-to-right, 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 $\pi^b$ from left and the $i$-th
            element 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 $\pi^*$ to the identity permutation $(1, 2, \ldots, 2n + 1)$;
       for each 2-cycle $(\pi_i, \pi_j)$ computed at Step 5, set $\pi^*_{\pi_i} = \pi_j$ and $\pi^*_{\pi_j} = \pi_i$;

    7. Return the permutation $\pi^*$ (which by construction is self-inverting);
```

Algorithm `Decode_SiP.to.w` (Chroni and Nikolopoulos [16])

```
    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 $\pi^b$ 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 $c = (a)$:
                   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 $\pi^b$ and then the decreas-
       ing 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 $B = b'_{n+1}b'_{n+2} \cdots b'_{n^*+1}$;
```

Both of two algorithms run in $O(n)$ time, where n is the length of the binary representation of the integer w. Moreover, they proposed several algorithms for multiple encoding the same watermark number w into many different reducible permutation graphs $F_i[\pi^*]$, $i > 1$, through the use of the encoding self-inverting permutation π^* [14, 15]. These results are summarized in the following theorems.

Theorem 3.1 *Let w be an integer and let $b_1, b_2, \ldots b_n$ 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 3.2 *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.*

Theorem 3.3 *It can produced more than one reducible permutation flow-graphs $F_1[\pi^*]$,$F_2[\pi^*]$, $\ldots, F_n[\pi^*]$ which encode the same watermark integer w through the use of the self-inverting permutation π^*.*

Example 3.1 W-to-SiP: Let $w = 10$ be the given watermark integer. We first compute the binary representation $B = 1010$ of the number 10, then we construct the binary number $B = 0000||1010||1$ and the binary sequence $B^* = (1,1,1,1,0,1,0,1,0)$ by flipping the elements of B'; we compute the sequences $X = (5,7,9)$ and $Y = (1,2,3,4,6,8)$ by taking into account the indices of 0s and 1s in B^*, and then the bitonic permutation $\pi^b = (5,7,9,8,6,4,3,2,1)$ on $n' = 9$ numbers by taking the sequence $X||Y^R$; since n' is odd, we select 4 cycles $(5,1), (7,2), (9,3), (8,4)$ of lengths 2 and one cycle (6) of length 1, and then based on the selected cycles construct the self-inverting permutation $\pi^* = (5,7,9,8,1,6,2,4,3)$.

Example 3.2 SiP-to-W: Let $\pi^* = (5,7,9,8,1,6,2,4,3)$ be the given self-inverting permutation produced by our method. The cycle representation of π^* is the following: $(1,5)$, $(2,7)$, $(3,9)$, $(4,8)$, (6); from the cycles we construct the permutation $\pi^b = (5,7,9,8,6,4,3,2,1)$; then, we compute the first increasing subsequence $X = (5,7,9)$ and the first decreasing subsequence $Y = (8,6,4,3,2,1)$; and construct the binary sequence $B^* = (1,1,1,1,0,1,0,1,0)$ of length 9; we flip the elements of B^* and construct the sequence $B' = (0,0,0,0,1,0,1,0,1)$; the binary number 1010 is the integer $w = 10$.

3.1.4 Reducible Permutation Graphs (RPGs)

A *flow-graph* is a directed graph F with an initial node s from which all other nodes are reachable. A directed graph G is *strongly connected* when there is a path $x \rightarrow y$ for all nodes x, y in $V(G)$. A node $u \in V(G)$ is an *entry* for a subgraph H of the graph G when there is a path $p = (y_1, y_2, \ldots, y_k, x)$ such that $p \cap H = \{x\}$ (see, [34, 35]).

Definition 3.3 A flow-graph is *reducible* when it does not have a strongly connected subgraph with two (or more) entries.

There are some other equivalent definitions of the reducible flow-graphs which use a few more graph-theoretic concepts. A depth first search (DFS) of a flow-graph partitions its edges into tree, forward, back, and cross edges. It is well known that tree, forward, and cross edges form a dag known as a DFS dag. Hecht and Ullman [34, 35] have proven the following theorem:

Theorem 3.4 *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 $\mathcal{T}_\infty$ and $\mathcal{T}_\in$, where $\mathcal{T}_\infty$ removes a cycle-edge, and $\mathcal{T}_\in$ picks a non-initial node y that has only one incoming edge (x, y) and glues nodes x and y.

Recently, a wide range of software watermarking techniques has been proposed among which the graph-based methods that encode watermark numbers w as reducible flow-graph structures F capturing such properties which make them resilient to attacks.

3.2 Properties

To be effective, a graph watermark system needs to provide several key properties. In this section, we analyze the structures of the three main components of our proposed codec system, that are, the self-inverting permutation π^*, reducible permutation graph $F[\pi^*]$ and $1 - cycle$ and discuss their properties with respect to resilience to attacks.

3.2.1 Properties of permutation π^*

The main properties of self-inverting permutation π^* produced by the algorithm `Encode_W. to.SiP` can be summarized into the following four categories:

- **Odd-lenght 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 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 and thus the cycles of π^*

We next analyze the structure of a self-inverting permutation π^* produced by the encoding algorithm `Encode_W.to.SiP`. Let π^* be the self-inverting permutation of length n^*, and let $b_1^*, b_2^*, \ldots, b_k^*$ be the bitonic subsequences forming the permutation π^*; note that, π^* encodes a watermark number w of binary length n and $n^* = 2n + 1$. Then, $b_1^*, b_2^*, \ldots, b_k^*$ have the following properties:

(P_1) Sequence b_1^* is a id-bitonic, $b_2^*, b_3^*, \ldots, b_{k-1}^*$ are either id-bitonic or d-bitonic sequences, while b_k^* is either a id-bitonic, i-bitonic, or d-bitonic sequence.

(P_2) Sequence b_1^* contains the min element π_{min}^* of permutation π^* and has always length $\lceil n^*/2 \rceil$; note that, $\pi_{max}^* = top(b_1^*) = 2n + 1$ and $\pi_{min}^* = 1$;

(P_3) The last element $last(b_k^*)$ of sequence b_k^* is equal to the index of the max element $\pi_{max}^* = 2n + 1$ in b_1^*.

Furthermore, it is easy to see that the elements of subsequence b_1^* are in the range $[n{+}1, 2n{+}1] \bigcup \{1\}$ and the elements 1, 2, and $n{+}1$ are never top elements of some subsequence b_i^*. Finally, since we computed all bitonic subsequences from self-inverting permutation π^*, it is very important to us to know how many these subsequences can be existed. So, if l is the number of all possible bitonic subsequences in a self-inverting permutation π^*, then

$$2 \leq l \leq \lfloor \tfrac{n+1}{2} \rfloor + 1.$$

Example 3.3 Let $w_1 = 20$ and $w_2 = 45$ be two watermark numbers. For these two watermarks, the encoding algorithm Encode_W.to.SIP produces the self-inverting permutations

- $\pi_1^* = (6, 8, 11, 10, 9, 1, 7, 2, 5, 4, 3)$, and
- $\pi_2^* = (7, 9, 10, 12, 13, 11, 1, 8, 2, 3, 6, 4, 5)$

of lengths $n_1^* = 2n_1 + 1 = 11$ and $n_2^* = 2n_2 + 1 = 13$, respectively; note that, $n_1 = 5$ is the length of the binary representation of number 20 (i.e., 10100), while $n_2 = 6$ is that of number 44 (i.e., 101101). The permutations π_1^* and π_2^* are composed by the following three and four, respectively, bitonic subsequences:

- π_1^* : $(6, 8, 11, 10, 9, 1) \ || \ (7, 2) \ || \ (5, 4, 3)$
- π_2^* : $(7, 9, 10, 12, 13, 11, 1) \ || \ (8, 2) \ || \ (3, 6, 4) \ || \ (5)$

We observe that all the bitonic subsequences of both permutations π_1^* and π_2^* satisfy the properties (P_1), (P_2), and (P_3). Indeed, for example, the subsequence $b_1^* = (6, 8, 11, 10, 9, 1)$ of permutation π_1^* is id-bitonic, contains the min elements of π_1^*, has length $6 = \lceil n_1^*/2 \rceil$, where $n_1^* = 11$ is the length of π_1^*, and the last element of subsequence $b_3^* = (5, 4, 3)$ (i.e., 3) is equal to the index of the max element of π_1^* in sequence b_1^* (i.e., $\pi_{max}^* = 11$).

3.2.2 Properties of permutation $F[\pi^*]$

In a graph-based watermarking environment, the watermark data can be encoded in the structures of various types of graphs. A graph representing a watermark data is called a *watermarking graph*. To be effective, a graph watermark system needs to provide several key properties. In general, the watermark will be difficult to detect (and remove) by potential attackers, and adding the watermark to the graph has minimal impact on the graph structure and its utility, so, the watermark graph should not differ from the graph data structures built by real programs. Important properties are graph watermarks should be relatively small and, the existence of a unique root node so that all other nodes can be reached from it. Finally, both the embedding and extraction of graph watermarks should be efficient, even for extremely large graph datasets with billions of nodes and edges.

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, where $n^* = 2n + 1$ and $n = \lceil \log_2 w \rceil$, with outdegree exactly two; that is, it has low max-outdegree and, thus, it matches real program graphs.

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

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

  (ii) **A footer node**: it is a node with outdegree zero that is reachable from every other node of the graph and is denoted by t. Every control flow-graph has such a node, representing the method exit.

  (iii) **A linked list**: it consists of n nodes $u_1, u_2, \ldots, u_n$ each with outdegree two. In particular, each node u_i $(1 \leq i \leq n)$ has exactly two outpointers: one points to node u_{i-1}, which we call list pointer, and the other points to node u_m, where $m > i$, where $u_m > u_i > u_{i-1}$.

  Thus, the graph $F[\pi^*]$ enables us to correct single edge modifications, i.e., edge-flips, edge-additions, or edge-deletions.

- **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, the size of $F[\pi^*]$ is $O(\log_2 w)$, since $n^* = 2n + 1$ and $n = \lceil \log_2 w \rceil$.

- **Efficient codecs**: The codec $(\texttt{encode,decode})_{F[\pi^*]}$ has linear time and space complexity.

It is worth noting that our codec system use basic data structures and operations, and thus they are easily implementable.

Furthermore, there is a relation between structure of $F[\pi^*]$ and 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 $G_i = G \setminus (v_1, v_2, \ldots, v_i)$, $1 \leq i \leq n-1$. It is important to note that any reducable flow-graph haw at most one Hamilton Path [27]. It is not difficult to see that the reducible permutation graphs $F[\pi^*]$ constructed by Algorithms `Encode_SiP.to.W-Bitonic-1` and `Encode_SiP.to.W-Bitonic-2` have a unique Hamiltonian path, denoted by HP($F[\pi^*]$); this is precisely the path $u_{n^*+1}u_{n^*} \ldots u_1 u_0$. Such a path can be found in time linear in the size of $F[\pi^*]$. Chroni and Nikolopoulos [16] proposed the algorithm `Unique_HP`, which takes as input a graph $F[\pi^*]$ on $n^* + 2$ nodes and produces its unique Hamiltonian path HP($F[\pi^*]$). The time and space complexity of this algorithm is summarized in the following theorem.

Theorem 3.1 *Let $F[\pi^*]$ be a reducible permutation graph of size $O(n^*)$ constructed by algorithm* `Encode_W.to.SiP`*. The algorithm* `Unique_HP` *correctly computes the unique Hamiltonian path of $F[\pi^*]$ in $O(n^*)$ time and space.*

3.2.3 Properties of $1 - cycle$

By definition, a self-inverting permutation π^* contains only one 1-cycle (x). This 1-cycle represents the index of the first 0 in the binary representation of the integer w. The 1-cycle is in the $n + k$ vertex, where n is the length of the binary representation of the integer w and k is the index of the first 0 in the binary representation; thus, $n + 2 \leq x \leq 2n + 1$. The last number of a range of bits, which does not have zeros, the 1-cycle is the maximum element π^*_{max}, i.e, $x = 2n + 1$. Moreover, the 1-cycle is the maximum element after the first bitonic subsequence b_1^* of self-inverting π^*. Taking into consideration that we can find which element is the 1-cycle if we know the first increasing subsequence from π^*, which is the elements between $n+1$ to $2n+1$. The 1-cycle is the first consecutive element which is missing from the first bitonic subsequence b_1^*.

Example 3.4 The permutation $\pi^* = (6, 8, 10, 11, 9, 1, 7, 2, 5, 3, 4)$ which decodes the number $w = 21$ with binary representation 10101. Obviously, the first bitonic subsequence is $b_1^* = (6, 8, 10, 11, 9, 1)$ and the 1-cycle is $7 = n + k$, where $n = 5$ and the index of first zero is $k = 2$. Finally, it is easy to see that the 1-cycle is the maximum element in the next bitonic subsequences $b_2^* = (7, 2)$, $b_3^* = (5, 3)$, $b_4^* = (4)$ and 7 is the first sequential element which is missing from the first increasing subsequence $(6, 8, 10, 11)$.

There are 2^{n-1} different integers w_i in the range n, where n is the number of bits in the binary representation of w_i. The encoding algorithm Encode_W.to.SiP produces 2^{n-1} different self-inverting permutations π^* and thus, there are n different 1-cycles $(x_1), (x_2), \ldots, (x_n)$, where $x_i = \pi^*_{n+i+1}$ and $x_n = \pi^*_{max}$. The 1-cycle (x_i) appears 2^{n-i-1} times in different self-inverting permutations, $i < n$, and the 1-cycle (π^*_{max}) appears only one time; note that it happens when w is the last integer in range n, that $w = 2^n - 1$.

Example 3.5 For $n = 3$, there are $4 (= 2^{n-1})$ different integers w_i. So, the self-inverting permutations produced by algorithm Encode_W.to.SiP, are:

- $w_1 = 4$: $\pi_1^* = (4, 7, 6, 1, 5, 3, 2)$,
- $w_2 = 5$: $\pi_2^* = (4, 6, 7, 1, 5, 2, 3)$,
- $w_3 = 6$: $\pi_3^* = (4, 5, 7, 1, 2, 6, 3)$, and
- $w_4 = 7$: $\pi_4^* = (4, 5, 6, 1, 2, 3, 7)$

with 3 different 1-cycles $x_1 = 5 = n + i + 1$, $x_2 = 6$ and $x_3 = 7 = \pi^*_{max}$. We observe that element 5 appears $2 (= 2^{n-i-1})$ times, as 1-cycle x_1 in π_1^* and π_2^*.

Chapter 4

Watermarking Algorithms

4.1 S-bitonic Algorithm

Having presented an efficient codec algorithm for encoding a watermark number w as a self-inverting permutation π^* [13] and several codec algorithms for efficiently encoding the permutation π^* into different reducible permutation flow-graphs $F_i[\pi^*]$ $(i > 1)$, in this section we extend the types of such flow-graphs by proposing an algorithm for encoding a self-inverting permutation π^* into a reducible permutation graph $F[\pi^*]$ having properties which are derived from the bitonic subsequences composing the self-inverting permutation π^* (see, properties $P_1 - P_3$ in Subsection 3.2.1).

The encoding algorithm, which we call `Encode_SIP.to.RPG-Bitonic-S` is described below.

Algorithm `Encode_SIP.to.RPG-Bitonic-S`

1. Compute the bitonic subsequences $S_1, S_2, ..., S_k$ of the self-inverting permutation π^* and let $S_i = (i_1, i_2, ..., top(S_i), ..., i_t)$;

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

```
    2.1  $V(F_s[\pi^*]) = \{s = u_{n^*+1}, u_{n^*}, ..., u_1, u_0 = t\}$;
    2.2  for $i = n^*, n^* - 1, ..., 0$ do
             add the edge $(u_{i+1}, u_i)$ in $E(F_s[\pi^*])$;
3.  For each bitonic subsequence $S_i$, $1 \leq i \leq k$, do
    3.1  add the edge $(u_{top(S_i)}, s)$ in $E(F_s[\pi^*])$;
    3.2  for $j = 1, 2, \ldots, top(S_i), \ldots, t - 1$ do
             if $i_j < i_{j+1}$ then add the edge $(u_{i_j}, u_{i_{j+1}})$
             else the edge $(u_{i_{j+1}}, u_{i_j})$ in $E(F_1[\pi^*])$;
4.  Return the graph $F_s[\pi^*]$;
```

Figure 4.1 shows the encoding of the SiP $\pi^* = (6, 8, 11, 10, 9, 1, 7, 2, 5, 4, 3)$ into the reducible permutation flow-graph $F_s[\pi^*]$; note that, π^* encodes the watermark number $w = 20$.

Time and Space Complexity. Let w be a watermark number and π^* be the self-inverting permutation of length $n^* = 2n + 1$ which encodes watermark w, where n is the length of the binary representation of number w. The subsequences $S_1, S_2, \ldots, S_k$ of π^* can be computed in $O(n^*)$ time and space. Furthermore, the construction of the graph $F[\pi^*]$ also takes $O(n^*)$ time and space. Thus, the following theorem holds.

Theorem 4.1 *The algorithm* `Encode_SIP.to.RPG-Bitonic-S` *for encoding the permutation π^* of length n^* as a reducible permutation flow-graph $F_s[\pi^*]$ requires $O(N)$ time and space, where $N = n^* + 2$.*

We next present the decoding algorithm `Decode_RPG.to.SIP-Bitonic-S`, which takes as input a reducible permutation flow-graph $F_s[\pi^*]$ on $n^* + 2$ nodes produced by algorithm `Encode_SIP.to.RPG-Bitonic-S` and correctly extract the self-inverting permutation π^* from the graph $F_s[\pi^*]$; the algorithm is described below.

Algorithm `Decode_RPG.to.SIP-Bitonic-S`

```
1.  Delete the directed edges $(u_{i+1}, u_i)$, $1 \leq i \leq n$, and the node $t = u_0$ from $F_s[\pi^*]$,
    and flip all the remaining directed edges in $F_s[\pi^*]$; let $s = u_0, u_1, u_2, \ldots, u_n$ be
    the nodes in the resulting graph $T_1[\pi^*]$;
```

2. Compute the set $R = \{u_j \mid (s, u_j) \in T_1[\pi^*]\}$ and delete the directed edges (s, u_j) from the graph $T_1[\pi^*]$;

3. Sort the nodes of set R in descending order according to their labels and let $R^* = (r_1, r_2, ..., r_k)$ be the resulting sorted sequence, $1 \leq k < n$;

4. Construct the underlying graph $H[\pi^*]$ of the directed graph $T_1[\pi^*]$ and let $C(r_1)$, $C(r_2)$, . . ., $C(r_k)$ be the connected components of the graph $H[\pi^*]$ which contain the nodes $r_1, r_2, ..., r_k$, respectively;

5. For each node $r_i \in R^*$, $i = 1, 2, ..., k$, perform BFS-search in graph $C(r_i)$ starting at node u and compute the sequence b_i^* of the nodes of $C(r_i)$ taken by the order in which they are BFS-discovered; the starting node u is selected as follows:

   ◦ if $i < k$ and $deg(r_i) = 2$, then u is the node with minimum label in $C(r_i)$;

   ◦ if $i < k$ and $deg(r_i) \leq 1$, then u is the node with maximum label in $C(r_i)$;

   ◦ if $i = k$, then u is the node with label ℓ_{max} in $C(r_i)$, where ℓ_{max} is the index of the max element in sequence $(b_1^*)^R$;

   recall that, $(b_i^*)^R$ denotes the reverse sequence of b_i^*, $1 \leq i \leq k$;

6. Return $\pi^* = (b_1^*)^R || b_2^* || \cdots || b_{k-1}^* || (b_k^*)^R$;

Figure 4.1 shows the extraction of the self-inverting permutation $\pi^* = (6, 8, 11, 10, 9, 1, 7, 2, 5, 4, 3)$, which encodes the watermark number $w = 20$, from the reducible permutation flow-graph $F_s[\pi^*]$ using the tree $T_1[\pi^*]$. Note that a reducible permutation flow-graph $F_s[\pi^*]$ of size $n^* + 2$ encodes a self-inverting permutation π^* of length n^*.

Time and Space Complexity. The size of the reducible permutation graph $F_s[\pi^*]$ constructed by the algorithm Encode_RPG.to.SIP-Bitonic-S is $O(N)$, where $N = n^* + 2$ and n^* is the length of the permutation π^*, and thus the size of the resulting graph $T_1[\pi^*]$ is also $O(N)$. The construction of underlying graph $H[\pi^*]$ takes $O(N)$ time and space. It is well known that the BFS-search on the connected components $C(r_1)$, $C(r_2)$, . . ., $C(r_k)$ of the graph $H[\pi^*]$ takes time linear. Thus, the decoding algorithm is executed in $O(N)$ time using $O(N)$ space. Thus, the following theorem holds:

Theorem 4.2 *Let $F_s[\pi^*]$ be a reducible permutation flow-graph of size $O(N)$ produced by algorithm* Encode_SiP.to.RPG-Bitonic-S. *The algorithm* Decode_RPG.to.SIP-Bitonic-S *decodes the flow-graph $F_s[\pi^*]$ in $O(N)$ time and space.*

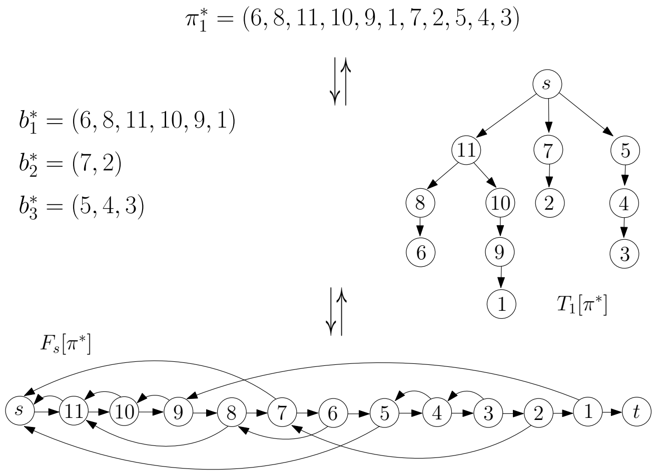

Figure 4.1: The main structures used or constructed by the codec algorithms `Encode SiP.to. RPG-Bitonic-S` and `Decode RPG.to.SiP-Bitonic-S`.

4.2 T-bitonic Algorithm

In this section we enrich the repository of reducible permutation flow-graphs $F[\pi^*]$ which can encode a self-inverting permutation π^* or, equivalently, a watermark number w by proposing a reducible permutation flow-graph $F_t[\pi^*]$, different from $F_s[\pi^*]$ but of the same type, having also important properties deriving from the bitonic subsequences of π^*.

By construction, the indegree of the first node $s = u_{n^*+1}$ of the flow-graph $F_s[\pi^*]$ is equal to the number of bitonic subsequences $b_1^*, b_2^*, \cdots, b_k^*$ of π^*, while the indegree of the first node of the graph $F_t[\pi^*]$ is much smaller that k. This property causes $F_t[\pi^*]$ more appropriate, in same cases, since it does not contain an extreme characteristic thereby enhancing the resilience of graph-structure to attacks. The proposed algorithm `Encode_SIP.to.RPG-Bitonic-T` for encoding a self-inverting permutation π^* into a reducible permutation graph $F_t[\pi^*]$ is described below.

Algorithm `Encode_SIP.to.RPG-Bitonic-T`

1. Execute algorithm `Encode_SIP.to.RPG-Bitonic-S` and compute the bitonic subsequences $S_1, S_2, ..., S_k$ of π^* and the graph $F_s[\pi^*]$; Set $F_t[\pi^*] \leftarrow F_s[\pi^*]$;

2. For each edge $(u_{top(S_i)}, s)$ in $F_t[\pi^*]$, $2 \leq i \leq k$, do

   if S_i is an id-bitonic sequence, then

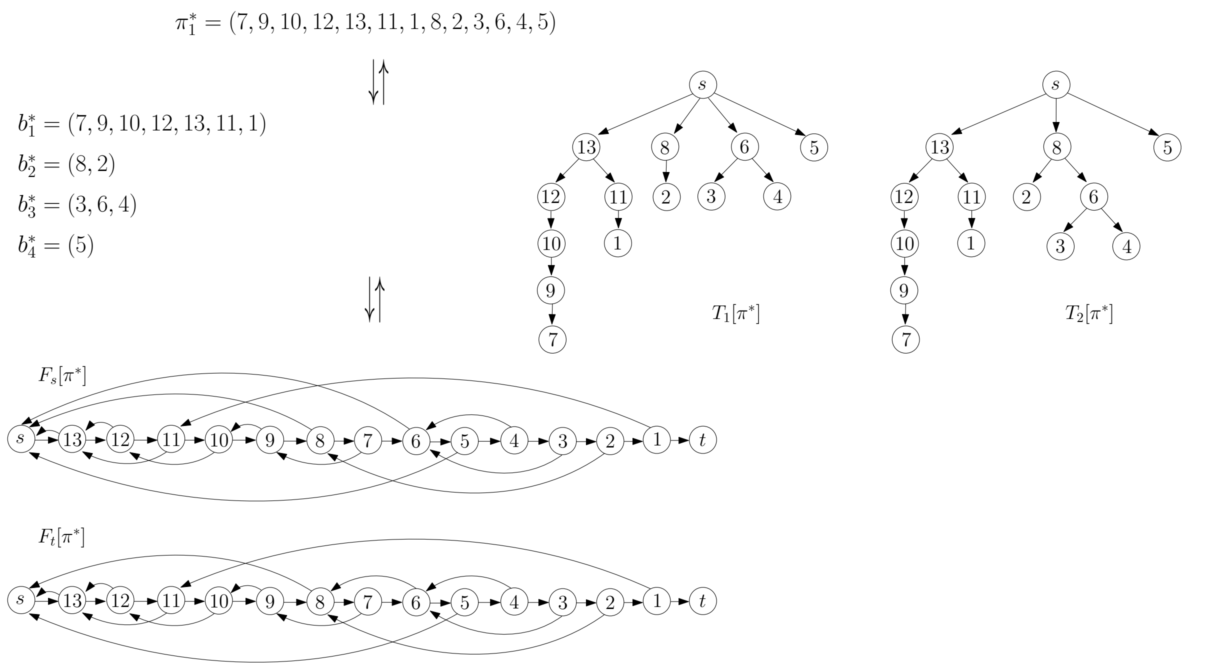

Figure 4.2: The main structures used or constructed by the codec algorithms `S-bitonic` and `T-bitonic`.

- ○ delete the edge $(u_{top(S_i)}, s)$ and
- ○ add the edge $(u_{top(S_i)}, u_{top(S_{i-1})})$ in $E(F_2[\pi^*])$;

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

We next describe the corresponding decoding algorithm for extracting the permutation π^* from the flow-graph $F_t[\pi^*]$.

Algorithm `Decode_RPG.to.SIP-Bitonic-T`

1. Execute Steps 1 and 2 of algorithm `Decode_RPG.to.SIP-Bitonic-S` on graph $F_t[\pi^*]$ and compute the directed graph $T_2[\pi^*]$ and the node set R;

2. Compute the node set

   $R' = \{u_j \mid (u_i, u_j) \in T_2[\pi^*] \text{ and } outdeg(u_j) \geq 2\}$,

   delete the directed edges (u_i, u_j) from the graph $T_2[\pi^*]$, and set $R \leftarrow R \cup R'$;

3. Execute Steps 3, 4 and 5 of the decode algorithm `Decode_RPG.to.SIP-Bitonic-S` and compute the sequences $b_1^*, b_2^*, \ldots, b_k^*$;

4. Return $\pi^* = (b_1^*)^R || b_2^* || \cdots || b_{k-1}^* || (b_k^*)^R$;

In the example of Figure 4.2, $R = \{13, 8, 5\}$ and $R' = \{6\}$. Recall that, the self-inverting permutation which encodes watermark w is of length $n^* = 2n + 1$, where n is the binary length of the watermark number w, while the reducible permutation flow-graph $F_t[\pi^*]$ is of size $n^* + 2$.

The results of this section concerning the correctness and the time and space complexity of both algorithms are summarized in the following theorem.

Theorem 4.3 *The algorithm* `Encode_SIP.to.RPG-Bitonic-T` *encodes a permutation π^* of length n^* into a reducible permutation flow-graph $F_t[\pi^*]$ in $O(N)$ time and space, where $N = n^* + 2$, and the corresponding decoding algorithm* `Decode_RPG.to.SIP-Bitonic-t` *correctly extract π^* from the flow-graph $F_t[\pi^*]$ within the same time and space complexity.*

4.3 Comparison of our algorithms

It would be very interesting to compare the two codec algorithms, S-bitonic Algorithm and T-bitonic Algorithm. More precisely, we compare the structures of reducible permutation flow-graphs $F_s[\pi^*]$ and $F_t[\pi^*]$, their properties, and also the space and time complexity which are required.

First of all, the structures of the watermark graphs which are produced by Algorithm `Encode_ SIP.to.RPG-Bitonic-S` and Algorithm `Encode_SIP.to.RPG-Bitonic-T` encoding the same self-inverting permutation π^* are similar. The reducible permutation flow-graphs $F_s[\pi^*]$ and $F_t[\pi^*]$ are node-labeled on $n^* + 2$ nodes, where n^* is the length of π^*. Indeed, the labels of $F_s[\pi^*]$ and $F_t[\pi^*]$ 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}, $1 \leq i \leq n$. Graphs are directed on $n^* + 2$ nodes with outdegree at most two. More precisely, the body nodes have $outdeg(u_{n^*+1-i}) = 2$, the initial node s has $outdeg(s) = 1$ and the terminal node t with $outdeg(t) = 0$. Obviously, they have exactly the same number of edges. For every node u of either $F_s[\pi^*]$ or $F_t[\pi^*]$, there is an edge (v, u), where $u + 1 \leq v \leq 2n + 1$ or $v = s$ or $v = u - 1$. Moreover, it is easy to see that both of these graphs have the same Hamiltonian path.

There is always $(2n + 1, s)$ edge in both of these flow-graphs $F_s[\pi^*]$ and $F_t[\pi^*]$, because $2n+1$ is the maximum element of self-inverting permutation which is in its first bitonic subsequence b_1^*.

From Theorems 4.1, 4.2 and 4.3, we observe that both codec systems require $O(N)$ time and space for encoding the permutation π^* as a reducible permutation flow-graph $F[\pi^*]$ and decoding π^* from theses graphs. Although, the Algorithm `Encode_RPG.to.SIP-Bitonic-t` check whether any subsequence b_i^* is id-bitonic and computes one more node set R', it does not effect its time and space complexity.

Therefore, it is also important to point out that, if self-inverting permutation π^* does not have id-bitonic subsequence, the reducible permutation flow-graphs $F_s[\pi^*]$ and $F_t[\pi^*]$ have identical structures.

On the other hand, we proposed two flow-graphs $F_s[\pi^*]$ and $F_t[\pi^*]$ which by construction have some different properties so that they follow aim which is to enhance our ability to select a graph structure being more similar to the structure of a given application program P. The main difference is based on the way in which the edges are constructed. The former links all top elements of subsequences b_i^* with s node in $F_s[\pi^*]$, while the latter checks whether the bitonic subsequences b_i^* of the self-inverting permutation π^* are id-bitonic or not and if bitonic subsequence b_i^* is id-bitonic, the $top(b_i^*)$ is linked with $top(b_{i-1}^*)$. On the contrary, from the S-bitonic Algorithm. It implies that

$$indeg_{F_s[]}(s) \geq indeg_{F_t[]}(s)$$

.

It is also necessary to consider the number of nodes that a node can be linked. We distinguish the following five cases for $F_s[\pi^*]$ and $F_t[\pi^*]$:

- If $u_k \in V(F_s[\pi^*])$:
  - ○ $k=1$, then u_k can be linked with n nodes, because the edge (u_k, u_i) can exist, if $i \in [n+3, 2n+1] \cup \{t\}$.
  - ○ $k=2$: the node u_k can be linked with $n+1$ nodes, because the edge (u_k, u_i) can exist, if $i \in [k+1, n] \cup \{n+k\} \cup \{n+k+1\} \cup \{1\}$.
  - ○ $2 < k \leq n-1$: the node u_k can be linked with $n-k+4$ nodes, because the edge (u_k, u_i) can exist, if $i \in [k+1, n] \cup \{n+k\} \cup \{n+k+1\} \cup \{s\} \cup \{k-1\}$.
  - ○ $n \leq k < 2n+2$: the node u_k can be linked with $2n-k+1$ nodes, because the edge (u_k, u_i) can exist, if $i \in [k+1, 2n+1] \cup \{k-1\}$.

- $n = 2n + 1$: the node u_k can be linked with 2 nodes, because there are edges (u_{2n+1}, s) and (u_{2n}, u_{2n+1}).

- If $u_k \in V(F_t[\pi^*])$:
  - if $k = 1$: it is the same with $F_s[\pi^*]$.
  - if $k = 2$: it is the same with $F_s[\pi^*]$.
  - if $2 < k \leq n - 1$: the node u_k can be linked with $n + 1$ nodes, because the edge (u_k, u_i) can exist, if $i \in [k + 1, n] \cup [n + 2, n + k + 1] \cup \{k - 1\}$.
  - if $n \leq k < 2n + 1$: it is the same with $F_s[\pi^*]$.
  - if $n = 2n + 1$: it is the same with $F_s[\pi^*]$.

Chapter 5

Attacks

A watermarking system is usually divided into three distinct steps, embedding, attack, and extracting. In this section, we show that the malicious intentions of an attacker to a reducible permutation graph $F[\pi^*]$ by modifying some node-labels or edges of the graph $F[\pi^*]$ can be efficiently detected. Also, we present all probabilities of edge and label modifications.

5.1 Node-Label and Edges Modification

In Chapter 2, we introduced the ways to attack a watermark in an application program. In this part, we present how we can detect an attack in our graph watermark, like node modification or edge modification.

The labels of $F[\pi^*]$ by encoding algorithms Encode_SiP.to.RPG-Bitonic-S or Encode_ SiP.to.RPG-Bitonic-T are numbers of the set $\{0, 1, \ldots, n^* + 1 \}$, where n^* is the length of π^*. We have considered that the header node is $s = u_{n^*+1}$, the footer node is $t = u_0$ and body nodes are u_{n^*+1-i}, $1 \leq i \leq n$.

A node-label attack of $F[\pi^*]$, can mainly occur in the following three ways:

1. Swapping of the labels of two nodes of $F[\pi^*]$,
2. Altering the value of the label of a node, and
3. Node-unlabeled graph, i.e, removing all the labels of the graph $F[\pi^*]$.

The algorithms `Decode_RPG.to.SiP-Bitonic-S` and `Decode_RPG.to.SiP-Bitonic-T` rely on the labels of the flow graph $F[\pi^*]$. It would be problem for our code system $(\texttt{encode, decode})_{F[\pi^*]}$, if it is susceptible to node-label modification attack. But we have a way to know if our flow graph $F[\pi^*]$ has been attacked and we can be able to correct the label of nodes. 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.

Lemma 5.1 *Let $F[\pi^*]$ be a reducible permutation graph of size $O(N)$ produced by either algorithm* `Encode_SiP.to.RPG-Bitonic-S` *or -2, 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.*

However, an attacker can modify the edges of reducible permutation graph $F[\pi^*]$, so our codec system should have appropriate properties with respect to resilience to these attacks. During the construction of a RPG $F[\pi^*]$, an edge can be classified as one of two types: forward edge (f_1, f_2), $f_1 < f_2$ or backward edge (b_1, b_2), $b_1 > b_2$. The forward edges can be retrieved with the same way as label-node from HP($F[\pi^*]$). The backward edges require more attention for their retrieving. Let $F[\pi^*]$ be a flow-graph which encodes the SiP representing an integer w and let $F'[\pi^*]$ be the graph resulting from $F[\pi^*]$ after an edge modification.
There are two cases:

(i) our codec system fail to return the SiP, or

(ii) our codec system extracts $F'[\pi^*]$ and returns a SiP $\pi' \neq \pi^*$

The properties of self-inverting permutation π^* (see Subsection 3.2.1), like odd-length property or 1-cycle property, are incorporated in the structure of the reducible permutation graph $F[\pi^*]$. Thus, if an attacker makes any single node-modification in

$F[\pi^*]$, i.e., node-addition or node-deletion, it can be easily identified. Although, it is the first case, if at least one of the SiP properties does not hold. If $F[\pi^*]$ decodes a permutation $\pi' \neq \pi^*$, then the subsequence X (Y, resp.) may not be increasing and the bitonic property does not hold. In the second case, an attacker makes appropriate edge-modifications to $F[\pi^*]$ so that the resulting graph $F'[\pi^*]$ decodes a permutation $\pi^{*\prime}$ 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.

In addition, we can use the properties of 1-cycle of self-inverting permutation π^* for detecting an edge modification in $F[\pi^*]$. For example, we always know where the edge of 1-cycle node is linked.

Property 5.1 *Let π^* be a self-inverting permutation. If the first increasing subsequence of π^* is known, we can construct the SiP $\pi^* = (\pi_1^*, \pi_2^*, \ldots, \pi_n^*)$.*

Proof. The proof is based on properties of self-inverting permutation π^* and $1-cycle$ (Chapter 3). From the first increasing subsequence of π^*, we can compute which element is the $1-cycle$ because it is the first missing consecutive element from this subsequence. Also we know from the properties of partition $b_1^*, b_2^*, \ldots, b_k^*$ of π^*, that b_1^* contains all elements between $n+1$ to $2n+1$. Obviously, π_{max}^* is the top element of b_1^* and after that there is a decreasing subsequence. The final step is easy because it is self-inverting permutation and applies $\pi_{\pi_i} = i$, so we know all elements and its indexes of SiP π^*. □

Example 5.1 If the first increasing subsequence of π^* is $(6, 11)$, we can extract all elements of self-inverting permutation $\pi^* = (\pi_1^*, \pi_2^*, \ldots, \pi_{11}^*)$. First of all, we know that $\pi_{max}^* = 11$, $\pi_6^* = 1$ and $\pi_{11}^* = 2$. Also, from properties of $1-cycle$, the 1-cycle is 7, so $\pi_7^* = 7$. We know that the first bitonic subsequence b_1^* contains all elements between $n+1$ to $2n+1$. It is obviously that we have an increasing subsequence from $n+1$ until $2n+1$ element and after a decreasing subsequence until minimum element $\pi_{min}^* = 1$. So, $\pi_3^* = 10$, $\pi_4^* = 9$, $\pi_5^* = 8$. Finally from $\pi_{\pi_i} = i$, we can find all elements from indexes of elements of b_1^*. Thus, the self-inverting permutation is $\pi^* = (6, 11, 10, 9, 8, 1, 7, 5, 4, 3, 2)$.

To sum up, if an attacker does not change the edges from the nodes with labels, which are the first increasing subsequense's elements of self-inverting permutation π^*, we can construct the right $F[\pi^*]$ and extract the original π^*.

5.2 Probability for Modification

First of all, there are many parameters which affect the probability of edge existence in $F[\pi^*]$. Let a reducible permutation graph $F_s[\pi^*]$ produced by Algorithm `Encode_SIP.to.RPG-Bitonic-S` and a RPG $F_t[\pi^*]$ produced by Algorithm `Encode_SIP.to.RPG-Bitonic-T` on $n^* + 2$ vertices, where $n^* = 2n + 1$. The number of all nodes that have a backward edge, is $2n + 1$ (s is header node and $outdeg(t) = 0$). Let we have a node u_k, where k is its label. We can compute the probabilities for any "real" edge in a RPG encoding by S-bitonic and T-bitonic Algorithm (see Section 5.1), which is given by the following:

- If $k = 1$, then the probability of its node to linked is $p_1 = \frac{1}{n-1}$, $u_k \in V(F_1[\pi^*])$ or $u_k \in V(F_2[\pi^*])$.
- If $k = 2$, then the probability of its node to linked is $p_2 = \frac{1}{n-1}$, $u_k \in V(F_1[\pi^*])$ or $u_k \in V(F_2[\pi^*])$.
- If $2 < k < n$, then the probability of its node to linked is $p_k = \frac{1}{n-k+3}$, $u_k \in V(F_1[\pi^*])$ or $p_k = \frac{1}{n}$, $u_k \in V(F_2[\pi^*])$.
- If $n \leq k \leq 2n + 1$, then the probability of its node to linked is $p_k = \frac{1}{2n-k+1}$, $u_k \in V(F_1[\pi^*])$ or $u_k \in V(F_2[\pi^*])$.

However, if an attacker knows that it should change only the backward edges in order to succeed, the probability of construction an edge from u_k node is:

$$p_k = \frac{1}{2n + 1 - k} \tag{5.1}$$

In addition, we consider the probability an attack to our flow-graph $F_\bullet[\pi^*]$, when $F_\bullet[\pi^*] \in \{F_\bullet[\pi^*], F_\bullet[\pi^*]\}$ to be "almost" successful. It means that our codec system extracts $F'[\pi^*]$ and returns a SiP $\pi^{*\prime} \neq \pi^*$ (Case (ii) in Subsection 5.1). Let S be the set of all pairs $(F_i[\pi^*], F_j[\pi^*])$ of RPGs of order $2n + 3$ and let $emf(i, j)$ denote the number of edge modifications made in $F_i[\pi^*]$ so that $F_i[\pi^*] = F_j[\pi^*]$. Let S' be the set of all pairs $(F_i[\pi^*], F_j[\pi^*])$ with the minimum value of $emf(i, j)$. The first step towards to compute the above probability is to find set S'. The minimum number of edge modifications is two, but there is low probability that it can occur. We want to know all pairs of integers (w_1, w_2), where have different SiPs π_1^* and π_2^* but their flow graphs $F_s[\pi^*]$ or $F_t[\pi^*]$, from Algorithms `Encode_SIP.to.RPG-Bitonic-S` and `-2`,

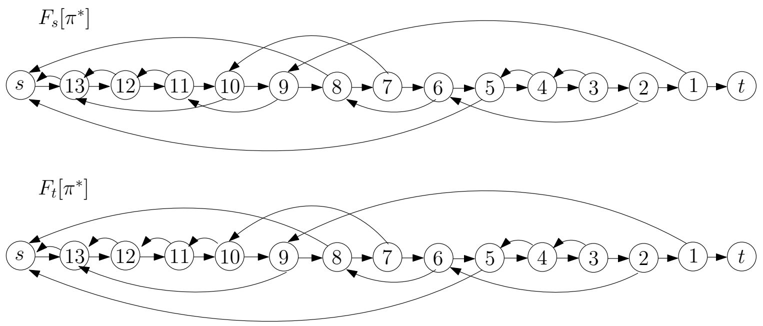

Figure 5.1: The RPGs of $\pi_1^* = (7, 10, 13, 12, 11, 9, 1, 8, 6, 2, 5, 4, 3)$ from $w_1 = 36$ and $\pi_2^* = (7, 10, 11, 12, 13, 9, 1, 8, 6, 2, 3, 4, 5)$ from $w_2 = 39$.

differ in two edges. All these pairs are (w_1, w_2), where $w_1 = a010^b$ and $w_2 = a011^b$, a is a binary number and $1 \leq b \leq n - 3$. There are 2 edge modifications in their RPGs because only two elements of their SiPs change their indices.

Example 5.2 Let $w_1 = 36$ with binary representation 100100 and $w_2 = 39$ with binary representation 100111 be two watermarks. Their self-inverting permutations are $\pi_1^* = (7, 10, 13, 12, 11, 9, 1, 8, 6, 2, 5, 4, 3)$ and $\pi_2^* = (7, 10, 11, 12, 13, 9, 1, 8, 6, 2, 3, 4, 5)$. Figure 5.1 shows the encoding of the self-inverting permutations π_1^* and π_2^* into the reducible permutation flow graphs $F_s[\pi^*]$ and $F_t[\pi^*]$. Obviously, the different edges from $F_s[\pi^*]$ to $F_t[\pi^*]$ are $(9, 11)$ and $(11, 12)$. The main difference is the bitonic subsequences. The bitonic subsequences are $b_{11}^* = (7, 10, 13, 12, 11, 9, 1)$, $b_{12}^* = (8, 6, 2)$, $b_{13}^* = (5, 4, 3)$ and $b_{21}^* = (7, 10, 11, 12, 13, 9, 1)$, $b_{22}^* = (8, 6, 2)$, $b_{23}^* = (3, 4, 5)$. In consequence, $b_{11}^* \neq b_{12}^*$ with 3 different indices of elements, $b_{12}^* = b_{22}^*$ and $b_{13}^* = (b_{23}^*)^R$. Now, it is easy to see that we need 2 edge modifications that $E(F_s[\pi^*]) = E(F_t[\pi^*])$.

The probability to choose a pair of integers $(w_i, w_j)_n$ such that the corresponding RPGs $F_i[\pi^*]$ and $F_j[\pi^*]$ belong to S' is based on the following result.

Lemma 5.2 *Let $(w_1, w_2)_n$ be a pair of watermark integers with the same length n of their binary representations, which encode $(F_1[\pi_1^*], F_2[\pi_2^*])_n$ (F_1 and F_2 are produced by Algorithm* `Encode_SIP.to.RPG-Bitonic-S` *or* `-2`*). The probability p_n that $F_1[\pi^*]$ and $F_2[\pi^*]$ differ in 2 edges, is*

$$p_n = \begin{cases} 0, n \leq 3 \\ \\ \frac{2^{n-2}-1}{2^{n-1}-1} \cdot p_{n-1} + \frac{1}{2^{n-2}(2^{n-1}-1)}, n > 3 \end{cases} \tag{5.2}$$

Proof. There are 2^{n-1} different integers in the same range n, where n the length of the binary representation of them and there are $2^{n-2}(2^{n-1}-1)$ possible pairs. The number of pairs $(w_1, w_2)_n$, that w_1 encodes a RPG which differs only in 2 edges with RPG of w_2, is $b_n = 2b_{n-1} + 1$. So, we have the following probabilities:

$$p_n = \frac{b_n}{2^{n-2}(2^{n-1}-1)} = \frac{2b_{n-1}+1}{2^{n-2}(2^{n-1}-1)} \tag{5.3}$$

$$p_{n-1} = \frac{b_{n-1}}{2^{n-3}(2^{n-2}-1)} \Leftrightarrow b_{n-1} = p_{n-1}2^{n-3}(2^{n-2}-1) \tag{5.4}$$

From 5.2 and 5.3:

$$p_n = \frac{2p_{n-1}2^{n-3}(2^{n-2}-1)+1}{2^{n-2}(2^{n-1}-1)} = \frac{2^{n-2}-1}{2^{n-1}-1}p_{n-1} + \frac{1}{2^{n-2}(2^{n-1}-1)}$$

Then

$$p_n = \begin{cases} 0, n \leq 3 \\ \frac{2^{n-2}-1}{2^{n-1}-1}p_{n-1} + \frac{1}{2^{n-2}(2^{n-1}-1)}, n > 3. \end{cases}$$

□

It is easy to see from Lemma 5.2 that, when we have a big length n of binary represantation, this probability p_n decreases if we choose a random integer w_n for coding. Also, these edge modifications are only in the nodes that its labels represent the elements of first bitonic subsequence of a self-inverting permutation.

The integers w_1,w_2 of pair $(w_1, w_2)_n$ of the above lemma are called as *weak*. Next, we give the formal definition of the $emf(F[\pi_1^*], F[\pi_2^*])_n$, which is the number of edge modification in $F[\pi_1^*]$ and $F[\pi_2^*]$.

Definition 5.1 Let $F[\pi_1^*], F[\pi_2^*]$ be two different reducible permutation graphs. We define $emf(F[\pi_1^*], F[\pi_2^*])_n$ to be the number of edge modification from $F[\pi_1^*]$ to $F[\pi_2^*]$ and n is the length of binary representation w_1 and w_2 where encode $F[\pi_1^*]$ and $F[\pi_2^*]$.

All binary numbers have a recursive relationship. Specifically, a binary number, in range n, $n \geq 2$, has binary subsequences which are binary numbers in range n',

$n' < n$. It implies that there are RPGs $(F[\pi_1^*], F[\pi_2^*])_n$ that need the same numbers of edges modification with RPGs $(F[\pi_1^*], F[\pi_2^*])_{n'}$ if they are the form of $(w_1)_n = 1^k(w_1')_{n'}$ and $(w_2)_n = 1^k(w_2')_{n'}$, $k = n - n'$ where encode $F[\pi_1^*]$ and $F[\pi_2^*]$. For example, let $F[\pi_1^*]$ encodes the watermark integer $w_1 = 11000$, $F[\pi_2^*]$ encodes the integer $w_2 = 11001$ and the length of binary representation $n = 5$. The RPGs have $emf(F[\pi_1^*], F[\pi_2^*])_5 = 4$ and the RPGs also have $emf(F'[\pi_1^*], F'[\pi_2^*])_4 = 4$, where $F'[\pi_1^*]$ decodes $w_1' = 1000$ (w_1' is a subsequence of w_1) and $F'[\pi_2^*]$ decodes $w_2' = 1001$ (w_2' is a subsequence of w_2).

Let $emf(F[\pi_1^*], F[\pi_2^*])_n = l$, and let $F[\pi_1^*]$ encodes w_1 and $F[\pi_2^*]$ encodes w_2. In order to help us to find the value of $emf(F'[\pi_1^*], F'[\pi_2^*])_{n-k+1}$, where $F'[\pi_1^*]$ encodes w_1', $w = 1^k w_1'$ and $F'[\pi_2^*]$ encodes w_2', where $w_2 = 1^k w_2'$, $w_1 < w_2$ and are binary numbers with n the length of binary representation, we next present the following result:

Lemma 5.3 *Let $F[\pi_1^*], F[\pi_2^*]$ be two different reducible permutation graphs with*

$$emf(F[\pi_1^*], F[\pi_2^*])_n = l$$

and let n be the length of binary representation of $w_1 = 1^k w_1'$ and $w_2 = 1^k w_2'$ where encode $F[\pi_1^*]$ and $F[\pi_2^*]$. It implies that $emf(F[\pi_1^*], F[\pi_2^*])_i = l, i < n - k + 1$. There are two cases for $emf(F[\pi_1^*], F[\pi_2^*])_{n-k+1}$:

(i) If w_1' and w_2' have the same first digit then $emf(F[\pi_1^*], F[\pi_2^*])_{n-k+1} = l$,

(ii) otherwise $emf(F[\pi_1^*], F[\pi_2^*])_{n-k+1} = l - 1$.

Proof. The proof of this property is based on construction of self inverting permutation. The first increasing subsequence of SiP, which is encoded by w_1 in range n, indicate the sequence Y^R of the 1s positions (See Subsection 3.1.3). So, the nodes with its elements as label will linked consecutively. If we add or remove first 1 bit in an integer w (if it is possible in range $n + 1$ or $n - 1$), the length of this sequence will be changed but not its structure, i.e., the consecutive link of nodes. Let be w_1' and w_2' have different first digit and 0 is the first bit of w_1'. The b_2^* of SiP of $1^k w_1'$ and $1^k w_2'$ are id-bitonic subsequences. But the b_2^* of SiP of $1w_1'$ is d-bitonic and the b_2^* of SiP of $1w_2'$ (its first bit of w_2' is 1) is id-bitonic. So, there is one less edge modification, $emf(F[\pi_1^*], F[\pi_2^*])_{n-k+1} = l - 1$. If w_1' and w_2' have the same first digit (i.e., the bit 0), it means that all subsequence of SiPs from $1w_1'$ and $1w_2'$ are the same kind of bitonic sequence (id-bitonic, i-bitonic and d-bitonic) with bitonic subsequences of SiPs from $1^k w_1'$ and $1^k w_2'$. □

To sum up, when we know all edge modifications that can be done between $(F[\pi_1^*], F[\pi_2^*])_n$ decoding pair $(w_1, w_2)_n$ from the same range n, we can know all edge modifications that can be done in range $n-i$, $i < n$. In addition, we can compute the edge modifications between $(F[\pi_1^*], F[\pi_2^*])_{n+1}$ of integer pairs $(w_1, w_2)_n$ with $w_1, w_2 \leq 2^{n'-1} + 2^{n'-2}$ in the range $n' = n+1$.

Let $w_1 = 10$ and $w_2 = 11$ be binary numbers which encode $F[\pi_1^*]$ and $F[\pi_2^*]$ using Algorithm `Encode_SIP.to.RPG-Bitonic-S` or `-2` and let $emf(F[\pi_1^*], F[\pi_2^*])_2 = 3$. Based on previous results, we should notice that any RPG $F[\pi^*]$ from the penultimate integer with n bits, i.e., $2^n - 2$, always differs 4 edges from $F'[\pi]$ which decodes the last one integer in the same range, i.e., $2^n - 1$. But, we point out that this difference is the smallest on between $F[\pi^*]$ and any other RPG by two different SiPs, which have different indices of 1-cycle, in the same range n. The 1-cycle of its self-inverting permutation is unique and it change from SiP of the previous or next watermark integer. Thus, we have the following lemma:

Lemma 5.4 *Let π^* be the self-inverting permutation of length n^*, where $n^* = 2n+1$ and $n^* \geq 3$, with its 1-cycle having index i. We need $emf(F[\pi^*], F'[\pi^*])_n \geq 4$, where $F'[\pi^*]$ decodes a SiP by* `S-bitonic Algorithm` *with 1-cycle is in index $i+1$, $n+2 \leq i \leq n^*$.*

Proof. Let π_1^* and π_2^* be the self-inverting permutations of length n^*, where $n^* = 2n+1$ and $n^* \geq 3$ and $F_1[\pi^*], F_2[\pi^*]$ the corresponding reducible permutation graphs. The 1-cycle of π_1^* is the index i and the 1-cycle of π_2^* is the index $i+1$. When the index of 1-cycle change, from i to $i+1$, it implies that two elements of SiP p^* are swapped by b_1^* and b_2^* bitonic subsequences. We know from properties of 1-cycle that b_2^* always contains the 1-cycle, which is the top element of it. Encoding $F_1[\pi^*]$ and $F_2[\pi^*]$ from `S-bitonic Algorithm`, it needs at least 4 edge modifications, because the nodes with label, which is the top element of any bitonic subsequences b_i^*, are linked with initial node s, so, $emf(F[\pi^*], F'[\pi^*])_n \geq 4$. □

Lemma 5.5 *Let π^* be the self-inverting permutation of length n^*, where $n^* = 2n+1$ and $n^* \geq 3$, with its 1-cycle having index i. We need $emf(F[\pi^*], F'[\pi^*])_n \geq 4$, where $F'[\pi^*]$ decodes a SiP by* `T-bitonic Algorithm` *with 1-cycle is in index $i+1$, $n+2 \leq i \leq n^*$.*

Proof. The proof is similar with proof of Lemma 5.4 and it also needs at least 4 edge modifications but it is necessary to research more because a RPG from `T-bitonic Algorithm` has more properties in construction. Encoding $F_1[\pi^*]$ and $F_2[\pi^*]$

from `T-bitonic Algorithm`, let be a node $u_{i_k} \in F_i[\pi^*]$ with label k. In $F_s[\pi^*]$, there are the edges $(s, u_{1_{1-cycle}})$ and $(u_{1_{max}}, u_{1_j})$, $j \in b^*_{1_1}$. In $F_t[\pi^*]$, there are the edges $u_{2_{max}}, u_{2_{1-cycle}}$ (because $b^*_{2_2}$ is id-bitonic) and $(u_{2_{max}}, u_{2_j})$. We observe that it is possible that $u_{1_{1-cycle}} = u_{2_j}$ and $u_{1_j} = u_{2_{1-cycle}}$. So there is not other edge modification. It implies $emf(F[\pi^*], F'[\pi^*])_n \geq 4$. □

A watermark w is called *strong* if its reducible permutation graph $F[\pi^*]$ produced by Algorithm `Encode_SIP.to.RPG-Bitonic-S` (or `-T`) has more different backward edges of any other $F'[\pi^*]$ decoding a watermark $w' \neq w$ of same length n. We can find a *strong* watermark integer w by exploiting the properties of $F[\pi^*]$ and $1 - cycle$ of π^*. In our case, the *strong* watermark in the range n (length of binary representation), is the number $2^n - 2$. This watermark integer has the following properties:

(a) its sum of all edge modifications which need to fit in any other RPG, is always maximum, and

(b) its 1-cycle of π^* is unique.

We thus conclude that, if we choose a large n, and the integer $2^n - 2$ which encodes a reducible permutation graph, the probability for ”almost” successful attack decreases.

Chapter 6

Comparison of Related Algorithms

This section is based on a comparative study between different watermarking techniques.

6.1 Encoding SiPs as Reducible Permutation Graphs

Graph-based watermarking schemes have received a lot of attention ever since and due emphasis must be given to the contributions of Collberg et al. in a series of papers [27, 28]. More recently, Chroni and Nikolopoulos presented an ingenious such scheme [13, 14, 15], where the generated watermark graphs constitute a subclass of reducible flow graphs. It would be very interesting to compare their codec watermarking scheme with ours because the reducible permutation graphs are produced by all these algorithms encode the same structure, that is, the self-inverting permutation π^*.

First of all, we present their proposed encoding algorithm `Encode_SIP.to.RPG-I`. They use dmax-domination relation, i.e., $d - dom(j)$ is the set of all the elements of the permutation π^* which d-dominates the element j, $dmax(j)$ is the element of the set $d - dom(j)$ with maximum value and 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 algorithm Encode_SIP.to.RPG computes the dmax-domination relation of each of the n elements of the self-inverting permutation π^*, and then, it constructs a directed graph $F_1[\pi^*]$ on $n^* + 2$ nodes using the dmax-domination relation of the elements of the permutation π^*.

Next, this encoding algorithm Encode_SIP.to.RPG-I is presented in detail (see, Figure 6.1).

Algorithm Encode_SIP.to.RPG-I

```
1. for each element $i \in \pi^*$, $1 \leq i \leq n^*$, do;
     ◦ set $P(i) = m$, where $m = dmax(i)$, i.e., $m$ is the element from $d - dom(i)$
       with the maximum value;
2. Construct a directed graph $F_1[\pi^*]$ on $n^* + 2$ vertices as follows:
     ◦ $V(F_1[\pi^*]) = \{s = u_{n^*+1}, u_{n^*}, ..., u_1, u_0 = t\}$;
     ◦ for $i = n^*, n^* - 1, ..., 0$ do
           add the edge $(u_{i+1}, u_i)$ in $E(F_1[\pi^*])$;
3. for each vertex $u_i \in V(F_1[\pi^*])$, $1 \leq i \leq n^*$, do
     ◦ add the edge $(u_i, u_m)$ in $E(F_1[\pi^*])$ where $m = P(i)$;
4. Return the graph $F_1[\pi^*]$;
```

Next, they propose a decoding algorithm, namely, Decode RPG.to.SiP-I, which takes as input a reducible permutation flow-graph $F_1[\pi^*]$ on $n^* + 2$ nodes constructed by algorithm Encode_SiP.to.RPG-I, and produces a self-inverting permutation π^* of length n^*. The only operation used by the algorithm are edge modifications on $F_1[\pi^*]$ and DFS-search on trees; it works as follows (see, Figure 6.1):

Algorithm Decode_RPG.to.SIP-I

```
1. Delete the directed edges $(u_{i+1}, u_i)$ from the edge set $E(F_1(\pi^*))$, $1 \leq i \leq n$, and
   the node $t = u_0$ from $V(F_1[\pi^*])$;
```

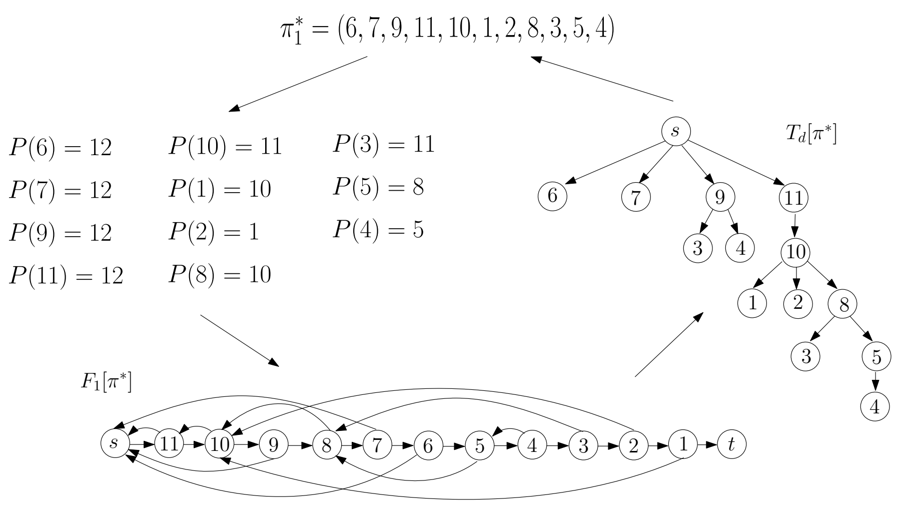

Figure 6.1: The main structures used or constructed by the codec algorithms `Encode_SiP.to.RPG` and `Decode_RPG.to.SiP`.

2. Flip all the remaining directed edges of the graph $F_1(\pi^*)$;
   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 $\pi^* = \pi$;

Furthermore, Chroni and Nikolopoulos propose another one coding system consisting of algorithms `Encode_SIP.to.RPG-II` and `Decode_RPG.to.SIP-II`. The first 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 $F_2[\pi^*]$ on $n^* + 2$ nodes using the subsequences $S_1, S_2, \ldots, S_k$.

Algorithm `Encode_SIP.to.RPG-II` is presented in detail (see, Figure 6.2).

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 $F_2[\pi^*]$ on $n^* + 2$ vertices as follows:
     ○ $V(F_2[\pi^*]) = \{s = u_{n^*+1}, u_{n^*}, ..., u_1, u_0 = t\}$;
     ○ for $i = n^*, n^* - 1, ..., 0$ do
           add the edge $(u_{i+1}, u_i)$ in $E(F_2[\pi^*])$;
3. For each decreasing subsequence $S_i = (i_1, i_2, \dots, i_t)$, $1 \leq i \leq k$, do
       add the edge $(u_{i_1}, s)$ in $E(F_2[\pi^*])$;
       for $j = t, t-1, \dots, 2$ do     add the edge $(u_{i_j}, u_{i_{j\text{-}1}})$ in $E(F_2[\pi^*])$;
4. Return the graph $F_2[\pi^*]$;
```

Having designed the encoding algorithm Encode_SiP.to.RPG-II, they next present the decoding algorithm Decode_RPG.to.SiP-II which takes as input a flow-graph $F_2[\pi^*]$ and extracts the self-inverting permutation π^* from $F_2[\pi^*]$ using the components of tree $T_s[\pi^*]$. It finds all pairs $P_1, P_2, \dots, P_k$ which are the cycles of self-inverting permutation π^*; it works as follows (see, Figure 6.2):

Algorithm Decode_RPG.to.SIP-II

```
1. Delete the directed edges $(u_{i+1}, u_i)$ from the edge set $E(F_2[\pi^*])$, $1 \leq i \leq n$, and
   the node $t = u_0$ from $V(F_2[\pi^*]) == \{s = u_{n^*+1}, u_{n^*}, ..., u_1, u_0 = t\}$;
2. Flip all the remaining directed edges of the graph $F_2[\pi^*]$; the resulting graph is
   a tree $T_s[\pi^*]$ rooted at $s = u_{n^*+1}$; let $v_{n^*+1}, v_{n^*} \dots, v_1$ 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 $\pi^* = (\pi^*_1, \pi^*_2, \dots, \pi^*_{n^*})$ to the identity permutation
   $(1, 2, \dots, n^*)$, and let $P$ be the set of all pairs $P_1, P_2, \dots, P_k$ computed at Step 3;
   then
     ○ for each pair $(v_i, v_j) \in P$, swap elements $\pi^*_i$ and $\pi^*_j$ in permutation $\pi^*$;
6. Return the self-inverting permutation $\pi^*$;
```

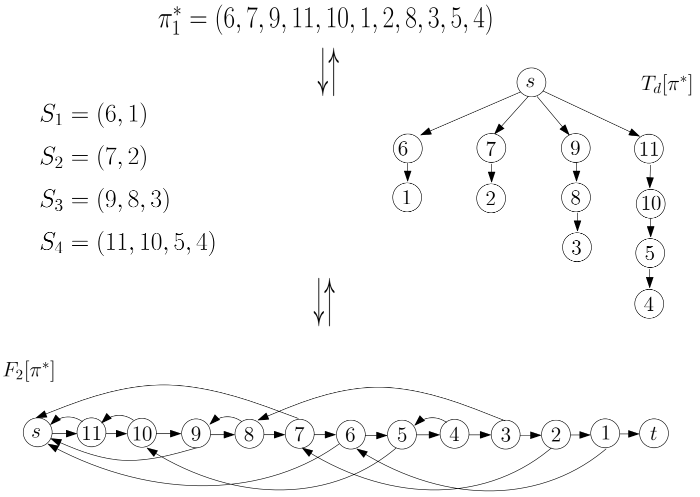

Figure 6.2: The main structures used or constructed by the codec algorithms `Encode SiP.to. RPG-II` and `Decode RPG.to.SiP-II`.

Their results are summarized in the following theorems.

Theorem 6.1. 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 6.2. 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`.

Chroni and Nikolopoulos extended the class of graph structures by proposing two different reducible permutation flow-graphs $F_1[\pi^*]$ and $F_2[\pi^*]$ and it would be very interesting to compare it with our proposed software watermarking model.

6.2 Our Algorithms

Our proposed algorithms are based on bitonic subsequences of self-inverting permutation π^*. The encoding algorithm `Encode_SiP.to.RPG-Bitonic-1` computes the bitonic subsequences $b_1^*, b_2^*, \ldots, b_k^*$ of SiP π^* and adds all forward edges in $E(F_s[\pi^*])$. Next, all nodes which their labels are the top elements of bitonic subsequences, are linked with

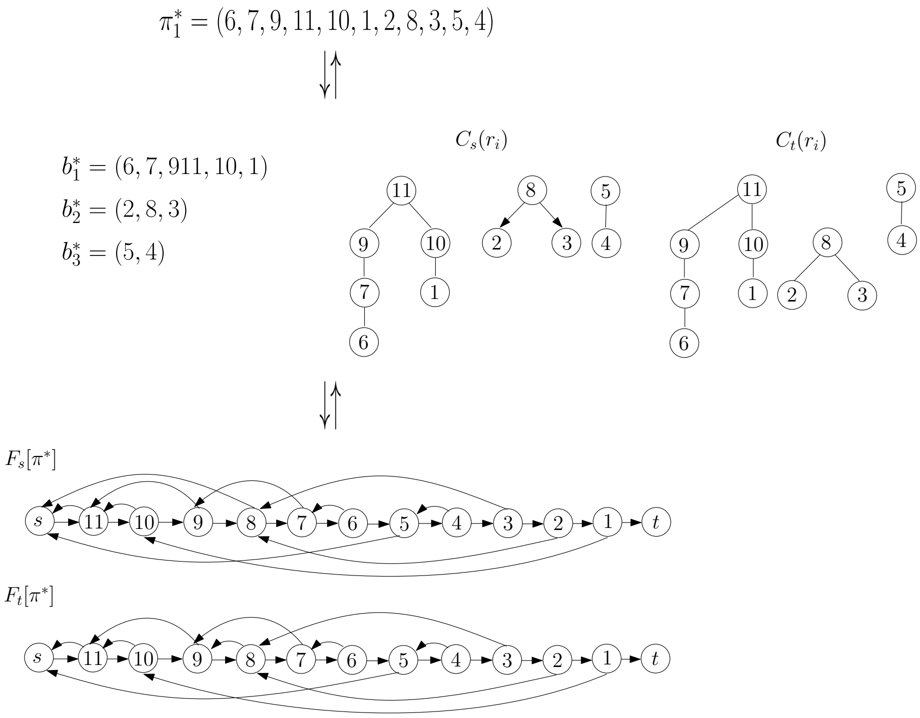

Figure 6.3: The main structures used or constructed by the codec system `S-bitonic Algorithm` and `t-Bitonic Algorithm`, that is, the self-inverting permutation π^*, the bitonic sequences b_i^*, the reducible permutation graphs $F_s[\pi^*]$ and $F_t[\pi^*]$, and connected components $C_s(r_k)$, $C_t(r_k)$ of underlying graphs.

header node s and the remaining nodes are linked in accordance with order. The decoding algorithm `Decode_SiP.to.RPG-Bitonic-1` extracts the self-inverting permutation π^* from RPG $F_s[\pi^*]$ by connecting components $C_s(r_k)$, $1 \leq k < n$, of undirected graph $H_s[\pi^*]$ and BFS-search for the construction of bitonic subsequences.

The encoding algorithm, `Encode_SiP.to.RPG-Bitonic-2`, exploits the id-bitonic subsequences (recall that, if either monotonically increases and then monotonically decreases) and the nodes, which are top elements of id-bitonic subsequences, are linked with node, which is top element of previously bitonic subsequence. The remaining edges of $F_t[\pi^*]$ coincide with $E(F_s[\pi^*])$. The decoding algorithm `Decode_SiP.to.RPG-Bitonic-2` extracts π^* from $F_t[\pi^*]$ by converting $F_t[\pi^*]$ into an undericted graph $H[\pi^*]$, too. Then, it applies BFS-search on connected components $C_t(r_k)$ of $H_t[\pi^*]$, $1 \leq k < n$, by construction of new set R', which has the nodes u_i with $outdeg(u_i) \geq 2$.

In Figure 6.3, we present an example of our encoding watermarking systems `S-bitonic` and `T-bitonic` with the same self-inverting permutation $\pi^* = (6, 7, 9, 11, 10, 1,$

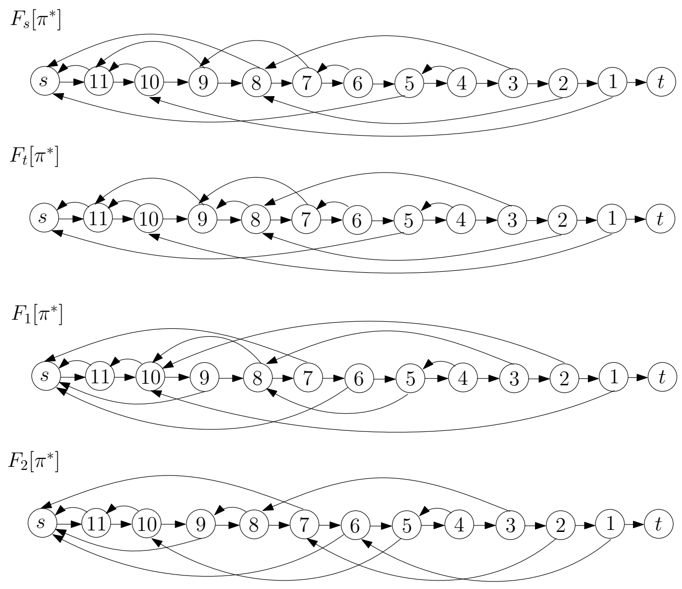

Figure 6.4: The RPGs $F_s[\pi^*]$, $F_2[\pi^*]$, $F_1[\pi^*]$ and $F_2[\pi^*]$ of $\pi^* = (6, 7, 9, 11, 10, 1, 2, 8, 3, 5, 4)$ from $w = 26$.

$2, 8, 3, 5, 4)$, which encodes the watermark number $w = 26$.

6.3 Comparison

It is worth mentioning that all comparative algorithms use the same self-inverting permutation π^* which is encoded by `Encode_W.to.SiP-I`. It implies that their reducible permutation graphs will have similar structure. Specifically, they are node-labeled on $n^* + 2$ nodes, where n^* is the length of permutation π^*. All encoding reducible permutation graphs have a header node s with $outdeg(s) = 1$, a terminal node t with $outdeg(t) = 0$ and body nodes, that have $outdeg(u_{n^*+1-i}) = 2$ (See Section 4.3).

It is worth notice that all algorithms take time and space linear in the size of the flow graph $F[\pi^*]$. So all proposed algorithms encode and decode SiP and RPG in $O(N)$ time and space, where $N = n^* + 2$ and $n^* = 2n + 1$.

Let s_s be the header node in $F_s[\pi^*]$ produced by `Encode_SiP.to.RPG-Bitonic-S`, s_t in $F_t[\pi^*]$ by `Encode_SiP.to.RPG-Bitonic-2`, s_1 in $F_1[\pi^*]$ by `Encode_SiP.to.RPG` and s_2 in $F_2[\pi^*]$ by `Encode_SiP.to.RPG-II`. We have the following result (see, Figure 6.4):

$$indeg(s_t) \leq indeg(s_s) \leq indeg(s_1) = indeg(s_2)$$

k	$F_1[\pi^*]$	$F_2[\pi^*]$	$F_s[\pi^*]$	$F_t[\pi^*]$
1	$\frac{n}{2n+3}$	$\frac{2}{2n+3}$	$\frac{n}{2n+3}$	$\frac{n}{2n+3}$
2	$\frac{2n}{2n+3}$	$\frac{n+1}{2n+3}$	$\frac{n+1}{2n+3}$	$\frac{n+1}{2n+3}$
$[3, n-2]$	$\frac{2n-k}{2n+3}$	$\frac{n+1}{2n+3}$	$\frac{n-k+4}{2n+3}$	$\frac{n+1}{2n+3}$
$n-1$	$\frac{2n-k}{2n+3}$	$\frac{n-1}{2n+3}$	$\frac{n-k+4}{2n+3}$	$\frac{n+1}{2n+3}$
n	$\frac{2n-k}{2n+3}$	$\frac{n+1}{2n+3}$	$\frac{2n-k+2}{2n+3}$	$\frac{2n-k+2}{2n+3}$
$n+1$	$\frac{2}{2n+3}$	$\frac{2}{2n+3}$	$\frac{2n-k+2}{2n+3}$	$\frac{2n-k+2}{2n+3}$
$[n+2, 2n]$	$\frac{2n-k+2}{2n+3}$	$\frac{3}{2n+3}$	$\frac{2n-k+2}{2n+3}$	$\frac{2n-k+2}{2n+3}$
$2n+1$	$\frac{2}{2n+3}$	$\frac{2}{2n+3}$	$\frac{2}{2n+3}$	$\frac{2}{2n+3}$

Table 6.1: All probabilities of any edge (u_k, u_i) existence in RPGs $F_1[\pi^*]$, $F_2[\pi^*]$, $F_s[\pi^*]$ and $F_t[\pi^*]$.

Let $F_1[\pi^*]$ be a reducible permutation flow-graph of size $n^* = 2n+1$ produced by the algorithm Encode_SiP.to.RPG.
If $u_k \in V(F_1[\pi^*])$:

- $k = 1$: the node u_k can be linked with $n+1$ nodes,
- $2 \leq k \leq n$: the node u_k can be linked with $2n-k$ nodes,
- $k = n+1$: the node u_k can be linked with 2 nodes,
- $n+1 < k \leq 2n+1$: the node u_k can be linked with $2n-k$ nodes.

Let $F_2[\pi^*]$ be a reducible permutation flow-graph of size $2n^*+1$ produced by the algorithm Encode_SiP.to.RPG-II.
If $k \in V(F_2[\pi^*])$:

- $k = 1$: the node u_k can be linked with 2 nodes,
- $2 \leq k < n-1$: the node u_k can be linked with $n+1$ nodes,
- $k = n-1$: the node u_k can be linked with $n-1$ nodes,
- $k = n$: the node u_k can be linked with $n+1$ nodes,
- $k = n+1$: the node k can be linked with 2 nodes,
- $n < k < 2n+1$: the node u_k can be linked with 3 nodes,

Algorithms	$min\{emf(F[\pi_1^*], F[\pi_2^*])\}$	probability of weak integers in range n
Encode SIP.to.RPG-Bitonic-S	2	$p_n = \begin{cases} 0, n \leq 3 \\ \frac{2^{n-2}-1}{2^{n-1}-1} p_{n-1} + \frac{1}{2^{n-2}(2^{n-1}-1)}, n > 3 \end{cases}$
Encode SIP.to.RPG-Bitonic-2	2	$p_n = \begin{cases} 0, n \leq 3 \\ \frac{2^{n-2}-1}{2^{n-1}-1} p_{n-1} + \frac{1}{2^{n-2}(2^{n-1}-1)}, n > 3 \end{cases}$
Encode SIP.to.RPG-I	3	$p_n = \begin{cases} 0, n \leq 3 \\ \frac{2^{n-2}-1}{2(2^{n-1}-1)} p_{n-1} + \frac{2^{n-3}+1}{2^{n-2}(2^{n-1}-1)}, n > 3 \end{cases}$
Encode SIP.to.RPG-II	1	$p_n = \frac{1}{2^{n-2}(2^{n-1}-1)}$

Table 6.2: The minimum number of edge modification between $(F[\pi_1^*], F[\pi_2^*])$ and the probabilities of weak watermark integers in range n for algorithms `Encode SiP.to.RPG-Bitonic-S`, `Encode SiP.to.RPG-Bitonic-2`, `Encode SiP.to.RPG-I` and `Encode SiP.to.RPG-II`.

- ○ $k = 2n + 1$: the node u_k can be linked with 2 nodes.

When an attacker has no knowledge about the structure of RPG, then there are $2n + 3$ options for edge modification. We know all properties of RPG but we have to examine if one edge is "real" for our codec system at higher level than the level of SiP construction. Table 6.1 illustrates the probabilities of a successful modification edge from an attacker in any reducible permutation graphs $F_s[\pi^*], F_t[\pi^*], F_1[\pi^*]$ and $F_2[\pi^*]$.

We saw in Section 5.2, the minimum number of edge modifications, that an attack is "almost" successful in our flow graph $F[\pi^*]$ (i.e., Case (ii) in Subsection 5.1), is 2 in $F_s[\pi^*]$ and $F_t[\pi^*]$ with probability in Lemma (5.2). In Algorithm `Encode_SIP.to.RPG`, the minimum number of edge modifications are 3, but the probability of a pair of watermark integers (w_1, w_2) in same range n, with n the length of the binary representation of them, where $F_1[\pi^*]$ has 3 different edges from $F_2[\pi^*]$ is

$$p_n = \begin{cases} 0, n \leq 3 \\ \\ \frac{2^{n-2}-1}{2(2^{n-1}-1)} \cdot p_{n-1} + \frac{2^{n-3}+1}{2^{n-2}(2^{n-1}-1)}, n > 3 \end{cases} \tag{6.1}$$

In Algorithm `Encode_SIP.to.RPG-II`, the minimum number of edge modifications is 1 because there is only one different edge between RPG which be extracted by $w = 2^n - 2$ and RPG which be extracted by $w = 2^n - 1$. Thus, the corresponding probability is

$$p_n = \frac{1}{2^{n-2}(2^{n-1} - 1)} \tag{6.2}$$

Table 6.2 summarizes the minimum number of edge modifications between $F[\pi_1^*]$ and $F[\pi_2^*]$, that is the value of $emf(i,j)$, which are produced by the same algorithms using the SiPs π_1^* and π_2^* in the same range n. Also it depicts all probabilities of weak integers $(w_1, w_2)_n$, which encode $(\pi_1^*, \pi_2^*)_n$, in range n for all algorithms under comparison, that is, `Encode_SiP.to.RPG-Bitonic-S`, `Encode_SiP.to.RPG-Bitonic-T`, `Encode_SiP.to.RPG-I` and `Encode_SiP.to.RPG-II`.

The reducible permutation graphs $F_s[\pi^*]$ and $F_t[\pi^*]$ have $b_n = 2b_{n-1} + 1$ pairs of week integers having $emf(\cdot) = 2$, $F_1[\pi^*]$ has $b_n = 2b_{n-1} + 2^{n-3} - 1$ pairs having $emf(\cdot) = 3$ and $F_2[\pi^*]$ has 1 pair having $emf(\cdot) = 1$, with n the length of the binary representation of integer w, which encodes the self-inverting permutation π^*.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In the last decade, a wide range of software watermarking techniques has been proposed among which the graph-based methods that encode watermark numbers as graphs whose structure resembles that of real program graphs. Recently, Chroni and Nikolopoulos [14, 15] proposed several algorithms for multiple encoding a watermark into a graph-structure: an integer (i.e., a watermark) is encoded first into a self-inverting permutation π^* and then into several reducible permutation graphs using Cographs [14] and Heap-ordered trees [15].

Following up on our recently proposed methods, in this paper, we extended the class of graph-structures by proposing two different reducible permutation flow-graphs $F_s[\pi^*]$ and $F_t[\pi^*]$ incorporating important structural properties which are derived from the bitonic subsequences forming the self-inverting permutation π^*. These new flow-graphs enrich the repository of graphs available for multiple encoding a watermark number and, thus, it increases our ability to select a graph structure more similar to the structure of a given application program P thereby enhancing the resilience of our codec system to attacks.

We compared the two watermarking algorithms with two previous codec watermarking algorithms and presented similarities and differences about structure and complexity. In addition, we computed the probabilities of edge and label modifications in order to consider the resilience of our watermark systems.

7.2 Future Work

An interesting open question is whether the properties of the bitonic subsequences forming the self-inverting permutation π^* can help develop efficient graph structures having more similar structure to that of a given application program P. Can we use some of the bitonic subsequences $b_1^*, b_2^*, \cdots, b_k^*$ of permutation π^* or part of them in order to efficiently encode and decode a self-inverting permutation π^* into a reducible permutation flow-graph $F[\pi^*]$? In addition, we note that in light of our watermarking model it would be very interesting to compare it with other dynamic, or even static, already proposed software watermarking models, too. 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 investigation.

Bibliography

[1] E. Garfinkel, *Web Security, Privacy and Commerce*. O'Reilly, 2001.

[2] L. Chun-Shien, H. Shih-Kun, S. Chwen-Jye, and M. L. Hong-Yuan, "Cocktail watermarking for digital image protection," in *it IEEE Transactions on Multimedia 2:4*, pp. 209–224, 2000.

[3] J. C. Davis, "Intellectual property in cyberspace-what technological/legislative tools are necessary for building a sturdy global information infrastructure?," in *Proc. IEEE Int'l Symposium on Technology and Society*, pp. 66–74, LNCS 1174, 1997.

[4] X. Zhao, Q. Liu, L. Zhou, H. Zheng, and B. Y. Zhao, "Graph watermarks," arXiv:1506.00022, 2015.

[5] C. I. Podilchuk and E. J. Delp, "Digital watermarking: algorithms and applications," in *Signal Processing Magazine IEEE*, pp. 33–46, 2001.

[6] A. S. Panah, R. van Schyndel, T. Sellis, and E. Bertino, "On the properties of non-media digital watermarking: A review of state of the art techniques," in *DOI 10.1109/ACCESS.2016.2570812*, IEEE, 2016.

[7] A. Z. Tirkel, G. A. Rankin, R. M. V. Schyndel, W. J. Ho, N. R. A. Mee, and C. F. Osborne, "Electronic watermark," in *Digital Image Computing, Technology and Applications (DICTA'93)*, pp. 666–673, 1993.

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

[9] H. Tamada, M. Nakamura, A. Monden, and K. Matsumoto, "Design and evaluation of birthmarks for detecting theft of java programs," in *Proc. Int'l Conference on Software Engineering (IASTED SE'04)*, pp. 569–575, 2004.

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

[11] D. Grover, *The protection of Computer Software-Its Technology and Applications.* New York: Cambridge University Press, 1997.

[12] F. Y. Shih, *Digital Watermarking and Steganography: Fundamentals and Techniques.* Boca Raton, FL: CRC Press, 2007.

[13] M. Chroni and S. D. Nikolopoulos, "Encoding watermark integers as self-inverting permutations," in *Proc. Int'l Conference on Computer Systems and Technologies (CompSysTech'10)*, pp. 125–130, ACM ICPS 471, 2010.

[14] M. Chroni and S. D. Nikolopoulos, "Encoding watermark numbers as cographs using self-inverting permutations," in *Proc. Int'l Conference on Computer Systems and Technologies (CompSysTech'11)*, pp. 142–148, ACM ICPS 578, 2011.

[15] M. Chroni and S. D. Nikolopoulos, "Encoding numbers into reducible permutation graphs using heap-ordered trees," in *Proc. 19th Panhellenic Conference on Informatics*, ACM, 2015.

[16] M. Chroni and S. D. Nikolopoulos, "An efficient graph codec system for software watermarking," in *Proc. Int'l Conference on Computers, Software, and Applications (COMPSAC'12); Workshop STPSA'12*, pp. 595–600, IEEE Proceedings, 2012.

[17] G. Myles and C. Collberg, "Software watermarking via opaque predicates: Implementation, analysis and attacts," *Electronic Commerce Research*, vol. 6, pp. 155–171, 2006.

[18] C. Collberg and C. Thomborson, "Software watermarking: models and dynamic embeddings," in *Proc. 26th ACM SIGPLAN-SIGACT on Principles of Programming Languages (POPL'99)*, pp. 311–324, 1999.

[19] J. Palsberg, S. Krishnaswamy, M. Kwon, M. D, Q. Shao, and Y. Zhang, "Experience with software watermarking," in *Proc. 16th Annual Computer Security Applications Conference (ACSAC'00)*, pp. 308–316, 2000.

[20] D. Eppstein, M. T. Goodrich, J. Lam, N. Mamano, M. Mitzenmacher, and M. Torres, "Models and algorithms for graph watermarking," in *International Conference on Information Security*, Springer International Publishing, 2016.

[21] R. L. Davidson and N. Myhrvold, "Method and system for generating and auditinga signature for a computer program," in *US Patent 5.559.884*, Microsoft Corporation, 1996.

[22] R. Ghiya and L. J. Hendren, "Is it a tree, a dag, or a cyclic graph? a shape analysis for heapdirected pointers in c," in *Proc. 23rd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL'96)*, pp. 1–15, LNCS 1174, 1996.

[23] S. A. Moskowitz and M. Cooperman, "Method for stegacipher protection of computer code," US Patent 5.745.569, 1996.

[24] C. Collberg, C. Thomborson, and D. Low, "On the limits of software watermarking," in *Department of Computer Science, The University of Auckland*, Technical Report No 164, 1998.

[25] L. Zhang, Y. Yang, X. Niu, and S. Niu, "A survey on software watermarking," *Journal of Software*, vol. 14, pp. 268–277, 2003.

[26] W. Zhu, C. Thomborson, and F. Y. Wang, "A survey of software watermarking," in *Proc. IEEE Int'l Conference on Intelligence and Security Informatics (ISI'05)*, pp. 454–458, LNCS 3495, 2005.

[27] C. Collberg, S. Kobourov, E. Carter, and C. Thomborson, "Error-correcting graphs for software watermarking," in *Proc. of the 29th Workshop on Graph-Theoretic Concepts in Computer Science (WG'03)*, pp. 156–167, LNCS 2880, 2003.

[28] C. Collberg, A. Huntwork, E. Carter, G. Townsend, and M. Stepp, "More on graph theoretic software watermarks: Implementation, analysis, and attacks," in *Information and Software Technology 51*, pp. 56–67, 2009.

[29] R. Venkatesan, V. Vazirani, and S. Sinha, "A graph theoretic approach to software watermarking," in *Proc. of the 4th Int'l Workshop on Information Hiding (IH'01)*, pp. 157–168, LNCS 2137, 2001.

[30] I. Chionis, M. Chroni, and S. D. Nikolopoulos, "A dynamic watermarking model for embedding reducible permutation graphs into software.," in *Proc. 17th Panhellenic Conference on Informatics*, pp. 144–151, ACM, 2013.

[31] T. Xie and D. Notkin, "An empirical study of java dynamic call graph extractors," in *University of Washington CSE*, Technical Report 02-12-03, 2002.

[32] J. Graham, P. Kessler, and M. Mckusick, "Gprof: A call graph execution profiler," in *ACM SIGPLAN Notices 17(6)*, pp. 120–126, 1982.

[33] L. M. Bento, D. R. Boccardo, R. C. Machado, V. G. P. de Sa, and J. L. Szwarcfiter, "On the resilience of canonical reducible permutation graphs," *Discrete Applied Mathematics*, 2016.

[34] M. Hecht and J. Ullman, "Flow graph reducibility," *SIAM J. Computing*, vol. 1, pp. 188–202, 1972.

[35] M. Hecht and J. Ullman, "Characterizations of reducible flow graphs," *Journal of the ACM*, vol. 21, pp. 367–375, 1974.

Short Biography

Anna Mpanti received her B.Sc. degree in Mathematics (2015) from the Department of Mathematics of the University of Ioannina, Greece. Her research interests are focused on the design and analysis of algorithms, algorithms engineering, approximation algorithms, algorithmic graph theory, and information hiding. Anna has been an assistant in the laboratories of the undergraduate course on Introduction to Programming and has also been an assistant in the laboratories of the undergraduate course on Design and Analysis of Algorithms in the Department of Computer Science & Engineering, University of Ioannina.