Tamperproofing a Software Watermark by Encoding Constants

by

Yong He

A Thesis Submitted in Partial Fulfilment of the
Requirement for the Degree of

MASTER OF SCIENCE
in
Computer Science

University of Auckland

Abstract

Software Watermarking is widely used for software ownership authentication, but it is susceptible to various de-watermarking attacks such as obfuscation. Dynamic Graph Watermarking is a relatively new technology for software watermarking, and is believed the most likely to withstand attacks, which are trying to distort watermark structure.

In this thesis, we present a new technology for protecting Dynamic Graph Watermarks. This technology encodes some of the constants, which are found in a software program, into a tree structure that is similar to the watermark, and generates decoding functions to retrieve the value of the constants at program execution time. If the constant tree is modified, the value of some constants will be affected, destroying program correctness. An attacker cannot reliably distinguish the watermark tree from the constant tree, so they must preserve the watermark tree or risk introducing bugs into the program.

Constant Encoding technology can be included in Dynamic Graph Watermarking systems as a plug-in module to improve the Dynamic Graph Watermark protection. In this thesis, we present a prototyping program for Constant Encoding technology, which we call the JSafeMark encoder. Besides addressing the issues about Constant Encoding technology, we also discuss the design and implementation of our JSafeMark encoder, and give a practical example to show how this technology can protect Dynamic Graph Watermarking.

Acknowledgement

I would like to thank Professor Clark Thomborson, my supervisor, for his guidance and enthusiasm. Throughout my thesis year, he provided encouragement, sound advice, and many good ideas.

I want to express my gratitude to Dr. Christian Collberg for his invaluable advice as well as a lot of technical support.

I wish to thank Jas Nagra for numerous stimulating discussions.

I would also like to thank Dr. Ben Andrew for his precious time to give me a proofreading.

I am grateful to all my friends for their time to help me on various problems.

Lastly, and most importantly, I wish to thank my family for their understanding, patient and encouragement, without whose help and support, I would have had a far more difficult time to finish this thesis.

Table of Contents

Abstract ..... **i**

Acknowledgement ..... **iii**

Table of Contents ..... **v**

Table of Figures ..... **ix**

Chapter 1 *Introduction* **1**

1.1 Software Piracy 1

1.2 Ownership Authentication 1

1.3 Threat to Java Software Protection 2

1.4 Target of this Thesis 3

1.5 Related Work 6

1.6 Organization of this Thesis 7

Chapter 2 *Survey of Watermarking Techniques* **9**

2.1 Introduction 9

2.2 Watermark Technique Overview 10

2.3 Dynamic Graph Watermark (DGW) 11

2.3.1 Dynamic Graph Watermarking Process 11

2.3.2 DWG Embedding Enumeration Methods 13

2.3.3 Current Implementations of DGW 17

2.4 Attacks on Software Watermark 21

2.4.1 Overview of Attacks on Software Watermark 22

2.4.2 Attacks on DGW watermark 23

2.5 Watermark Protection 26

2.5.1 Overview of Protection Techniques 27

2.5.2 Protections for DGW watermark 28

2.5.3 Integration of Protection Methods 30

2.6 Discussion 30

2.7 Summary 30

Chapter 3 *Constant Encoding* **33**

3.1 Introduction 33

3.2 Technology Overview 36

3.3 System Structure for JSafeMark Plug-in 38

3.4 Constant Encoding Process 40

3.4.1 Work Flow 40

3.4.2 Constant Analysis by the DGW System 44

3.4.3 Building Constant Tree 44

3.4.4 Building Constant Graph (CG) 45

3.4.5 Searching for Constant Substructures in Dynamic structures 57

3.4.6 Generating the Decoding Method 60

3.4.7 Process of Modifying Source Code by the DGW System 68

3.5 Discussion 68

3.5.1 Possibility of finding constants in Programs 68

3.5.2 Possibility of finding integers in a PPCT 69

3.6 Summary 74

Chapter 4 *Overview of Our Prototyping* **75**

4.1 Introduction 75

4.2 System Structure 75

4.3 Information Exchanging 78

4.4 Summary 82

Chapter 5 *Developing the JSafeMark Tester and Codec Library* **83**

5.1 Introduction 83

5.2 Building JSafeMark Tester 83

5.2.1 Overview of the JSafeMark Tester 84

5.2.2 JSafeMark Tester Structure 84

5.2.3 Using JSafeMark Tester 85

5.3 Building the PPCT Codec Library 88

5.4 Summary 90

Chapter 6 *Developing a JSafeMark Encoder* **91**

6.1 Introduction 91

6.2 Analysis 92

6.2.1 Functional requirements: 92

6.2.2 Non-functional Requirements 92
6.2.3 Use Case View: Requirement Description 92
6.2.4 Sequence Diagrams: Conceptual Dynamic Behavior 94
6.2.5 Class Diagrams: Conceptual Static Structure 95
6.3 Design 96
6.3.1 Static Design: Design Level Class Diagrams 96
6.3.2 Dynamic Design: Design Level Sequence Diagrams 98
6.4 Implementation 100
6.5 Summary 101

Chapter 7 *JSafeMark in Practice* 103
7.1 Introduction 103
7.2 Example of Using JSafeMark encoder 103
7.2.1 Import Constants (WF1) 104
7.2.2 Building PPCT Constant Tree (WF2) 104
7.2.3 Convert Constant to Integer (WF3) 105
7.2.4 Generate Source Code to Reverse WF3 (WF4) 105
7.2.5 Convert Integer to Graph Structure (WF5) 106
7.2.6 Generate Source code to Reverse WF5 (WF6) 107
7.2.7 Searching (WF7) 107
7.2.8 Expand the Constant Tree (WF8) 107
7.2.9 Generate Referencing information (WF9) 109
7.2.10 Generate and Decoding Source Code (WF10, 11) 111
7.3 Discussion 112
7.4 Summary 112

Chapter 8 *Conclusion* 113
8.1 Pros and Cons of JSafeMark 113
8.2 Future Work 114
8.2.1 Protect the Return Value of the Decoding Function 114
8.2.2 Changing Constant Tree at Runtime 115
8.2.3 Encode Constant into Watermark Structures 116
8.2.4 Using Code that builds Watermarks to build a Constant Tree 117
8.2.5 Using a Set of Constant Trees 117
8.2.6 Using Constant Subtrees as Masks 118
8.3 Summary 118

Appendix A. ***CLOC Encoding Implementation*** **.....119**

Appendix B. ***Experimental Data of Integer Frequency in PPCT Structures*** **.....129**

Appendix C. ***The output of the generated Source Code in testing*** **.....133**

Bibliography **.....139**

Table of Figures

Figure 1-1 A program with no watermark is listed on the left. This program builds a single dynamic data structure to hold the value of its variable a, which is shown on the right side. 5

Figure 1-2 The program of Figure 1-1, modified so that a watermark is embedded as a dynamic graph structure g (shown on the right). Static analysis is enough to discover that the dynamic watermark structure g doesn't affect the program output, and therefore this watermark can be removed. 5

Figure 1-3 The program of Figure 1-2, modified to include a dependency between the assignment of 2 to variable a and the value of g. The assignment statement "a = 2" in Figure 1-2 is replaced by two statements "s = f(g)" and "a = d(s)". The value 2 of variable a is obtained by evaluating function d(s), which takes a part of the structure g as its parameter. Statement "s = f(g)" is used to locate s inside structure g. 5

Figure 2-1 Dynamic watermark embedding process 12

Figure 2-2 A Planted Plane Cubic Tree example 14

Figure 2-3 A PPCT structure, with two outgoing pointers for all nodes 14

Figure 2-4 Enumeration of PPCTs with 4 Leaves 15

Figure 2-5 Radix-*k* enumeration example with k = 4 16

Figure 2-6 Oriented parent-pointer tree enumeration with four nodes 16

Figure 2-7 A circular linked list for permutation encoding number 29 17

Figure 2-8 Some PPCT structures of index = 0, with different numbers of leaves 18

Figure 2-9 Attacks on software watermark 24

Figure 2-10 Tamperproofing a DGW against a subtractive attack 25

Figure 2-11 Attacks on DGW watermark 26

Figure 2-12 Watermark protection techniques: obfuscation and tamperproofing. 27

Figure 2-13 Protection from node splitting. 29

Figure 3-1Comparison of subtree and substructures. 35

Figure 3-2 Constant encoding process 37

Figure 3-3 Proposed watermark embedding and constant encoding process flow 39

Figure 3-4 System structure for the DGW system and the JSafeMark encoder 40

Figure 3-5 Control flow of constant encoding process 43

Figure 3-6 Relationships between codecs used in watermarking and in constant encoding. ..46
Figure 3-7 Java types classification 47
Figure 3-8 CLOC index of PPCTs with 1 to 4 leaves..... 50
Figure 3-9 Four PPCTs that represent the integer 0 52
Figure 3-10 Structure for Bit-Oriented Conversion..... 52
Figure 3-11 Integer to graph encoding using BOC..... 55
Figure 3-12 The CUIC decoding function CUIC(T) for all PPCTs with 1 to 4 leaves 57
Figure 3-13 Referencing a substructure at runtime..... 62
Figure 3-14 Finding the root of substructure by path and depth..... 63
Figure 3-15 Masking technology illustration..... 63
Figure 3-16 Estimated Probability (Expressed as a percentage) that an integer *i* appears in a 50 node (25 leaf) tree. 72
Figure 3-17 Estimated Probability (Expressed as a percentage) that an integer *i* appears in a 100 node (50 leaf) tree. 72
Figure 3-18 Estimated Probability (Expressed as a percentage) that an integer *i* appears in a 200 node (100 leaf) tree. 73
Figure 3-19 Estimated Probability (Expressed as a percentage) that an integer *i* appears in a 300 node (150 leaf) tree. 73
Figure 3-20 Estimated Probability (Expressed as a percentage) that an integer *i* appears in a 500 node (250 leaf) tree. 73
Figure 4-1 Overview of an expected system structure for the JSafeMark encoder. 77
Figure 4-2 Implemented system structure in prototyping. 78
Figure 4-3 Sample tree structure for structure passing. 81
Figure 4-4 Data Type for information exchanging 81
Figure 5-1 The JSafeMark tester structure..... 85
Figure 5-2 The JSafeMark tester running in “manual” and “normal display” mode, showing a PPCT structure for number 0 with 5 leaves. 87
Figure 5-3 The JSafemark tester in “auto increase” and "compact display" mode, showing a tree structure for number 1500000 with 15 leaves..... 87
Figure 5-4 Internal structure for a codec..... 88
Figure 5-5 PPCT codec class diagram 89
Figure 6-1 Use Case Diagram..... 93
Figure 6-2 Conceptual sequence diagram for the encoding function. 95
Figure 6-3 Conceptual class diagram for JSafeMark encoder 96
Figure 6-4 GraphNode class, DecodingMethodGenerator class, Error Classes and searchingMethod package constantConvertingMethod package example classes 97
Figure 6-5 Design level class diagram..... 98

Figure 6-6 Design level sequence diagram for constant encoding 99
Figure 7-1 The random constant tree of index 8 with 5 leaves.....105
Figure 7-2 PPCT structures for first integer = 1 and second integer = 4 106
Figure 7-3 Constant tree and the substructure for integers 1 and 4 109
Figure 7-4 Referencing to the substructures for integers 1 and 4 110
Figure 7-5 intersecting mask and CT to find the boundary information for integers 1 and 4110
Figure 8-1 Changing constant tree at runtime. Constant tree is changed when executing a sequence of statements. Decoding function works well. 116
Figure 8-2 Changing the constant tree can cause errors. 116

Chapter 1

Introduction

1.1 Software Piracy

Due to the dramatically increased usage of the Internet, software piracy has become more and more serious. The ease on downloading from the Net seems to encourage people to use software without authorization. According to Curtis [Curtis 1994], software piracy can be defined as the illegal distribution, duplication, using and selling of a software product, which is protected by copyright laws or covered by an applicable license agreement. The dollar losses due to software piracy were estimated at $11.75 billion in year 2000, and the software piracy rate rose from 36% in 1999 to 37% in 2000 [Business Software Alliance 2001]. The software industry continues to be challenged to find ways of protecting software products and worldwide economy.

1.2 Ownership Authentication

People are trying hard to find methods to stop software piracy, and one of the important ways is software ownership authentication through watermark embedding. Watermark embedding is any technique that embeds a secret message into a cover message. In the case of software watermarking for authentication, the secret message is the digital rights

management information (such as a license number, and license restrictions). The cover message is the license-controlled software. The secret message in a software-watermark will be retrieved when this information is required for authentication purposes. In early years, watermarking technology was used to imprint authentication information on stationery [Webopedia DW]. Recent work in computer watermarking technology has developed techniques for embedding a watermark into various media, such as images [Chuan-Fu and Wen-Shyong 2000] [Kahn 1996] [Ping Wah and Memon 2001], audio [Kirovski and Malvar 2001] or video files [Hartung and Girod 1998]. In recent years, some articles [Collberg and Thomborson 1999] [Collberg, Thomborson, et al. 1997] [Palsberg, Krishnaswamy, et al. 2000] and patents [Holmes 1994] [Moskowitz and Cooperman 1996] [Samson 1994] have been published, showing how watermarks, such as copyright notices, can be embedded into software programs.

1.3 Threat to Java Software Protection

In order to avoid being punished, counterfeiters sometimes go to great lengths to make their fake software look legal. One of the most often used technologies for understanding and modifying someone else's software is reverse engineering [Chikofsky and Cross, 1990].

Reverse engineering originated in hardware technology. It is now widely used in software products. It is a process that takes apart an object to analyze its internal structure and its inner relationships, in order to duplicate the object or to improve it [Chikofsky and Cross 1990]. Reverse engineering is not always a bad action. It is widely used to improve productivity and interoperability by manufacturers. Sometimes it is necessary for the author of software to reverse-engineer it, for example when the original source code has been lost, or was written in a language for which software maintenance tools no longer exist. Crackers typically use reverse engineering techniques to convert software from a bytecode or machine code level back into the source code level [Byrne 1992] to analyze how it works, and then modify it to suit their own requirements. Decompilers and disassemblers are often used for reverse engineering, together with some additional tools such as program slicers [Tip 1995] and debuggers.

The Java Virtual Machine is designed to be independent of the Java Development Kit. Java source code is first compiled into byte code format, and stored in a class file. The class file does not contain information for operation on a particular platform or hardware. Instead,

a Virtual Machine, written for each platform, recognizes the bytecode format and then translates the instructions into platform-dependent code [Lindholm and Yellin 1997]. To allow platform independence and to ensure platform security, a lot of information from the source code must remain in the bytecode format, such as type information for object linking. The highly informative class file format makes reverse-engineering and static analysis of a compiled Java program much easier than in other executable formats such as native machine code. Therefore Java bytecode is prone to decompilation [Proebsting and Watterson 1997] and is considered to be insecure for information hiding, unless an "obfuscator" is used to make it difficult for the reverse engineer to understand the program's structure [Collberg, Thomborson, et al. 1998a].

Because of its platform independent features, the Java language is widely used for commercial product development. This gives software piracy a great chance to grow, and has also made hiding watermarks inside source codes more difficult. Crackers can easily reverse-engineer most Java bytecodes, understand the work flow and the function of each part, then insert or remove some important information such as a license validity check. They may even be able to recognize code whose only function is to watermark the code. If they can reliably recognize this code, they are usually able to disable it without affecting the correctness of the code.

1.4 Target of this Thesis

One instance of the newly developed software watermarking technologies is dynamic graph watermarking (DGW) [Collberg and Thomborson 1999]. This technology uses a dynamically created graph structure to represent a watermark message at the software execution time, instead of directly embedding a watermark message into the software program. Our aim in this thesis is to demonstrate a new way of protecting dynamic graph watermarks, called Constant Encoding. This technology introduces code that closely resembles watermark code, but which is necessary for program correctness.

The dynamic graph watermarking techniques will be discussed in Chapter 2, and detailed constant encoding techniques will be discussed in Chapter 3. Now we just give a brief overview of what this technology is like.

Please have a look at Figure 1-1, Figure 1-2 and Figure 1-3. The original program in Figure 1-1 is not watermarked. In Figure 1-2, a dynamic graph watermark g has been

embedded into the program of Figure 1-1. The dynamic watermark *g* is a data structure in the shape of a Planted Plane Cubic Tree (PPCT), where *g* is the seventh PPCT in some enumeration of PPCTs. Therefore, we say that this program carries the watermark "7".

Now we notice that the execution and output of the original program does not depend on the watermark code. Static analysis is enough to discover this and remove the watermark code.

In order to protect the dynamic graph watermark, in Figure 1-3, we create another dynamic structure *ct* that has the same data structure as the watermark structure *g*. Because of the features of alias analysis [Burke, Carini, et al. 1999] in a dynamic tree structure, an attacker cannot reliably recognize which is the watermark structure. After that, we create a dependency from the assignment statement "*a = 2*" in the watermarked program to the dynamic data structure *ct*. We do this by replacing the constant 2 with a retrieving function *d(s*). The function *d(s)* is evaluated only at runtime. It takes an argument *s*, where *s* is a part of the dynamic data structure *ct*, and will always evaluate to 2. Thus, if an adversary tries to find the watermark structure and remove it, the adversary is not quite able to remove the watermark structure without modifying the structure *ct*, because *ct* is very similar to the watermark tree *g*. However, if *ct* is modified, the structure *s* in *ct* will be no longer be referenced correctly at runtime. Thus, function *d(s)* cannot be evaluated correctly any more and the constant value of integer 2 cannot be retrieved. In order to remove the watermark structure, adversary must analyze the program's dynamic behavior, to discover that *d(s)* will always evaluate to 2, or accurately determine the boundaries of the watermark structure. However, dynamic analysis is known to be computationally intractable in general.

From the above analysis, we see that we have created a false dependency from the watermarked program to the embedded watermark structure. Figure 1-1, Figure 1-2 and Figure 1-3 give a rough idea of how our technology works. In practice, the method we used for encoding and decoding constants will be much complex than the one shown above.

Our target, in this thesis, is to build the second tree *ct* and design a retrieval function *d(s)* that might someday be proved computationally intractable. Recent theoretical work [Wang 2000] indicates that such proofs may be possible, but far too complex and subtle to be attempted in a Master's thesis.

```
public class A{
    int a;
    public  void print(){
        a = 2;
        System.out.println(a);
    }
    public static void main(String[] args){
        new A().print();
    }
}
```

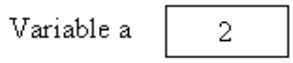


Figure 1-1 A program with no watermark is listed on the left. This program builds a single dynamic data structure to hold the value of its variable a, which is shown on the right side.

```
public class A{
    int a;
    DGW g;      //Dynamic Graph Watermark
    public  void print(){
        g = build_DGW_watermark( 7 );
        a = 2;
        System.out.println(a);
    }
    public static void main(String[] args){
        new A().print();
    }
}
```

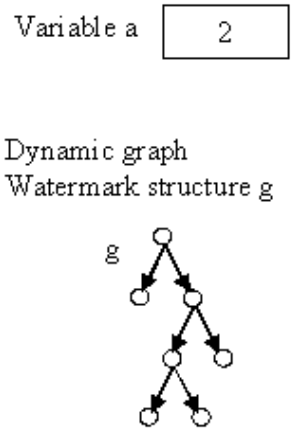


Figure 1-2 The program of Figure 1-1, modified so that a watermark is embedded as a dynamic graph structure g (shown on the right). Static analysis is enough to discover that the dynamic watermark structure g doesn't affect the program output, and therefore this watermark can be removed.

```
public class A{
    int a;
    DGW g;      //dynamic graph watermark
    DGW ct;     //constant tree
    DGW s;      //substructure of ct
    public  void print(){
        ct = build_DGW_for_Constant();
        g = build_DGW_watermark( 7 );
        s = f(ct);      //finding substructure s in ct
        a = d(s);       //retrieved value 2 from s
        System.out.println(a);
    }
    public static void main(String[] args){
        new A().print();
    }
}
```

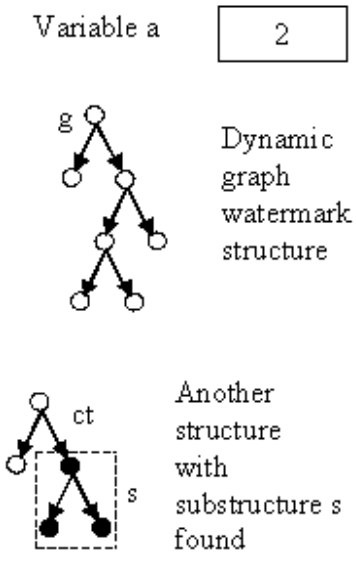


Figure 1-3 The program of Figure 1-2, modified to include a dependency between the assignment of 2 to variable a and the value of g. The assignment statement "a = 2" in Figure 1-2 is replaced by two statements "s = f(g)" and "a = d(s)". The value 2 of variable a is obtained by evaluating function d(s), which takes a part of the structure g as its parameter. Statement "s = f(g)" is used to locate s inside structure g.

Another concept, fingerprinting, is closely related to watermarking. The fingerprinting technique is embedding different watermarks into different copies of the same program. However, this technique is beyond our interest in this thesis, and we will focus on watermark protection only.

1.5 Related Work

Dynamic graph watermarking is a relatively new technique, first published in 1999. So far, the technology of protecting dynamic graph watermark in software has not received much academic attention.

Collberg and Thomborson suggest splitting a watermark structure into pieces, and building the watermark pieces separately according to a special input sequence. These pieces of watermark structures will be merged together after all the input sequence has been accepted [Collberg and Thomborson 1999]. Their Sandmark System [Collberg, Townsend, et al. 2001], which is an implementation of their DGW technology, adopts this technique. This technique brings difficulties to attackers on finding the watermark structure and figuring out what the special input sequence is.

Based on earlier work by Collberg and Thomborson in [Collberg and Thomborson 1999], Palsberg performs experiments on his implementation of a dynamic watermark [Palsberg, Krishnaswamy, et al. 2000]. He suggests adding a dependency to a graph g' of the same type as the watermark graph g, by adding an if-statement with a predicate depending on *g'* before a number of statements in the original program. Presumably, the attacker will not be able to remove g without also removing g'. The if-statements test whether two nodes in the graph structure are distinct. This predicate is a typical example of the "opaque predicates" proposed by Collberg et al [Collberg, Thomborson, et al. 1998b]. Such if statements will always evaluate to true and will not affect the execution of the original program, but will create dependencies from the original program to the data types used to build the watermark graph. We will discuss Palsberg's technique further in Section 2.5.2.

Apart from the above algorithms, there is no protection technology discussed specifically for dynamic graph structure. However, technologies such as tamperproofing and obfuscation have been suggested as ways to protect dynamic graph watermark [Collberg and Thomborson 1999] [Palsberg, Krishnaswamy, et al. 2000].

Obfuscation is the process of modifying a source code or bytecode, to hide the program

information from being attacked while preserving the program's semantics [Collberg, Thomborson, et al. 1997] [Collberg, Thomborson, et al. 1998b] [Low]. Through obfuscation, the low level program either can not be decompiled due to its high level code, or else the decompiled high level code is hard to understand, and might not be recompilable.

Tamperproofing is a technique that can detect if a program has been altered, and if so, then the program will fail to function properly [Collberg and Thomborson 2000].

Another concept called randomization is mentioned by Palsberg. His idea is to weave the program code and watermark code in a random way, to protect against malicious attacks [Palsberg, Krishnaswamy, et al. 2000]. It will bring difficulties to attackers who try to compare two different copies of the same program, maybe with different watermarks (called *fingerprint*). Randomization will not be our interest in this thesis because it will not help too much on watermark technology where watermarks in different copies of the same program are the same, also fingerprinting is out of our focus in this thesis.

1.6 Organization of this Thesis

This thesis is divided into eight chapters. In the first chapter, the Introduction, we have outlined some background information on watermarking, including software piracy, the need for software protection, current research on watermarking and related technologies. We also give a brief overview of the ideas we will talk about in this thesis.

Chapter 2 overviews the watermarking techniques currently used in software protection, compares the main differences among them, examines the attacks on watermarking and surveys defenses against these attacks. In each topic in this chapter, we emphasize dynamic graph watermarking technology, show the outstanding features of it, and also discuss the pros and cons as well as the attacks and protection of it.

Chapter 3 discusses constant encoding technology in detail; it explains the relation between DGW watermarking technology and constant encoding technology. At the end of this chapter, some practical work shows the applicability of our constant encoding algorithm.

Chapter 4 provides a high-level description of our JSafeMark encoder prototyping, and discusses the system structure as well as how the information is exchanged in the system.

Chapter 5 discusses the JSafeMark tester, which we designed to tester our encoder. Meanwhile, the codec library is also discussed in this chapter.

Chapter 6 presents details of how our JSafeMark encoder is designed in the style of Software Engineering. Rational Rose is used throughout the implementation.

Chapter 7 evaluates our partially developed JSafeMark encoder with concrete examples.

Chapter 8 discusses the overall pros and cons of our JSafeMark algorithm and system. It also suggests some improvements that can be made in the future.

Chapter 2

Survey of Watermarking Techniques

2.1 Introduction

Watermarking embeds a secret message into a cover message. In this thesis, we are only interested in the type where the cover message is a software program. This is called software watermarking [Collberg and Thomborson 2000]. Dynamic Graph Watermarking (DGW), which embeds a watermark number into a graph topology that is dynamically built at program runtime, is one of the software watermarking technologies published by Collberg and Thomborson [Collberg and Thomborson 1999]. It inherits the benefits of earlier watermarking techniques, also, it has its own advantages obtained from the graph structure.

In this chapter, we will introduce some background knowledge about software watermarking including attacks and defenses. Dynamic Graph watermarking will be the focus of our discussion. We will describe two watermarking systems, which are built on Dynamic Graph watermark technology. The concept of the Planted Plane Cubic Tree will be emphasized during the introduction, since we will be using this graph structure extensively.

2.2 Watermark Technique Overview

Watermarks can be embedded into software programs in a number of ways, and they can be classified into two categories: static watermarks and dynamic watermarks [Collberg and Thomborson 1999].

A static watermark is stored inside program code in a certain format, and it does not change during the program execution. According to the representation of watermark information, there are two types of static watermarks: data watermarks and code watermarks. A data watermark stores watermark information as program data, and can be stored anywhere inside a program, such as in comments or in variables. A code watermark is represented by choosing a particular sequence of instructions, in cases (and these are common) where more than one sequence of instructions has an equivalent effect. A static code watermark may also be stored in "dead code" (which is never executed); any sequence of instructions may be used with equivalent effect in a dead-code area. For example, in a Java program, a particular order of cases in a switch statement can be used to represent a watermark number. Further discussion of static watermarking issues can be found elsewhere [Davidson and Myhrvold 1996] [Moskowitz and Cooperman 1996] [Venkatesan, Vazirani, et al. 2001].

A dynamic watermark is built during program execution, perhaps only after a particular sequence of input. It might be retrieved by analyzing the data structures built when watermarked program is running. In other cases, tracing the program execution may be required. There are three kinds of dynamic watermarks: *Easter Eggs*, *Execution Trace Watermarks*, and *Dynamic Data Structure Watermarks* [Collberg and Thomborson 1999].

Easter Egg Watermarks. These dynamic watermarks are revealed at the user interface, by a secret sequence of input. These secret messages may be copyright information or an unexpected text or image. There are many resources about Easter Egg Watermarks and examples can be found on the Web [EEGGS]. For example, here is an undocumented Easter Egg distributed with Microsoft Windows 2000: start a new FreeCell game in Windows 2000, press Ctl+Shift+F10, click Abort, double click on the cards - you win!

Execution Trace Watermarks. When this type of watermarked program is run, the watermark information will be presented in the instruction or data-reference trace of program execution sequence. A special sequence of inputs may be required. This watermark can be extracted by analyzing the address trace and/or the sequence of operations of a watermarked program.

Data structure watermarks. When this type of watermarked program is executed with a special input sequence, watermark information will be left within the state of the program such as the heap or the stack. This watermark can be retrieved by analyzing the runtime memory state of the program. The Dynamic Graph Watermark (DGW) described by Collberg and Thomborson in [Collberg and Thomborson 1999] belongs to this category. Our new constant encoding idea is an improvement on DGW technology.

2.3 Dynamic Graph Watermark (DGW)

Collberg and Thomborson discussed watermarking technologies and their protections systematically in their paper [Collberg and Thomborson 1999]. They pointed out that current watermarking techniques could be attacked successfully by typical obfuscating transformations. Starting from this point, they presented a new watermarking technique called Dynamic Graph Watermarking. DGW is a watermark that is represented in a graph topology, which is created only at the runtime of a program, triggered by a special sequence of input. Their DGW technique is much less susceptible than static watermarks from de-watermarking attacks such as code optimization and code obfuscation.

2.3.1 Dynamic Graph Watermarking Process

In the following discussion of watermarking techniques, the term “Candidate Program” will be used to refer to a software program that needs to be watermarked. According to [Collberg and Thomborson 1999], the Dynamic Graph Watermarking process can be described as:

(1) Choosing a watermark number.

(2) Representing the watermark number with a graph structure.

(3) Building a watermark code for generating the graph structure at runtime.

(4) Embedding this code into a software program, so that the watermark is built only when a special input is presented to the resulting watermarked program.

Steps two to four above can be illustrated as in Figure 2-1. The watermarking process shown in (1) of Figure 2-1 shows a watermark number can be turned into a graph structure G according to a certain enumeration method (step (2) in the above list). The process (2) in Figure 2-1 builds a source code for generating the watermark structure G (step (3) in the

above list). When this source code is executed at runtime, the watermark graph G will be generated. The process (3) in Figure 2-1 inserts the source code into the candidate program, and (4) of Figure 2-1, ensures that the watermark will only be built when a special input is presented to the program (step (4) in the above list). To retrieve the watermark, one needs to run the watermarked program with the special input sequence, and then examine the objects in the heap dump.

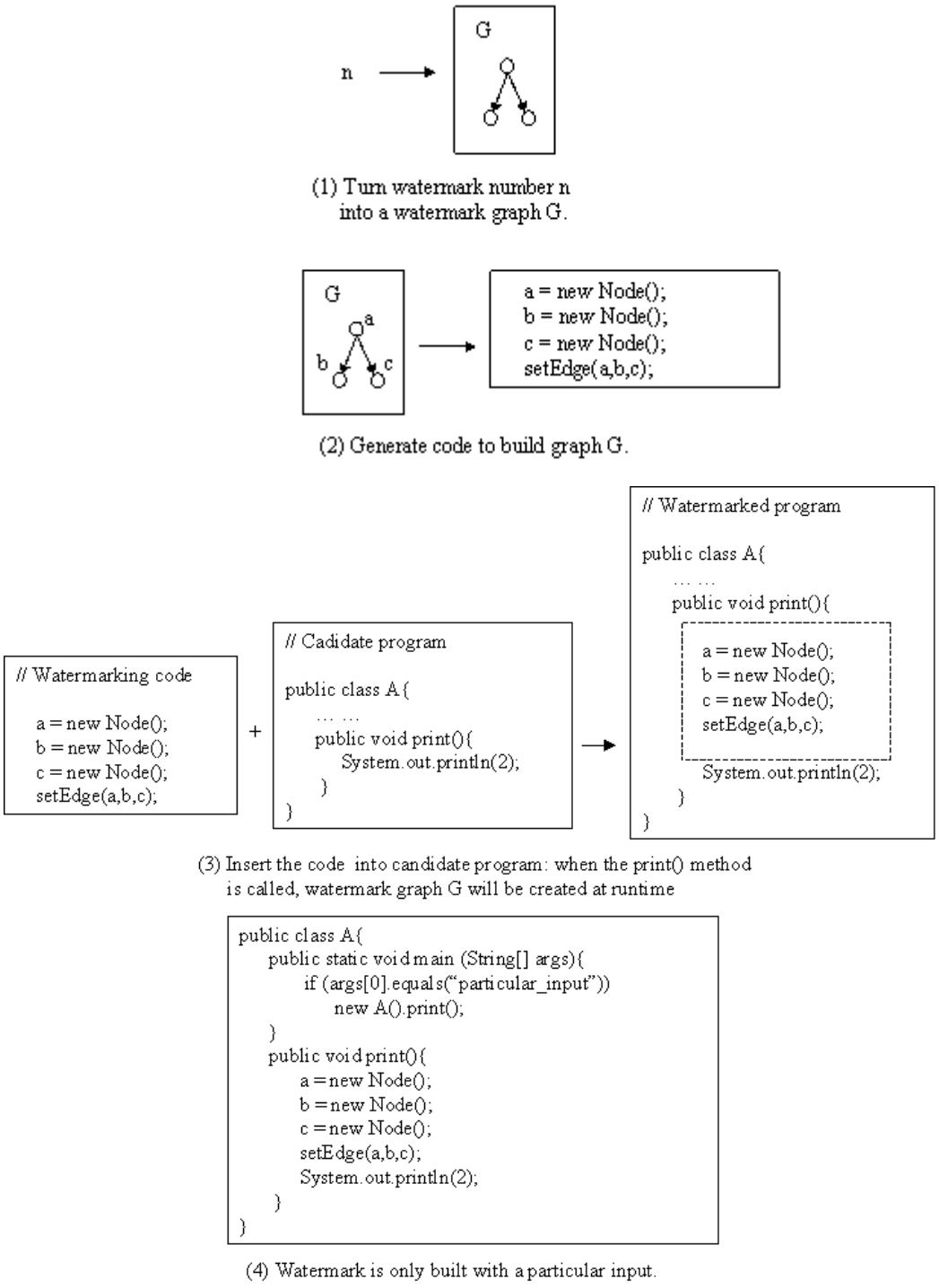


Figure 2-1 Dynamic watermark embedding process

In paper [Collberg and Thomborson 1999], Collberg and Thomborson also mentioned two improvements on the basic DGW method. One is to split the DGW structure into pieces that are built separately. After a particular sequence of input is accepted, all partitions of DGW structures have been built and can be put together. We have mentioned this in Section 1.5, and we will discuss the implementation in detail in Section 2.3.3. The other improvement is to integrate tamperproofing technologies into the DGW process. For simplicity, we did not show this idea in Figure 2-1. Details of watermark protection techniques will be discussed in Section 2.5.

DGW watermarking represents the watermark with a graph structure constructed at runtime. Its biggest advantage over static watermarking is that a dynamic watermark graph structure contains many pointers, and it is hard to analyze pointers at runtime. Also, because a DGW watermark is constructed dynamically, runtime information must be gathered to analyze the watermark structure. Hackers need more effort to analyze stack and heap dumps than to analyze plain language code. All of these features ensure that a DGW watermark gains a certain degree of protection simply by its method of construction.

2.3.2 DWG Embedding Enumeration Methods

Step (2) of the watermark embedding process, discussed in the previous section, requires us to find a graph corresponding to a desired watermark number. This type of problem has been studied extensively in combinatorics [Goulden and Jackson 1983]. The usual approach is to first index a set of graph structures according to a certain enumeration method, and then use the index number to represent the watermark. Thus, a watermark should be representable as an integer.

In most cases, the index – to – graph structure conversion is performed in a *codec*. A *codec* is the short term for ***co****mpressor/****dec****ompressor,* which is used for compressing and decompressing data [Webopedia Codec]. In this thesis, a *codec* refer to the software that, given an index number, generate a graph structure, and vise versa.

Obviously, there are many enumeration methods and families of graphs that can be used for embedding. In [Collberg and Thomborson 1999], four graph enumerations are discussed. They are: (1) Planted Plane Cubic Tree encoding, (2) Radix-k encoding, (3) Parent Pointer tree encoding, and (4) Permutation encoding. Now we will give an overview of how these enumeration methods works. We will emphasize the enumeration for Planted Plane Cubic Tree structure, which we will use throughout this thesis.

Planted Plane Cubic Tree (PPCT) encoding enumeration

First, we need to define the class of Planted Plane Cubic Trees (PPCTs). Following the definition given by Goulden, a Planted Plane Cubic Tree has these features [Goulden and Jackson 1983]:

(1) Has a root node, which is a single vertex that is distinguished.

(2) The root is monovalent.

(3) The tree is embedded in the plane.

(4) All vertices are either monovalent or trivalent.

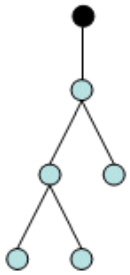


Figure 2-2 A Planted Plane Cubic Tree example

The graph shown in Figure 2-2 has all these features. Its root node is drawn in black. All the internal nodes of the tree are trivalent. None of its edges intersect in this drawing, thus we have a planar embedding.

The PPCT structure is used for dynamic watermark embedding. The data structure representation chosen for PPCTs by Collberg and Thomborson has two outgoing pointers for every node [Collberg and Thomborson 1999]. The single root is called *Origin*, and the node the origin points to is called *Root*. The right-pointers of the leaves are self-pointers, and the left pointers of the leaves point to the next leaf node on its left. The left pointer of the leftmost leaf points to the origin. The Left pointer of the origin points to the rightmost leaf node, and the right pointer points to the root. (Figure 2-3)

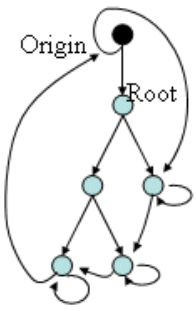


Figure 2-3 A PPCT structure, with two outgoing pointers for all nodes

The extended Planted Plane Cubic Tree has some useful properties. For example, a PPCT is a binary tree and no pointer in the tree structure is null. Further, the node origin points to the root of the binary structure. Lastly, all the leaf nodes are linked together by left pointers and the left most node points to the origin, and finally the origin points to right most leaf node. This structure can help us locate the tree structure, for example, to find the origin node from any node inside a PPCT structure.

According to Catalan number theory [Goulden and Jackson 1983], a PPCT with n leaves (2n nodes), can represent any integer in the following set:

$$\left\{0, 1, 2, \ldots \quad \frac{1}{n}\binom{2n-2}{n-1} - 1\right\}$$

For example, a simple enumeration of PPCTs with 4 leaves is shown in Figure 2-4. For an algorithmic implementation of a PPCT enumeration, please refer to Section 3.4.3.

Radix-k Encoding Enumeration

Radix-k encoding is based on a circular linked list of $k - 1$ nodes, with a distinguished first node. The index number is decided according to the following function:

$$index = a_1 \times k^0 + a_2 \times k^1 + \ldots + a_i \times k^{i-1} + \ldots + a_{k-1} \times k^{k-2}$$

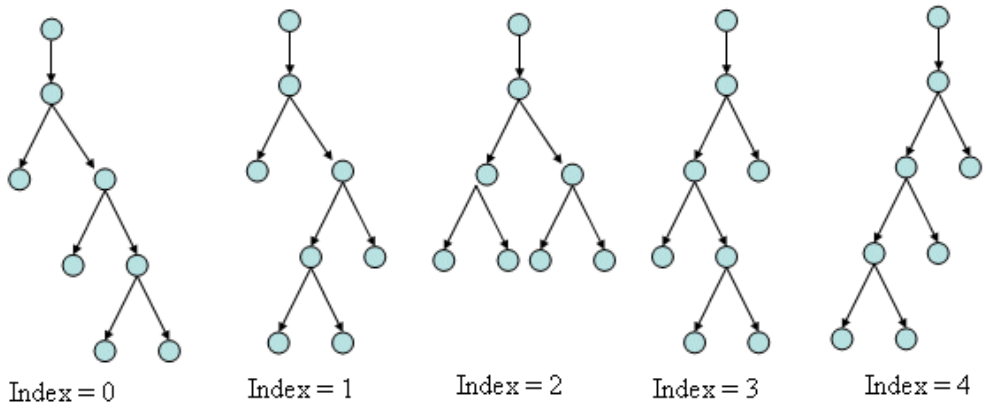


Figure 2-4 Enumeration of PPCTs with 4 Leaves

A node in the circular linked list contains 2 pointers, left and right. The value of a_i is decided by the left pointer of the node, based on following rules:

$a_i = 0$, if the left pointer is null;
$a_i = 1$, if the left pointer point to itself;
$a_i = 2$, if the left pointer points to the next node;
……

$a_i = k$-1, if the left pointer points to the previous node in the circular list.

An example of Radix-k enumeration is shown in Figure 2-5. The figure shows the first eight radix-4 structures.

According to the above enumeration method, a linked list of k - 1 nodes can represent any integer in the range from 0 to $k^{k-1} - 1$ [Collberg and Thomborson 1999].

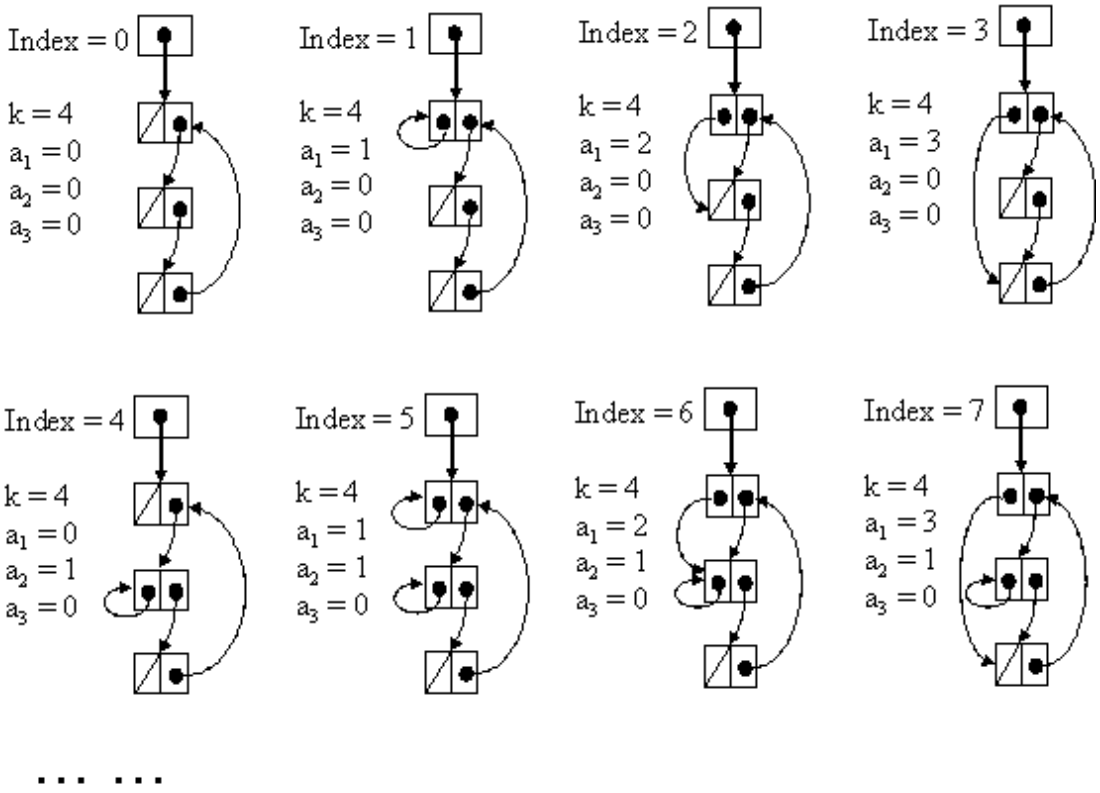


Figure 2-5 Radix-*k* enumeration example with k = 4

Parent-Pointer Tree Encoding Enumeration

Another way of representing a watermark with an indexed graph is by using a Parent-Pointer Tree. In a Parent-Pointer tree, each node has a pointer to its parent. Figure 2-6 shows all of the Parent-Pointer trees with four nodes.

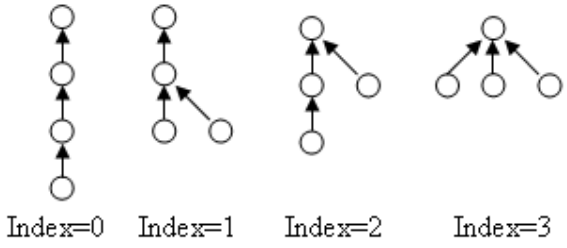


Figure 2-6 Oriented parent-pointer tree enumeration with four nodes

Permutation Encoding

The permutation encoding represents a watermark with a permutation of the numbers ⟨0, …, n-1⟩. For example, watermark number 29 could be represented by the permutation ⟨4, 5, 3, 1, 2⟩ of the numbers ⟨1, 2, 3, 4, 5⟩. A circular linked list can be used for the data structure for embedding. Figure 2-7 shows a circular linked list for representing the number 29, where the 1st number 4 in the permutation is represented by a pointer from the 1st node to the 4th node, and the 2nd number 5 is represented by a pointer from the 2nd node to the 5th node, and so on.

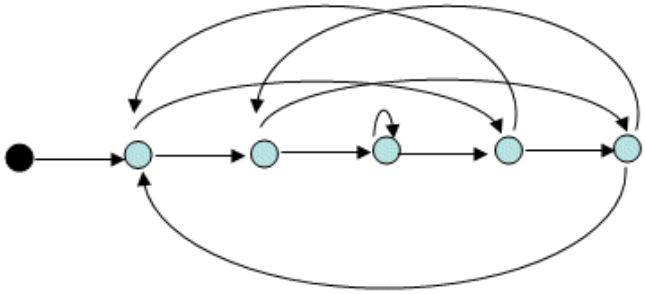


Figure 2-7 A circular linked list for permutation encoding number 29

2.3.3 Current Implementations of DGW

Since the announcement of the DGW algorithm, there have been three implementations of DGW. They are: the Sandmark System from University of Arizona [Collberg, Townsend, et al. 2001], the JavaWiz System from Purdue University [Palsberg SWM], and uwstego from University of Wisconsin-Madison [Jha 2002]. The uwstego system is still under implementation and we cannot find enough information to describe in any detail. Thus, in this section, we will only briefly introduce Sandmark and JavaWiz.

JavaWiz System

The JavaWiz System was built by Palsberg et al. at Purdue University. It targets at Java source code and its implementation is entirely in the Java Language. The JavaWiz System uses a Planted Plane Cubic Tree (PPCT) as its dynamic watermark graph structure and embeds watermark code into the source code of a Java program. If a bytecode program needs to be watermarked, the bytecode must be decompiled into source code, and then it can be watermarked. The JavaWiz System does not need to access all parts of the Java source code and does not touch the standard library [Palsberg SWM].

The source code of JavaWiz is not open-source, but the bytecode package is available online for watermark process testing [Palsberg Source]. From our experience, the function of the JavaWiz System is quite basic. It doesn't need a special input sequence to trigger the watermark-building code. Instead, the watermarked code will build a watermark graph structure when a pre-selected class is called during the program running. It seems that the protection methods mentioned in [Palsberg, Krishnaswamy, et al. 2000] have not been integrated into the system yet.

The watermarking process of Palsberg's JavaWiz System is described in more detail below, in terms of the following aspects: watermark enumeration, watermark embedding, and watermark retrieval.

(1) Watermark Enumeration

The watermark structure used is the Planted Plane Cubic Tree. The enumeration is based on Catalan number as we mentioned in previous section. A Catalan number defines the total number of distinct tree structures with certain leaves. Thus, the JavaWiz system indexes PPCTs according to their leaf number, which means that an integer may be represented by many different PPCT structures. For example, the integer 0 can be represented by any PPCT with index = 0. Four such structures, with varying numbers of leaves, are shown in Figure 2-8. The enumeration method used in JavaWiz will be discussed in detail in the first part of Section 3.4.4.

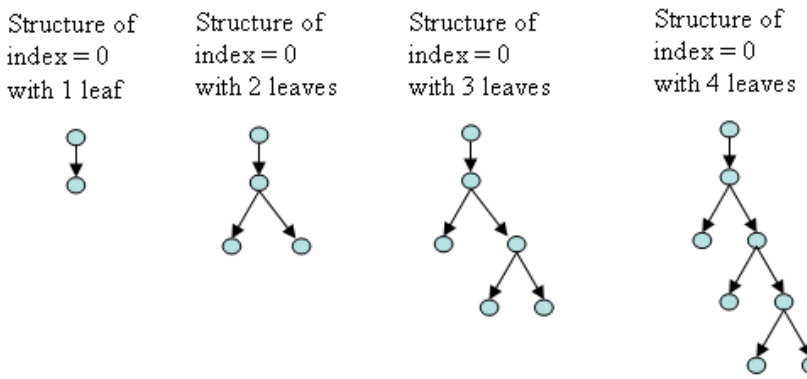


Figure 2-8 Some PPCT structures of index = 0, with different numbers of leaves

(2) Watermark embedding

The watermark embedding process in the JavaWiz system is quite similar to the steps we introduced in Section 2.3.1. Please review Figure 2-1 on page 12. We will point out some features in JavaWiz System that differ from those outlined in that section.

According to the description file [Palsberg Source] and from our examination of the watermarked source code produced by JavaWiz, the embedding process has the following features.

First, the node class used to build the watermark structure is obtained from the candidate program. The user of the system needs to choose a class in the candidate program that can be used as the node class for the PPCT structure. The JavaWiz system will turn that class into the node class, by adding extra outgoing pointers.

Second, the execution point of building the watermark structure is manually selected. The user needs to choose a method in a proper class into which the watermark code will be inserted.

Third, a particular sequence of input is not explicitly required to when building the watermark structure at runtime. The JavaWiz system will insert watermark code in the user-selected method and once the method is called, the watermark structure will be built without a secret input. Note that the user of JavaWiz may also select a method that is called only when a secret input is presented to the program. However, this is not explicitly stated in the JavaWiz system.

Last, the whole watermark graph is built all at once. Collberg and Thomborson suggest building different parts of a watermark structure in different places of a watermarked program, at various times during the program execution. From the source code generated by the JavaWiz system, we see that the watermark source code is inserted into a user-selected method, mixing with the original statements in the method. Thus, the whole watermark graph is built during the execution of a single method, that is, essentially all at one time.

(3) Watermark Retrieval

A watermark is retrieved by analyzing the heap dump at a certain stage of the program execution. Because the owner knows some special properties of the watermark structure, for example, the size of the watermark tree and the type of the watermark structure (in this case, a PPCT), the retrieving process will be efficient.

According to practical results given by Palsberg [Palsberg, Krishnaswamy, et al. 2000], the JavaWiz System will add 4KB to 12 KB extra code to the candidate program, and the program's execution time will increase by at most 7%, and often much less than this.

Sandmark system

The Sandmark System was built by Collberg and Townsend from the University of Arizona [Collberg Page]. It is closely based on Collberg and Thomborson's theory in [Collberg and Thomborson 1999]. Its target program language is Java. Sandmark is not only a tool for software watermarking; it also can be used to study software obfuscation, watermarking and tamper-proofing techniques. The most important point of the Sandmark System is that it shows the possibility of combining and using different algorithms and methods in a program.

The first version of the Sandmark System was written in the Java, Perl, and Icon languages, and the source code is now freely distributed. Recently, a newer version has been built purely in Java [Collberg 2002].

(1) Watermarking enumeration

Currently, there are three *codecs* (refer to the definition in Section 2.3.2) included in the Sandmark system based on three different enumeration algorithms. They are the Radix-k, the permutation, and the PPCT codec. The general idea of these enumeration methods has been introduced in Section 2.3.2, and PPCT enumeration will be further discussed in detail in Section 3.4.4, because it is closely related to the topic of this thesis.

(2) Watermark embedding

The watermark embedding process is more complex than that in the JavaWiz system. The features of this process can be described in the following aspects.

First, the Sandmark system automatically chooses where to insert watermark code. The system will find all possible places for inserting watermark source code and then decide which places will be used.

Second, different parts of the watermark graph are built in different sections of the program and at different times during the program execution. This requires more complicated mechanism to be used to find the place for embedding. The watermark structure will be divided into partitions, which will be built separately during program execution with a particular input. After all of the particular input is accepted, all partitions of the watermark structures are built and then the entire watermark structure will be created so that it can be recognized by a program that analyzes the heap-allocated data structures of the watermarked program. This algorithm is successfully implemented in the

Sandmark system.

Last, watermark pieces are only built when a predefined secret input is presented. This ensures that if the particular input is not present, some portions of the watermark structure, or even none of them, will be built. This technique hides the watermark code away from the attacker's focus, so that a simple dynamic data analysis and execution trace analysis will not find the watermark structure easily.

The embedding process is done in the following steps [Collberg 2002]:

- Annotation: This step is used to find out where, inside a candidate program, a piece of watermark code can be inserted. All potential places will be marked.
- Tracing: This step is used to find out which marked potential places are related to a predefined secret input sequence. During the execution of the program with that secret input sequence, some of the marked places will be hit, and the places that are hit will be used for inserting watermark code.
- Embedding: This is the real embedding step. During this step, a graph structure for a given watermark number (or string) is generated and, depending on the number of places hit, the watermark graph is split into a certain number of partitions, and the code for building those partitions are inserted into the hit places inside the candidate program. Information about root nodes of all partitions is also stored so as to merge the partitions into an entire watermark structure.

(3) Watermark Recognition:

By executing the program with the predefined secret input sequence, all the watermark partitions will be created and merged in the end. By examining the heap dump, the watermark graph structure will be discovered and the watermark is retrieved.

2.4 Attacks on Software Watermark

Attacks on software programs can be classified into two different types: malicious client attacks and malicious host attacks [Collberg and Thomborson 2000]. For the case of a malicious client attacking a benign host computer, the privilege of the malicious client can be limited by the benign host. Thus, the threat of a malicious client attack is not as great as in the case of malicious host attacks, in which case the host has full privilege to examine and modify the client code. Since watermark information is embedded in client program, the

attacking of watermarks is in the category of malicious host attack.

2.4.1 Overview of Attacks on Software Watermark

Because of the privilege of a malicious host, various technologies can be used for de-watermarking. The possible methods that could be used include reverse engineering, source code/bytecode analyzing, program output analyzing, execution trace analyzing, stack and heap analyzing, and so on.

Static data watermarks and code watermarks are highly susceptible to attacks using code obfuscation techniques [Collberg, Thomborson, et al. 1998a] [Collberg, Thomborson, et al. 1998b]. This is because the obfuscation techniques will change the original representation of the static data in a program. It will also change the order of some instructions, where the order of these instructions is not critical. Obfuscation may also replace instruction sequences by equivalent instruction sequences. For example, obfuscation may break strings to substrings or change the order of the cases in a switch statement. If the watermark information depends on the strings, or on the order of the cases, this could make the watermark unrecognizable.

Some dynamic watermarks can also be attacked by the obfuscation techniques. Execution trace watermarks are especially susceptible to obfuscation attacks. Easter Egg watermarks are unaffected by obfuscation attacks. Dynamic data structure watermarks are susceptible to some advanced data obfuscation attacks, for example, when the nodes representing a DGW are “split” or “merged”. Further analysis of obfuscation attacks can be found in [Collberg, Thomborson, et al. 1998a].

There are many ways of attacking watermarks other than obfuscation. According to Collberg and Thomborson [Collberg and Thomborson 1999], attacks can be classified in the three ways shown in Figure 2-9, and described briefly below.

- Additive attacks: the attacker inserts his own watermark. The result of this attempt is overriding the original watermark, or at least making it plausible that the original watermark was not inserted before the attacker’s. See Figure 2-9 (ii).
- Subtractive attacks: the attacker tries to locate and remove a watermark without affecting the usability of the software program. See Figure 2-9 (iii).
- Distortive attacks: the attacker tries to modify the watermark without damaging the usability of the software program. The result of this attack is that the watermark is not

recognizable, but the software is still of value to the attacker. See Figure 2-9 (iv).

Many tools are available to an attacker. Jobe [Jokipii] can change the identifiers of a program. KlassMaster [Klassmaster] not only can transform the names and strings, but also can offer flow obfuscation that changes most selections (e.g. if...else) and loops (e.g. while and for). Jad [Kouznetsov] can decompile Java class files to source code. The use of these tools makes the attacker more effective and efficient.

2.4.2 Attacks on DGW watermark

Now let us examine in detail how these attacks can affect Dynamic Graph watermarks.

Most of the technologies for attacking other types of watermarks, such as semantics-preserving transformation, and code obfuscation, have no effect on DGW watermark. However, some de-watermarking techniques can effectively make the DGW watermark retrieving process fail. According to [Collberg and Thomborson 1999], an attack on DGW watermark can be an additive attack, a subtractive attack or a distortive attack.

(1) Additive Attack:

Add a new watermark as discussed before. This attack will work in the same way as for other types of watermarks.

(2) Subtractive Attack:

Remove the watermark completely, so that no watermark can be retrieved for software authentication.

Because of the difficulty of alias analysis, a dynamic graph watermark is hard to remove. However, once the attackers find any part of the DGW structure, they can eliminate all node (or fields) of that type from the program, and then delete all references to these nodes (or fields), unless there is a tamperproofing technique to protect the watermark.

Please look at Figure 2-10. The method in Figure 2-10 uses opaque predicates, creating two pointers to a watermark structure *wm* (discussed in Section 2.3.3). The attacker must succeed in an alias analysis to replace the predicate (n_1 != n_2) by the appropriate constant value 1 (true).

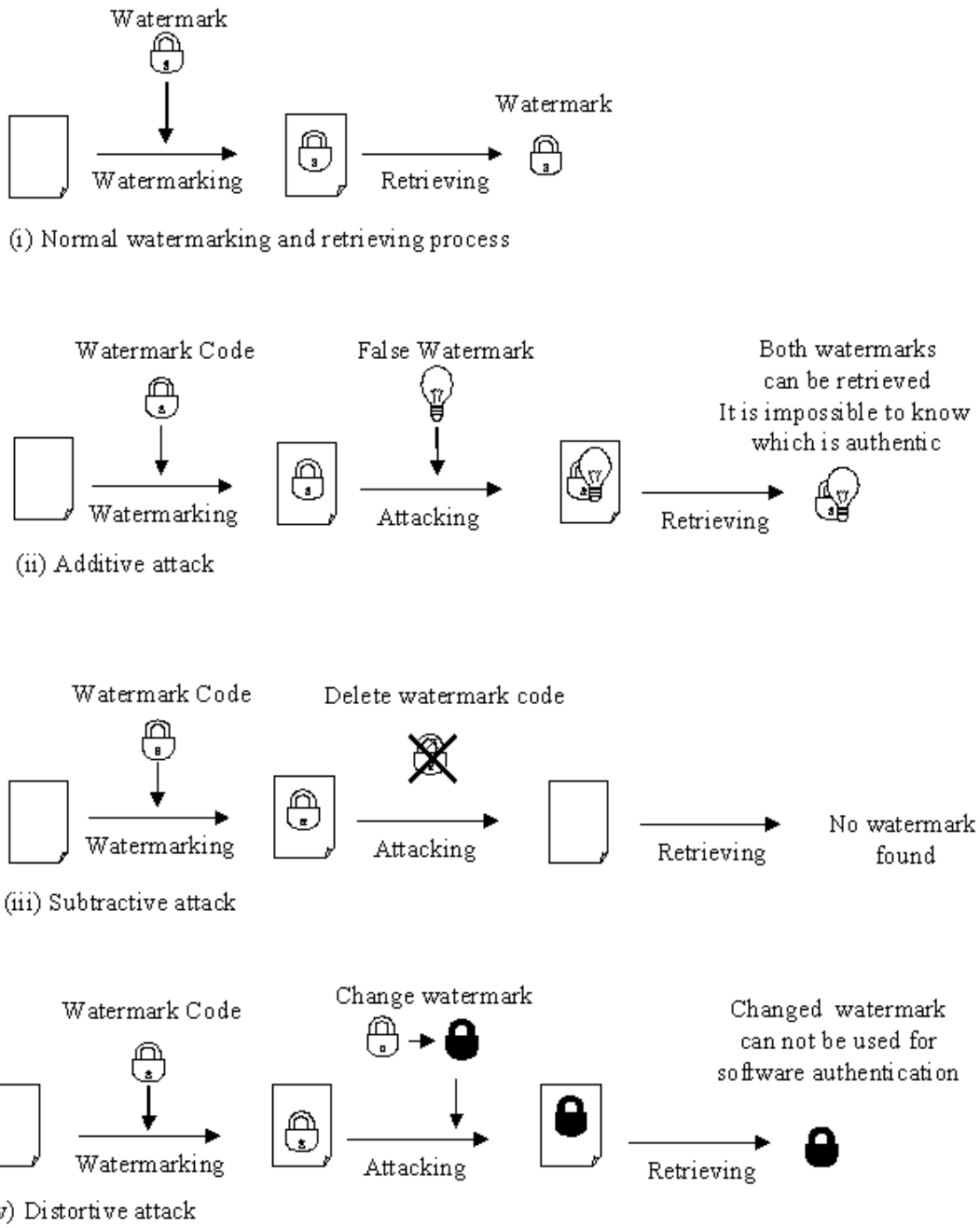


Figure 2-9 Attacks on software watermark

A second type of tamperproofing is illustrated in Figure 2-10 in which we make use of the DGW type for some useful purpose in the program. If the attacker removes all nodes of type DGW from the program, then they will also remove the usefulStructure of this type. Distinguishing the *wm* from the usefulStructure requires an accurate alias analysis. The first of these two tamperproofing methods was briefly suggested in [Collberg and Thomborson 1999]. Both methods were developed more fully in [Palsberg, Krishnaswamy, et al. 2000].

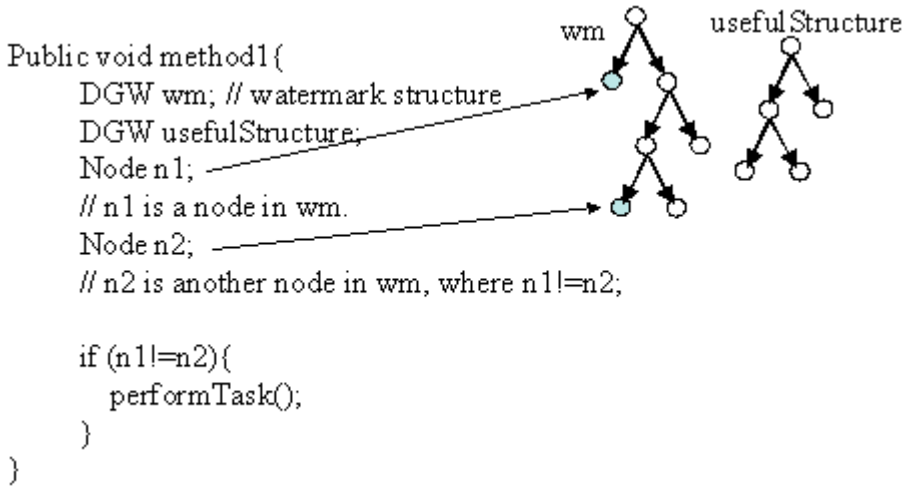


Figure 2-10 Tamperproofing a DGW against a subtractive attack

(3) Distortive Attack (Figure 2-11):

We identify five types of distortive attacks, and discuss each in turn below. See Figure 2-11.

- Add extra pointers to the graph nodes. An attacker may modify the node class of the watermark graph structure, making the watermark retrieving process more difficult. For example, by adding an additional field to the node class, the original watermark structure *G* (shown in Figure 2-11 (i)) looks like the one in Figure 2-11 (ii).
- Delete part of the structure. An attacker may delete part of the watermark structure, making the retrieving process fail. For example, the structure G_2 in Figure 2-11 (iii) shows the node *d* and *e* in the original structure *G* have been removed.
- Split nodes. The node class of the watermark structure has been split into two (Figure 2-11 (iv)). This action adds an extra level of reference to each node inside the original watermark structure *G*, so that the original status of links between the nodes is changed, and the data structure of the watermark tree may also be changed.
- Add extra nodes to the graph structure. In Figure 2-11 (v), three extra nodes (*f, g, h*) have been added into the original graph *G*. This will make the retrieving process difficult because of the change of the graph structure.

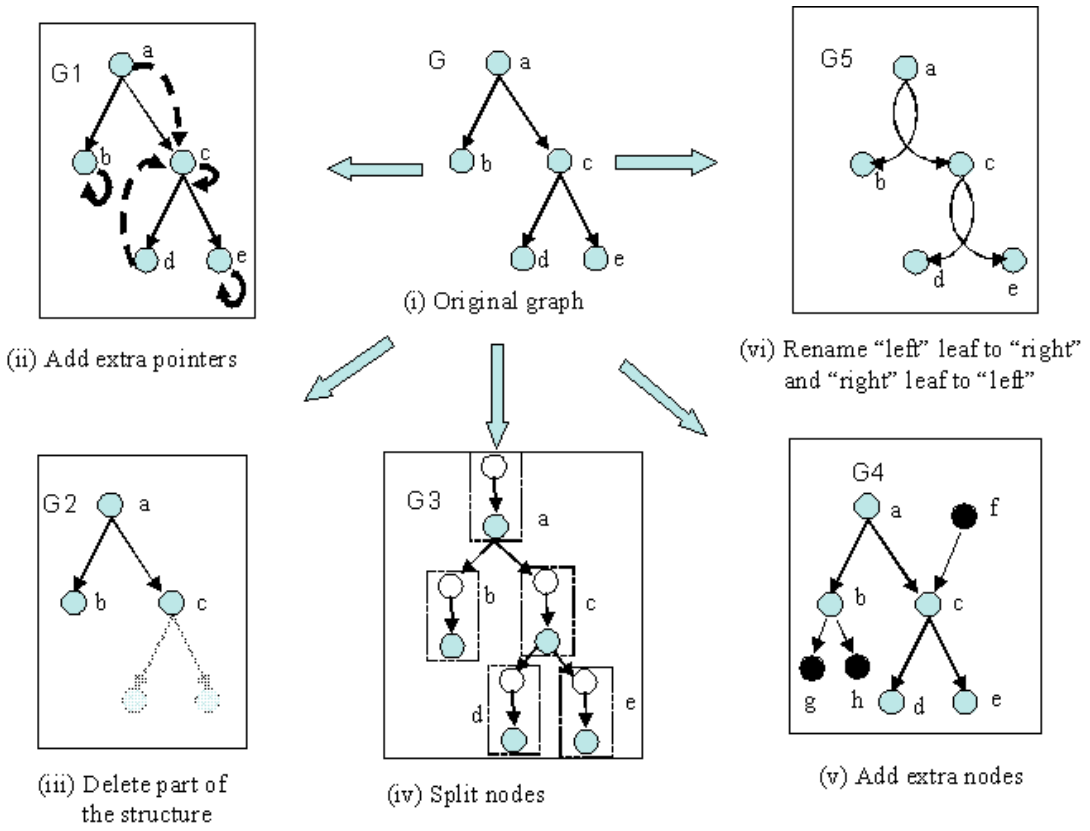


Figure 2-11 Attacks on DGW watermark

- Rename or reorder the fields inside a node. Figure 2-11 (vi) shows the left and right pointer name in the original graph *G* have been swapped. The result is the left subtree, which was the three node subtree, becomes a single node, and the right subtree, which was a single node, becomes a three node subtree. This change may affect the result of the watermark retrieving. Note: the term "subtree" will be defined formally in Section 3.1.

2.5 Watermark Protection

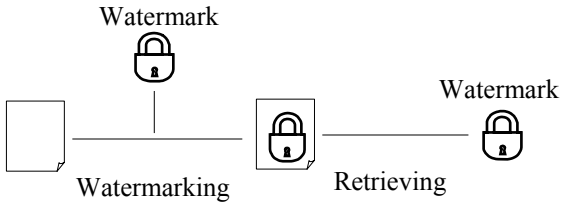
The goal of a software watermark protection technique is either to make it difficult for an attacker to analyze watermarked programs so as to prevent the watermark from being found, or to prevent the watermark code itself being modified or removed. However, software watermark protection has not been given much attention even in recent years. Different kinds of watermarks may need different protection techniques, and protection methods for newly developed watermarking techniques, such as DGW, have not received much attention.

In this section, we will discuss the few protection technologies that have been published for software watermarks. First, we will give an overview of protection technologies. Then,

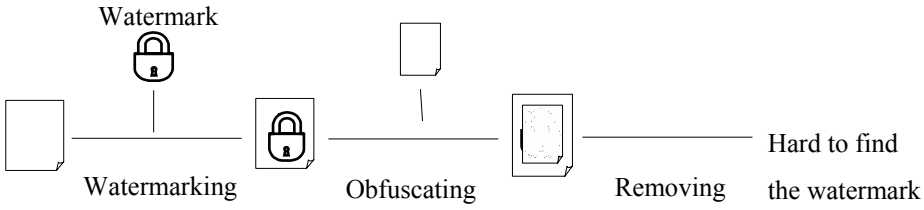
we will focus on the methods currently available for protecting DGWs.

2.5.1 Overview of Protection Techniques

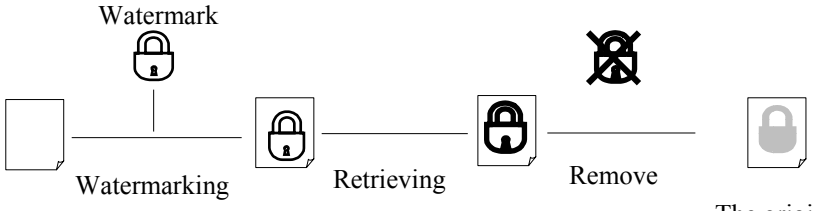
Obfuscation techniques can make it difficult for an attacker to analyze source code. Tamperproofing can make it difficult for an attacker to modify a software program without affecting the usability of the program. These concepts, of obfuscation and tamperproofing, have already been introduced in Section 1.5 and in Section 2.4.2. Figure 2-12 shows how these techniques are used to protect software from attackers.



(i) Normal watermarking and retrieving process



(ii) Obfuscation: makes watermark stealthier



The original program is changed and can't be executed properly.

(iii) Tamperproofing: makes watermark more resilient to attacks, if watermark is removed, the original program will have execution errors.

Figure 2-12 Watermark protection techniques: obfuscation and tamperproofing.

In Figure 2-12(ii), obfuscation makes the watermark stealthier, making it hard for the attackers to analyze the watermarked program. Obviously, if the attacker cannot find the watermark, he will have less chance to perform a fatal attack. Figure 2-12(iii) shows another possibility, that the attacker may have some way to remove the watermark, or at least to destroy part of the watermark. This can be prevented by tamperproofing. Tamperproofing

typically creates a dependency from the watermarked program to the embedded watermark structure. If the watermark structure is modified or removed, the watermarked program might function incorrectly.

However, the above protection methods may not be applied to all kinds of watermarks. For example, a static watermark may be destroyed, rather than protected, by obfuscation. Thus we see that the way in which a watermark is built can play an important role in its protection.

2.5.2 Protections for DGW watermark

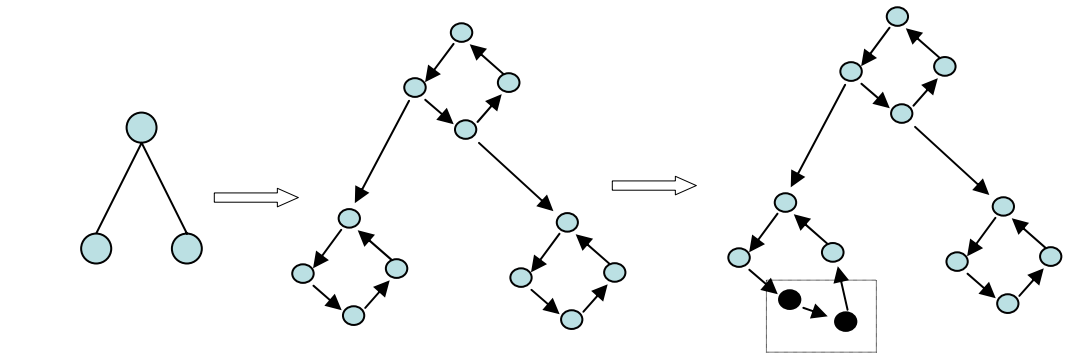
If an attacker does not know how the watermark is embedded, they will have to apply general methods, which are effective at attacking the most common watermarks. Fortunately, the DGW watermark algorithm has some intrinsic features that protect it from attacks, as discussed in [Collberg and Thomborson 1999]. Thus, a general attack on a watermark may not be a severe danger to a DGW watermark. For example, any semantic-transforming obfuscation will not affect a DGW watermark. However, if the attacker possesses some information about the watermark, for example that it is a DW, or specifically a PPCT DGW, or even more specifically, the root node of the PPCT DGW, their attack will be more serious or even fatal.

As we discussed in Section 2.4, an attack normally happens in one of two ways: either someone is trying to destroy or distort the watermark by means such as obfuscation when the attacker does not know the location of the watermark, or someone is analyzing the watermarked program and trying to find the location of the watermark and then applying attacks that aim particularly at that watermark. Therefore, we can improve the watermark protection in three ways. First, we can make the DGW watermark more robust, so that the DGW watermark could not be affected by advanced obfuscations such as data structure transformations. Second, we can prevent attackers from finding the watermark structure. Finally, we need to prevent the watermark from being removed, even when an attacker has discovered the location of the watermark. Of course, a protection method may have one or more functions among the three ways listed above.

In [Collberg and Thomborson 1999], Collberg and Thomborson discussed an improvement to DGW watermarking to protect against node splitting attacks. We know that if a node in a DGW watermark has been split into multiple parts or if an extra layer of redirection is added into a node class, as shown in Figure 2-11(iv), then the DGW watermark

structure may no longer be recognizable. Collberg and Thomborson suggest replacing each node in a DGW watermark structure with a cycle of nodes, as shown in Figure 2-13. In this case even when an attacker tries to destroy the watermark structure by splitting a node into two (Figure 2-13(iii)), we can still merge each node cycle into a single node so that the original shape is acquired. Any further splitting of the node just makes a cycle bigger, but does not affect watermark retrieval. However, as also mentioned in [Collberg and Thomborson 1999], this method requires extra space to store many extra nodes, which seems uneconomical.

In the same paper, Collberg and Thomborson also mentioned a way of protecting by using reflection. Reflection in the Java language can be used to get information about a program, such as the method, fields, and constructors [Sun Microsystems REF]. This information can be used to verify that the node class for a watermark structure remains unchanged. Thus, Java reflection can be used to protect against many types of attacks such as node reordering and renaming. However, as they pointed out, it is unstealthy to use reflection in a program that otherwise does not use reflection.



(i) Original structure (ii) Node been replaced with a cycle of nodes (iii) Attacker splits a node into two

Figure 2-13 Protection from node splitting.

Palsberg, in [Palsberg, Krishnaswamy, et al. 2000], described using an if-statement to create a dependency from watermarked program to the watermark structure. We mentioned this idea in Section 1.5 and again in Section 2.4.2. Palsberg's method transforms a statement "S" in the candidate program into the following if-statement, where x and y are references to distinct nodes in the watermark structure:

```
if (x != y) S
```

The predicate of this if-statement will always evaluate true. Thus, the statement S will always be executed. However, it is hard for the attacker to know whether x and y will always be

different. Without this knowledge, the attacker cannot safely remove x and y from the program, so at least this part of the watermark structure must be preserved.

Palsberg does not propose to protect the watermark structure directly by these if-statements. Instead, Palsberg suggests creating a second tree structure at the beginning of the watermarked program. This second structure is used as the source of distinct nodes for tests of the form x != y. However, the second watermark tree is still somewhat exposed to attack. Even so, this is a very efficient algorithm, which can build dependencies without too much cost. We believe Palsberg's algorithm still needs further development, in order to prevent pattern-matching attacks.

2.5.3 Integration of Protection Methods

No technology can guarantee immunity to all attacks. Thus, we might try to use all known methods together to achieve a higher level of software protection. The three general methods (watermarking, obfuscation, tamperproofing) were originally developed to control software piracy, but now they can be used for watermark protection.

2.6 Discussion

The sole purpose of the code that builds a DGW is to protect a software program. This leads to a dangerous situation: the watermark code does not relate closely to the candidate program. Therefore, an adversary might not find it too difficult to recognize the watermark code from watermarked program. Obfuscation can protect a program from static analysis, increasing the difficulty of such attacks, however, and this makes attacking more difficult, even obfuscation cannot prevent all recognition attacks with certainty.

Currently, systems based on DGW technology achieve a certain degree of protection. This is because of the nature of DGW technology as we discussed before. Some ways of protecting DGWs have been discussed, but it is impossible to achieve full protection. Thus, finding a more effective method for protecting DGWs is still an open problem.

2.7 Summary

In this chapter, we discussed the concept of watermarking and outlined technologies used in software watermarking such as static watermark and dynamic watermark. We emphasized Dynamic Graph watermarking, which will be useful in the following chapters. We discussed

protection technologies that increase the resilience of watermark, with special attention to DGWs. In the end, we pointed out that no defensive method can protect a watermark thoroughly. However combining different technologies may achieve a better result. Increasing the level of protection for DGW watermarks is still an open problem. In the following chapter, we will discuss a new algorithm for DGW watermark protection.

Chapter 3

Constant Encoding

3.1 Introduction

As we discussed in Chapter 2, a dynamic graph watermark is built at runtime in the program-controlled memory, and it is difficult to analyze or attack such watermarks. However, the DGW watermark does not relate closely to the functions of its candidate program. Thus, the DGW watermark still can be attacked by intensive analysis and modification to the watermarked program. Currently, there are several different ways of protecting DGW watermarks, such as creating the watermark only on specific inputs, and inserting opaque predicates as we have discussed in Section 2.5. These technologies, together with some general methods such as obfuscation and tamperproofing, can cause difficulties to attackers. However, there is still a need for improvements in DGW watermark protection.

In this Chapter, we discuss a new technology of DGW watermark protection that creates false dependencies from the candidate program to the DGW watermark code embedded in the program. In other words, we will show an algorithm that, in normal execution, builds a second tree that is the same type as the watermark tree, and makes the execution of the program dependent on the correctness of the second tree structure. Because of the difficulty of alias analysis, since the second tree is the same type as the watermark tree, attackers will have great difficulty in analyzing which is the watermark tree (which he wants to attack) and

which is the second tree (which he must not modify). Thus, if an adversary tries to remove or modify the watermark code without distinguishing the watermarked tree and the second tree, he will have a significant danger of rendering the program unusable. Moreover, if we apply opaque predicate tamperproofing such as Palsberg's if-statement discussed in Section 2.5.2, the second tree can be used as another source of opaque predicates. This makes the watermark structure even more secure.

Our algorithm is based on the idea of encoding constants into the watermark structure. We describe how this algorithm works from Section 0 to Section 3.4. After the process has been illustrated, we discuss some issues about this algorithm in Section 3.5. In later chapters, we show how the algorithm can be realized by building a prototyping system. We give a name to our prototyping constant encoding software - JSafeMark encoder. Thus, from now on we call the constant encoding technique JSafeMark Technology.

The JSafeMark technology manipulates the constants inside a candidate program, by converting each constant into a graph topology. The graph structure it handles is a PPCT. See Section 2.3.2. The technology will be discussed in detail later on. In some aspects, the encoding process that the JSafeMark carries out is quite similar to the process of turning a watermark number into a graph structure, when the watermark is embedded by a watermarking system. In order to avoid ambiguity, before we start to describe the constant encoding procedure, we need to introduce several terminologies that will be used later on in this thesis. Please note that we only deal with watermark trees that are PPCT structures in this thesis.

Embedding: refers to the process that embeds a watermark or a watermark graph structure into a candidate program; see Section 1.4 and Section 2.3 for more details.

Decoding: refers to the process of converting a graph structure into an integer

Encoding: depending on the context, encoding has two meanings. IN restricted contexts, it refers to the process of converting integers into graph structures. In general discussions, we speak of a "Constant Encoding Process", which introduces decoding and encoding processes into a candidate program.

Binary Tree: A directed graph in which each node (other than the root) has a parent node, and in which each node (other than the leaf nodes, is the parent of exactly two nodes. The leaf nodes of a tree have no children.

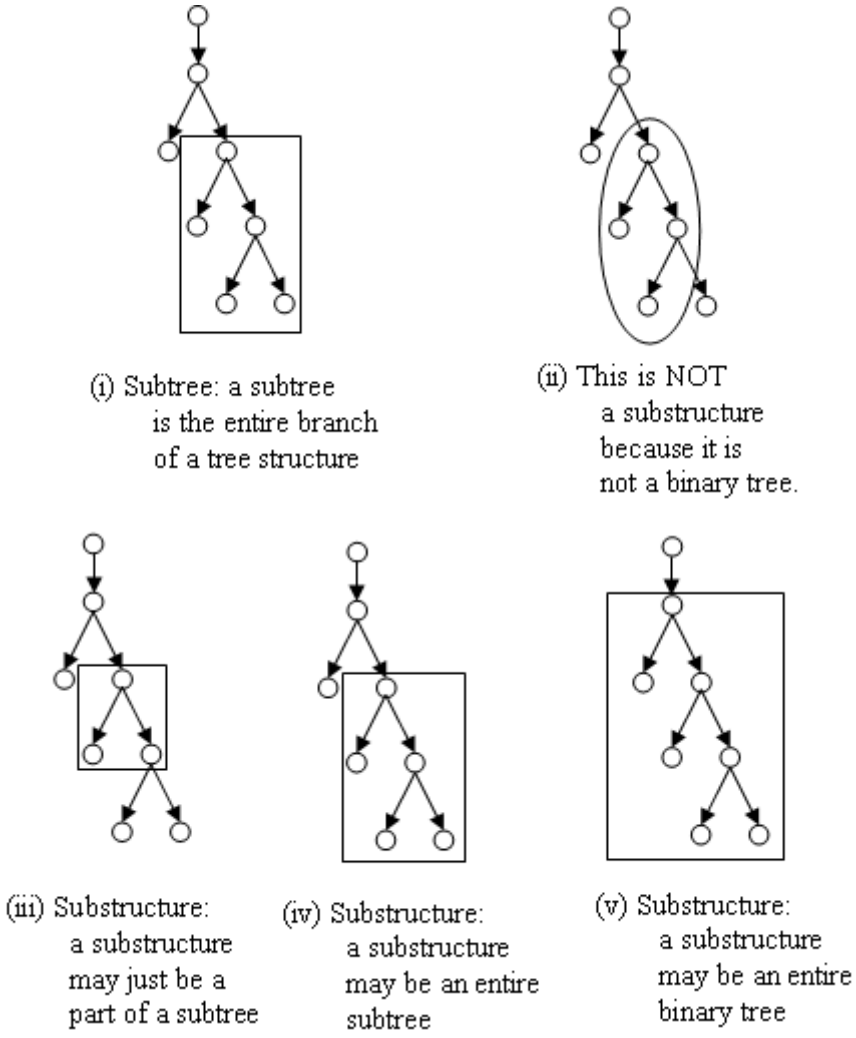


Figure 3-1Comparison of subtree and substructures.

Subtree: Refers to a branch of a binary tree. See Figure 3-1(i);

Substructure: refers to any binary tree that is a sub-graph of another binary tree. Note that any subtree is a substructure. However, a substructure is not necessarily a subtree. Three substructures are shown in Figure 3-1(iii), (iv) and (v). Figure 3-1(ii) shows a graph that is not a substructure, because it is not a binary tree.

Constant Substructure: refers to a substructure that encodes from a constant.

Constant Tree: Refers to a pseudo-randomly generated PPCT whose substructures encode constants. In the discussion at the beginning of this chapter, we called it a “second tree”. A constant tree is initially contains a randomly generated, relatively small PPCT. The constant tree may be expanded when encoding constants and its shape is fixed after the time of constant encoding.

E-Constant: a constant chosen from a candidate program that has been encoded into a

substructure of the constant tree.

Constant Graph: refers to a PPCT that encodes a constant. In the encoding process, we will ensure that the constant tree contains at least one substructure that is isomorphic to the constant graph of each *E-constant*.

3.2 Technology Overview

In chapter 2, we discussed the theory of dynamic graph watermarking, and we also showed an example of how to watermark a simple Java program with DGW watermarking techniques in Figure 2-1. From this example, we saw that the functionality of the original program did not depend on the watermark code. This is a weak point of the DGW watermarking algorithm. JSafeMark algorithm offers obfuscating and tamperproofing technologies to protect DGW watermarks by creating dependencies from a candidate program to a constant tree, which is hard to distinguish from the embedded DGW watermark code. JSafeMark is intended to be as an additional step in DGW Watermarking Systems, to protect watermarks against malicious attacks.

JSafeMark is designed for DGW watermarks with PPCT structures. We have already briefly illustrated its operation in Section 1.4. The idea of the protection algorithm is to convert some constants (called *E-Constants*), which are found inside a candidate program, into graphic structures that have the same PPCT structure as the watermark graph to be embedded. Graphs converted from constants are called constant graphs. For each constant graph, we try to find a substructure in the constant tree that has the same shape as the constant graph. We call this substructure a Constant Substructure. Then we generate a function that can be used to retrieve the constant value out of the constant substructure inside the constant tree. Thus, we can replace the constant with the constant retrieving function. This procedure creates dependencies from the candidate program to the constant tree structure.

What we expect from this algorithm is: if the attacker tries to modify the watermark code without distinguishing the watermark tree and constant tree, the constant tree may be modified or removed. Thus, the values of the encoded constants will not be retrieved correctly, thereby affecting the execution of the watermarked program. This algorithm can be illustrated in Figure 3-2.

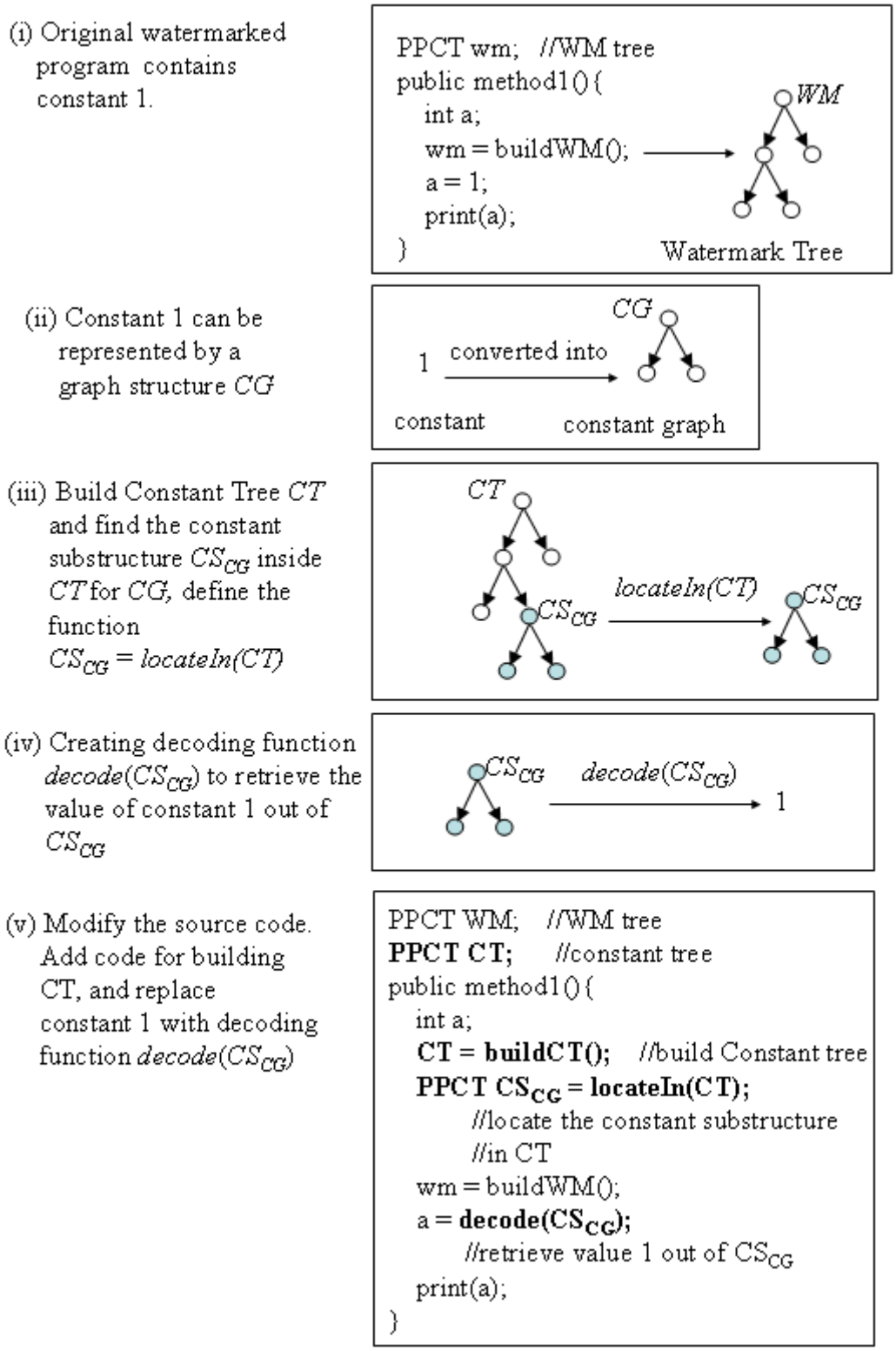


Figure 3-2 Constant encoding process

Figure 3-2 (i) shows a watermarked program. There is a watermark structure *WM* that may be created from the watermark code at some stage of program runtime according to the watermarking algorithm used. A constant integer 1 can be found inside the program. Figure

3-2 (ii) shows that the constant 1 can be represented by a graph structure *CG*. Figure 3-2 (iii) shows a constant tree *CT* is built and a constant substructure CS_{CG} is found for the constant graph *CG* inside *CT*. We can find CS_{CG} in *CT* by using function CS_{CG} = *locateIn(CT)*. Figure 3-2 (iv) shows that the value of constant 1 is decoded from the constant substructure CS_{CG} by using *decode*(CS_{CG}). Now we are ready to modify the source code. In Figure 3-2 (v), the source code of Figure 3-2 (i) is modified to include the decoding process. We eliminate the constant value 1 and replace it with a decoding function. We also add the code for building the constant tree and locating the constant substructure in the constant tree. Finally, the constant integer 1 inside the original program is replaced by the function *decode*(CS_{CG}). Thus, the proper running of the original program relays on the correctness of the constant tree.

By analyzing the process in Figure 3-2, we notice that the watermarked program relies on the constant tree *CT*. The value of constant 1 will be converted from a substructure of constant tree *CT* at the program's runtime. Since the decoding function *decode*(CS_{CG}) takes the subset CS_{CG} inside the constant tree *CT* as its parameter, and *CT* is built in the PPCT structure, which is also used by the watermark structure, it is hard for attackers to know which structure can be safely removed or modified.

A decoding function may have more parameters and look more complicated than the one shown in Figure 3-2. This will bring further difficulties to that attackers while they trying to analyze the program. In future work, we suggest using additional constant trees, and/or global pointers into constant tree, to make it even more difficult for the attacker to discover the watermark tree.

3.3 System Structure for JSafeMark Plug-in

Now we will have a look at the relationships between the watermarking techniques in the DGW watermarking system and JSafeMark technology. Our suggested process flow is illustrated in Figure 3-3. Please note, in this thesis, we use the phrase "program execution time" and "runtime" interchangeably.

The order of the watermarking process and the constant encoding process is not important in our implementation. However, if the constant encoding process is launched after the watermarking process, some of the constants in the watermarking code may also be used in constant encoding.

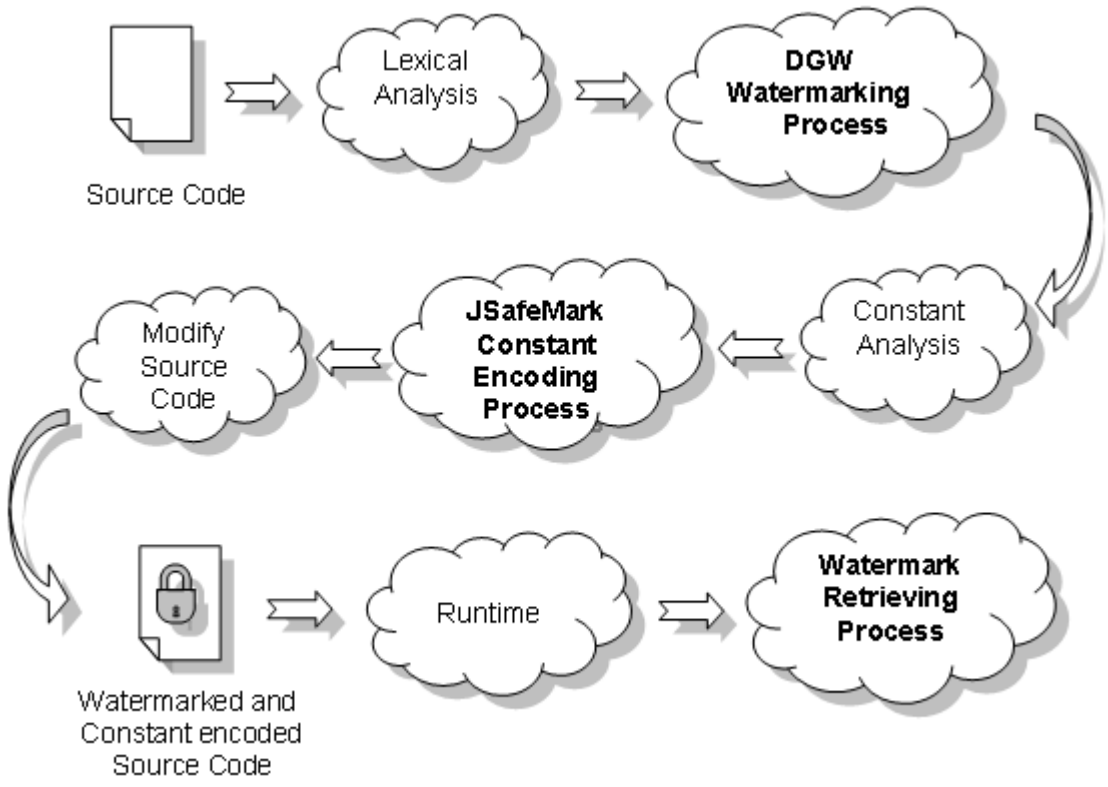


Figure 3-3 Proposed watermark embedding and constant encoding process flow

Both the JavaWiz and Sandmark system discussed in Section 2.3.3 work with Java source code, which means that these systems contain lexical analyzers, parsers and processes to modify source code. Rather than duplicating these functions, we assume the DGW systems will handle the entire source code handling process, including the constant analysis. JSafeMark Technology can thus be focused focus on the constant encoding and decoding method generation process (see the structure illustration in Figure 3-4; the import and export information will be discussed in Section 4.3 when the JSafeMark encoder interface design is introduced). This structure considers JSafeMark Technology to be a functional plug-in module to the DGW system. It seems that the DGW watermarking system has to do more work than before, but we can also see that the gain for such a scheme is much more than the loss. Firstly, the integrity of the DGW system is maintained. All the processes dealing with source code are handled by the DGW system. This makes the DGW system independent of the JSafeMark Technology Module. Secondly, the JSafeMark Module is optional to the DGW systems. This structure makes changing and updating JSafeMark encoder much easier, when compared with any scheme that includes the JSafeMark Technology as a fixed part.

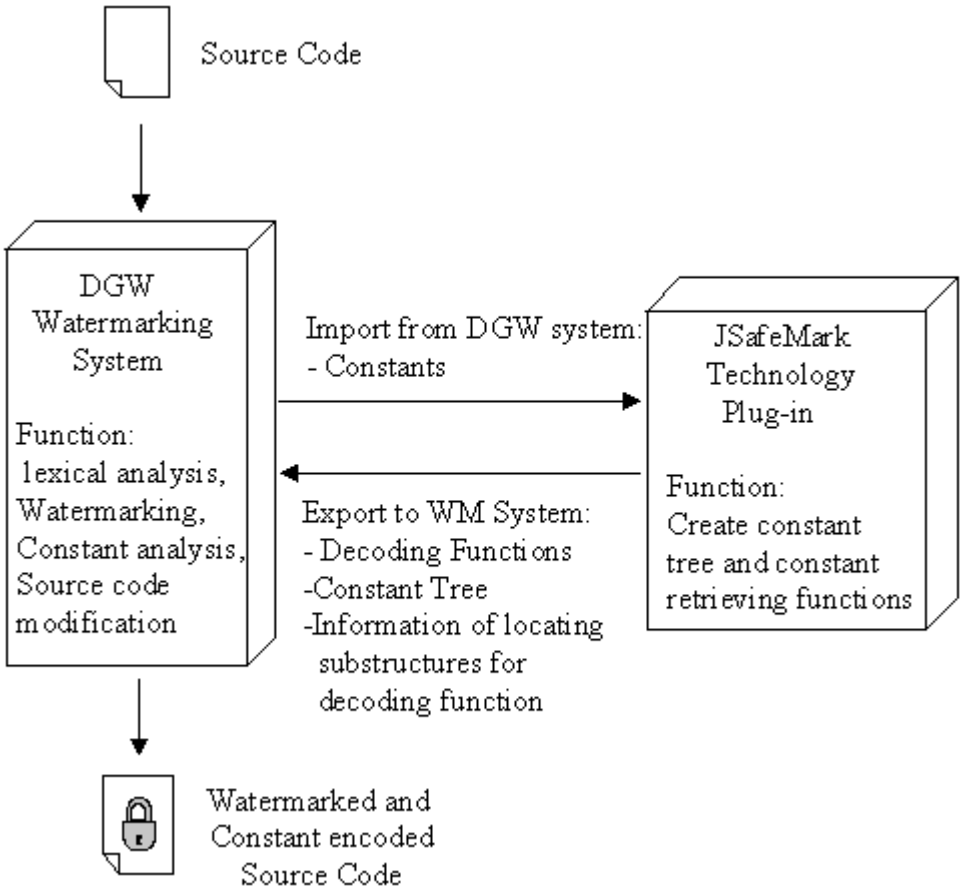


Figure 3-4 System structure for the DGW system and the JSafeMark encoder

3.4 Constant Encoding Process

Now that we have discussed the integration of the JSafeMark Technology with the DGW System, we can start to define the process of utilizing the JSafeMark algorithm in DGW systems.

3.4.1 Work Flow

The encoding process can be described in three phases. The first phase is encoding preparation, which is done in the DGW watermarking system. The second phase is the constant encoding phase, which is done by the JSafeMark module, which creates constant trees and decoding functions. The final phase is the modification of source code, which is handled by the DGW system. Our focus in the thesis is on the constant encoding phase of this process. The input to this phase is a list of E-constants identified by the preparation phase.

The preparation phase of the constant encoding must find the constants used by the constant encoding phase.

The constant encoding phase consists of the steps WF1 through WF10, listed below. Steps WF3 through WF10 are iterated, with all the E-constants are encoded and decoding functions are defined for each E-constant.

WF1 The encoder accepts a list of E-constants.

WF2 The encoder generates a pseudorandom PPCT as an initial constant tree. This PPCT should be big enough to have a good chance of encoding all E-constants. However, it should not be so big as to affect the performance of the watermarked program. The constant tree might be increased in size in step WF8.

WF3 For each constant, if the constant is too large to be encoded into a compact graph structure, the encoder will turn it into a suitable format for encoding. For example, the encoder will split a low precision double constant into two small integers.

WF4 The encoder generates code to invert the process of WF3, so that the original constant can be obtained at program execution time.

WF5 An E-constant is encoded into graph structure(s) using the data structure of the embedded watermark tree (that is, the PPCT structure).

WF6 The encoder generates code to invert the process of WF5, so that the value of the E-constant can be decoded from the graph structure at program execution time.

WF7 The encoder searches for a constant substructure in the constant tree, of the same shape as the constant graph (WF7-1). If the substructure is found, the encoder records the location of the root of the constant structure (WF7-2), and start to process the rest of the integers.

WF8 If a matching constant substructure is not found in step WF7, the encoder adds nodes to the constant tree so that it contains a substructure of the required shape.

WF9 The encoder generates code to derive a reference, which we call $Info_{CS}$, to the root of the constant substructure, given one (or more) references to the constant tree. This code will be called at runtime so that the decoding function in WF6 can be used.

WF10 The encoder generates code to call the functions generated in WF4, WF6 and WF9, to form a final decoding function. The decoding function takes the constant tree and the information for locating substructures in the constant tree as parameters, and retrieves the complete E-constant value out of the constant

substructure(s) in the constant tree. This function is slightly different from the function *decode*(CS_{CG}) shown in Figure 3-2, in which the constant substructure was referenced directly in order to simplify the description. In practice, we do not explicitly reference the constant substructure, to increase the stealthiness of our encodings.

WF11 The encoder exports the decoding functions for all E-constants and for generating the constant tree.

After the above process, the DGW system enters the final phase of constant encoding, in which the source code is modified. In this phase, the DGW system modifies the source code of the candidate program to include the source code of building the constant tree, and replaces the constants by the decoding function calls. The DGW system must ensure that the constant tree is built before any constant encoding function is called. Since the processes for modifying source code are out of the scope of our thesis, we will not discuss these here.

The workflow for the second stage of our constant encoding process is shown in Figure 3-5. The workflow in Figure 3-5 ignores the preparation phase and the final phase, for these are handled by the DGW system. At the bottom of the figure, in step WF11, the results of the second stage of the constant encoding process are exported to the DGW system. The results are the decoding functions, the constant tree (*CT*), and the information (*Info*) for finding each constant substructure (CS_i) in the constant tree.

For a much easier process in the step WF7 and WF8, if a suitable substructure cannot be found in the constant tree, we do not need to modify the constant tree at all. Instead, the encoder might inform the DGW system that this constant was not encoded. Alternatively, the encoder might return to step WF3, and choose a different splitting of a difficult-to-encode constant into smaller constants.(Note: small constants, such as 0, 1, or 2, can be encoded into very small PPCTs, which will always be found, even in modestly-sized constant trees.)

Starting form the next section, we introduce a more detailed description for the above steps. In Section 3.4.2, we give a few more details for the encoding preparation phase. The constant encoding phase, steps WF1 through WF11, discussed in great detail in Sections 3.4.3, through 3.4.6. In Section 3.4.7, we consider the final phase or source code modification.

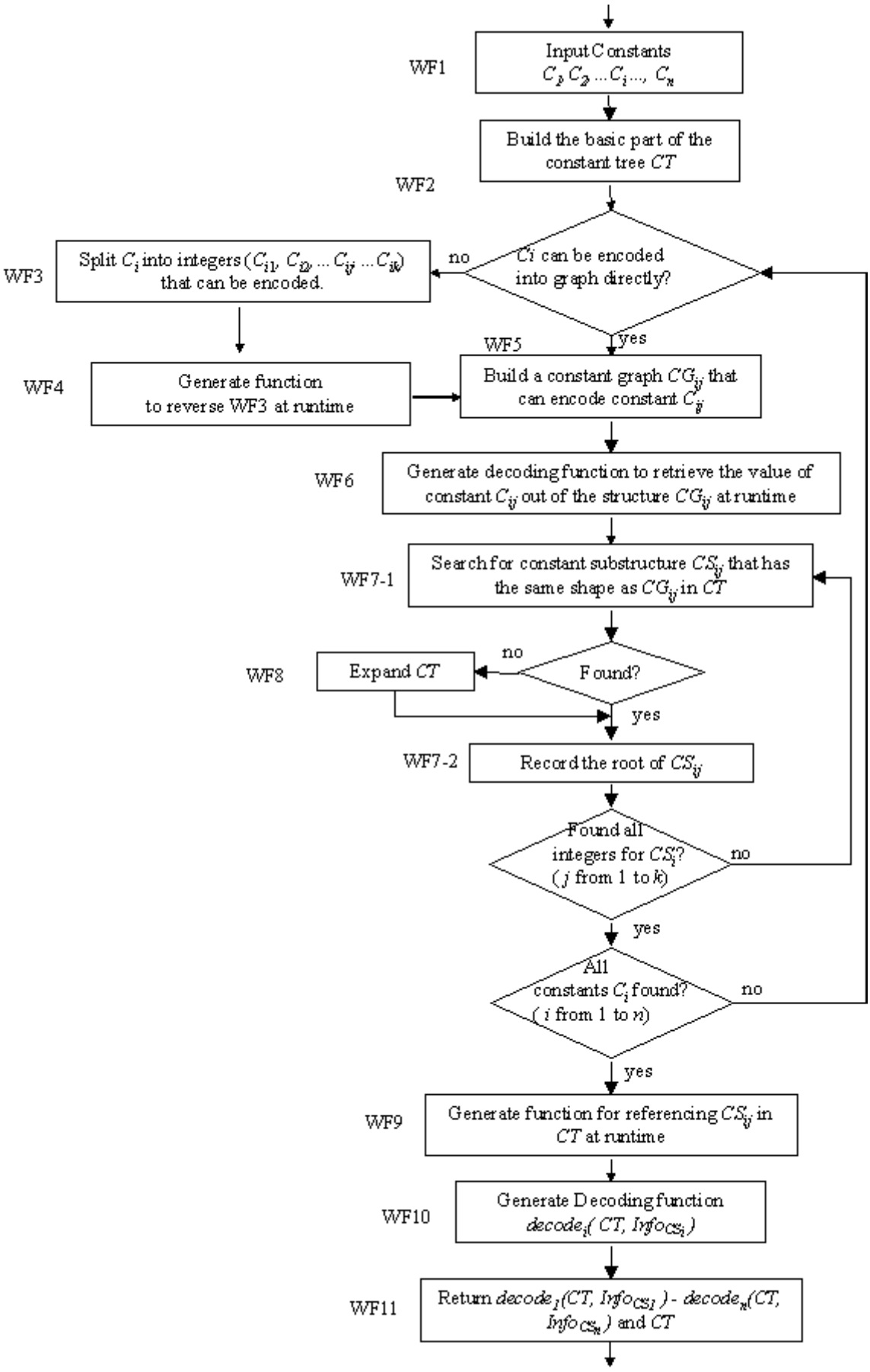


Figure 3-5 Control flow of constant encoding process

3.4.2 Constant Analysis by the DGW System

This process is launched by the DGW System and is not in the focus of our research. Briefly speaking, we expect the DGW System to perform the following tasks:

(1) Analyze a candidate program to find all the possible constants that can be evaluated by a decoding function. (Note that some constants may be required before the constant tree can be built).

(2) Decide which constants will be used for embedding (these are *E-constants*). A chosen *E-constant* should be easy to convert into an integer in a restricted range, say 0 … 9 or 0 … 255. Some large integers and non-integer constants can be handled by the methods described in Section 3.4.4.

(3) Record the locations of these constants in the source code, so that these can be replaces by decoding functions in the third and final phases, of source code modification.

3.4.3 Building Constant Tree

The first two steps in the second stage (See Figure 3-5) are inputting the E-constants in WF1, and building the initial constant tree in WF2. The basic idea of building a constant tree is simply to generate a structure that will look, to an attacker, very much like a watermark tree. However, this tree will be used only to encode E-constants. We believe an attacker will have great difficulty removing only the watermarking tree without also removing the constant tree. The removal of both trees, without affecting program correctness, would require extensive analysis on the part of the attacker.

The shape of a constant tree is variable during the constant encoding process, and is unchanging during the program's execution time.

The requirements for building a constant tree are:

(1) The constant tree for a watermarked program should use the same data structure as the one the watermark tree using in that program. In JSafeMark, this structure is the PPCT structure.

(2) The constant tree for a watermarked program should not be so big that it slows down the execution of that program.

To build a constant tree, JSafeMark will follow the methods below:

(1) We choose an integer uniformly at random, in the range 0 … *m*, where *m* is a suitably large integer (say, *m* = 1000) to be determined by practical considerations. If *m* is too small, then we will not be able to encode a wide variety of constants in our constant tree. If *m* is too large, then the runtime and space overhead will be excessive on our candidate program after constant encoding.

(2) The initial constant tree *CT* is computed as the *i*-th PPCT in a suitable enumeration of PPCTs. We will discuss enumerations in Section 3.4.4.

3.4.4 Building Constant Graph (CG)

Building a constant graph is a process that turns a constant into a dynamic graph structure. In JSafeMark, this is a PPCT structure. This process includes the step WF3 and WF5 in the workflow in Figure 3-5. The algorithm used to build constant graphs to encode an integer constant may be different from the one used by the DGW system to encode a watermark number. However, the algorithmic principles are the same. In both cases, we use what is called a "codec" – a pair of encoding and decoding functions. An encoder is a function that returns a graph, given an integer; the corresponding decoder returns the original integer given this graph.

As we will see later in this Chapter, there are many possible codecs for building a certain kind of constant graph structure. This choice is illustrated in Figure 3-6. In this Figure, we notice that building watermark trees and building constant trees can share the same library of codecs. The functionality of the codec library here is similar to the codecs defined in the Sandmark System [Collberg 2002], except that our codec must have functions to generate the source code for including the decoding method in the candidate program. In Sandmark and other watermarking systems, the decoding process is part of the watermark recognizer, which is generally not distributed with the source code. Besides showing the PPCT method library, Figure 3-6 also shows some codec libraries for other graph structures, in order to give a more complete view of the codecs used by the JSafeMark and the DGW system.

In this section, we will give some examples of how to convert a constant into a constant graph. By introducing several examples, we hope the converting process will become more clear to the reader, also, our examples will demonstrate some of the scope of the methods we envisage for encoding constants.

A convenient way of encoding an integer into a graph, and vice versa, is enumeration. In

an enumeration, each graph topology assigned an index [Goulden and Jackson 1983]. Using enumerations for this purpose in the DGW watermark embedding is suggested by Collberg and Thomborson in their paper [Collberg and Thomborson 1999]. Clearly, enumerations can also be used to convert between integers, constants, and graphs. In The JSafeMark, all constants must be converted into integer type before being converted into graphs.

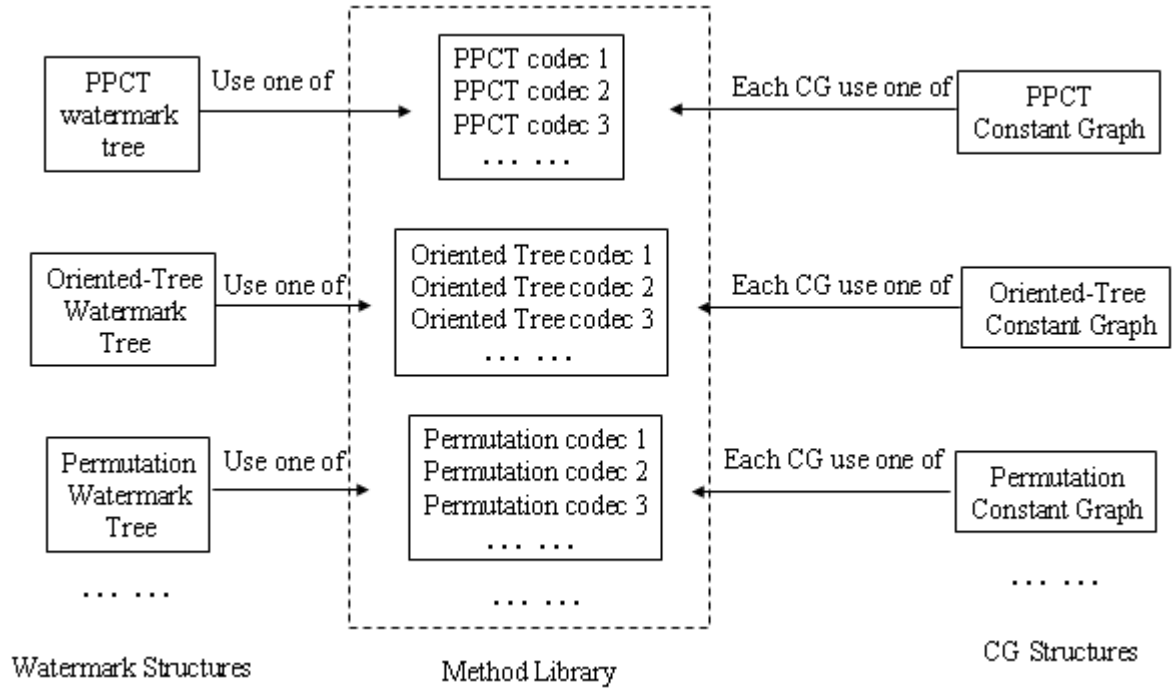


Figure 3-6 Relationships between codecs used in watermarking and in constant encoding.

Thus, in the process of converting constants to graphs, we need to follow two steps. First, define methods to convert all types of constants into integer. Second, define methods to turn the integers into graphs. When converting (decoding) a graph back into a constant, we need to reverse the above process of encoding.

Constant Types and Type Conversion:

Different languages have different sets of constant types. The target programming language of our implementation is Java. Therefore, in this part, we will focus on the constant type forms in Java. The algorithms we introduce in this thesis can be migrated to other programming languages if needed, but the basic idea of types and typecast will be the same.

The Java language is a strongly typed language. Types for all the variables must be specified at compile time. This allows more errors to be detected at compile time, and it ensures the safe execution of programs [Lindholm and Yellin 1997].

Data types in the Java language can be divided into two categories: primitive types and reference types [Gosling, Joy, et al. 2000], Figure 3-7 shows the type categories in detail. The types used for constants in Java program can be primitive types or reference types. Primitive types include float, double, byte, short, int, long, char, Boolean. Reference types can be an array of primitive types, or some built-in Classes, such as String, Integer, Double, Long, and so on.

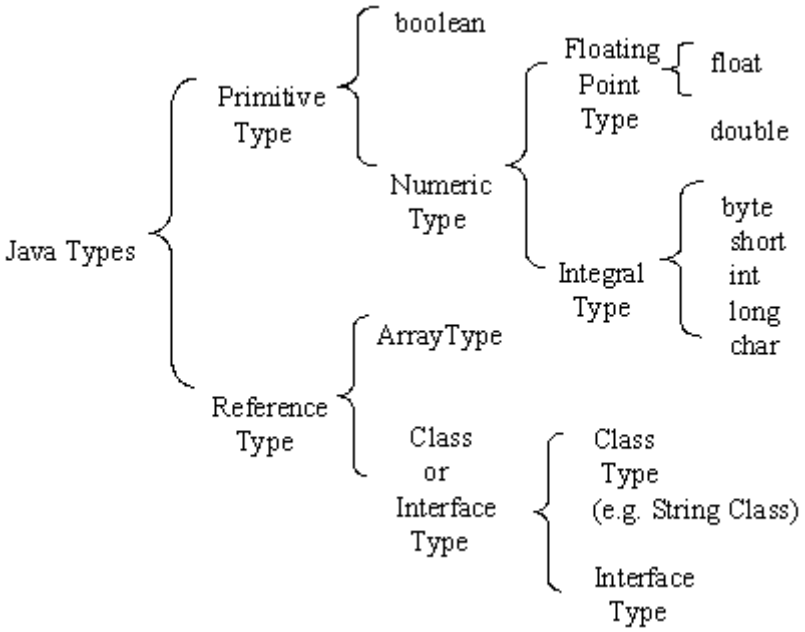


Figure 3-7 Java types classification

Constant to Integer Conversion: (WF3 in the work flow)

There are many ways of converting each constant type into one or more integers. We could form a conversion library, with several variants for each conversion, might give additional security to the watermark program, by making it more difficult for an attacker to mount a pattern matching attack. Normally a statement such as a switch statement is needed for encoding a constant into an integer in different ways according to its type, but this is not always the case. Sometimes we can pass a particular type of constant directly into a method that particular for that type by using method overloading. For example,

public void constantToInt(double d) and

public void constantToInt(String s)

will directly accept different types of constants, this is what we did in our prototyping. However, some constants with types other than int are hard to be converted, because of the extra overhead. Thus, only int's and small part of other types of constants can be selected by

the DGW system. We now give a simple example on how to convert a positive low-precision floating-point constant into an integer, for use when the constant is in the format of "x.y", where x and y are single digit integers. For example, when x = 3 and y = 2, the double constant is "3.2". We can use the following algorithm to turn such a low-precision positive double into two single digit integers in Java:

```
public static Vector constantToInt(double d){
        if (d<0) throw exception; // cannot handle negative values
        int firstInt, secondInt;
        firstInt = (int)d;
        if (firstInt>9) {
                throw exception; //integral part is greater than 9
        }
        if (((""+d).length()>3){
                throw exception; //two many digits
        }
        secondInt = (int)((d-(int)d)*10);
        Vector result = new Vector();
        result.add(new Integer(firstInt));
        result.add(new Integer(secondInt);
        return result;
}
```

Encoding Integer into Constant Graph (WF5 in the work flow)

Different graph structures need different encoders. For example, building PPCT structures, and building Oriented-Tree structures require different algorithms. In this thesis, we consider only PPCT structures

Below, we introduce three methods for encoding integers to PPCT graphs. We call these methods the Catalan Leaf-Oriented Conversion (CLOC), the Bit-Oriented Conversion (BOC), and the Catalan Unique Indexed Conversion (CUIC). We focus our attention on the first,

which is based on the Catalan number sequence [Dershowitz and Zaks 1980]. Part of the source code of this implementation can be found in Appendix A.

(1) Catalan Leaf-Oriented Conversion (CLOC):

After the PPCT enumeration for DGW watermark embedding was pointed out by Collberg and Thomborson [Collberg and Thomborson 1999], Palsberg et al. published recursive formulas for encoding and decoding these formulas use Catalan Numbers, which are named after the French mathematician Eugène Charles Catalan [Palsberg, Krishnaswamy, et al. 2000]. Our conversion method is based on Palsberg's formulas. However, during our implementation, we noticed that there is a bug in one of Palsberg's formulas. This will be pointed out later in this section. The Sandmark System currently adopts the core part of our implementation as its PPCT codec [Collberg and Townsend 2001].

The number of different PPCTs with n leaves can be computed by the following formula, for the *n*-th Catalan Number,

$$c(n) = \frac{1}{n}\binom{2n-2}{n-1}$$

The sequence *c(n)*, for n in the range 1 to 16, is [1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, 9694845].

In Figure 3-8, we draw all PPCTs of four or fewer leaves. This illustrates that $c(1) = 1$, $c(2) = 1$, $c(3) = 2$, and $c(4) = 5$.

We use the method and notation of Palsberg's paper [Palsberg, Krishnaswamy, et al. 2000], to decode graph structures into integers. See the definition of "*decoding*" on page 34.

We denote the CLOC index of a PPCT *T* as *int(T)*. We write *T.left* for the left sub-tree of *T*, *T.right* for the right sub-tree of *T*, *L* for the number of leaves in *T.left*, and *R* for the number of leaves in *T.right*. Then we have:

$$int(T) = 0, \quad \text{if } |T| = 1 \tag{3.4.1}$$

$$int(T) = int(T.left) \times c(R) + int(T.right) + min\_int(L, R), \quad if\ |T| > 1 \tag{3.4.2}$$

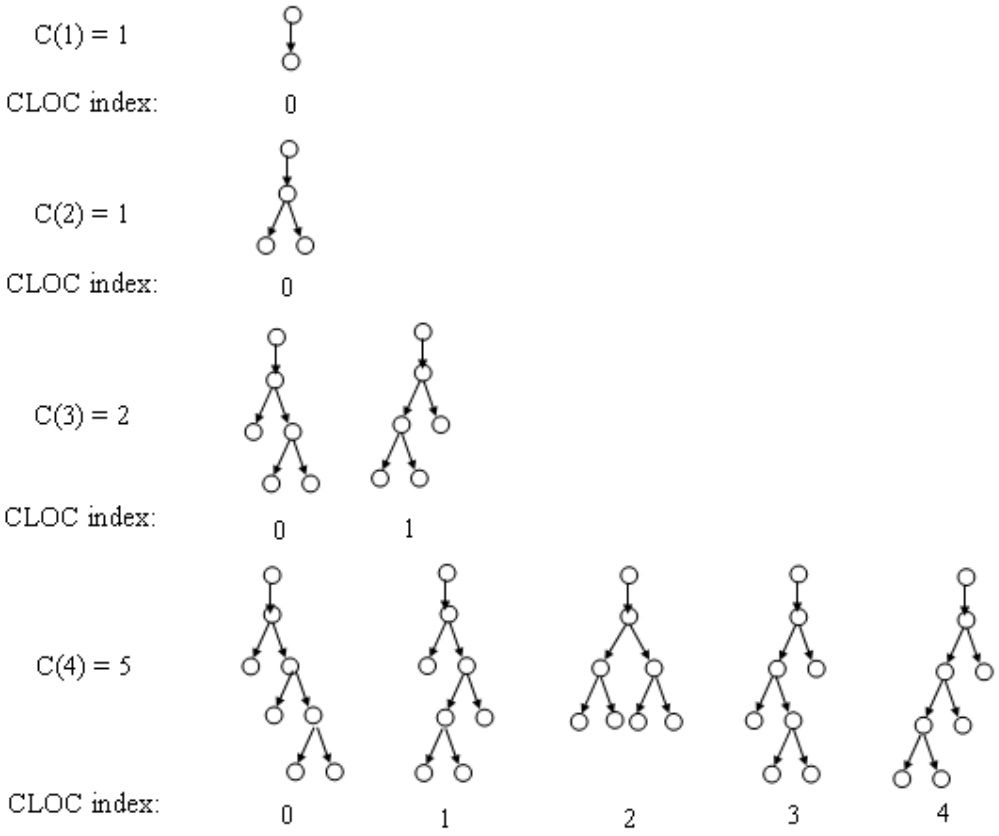


Figure 3-8 CLOC index of PPCTs with 1 to 4 leaves

Palsberg suggests the following formulas for computing *min_int(L, R)* recursively:

$$min\_int(1, R) = 0 \qquad (3.4.3)$$

$$min\_int(L, R) = min\_int(L-1, R+1) + c(L-1) \times R$$

Through our experimentation, we discovered that Palsberg's second formula is incorrect. A correct formula should use $c(L+1) \times c(R-1)$ instead of $c(L-1) \times R$, giving the following correct version of Palsberg's second formula:

$$min\_int(L, R) = min\_int(L-1, R+1) + c(L-1) \times c(R+1) \qquad (3.4.4)$$

The function *min_int(L,R)* can be described as follow.:

We can eliminate the recursion in equation (3.4.4).

$$min\_int(L,R) = \sum_{1<i<L} (c(L-i)c(R+i)) \qquad (3.4.5)$$

From expression (3.4.5) we can see that *min_int(L, R)* is equal to the total number of PPCTs on $L+R$ leaves with less than L leaves in their left subtree.

Because Catalan number *c( )* is monotone increasing, we believe it is possible to prove

what we have observed in practice on individual cases, that the name given by Palsberg to *min_int( )* is mnemonic:

$$min\_int(L,R) = \min_{T \in \mathcal{T}(L,R)} \{int(T)\}.$$

where $\mathcal{T}(L, R)$ is the set of all PPCTs with L leaves in the left subtree and R leaves in the right subtree.

After some analysis of equations (3.4.1) through (3.4.4), we wrote the following pseudo code for encoding a positive integer i in the CLOC enumeration of n-leaf PPCTs. We implemented and tested this pseudo code in our createTree() method in Appendix A, see page 119.

```
Node encode(int nodeNo, int i){
    int L, R;
    if (i >= c(n)) return null;          //the integer is too big to encode in a
                                         //n leaf PPCT, so we return an error signal
    if( i == 0 && nodeNo = 1) build and return a PPCT tree T with one leaf and
                                          CLOC index 0;
    else if( i == 0 && nodeNo = 2) build and return a PPCT with two leaf and
                                         CLOC index 0;
    else if ( i<3 ) return null;         //only i == 0 can be encoded in 1- and 2-leaf
                                         trees, see Figure 3-8
    L = 1;  while( c(L) <= i )  L++;
    R = n – L;
    // L-1 leaves in the left subtree, and R+1 leaves in the right subtree.
    Node left_subtree, right_subtree;
    left_subtree = encode( (L-1, i - min_int( L – 1 , R + 1 )) / c(R + 1) );
    right_subtree = encode((R+1, i - min_int( L – 1 , R + 1 )) % c( R + 1));
    return a PPCT whose subtrees are left_subtree and right_subtree;
}
```

Please note by using this algorithm, an integer may be represented by more than one structure. Figure 3-9 shows all the PPCTs with up to four leaves that can be used to represent the integer 0.

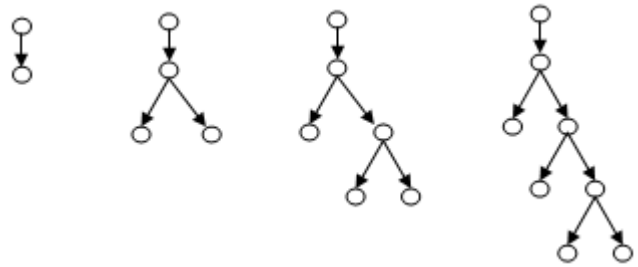


Figure 3-9 Four PPCTs that represent the integer 0

(2) Bit-Oriented Conversion (BOC):

The idea of this conversion is based on the binary tree feature of PPCT. Please refer to Figure 3-10(a). If we represent a single-leaf left subtree by the string "0", and a single-leaf right subtree by the string "1", we concatenate the string "1" for right subtree and the string "0" from the left subtree, then the two-leaf PPCT of Figure 3-10(a) can be represented by string "10".

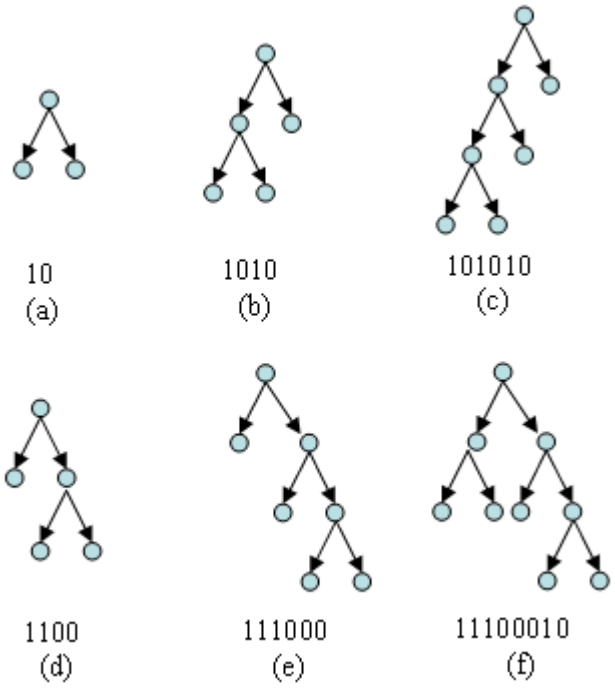


Figure 3-10 Structure for Bit-Oriented Conversion

We can extend this method to encode more complicated binary structures. For example

the three leaf tree of Figure 3-10(b). Its left-subtree is isomorphic to the tree in Figure 3-10(a), so it is decoded as the string “10”, with prefix “0” to indicate it is a left subtree (we use prefix “1” to indicate a right subtree when needed). The right subtree has a single leaf, so it is decoded as the string “1”. Concatenating the string “1” from the right subtree with the string “010” from the left subtree, we obtain the string “1010”, the complete decoding of the tree of Figure 3-10(b).

After a PPCT structure is converted into string representation, we need to represent the string by an integer.

For example, the string representation of the structure in Figure 3-10(a) is “10”. We need the integer 2, whose binary format is “10” (ignore the preceding 0’s), to represent the structure.

The pseudo code for decoding general PPCTs into bit-streams is shown as follows:

```
String decodeBOCInString(Node T){
    if( T is leaf) return “0”;
    String left_str, right_str;
    if( T.left is a leaf)
            left_str = “0”;
    else left_str = “0” + decodeBOCInString( T.left );
    if ( T.right is a leaf )
            right_str = “1”;
    else right_str = “1”+decodeBOCInString(T.right);
    return right_str + left_str;        //reversed order
}
```

The above algorithm can convert a structure into a string representation. By using an algorithm such as the following code, we can convert the string into a 32-bit integer representation.

```
public static int convertBOCStringToInt(String s){
    if (s.length()>7) {
```

```
            System.out.println( "String with incorrect length."); return -1;
            //test if the string is oversized. If yes, return -1 to indicate an error.
        }
        int result = 0;  //result integer to be returned.
        while(!s.equals("")){  //while the string is not empty
            String ss = s.substring(0,1);    //take the first character.
            if (result==0 && ss.equals("0")) continue;
                //remove the preceding 0's
            if (ss.equals("1")) {result*=2; result+=1;}
            else if (ss.equals("0")) result*=2;
            else {System.out.println("String format error");return -1;}
                //string s cannot contain characters other than 1 and 0.
            s=s.substring(1,s.length());     // remove the first character
        }
        return result;
    }
```

Until now, we have given the algorithm for the transition, trying to build a codec to decode a PPCT structure into an integer using BOC. Now we will give an method to encode an positive 32-bit integer into a PPCT structure.

We convert an integer into a string. For example, the binary format of the integer 10 is "0000,1010" (ignore the first 24 bit 0's). We convert it into a string and remove the preceding 0's, and get the string "1010". When doing the encoding, we first create a new stack *s*, and define a structure *T* as a new node, used to hold the result structure. Please refer to in Figure 3-11(a). We start working on the string from the left side. We do a push operation for character "1", and a pop operation for character "0". For the first character "1" in string "1010", we push the structure in *T* into the stack. We call the structure in the stack *T'*, and replace *T* with a new node (Figure 3-11(b)). For the second character "0" in string "1010", we pop the structure *T'* out of the stack, and construct a new structure with *T'* and *T* as its left and right subtree. Then we rename the new structure to be *T* (Figure 3-11(c)). For the third

character “1”, we push the structure of *T* into the stack, and create a new node for *T*. We still call the structure in stack *T’* (Figure 3-11(d). For the last character “0”, we pop *T*” out of the stack and construct a new structure with *T’* and *T* as its left and right subtree. We rename the new structure to be *T* (Figure 3-11(e). Since the whole string has been processed, we get the final structure *T*.

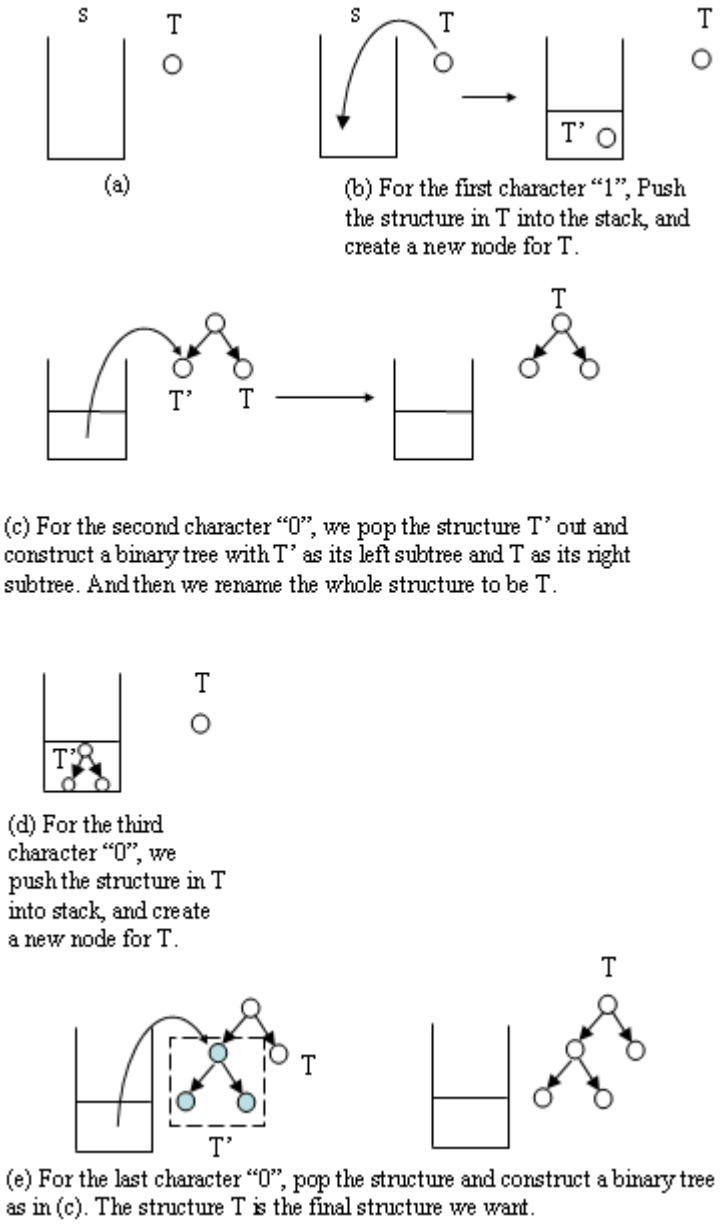


Figure 3-11 Integer to graph encoding using BOC

The pseudo code for encoding an integer into a binary structure by using BOC is listed below.

Node IntegerToGraph(int n){

```
    Node T = new Node();          //result tree, initially a single node.
    int m = n;       //local copy of n
    int b;   //least significant bit of n;
    Stack s = new Stack();          //initially empty
    while(m!=0){
        b = m&1;
        m = m / 2;
        if ( b ){
            if (s.empty()) throw Exception;
            Node temp = new node();
            temp.left = (Node)s.pop();     //use popped structure
                                           //as left subtree
            temp.right = T;          //using the existing structure
                                     //as right subtree
            T = temp;       //put the constructed structure into T
        }else{
            s. push(T);
            T = new Node();
        }
    }
    if (!s.empty()) throw Exception;
    return T;
}
```

Please note that not all integers can be encoded into binary structures and for ease on understanding, we did not remove the first and the last characters in the string, which are redundant and always are “1” and “0”.

(3) Catalan Unique Indexed Conversion (CUIC):

In the discussion in Catalan Leaf-Oriented Conversion, we saw that an integer might be represented by different PPCT structures with varying numbers of leaves. We will now show an efficient encoding that eliminates this duplication.

We write *CUIC(T)* to dente the CUIC index of a PPCT tree T, and using int(T) (recall that we used it when discussing CLOC in page 49) to be the CLOC index of the tree T. Then the indexing function for CUIC algorithm can be defined as follows.

$$CUIC(T) = int(T) + \sum_{n=1}^{LeafNum(T)-1} c(n)$$

This Conversion can be illustrated by Figure 3-12.

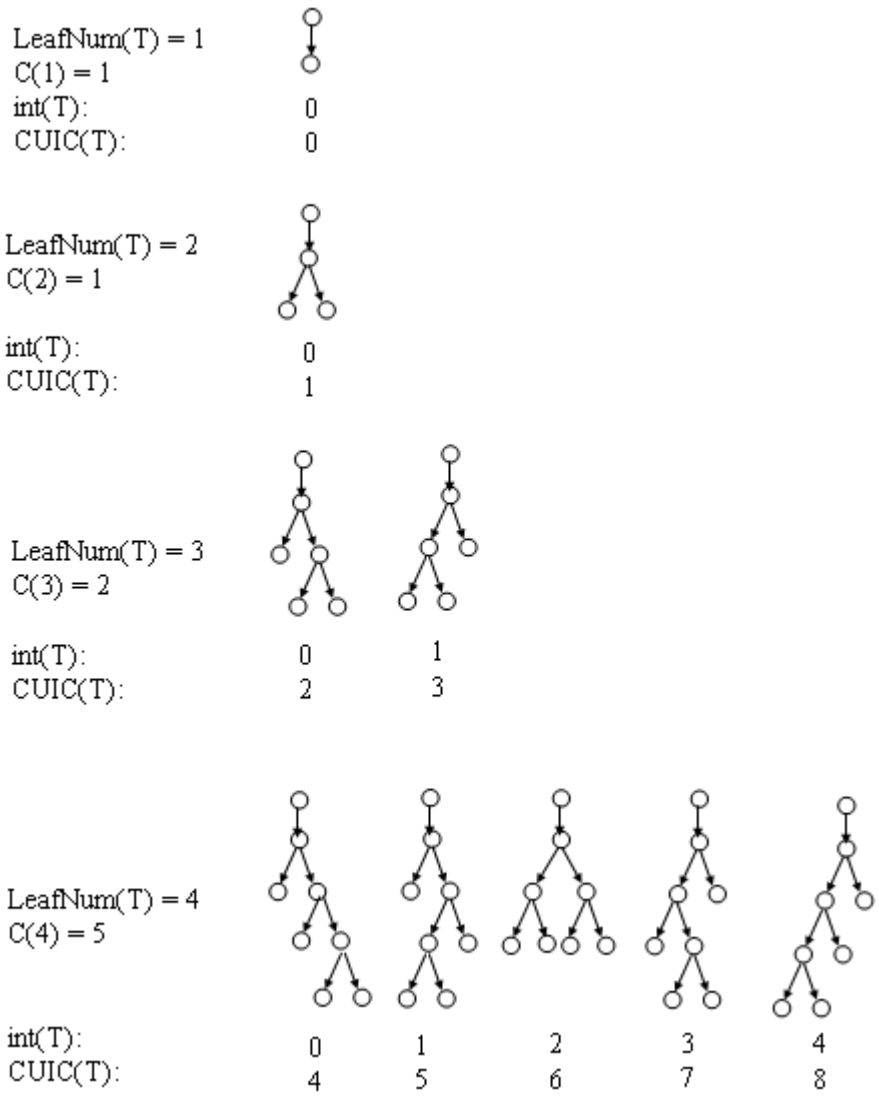


Figure 3-12 The CUIC decoding function CUIC(T) for all PPCTs with 1 to 4 leaves

3.4.5 Searching for Constant Substructures in Dynamic structures

The idea of the constant encoding algorithm is to encode constants into a constant tree, and to retrieve them during the execution of the program from a constant tree by using corresponding decoding functions. In the previous section, we introduced several methods for converting a constant into a constant graph. We now discuss step WF7 and WF8 of the workflow in Section 3.4.1, the processes of finding a constant substructure with a particular

shape inside a constant tree.

Our current method is to conduct an exhaustive search to look for a substructure with a particular shape in a constant tree. Although an exhaustive search may be inefficient, it is not a serious drawback to our algorithm because it happens only in the constant encoding process but not during program runtime. Also we use only small Constant Trees, so that an exhaustive search is not prohibitively expensive during the constant encoding process. If a substructure matching the constant graph can be found in the existing constant tree, the workflow of Figure 3-5 will go directly from WF7 to WF9. However, we cannot guarantee that the search will be successful.

If the constant encoder is unable to find a substructure inside the constant tree *CT* that matches the constant graph for CG_i, for an integer *i*, the method of Figure 3-5 is to follow approach A1 below.

(A1) Add additional nodes to the current constant tree CT, so that a substructure CS_i, which matches CG_i appears in the modified constant tree *CT*.

This is what we used in our prototyping. However, we are aware of other ways solve this problem, such as the two approaches listed below.

(A2) Find a decomposition into integers ($i_1, i_2, \ldots$) that can be encoded in *CT,* such that $f(i_1, i_2, \ldots) = i$, where $f( )$ is a suitable composition function. As a very simple example, we consider 2-way additive compositions of the form $i = i_1 + i$, in which case the composition function $f(i_1, i_2) = i_1 + i_2$. A slightly more complex composition function $f_d( )$ extracts the decimal digits of *i*

$$f_d(i) = \begin{cases} concatenate(i \bmod 10, \quad f_d(\lfloor i/10 \rfloor)), & if\ i \neq 0 \\ 0, & if\ i = 0 \end{cases}$$

where the corresponding composition function $f_d^{-1}( )$, required to decode the decomposed constant, is $f_d^{-1}(i_1, i_2, i_3 \ldots) = i_1 + f_d^{-1}(i_2, i_3 \ldots)$

(A3) Apply a different encoding algorithm *A'*, for example, *CUIC()* instead of *INT()*, to convert the integer *i* into a different constant graph CG_i', where CG_i' matches a substructure of *CT*.

An algorithm for our second approach A2 is relatively complex, but it has some interesting benefits, so we introduce it first. Here is some pseudo code to implement a simple (2-way additive) decomposition algorithm P2 for approach A2.

```
// Algorithm P2 for Approach A2, searching a substructure in CT for integer i,
//when f(i1, i2) = i1 + i2 is used as the combining function, and i = i1 + i2
Vector search(int i){
    //Return a Vector of two substructures = (CSj, CSk),
    //such that int(CSj) + int(CSk) =i
    int j;
    for(j=1; j<i/2; j++){
            Substructure CSj, CSk;
            Vector v = new Vector();



            CSj = exhaustiveSearch(j);
            //function exhaustiveSearch(j) searches exhaustively within CT for
            //a substructure CSj, such that int(CSj) = j, (this is actually the function
            //of step WF7 in Figure 3-5), and return the substructure that has been found.
            // if search was unsuccessful, return null.
            if (CSj != null) v.addElement(CSj);
            else continue;
            CSk =exhaustiveSearch(i-j);
            if (CSk != null) v.addElement(CSk);
            else continue;
            return v;
    }
    return null; //search() was unsuccessful
}
```

By slightly changing the algorithm P2, we can use other combining functions such as

$f(j,k) = j \times k$, so that *search(i)* looks for CS_j and CS_k, such that $int(CS_j) \times int(CS_k) = i$.

We now develop a specific algorithm E1 for approach A1 to step WF8 of Figure 3-5. Approach A1 also uses *search( )* for an exhaustive search. This step is entered only if step WF7, implemented as *exhaustiveSearch( )*, fails to find a matching substructure in *CT*. Our algorithm El randomly chooses a node in the constant tree, and attaches a subtree isomorphic to CG_i at this point.

Approach E1 clearly guarantees that the modified CT will have a sub-graph encoding. However, it has an obvious drawback. If too many large constants are encoded, the modified CT becomes so large that it will impose unacceptable runtime and space costs on the watermarked code. To prevent this drawback, we should not try to encode large constants. In our prototype implementation, for simplicity, we have only used algorithm E1 to encode integers. We did not test any mechanisms for avoiding large integers.

An ideal encoder would have many searching algorithms. If one algorithm fails, it will try another, unless it is running short of computational time or resources. Even if all the algorithms fail, our JSafeMark system will inform the DGW system that this integer is not encoded. This kind of process flow will end up with a variety of decoding functions being generated, which is arguably good, from the point of view of stealth – the attacker will have additional difficulty recognizing our constant – decoding functions.

3.4.6 Generating the Decoding Method

So far, we have introduced methods for building constant tree (WF2 in Figure 3-5) and constant graphs (WF3, WF5 in Figure 3-5). We have also introduced several ways to find substructures for a given constant inside a constant tree (WF7 in Figure 3-5), and to modify the constant tree so that it contains appropriate substructures (WF8 in Figure 3-5). During this discussion, we have ignored steps WF4, WF6, WF9 and WF10, which are used to generate decoding functions source code for the corresponding steps. However, some of the decoding algorithms have already been introduced in our discussion of the encoding functions. In this section, we will consider the problem of generating source code for the decoding functions to be inserted into the watermarked programs, for the retrieval of constants at runtime.

Following the workflow we introduced in Section 3.4.1, we identify the following steps in generating this source code.

(WF4) Generate source code for converting integer(s) into the value of a constant.

(WF6) Generate source code for decoding a substructure into an integer.

(WF9) Generate source code for referencing the substructure at runtime. This can be done in two steps.

(WF9-a) Generate a descriptor for a substructure *CS*, which includes the information about the root of the substructure *CS* in the constant tree and information on the boundary of this substructure.

(WF9-b) Generate source code for the decoding function based on the descriptor.

(WF10) Combine the above source code for all of the above.

We now consider each of these steps in detail.

1) Generate source code for converting integer(s) into the value of a constant. (WF4)

If the constant is an integer that can be directly encoded into a graph structure, in which case the step WF3 in Figure 3-5 was not performed, then this step will not be performed as well. However, when the constant has been processed before encoding in WF3, such as being split into two integers as discussed in Section 3.4.4 in page 48, this step is needed to reverse the process in WF3.

This step is relatively simple. However, each type of constant to integer conversion will need different process to reverse it. For example, the pseudo code for the process the algorithm introduced in Section 3.4.4, page 48 can be written as follows.

```
public double convertTwoDigitsToDouble(int firstInt, int secondInt){
    if ((firstInt < 0 || firstInt>9)||(secondInt<0 || secondInt >9)) throw exception;
    return firstInt + secondInt * 0.1;
}
```

2) Generate source code for decoding a substructure into an integer. (WF6)

The method declaration for decoding integer *i* out of a structure *CS* can be as follows:

```
public int decodingMethod (Node CS){}
```

The source code for different decoding functions is different. We must provide the encoder with the source code for each decoding function it uses, so that it can issue appropriate code in step WF6.

The source code for decoding an integer from a PPCT tree using CLUC encoding is listed in Appendix C, and the one using CLOC conversion is shown in Appendix A.

3) Generate a descriptor for a substructure CS_i, which includes the information about the root of the substructure CS_i in the constant tree and information on the boundary of this substructure. (WF9-a)

Suppose there is a constant tree *CT* rooted at R_{CT} (see Figure 3-13(1)). At runtime, the watermarked program needs to retrieve an integer value *i* out of a constant substructure CS_i rooted at node R_{CS} in *CT* (see Figure 3-13(2)). The descriptor of CS_i must contain the following information.

a) The position of the root node R_{CS} in the constant tree.

b) The boundary of *CS,* because the substructure CS_i may be only part of the subtree *CS'* rooted at R_{CS}. (See Figure 3-13(3))

Now we discuss a specific way to find R_{CS} at runtime. We used this method in our prototyping.

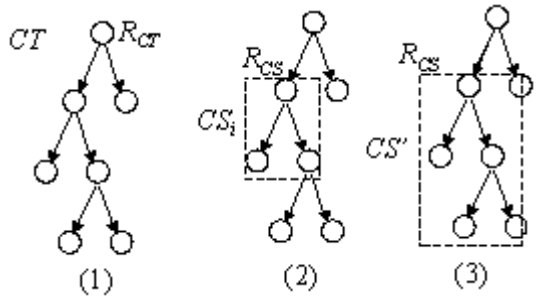


Figure 3-13 Referencing a substructure at runtime

The root node position of *CS* is defined by three parameters. 1. a node reference to the root of the constant tree, 2. An integer *depth*, indicating how far R_{CS} is from the root of the constant tree R_{CT}, and 3. An integer *path*, whose binary representation indicates the directions taken, when navigating the tree from R_{CT} to R_{CS}. See Figure 3-14.

As indicated in Figure 3-14, integer *path* defines the path from the root of the constant tree to the root of the substructure to be retrieved. For example, if *path* = 5, the least significant eight bits of this integer are 00000101. We interpret this as a bit-string, starting from its least-significant bit. Since PPCT structure is a binary tree, each non-leaf node has two children. A bit-value of 0 indicates that the path goes through the left child; a bit-balue of 1 indicates that the path goes to the right child. Starting from the least significant bit of a bit stream, the bit stream 0101 defines the path from root node going right – left – right – left.

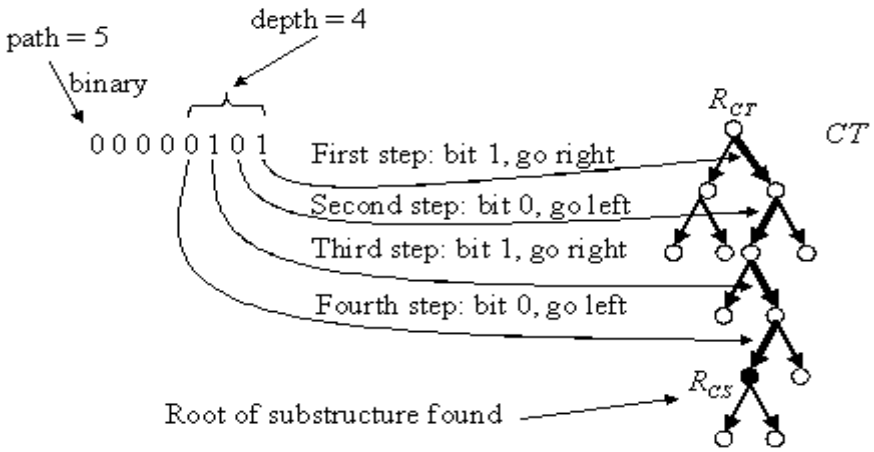


Figure 3-14 Finding the root of substructure by path and depth

Integer *depth* defines the length of the path. That is, how many steps to go. For example, if *depth* = 4, only the least-significant 4 bits of in the integer will be used to define the path from R_{CT} to R_{CS}.

The second part of step WF9 is to define the boundary of a substructure.

In our prototyping, we use a masking technique, in which an PPCT structure *mask* is used. The substructure CS_i is the intersection of the shapes of the mask tree and the subtree rooted at R_{CS}. See Figure 3-15 for an explanation.

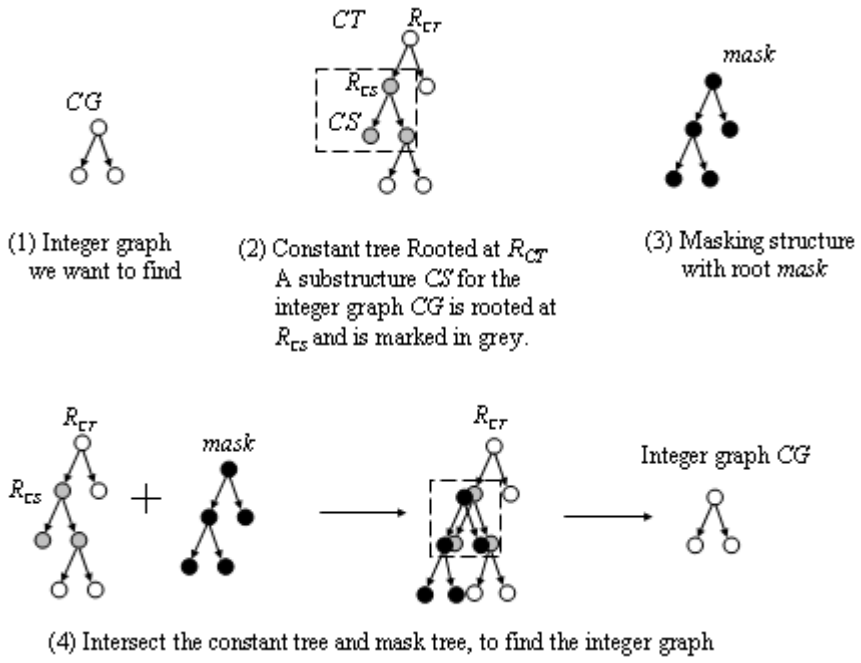


Figure 3-15 Masking technology illustration

The structure *mask* here is obtained in the following way. Suppose an integer *i* is converted into a constant graph *CG*, and a substructure *CS* inside the constant tree *CT* has been found that matches the shape of *CG*. The pseudo code for generating the *mask* structure can be as follows:

```
public Node getMask(Node CT, Node CG, Node CS){
    Node mask = local copy of CG; //construct an initial mask tree that is the same as CG
    for each leaf node leaf_n in mask {
        find the corresponding node leaf_n' in its matching substructure CS in CT;
        if leaf_n' is a leaf node of CT { // leaf_n' may not be a leaf node in CT
            add two children to leaf_n        // expand mask
        }
    }
    return mask;
}
```

After the above process, the information for referencing the substructure in a *CT* is obtained. Thus, the descriptor of *CS* includes the following information:

Integer *depth*;

Integer *path*;

PPCT structure *mask* (shown as a reference to the origin of the PPCT structure).

4) Generate source code for the decoding function based on the descriptor. (WF9-b)

From step WF9-a, we have the information needed to reference the substructure *CS* at runtime. We also need to generate source code so that the watermarked program can reference to the substructure at runtime.

According to the information in the descriptor, which includes two integers *depth*, *path* and a PPCT structure *mask*, the constant substructure *CS* can be obtained by the following method call:

Node CS = getSubstructure(R_{CT}, *path, depth, mask*);

To generate the source code for the above method, we first need to generate the source

code for getting the root of the substructure R_{CS} from R_{CT}, *path*, and *depth*. This source code can be as follows:

```
public Node getSubstructureRoot(Node R_CT, int path, int depth){
    Node result = R_CT;
    int temp = path; // create a local copy of integer path
    for(int i = 0; i<depth; i++){
        int least = temp&1; //get the least significant bit
        if (least == 1) result = result.right; //goto the right child
        else result = result.left; //goto the left child
        temp = temp>>1; //shift right
    }
    return result;
}
```

Now we have the root of the substructure *CS*. Next, we need to generate the source code for getting the substructure *CS* according to the boundary information stored in the *mask* tree. The following is the pseudo code for *getCS( )* method, which return a standalone PPCT structure *CS* (called *tempCS*) rather than only find the original *CS* in *CT*, so that *tempCS* can be used for integer retrieval.

```
public Node getCS(Node R_CS, Node mask){
    Node tempCS = copy of mask;        // create a local copy of mask to be the
                                       // initial structure of CS
    Node current = tempCS.right;//create a reference to the root of tempCS.
                                // because node tempCS is the
                                //origin of the PPCT.
    checkNode(R_CS, current);  // is a recursive method, explained below
    return tempCS;        // in chekNode(), tempCS is modified to be
                          //isomorphic to the substructure CS in CT
```

```
}
```

The above method, *checkNode( )*, recursively goes through all the nodes in the *tempCS* structure. It compares each node in the *tempCS* tree with the corresponding node in the *CS* substructure. By analyzing the structures in Figure 3-15(4) in page 63, we understand that, for a pair of corresponding nodes in mask tree and *CS*, if any of these two nodes is leaf, then that node is the leaf of the *CS* substructure. According to this analysis, we write the *checkNode( )* method as follows. For ease of understanding, we write it in pseudo code format.

```
public void checkNode(Node R_CS, Node current)
    if(R_CS is NOT leaf && current is NOT leaf){
        checkNode(R_CS.right, current.right); //recursively check the right child
        checkNode(R_CS.left, current.left); //recursively check the left child
    }else if (R_CS is leaf){
        cut off the child of current      //when R_CS is leaf, current node should
                                          //be corresponding to the leaf of the CS.
    }
    // when current is leaf, do not need to do anything
}
```

After the above analysis, the getSubstructure(R_{CT}, *path, depth, mask*) can be generated as follows.

```
public Node getSubstructure(Node R_CT, int path, int depth, Node mask){
    Node R_CS = getSubstructureRoot(R_CT.right, path, depth);
    return getCS (R_CS, mask);
}
```

5) putting source code together (WF10)

We need a final step to put all the generated source codes together, so as to form a series of method calls that can be inserted into the watermarked program.

A method call, which can be used to replace a constant, can be written as follows:

```
int const = getConstant(R_CT, path1, depth1, mask1, path2, depth2, mask2);
```

And the decoding method source code can be as follows:

```
public double getConstant(Node R_CT, int path1, int depth1, Node mask1, int path2, int
                depth2, Node mask2){
    int firstInt, secondInt;
    Node CS1 = getSubstructure(R_CT, path1, depth1, mask1);
    Node CS2 = getSubstructure(R_CT, path2, depth2, mask2);
    firstInt = decodeMethod(CS1);
    secondInt = decodeMethod(CS2);
    return convertTwoDigitsToDouble(firstInt, secondInt);
}
public Node getSubstructure(Node R_CT, int path, int depth, Node mask){
    Node R_CS = getSubstructureRoot(R_CT, path, depth);
    return getCS(R_CS, mask);
}
public Node getSubstructureRoot(Node R, int path, int depth){/*Please see page 65 */}
public Node getCS(Node R_CS, Node mask){/*Please see page 65*/}
public int decodeMethod(Node CS){/*Discussed in page 61*/}
public double convertTwoDigitsToDouble(int firstInt, int secondInt){
    /*Please see page 61*/
}
```

All the processes described in this section are illustrated in Section 7.2 with a sample encoding of the double constant 1.4, and the source code generated in the testing is listed in Appendix C. Please note, in Section 7.2, the PPCT structure, which our JSafeMark system exports to the DGW system, is in the format of a two dimensional array. This is the consideration of independency, and the JSafeMark expects the DGW system to turn the array representation of a PPCT into a tree structure. This will be further discussed in Section 4.3.

3.4.7 Process of Modifying Source Code by the DGW System

In the preceding sections, we have discussed steps WF1 through WF10 of Figure 3-5. We now discuss the final step, WF11, in which JSafeMark outputs the following information to the DGW system.

(1) The constant tree structure.

(2) The source code for decoding each constant from the constant tree. This code includes a method call, which can be used to replace a constant, and the source code for decoding method as described in the previous section.

In the third phase of constant encoding, described in Section 3.4.1 in page 40, the DGW system modifies the sources code of the program and makes sure that constant tree is built before any decoding function is called This phase is outside the scope of our research.

3.5 Discussion

Now that we have illustrated the whole process of constant encoding, we are able to discuss some important aspects of this algorithm. These discussions will cover the feasibility of applying this algorithm, and some other aspects.

3.5.1 Possibility of finding constants in Programs

Since our algorithm protects watermark by encoding constants, the applicability of our algorithm heavily depends on the likelihood of finding constants in a program. Thus we need to find out how many constants can be found in a typical program.

The testing is accomplished by analyzing constant accessing operations in Java programs class files.

In Java, the constant pool is used for storing several kinds of constants for a class or an interface. Numeric constants and constant references to objects are accessed via the constant pool, for example, constants of type int, long, float, double and references to an instance of a String object. The bytecode operations that access constant pool include ldc, ldc_w, ldc2_w, bipush, sipush, iconst_<i>, fconst_<f>, and dconst_<d>. Thus, by analyzing the constant pool accessing operations, we can roughly know the constants contained in a class file.

We choose some software to analyze the frequency of constant accesses. The programs we tested, and our results, are displayed in Table 3-1. These programs, which were randomly

chosen by Palsberg and recorded in his paper [Palsberg, Krishnaswamy, et al. 2000], were used to test the JavaWiz system. In all cases, we found thousands of constant accesses, strongly suggesting that our constant encoding processes will have plenty of opportunities to protect a watermarked program.

Program Name	Number of Constant References
Javac	7400
Java	4201
javadoc	772
javacup	6192
javawiz/JavaWiz2	8411
javawiz/Hipi	5495
JTB	9926

Table 3-1 constant loading operations found in Java class files

Some obfuscation techniques introduce additional constants (especially integers) in the transformed program. These integers also can be used in constant encoding, and our encoding process may increase the difficulty of deobfuscation, by obscuring the values of these obfuscating constants. However, we note that if an attacker can de-obfuscate the program, these additional constants are no longer essential for program correctness.

3.5.2 Possibility of finding integers in a PPCT

In order to find out how large a PPCT structure should be so that it can be used as a constant tree, and what range of integers is suitable for encoding, we performed the practical analysis described in this section.

Before describing our analysis, we need to define a concept to describe the possibility of finding an integer in a PPCT structure.

Given a collection of constant trees $\mathcal{T}$, each equally likely to be chosen as a constant tree *CT*, and given a fixed encoding function such as *CUIC()* decoding function defined in 56, we define *f(i)* as the probability of the existence of at least one constant substructure CS_i

encoding an integer *i*, in a randomly chosen $CT \in \mathcal{T}$ as follows.

$f(i) = Prob[\exists CS \ \ a\ substructure\ of\ CT : \ \ CUIC(i) = CS]$, where CT is chosen uniformly at random from $\mathcal{T}$.

Because *f(i)* is a probability, $0\% \le f(i) \le 100\%$.

We are interested in knowing the likelihood of being able to encode a constant integer *i* in a substructure that appears in a randomly chosen constant graph with a fixed number m of leaves. Thus, we introduce another parameter to our probability function.

We write $\mathcal{T}^{(m)}$ for the set of an PPCT with exactly *m* leaves.

We write $f^{(m)}{}_{(i)}$ to be the value of *f(i)*, when $\mathcal{T} = \mathcal{T}^{(m)}$.

We conduct the following Monte Carlo experiment to estimate $f^{(m)}{}_{(i)}$ for small values of *m* and *i*.

Our experiment consists of choosing an *n*-vector of samples $\vec{T}_i^{(m)}$ from $\mathcal{T}^{(m)}$ with uniform probability:

$$\vec{\mathcal{T}}^{(m)} = < T_1^{(m)}, T_2^{(m)}, \ldots \quad T_n^{(m)} >$$

Then we form our estimate $\hat{f}^{(m)}{}_{(i)}$ for $f^{(m)}{}_{(i)}$ by computing

$$\hat{f}^{(m)}(i) \equiv \frac{\sum_{1 \le j \le n} \delta[\exists CS \ a\ substructure\ of\ T_j^{(m)} : CUIC(i) = CS]}{n}$$

where the delta function $\delta(\ )$ defined in the usual manner as

$$\delta(x) = \begin{cases} 1, & if\ x\ is\ true \\ 0, & otherwise \end{cases}$$

We examined PPCT structures with $m \le 300$ leaves, and *i* in the range from 0 to 1000. We formed our *n*-vector $\vec{\mathcal{T}}^{(m)}$ of PPCTs by choosing n random numbers (r_1, r_2, ... r_n) uniformly in the range from 0 to *c(m)* – 1, where c( ) is the Catalan function defined in Section 3.4.4 in page 49. We encoded this number into PPCTs using the Catalan Leaf Oriented Conversion decoder $CLOC^{-1}(r_j)$ to obtain $T_j^{(m)}$

The specific values of *m* we tested were *m* = 25, 50, 150, 250. These trees have 50 nodes

(in the case that m = 25) up to 500 nodes (in the case that m = 250). In each case we formed our estimate $\hat{f}^{(m)}{}_{(i)}$ on the basis of n = 100randomly-chosen trees.

In Appendix B, we tabulate our estimate $\hat{f}^{(m)}{}_{(i)}$ for selected i in the range from 0 to 1000. The values of m are 25, 50, 150, 250. To conserve space, the table of Appendix B covers all i in the range o to 1000, but only i = 0 mode 5 for i between 101 and 200; only i = 0 mode 10 for i between 201 and 500; an only i = 0 mode 50 for i between 501 and 1000. In Appendix B and in the following figures, we report our estimated probabilities $\hat{f}^{(m)}{}_{(i)}$ as percentages 100* $\hat{f}^{(m)}{}_{(i)}$.

Please have a look at Figure 3-16. This is our estimates probability of encoding integers, ranging from 0 to 256, in the random 50-node PPCTs (such trees have m = 25 leaves). We conclude that all integers smaller than 8 are almost certain to be found in a randomly-chose 50-node tree. Integers in the range 9 to 22 have about a 95% possibility of appearing.

Figure 3-17 shows our result for 100 node trees (m = 50). In this case, integers ranging from 0 to 22 have a probability of about 100% of appearing. Integers in the range of 23 to 66 have a probability of at least 90% of appearing. Integers up to 196 have probabilities above 70%

Figure 3-18 shows our result for 200 node trees (m = 100). Here we see that integers in the range of 0 to 61, have a probability of about 100% of appearing.

Figure 3-19 shows that in a 300 node trees (m = 150). Integers, in the range of 0 to 180, have a probability of nearly 100% of appearing.

Finally, in Figure 3-20, in the 500 node trees (m = 250), we find that all of the integers from 0 to 200 have a probability of about 100% of appearing. Referring to appendix B, we see that most of the integers from 1 to 600 have a 95% or greater probability of appearing in trees with m = 250 leaves.

After the above analysis, we believe a small PPCT with 100 nodes, is large enough to encode integers less than 60, with better that 90% probability.

A bigger tree with 500 nodes, can encode integers from 0 to 600, with probability $\geq$ 95%.

Different encoding algorithms may give different results for the probabilities. In our Monte Carlo experiments, we used the CUIC decoding to find an integer in a constant tree. However, according to discussion in Section 3.4.4, if we use the Catalan Leaf Oriented

Conversion (CLOC), we will have much greater chance of finding small integers. This is because the CLOC encoding represents an integer by more than one PPCT structure. (Please see Figure 3-9 in page 52 for a reference). By contrast, the CUIC encoding eliminates the duplication of the CLOC encoding. A drawback of using CLOC encoding is that the chance of finding larger integers is decreased.

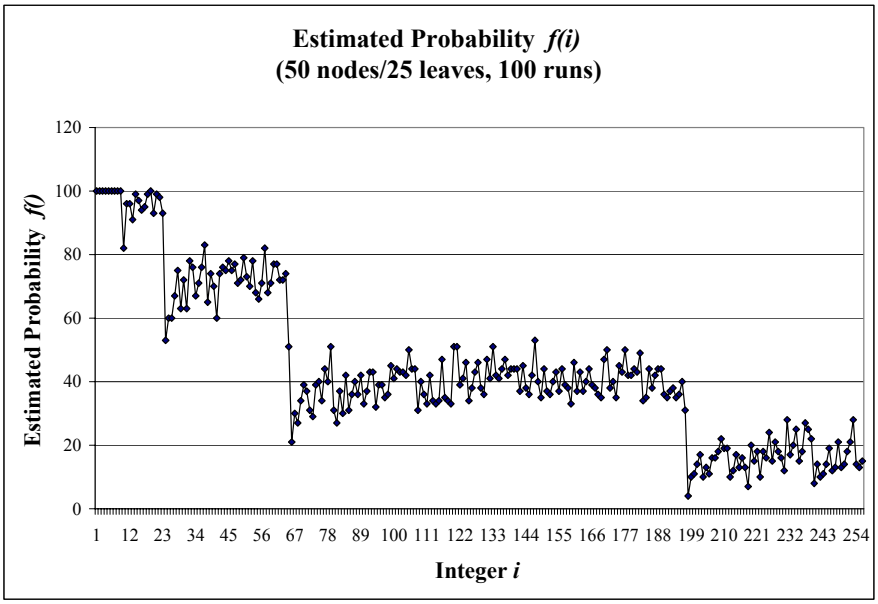


Figure 3-16 Estimated Probability (Expressed as a percentage) that an integer *i* appears in a 50 node (25 leaf) tree.

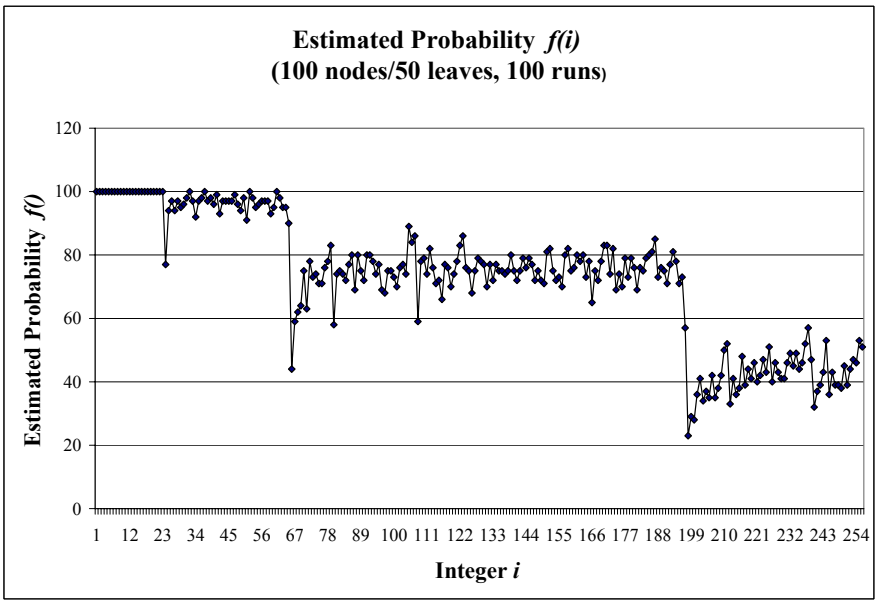


Figure 3-17 Estimated Probability (Expressed as a percentage) that an integer *i* appears in a 100 node (50 leaf) tree.

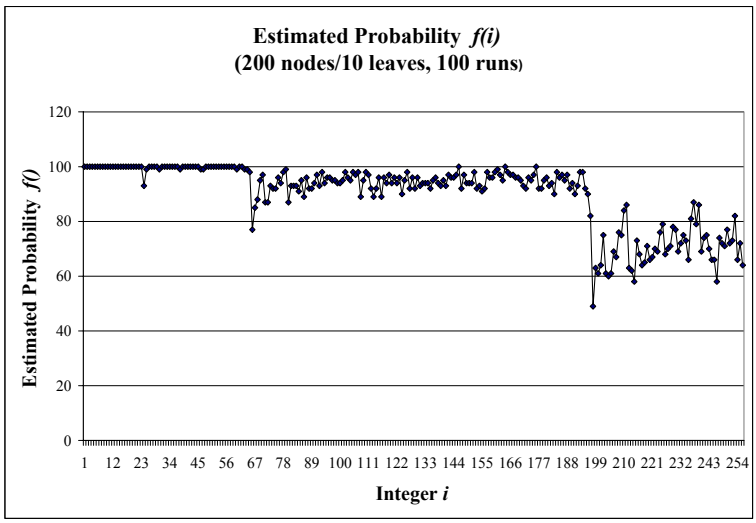


Figure 3-18 Estimated Probability (Expressed as a percentage) that an integer *i* appears in a 200 node (100 leaf) tree.

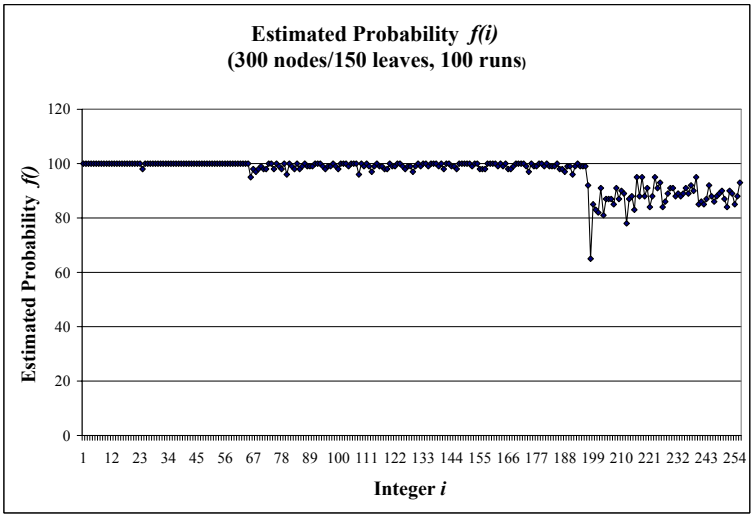


Figure 3-19 Estimated Probability (Expressed as a percentage) that an integer *i* appears in a 300 node (150 leaf) tree.

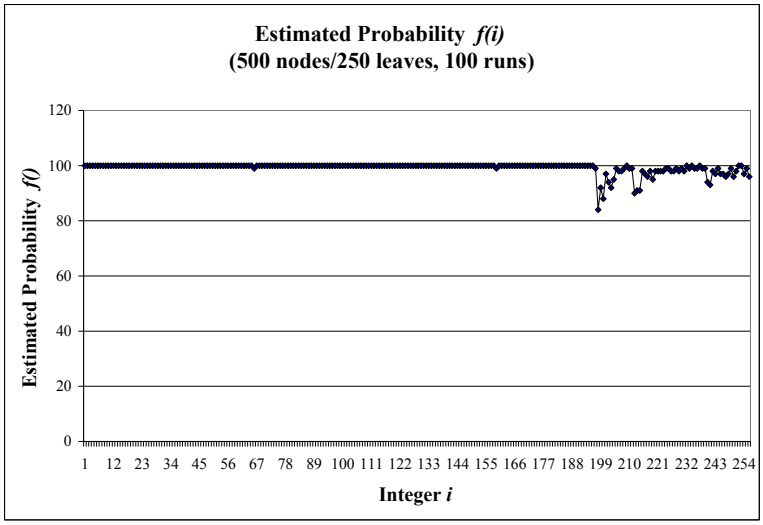


Figure 3-20 Estimated Probability (Expressed as a percentage) that an integer *i* appears in a 500 node (250 leaf) tree.

3.6 Summary

In this chapter, we have discussed the constant encoding algorithm. We discussed the process of encoding in detail together with some algorithms we used in prototyping. In Section 3.5.1, we presented experimental results indicating that typical Java classes load many constants. In Section 3.5.2 of this chapter, we presented experimental results demonstrating that small integers are very likely to be found in a randomly chosen, relatively small PPCT. In the following chapters, we will focus on the prototyping of the algorithms we discussed in this chapter.

Chapter 4

Overview of Our Prototyping

4.1 Introduction

Our JSafeMark encoder is the prototyping of our constant encoding algorithm. It is designed and implemented based on the issues discussed in the previous chapter. In this chapter, we will first give a brief introduction on our prototyping, showing the system structure of our prototyped system, which includes three parts: a JSafeMark tester for simulating a DGW system, a PPCT Codec library, and most importantly, our JSafeMark encoder, then we will introduce how the information is exchanged between the different parts of the system. In the next chapter, we will briefly introduce the design of the JSafeMark tester and the codec library. After that, in Chapter 6, we will concentrate on the design and implementation of our JSafeMark encoder.

4.2 System Structure

As we discussed in Section 3.3, the JSafeMark encoder works as a plug in module for a DGW watermark system. When designing the encoder, we try to make it work independently, which means the watermark functionality of the DGW system does not rely on the JSafeMark encoder and the encoder does not rely on the DGW system as well, so that it can be an

optional part of a DGW system. Thus, we adopt the encapsulation concept in the component-based software engineering [Brown 2000] and make the JSafeMark encoder a self-contained unit with its own functionality. It works as an optional part of a DGW system, to add extra obfuscating and tamperproofing functions, so as to protect the watermark structures generated by the DGW system. Meanwhile, we also adopt the interface concept in the component-based design for exchanging information among different part of our system.

The structure of a normal component includes three aspects: Export Interface, Component Body, and Import Interface [Mann and Borusan 2001]. The structure of our JSafeMark encoder is based on this scheme. It offers export and import interfaces for information exchanging. The JSafeMark encoder is connected to the DGW watermarking system through these interfaces and offers a constant encoding service.

The Import Interface defines the service the encoder requires from its external environment; for example, how the external DGW watermarking system can pass a series of constants into the JSafeMark encoder.

The Export Interface defines the service that the encoder offers to its external environment. These services include exporting the structure of a constant tree and a series of decoding functions.

The Encoder Body (shown as "JSafeMark" in Figure 4-1) is the core section of the JSafeMark encoder. It contains all the control and encoding/decoding logic for encoding the constants, and manages the interaction between the encoder and its external world.

The JSafeMark encoder structure, together with the whole constant encoding system structure, is expected as in Figure 4-1.

Figure 4-1 shows that in an expected implementation, the entire watermarking system, including constant encoding, has split into three parts. The DGW system and JSafeMark works individually. The third part is called Codec Library, which is a collection of codecs, offering the functionality of integer encoding and graph structure decoding.

In fact, the services, provided by codec library, for the JSafeMark and the DGW system are slightly different, because the encoder requires source code of encoding and decoding functions, which is not needed by the DGW system. The different services for the JSafeMark and the DGW system are obtained by calling different services in the interface of the codec library. By building the codec library as a separate part, the implementation can be simplified because we do not need to duplicate a codec library in the JSafeMark and the

DGW system.

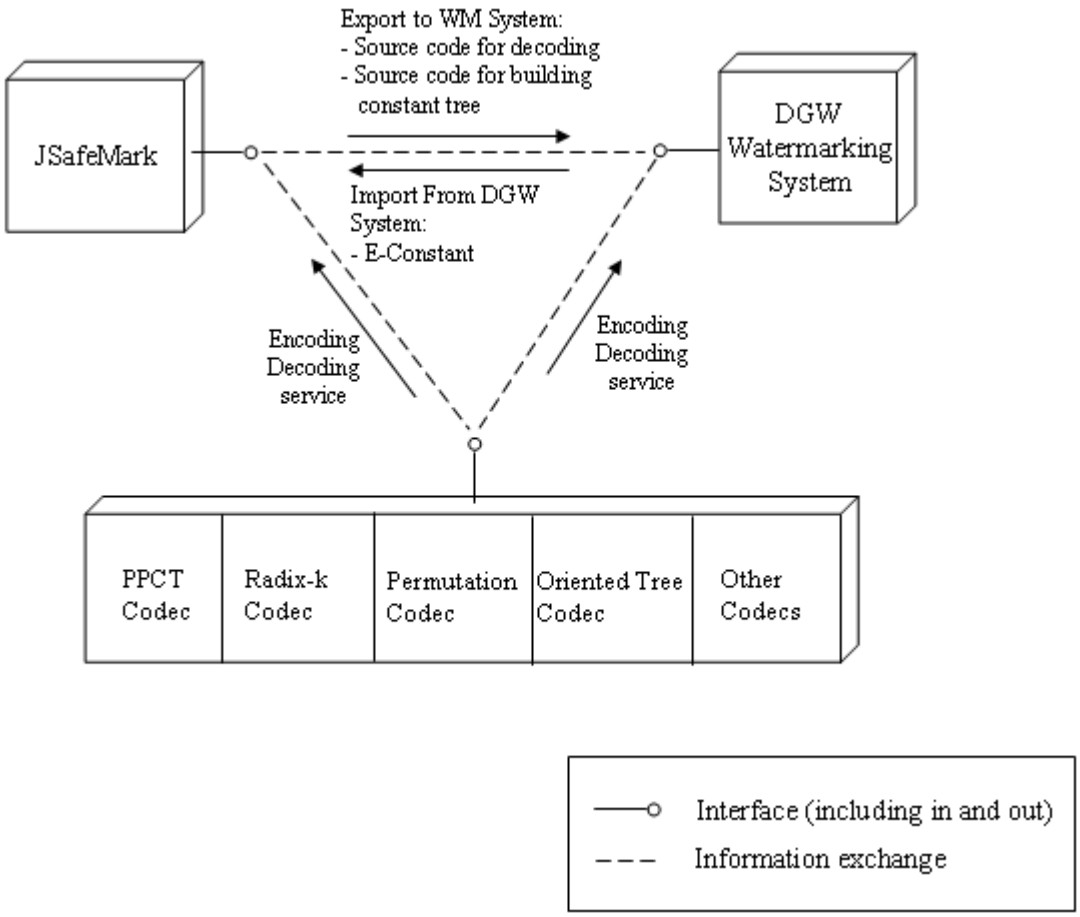


Figure 4-1 Overview of an expected system structure for the JSafeMark encoder.

So far we have discussed the expected system structure of a JSafeMark encoder and the DGW system, now we show how this is implemented in our prototyping.

In our prototyping, the JSafeMark encoder is built as described above. A DGW system simulator, called JSafeMark tester, which will be discussed in Chapter 5, is built to simulate a DGW watermarking system. The codec library is built within the tester. However, from the point of view of the JSafeMark encoder, the codec library is still a third party service. Both the JSafeMark encoder and the codec library handle only PPCT structures. Please see the structure in Figure 4-2.

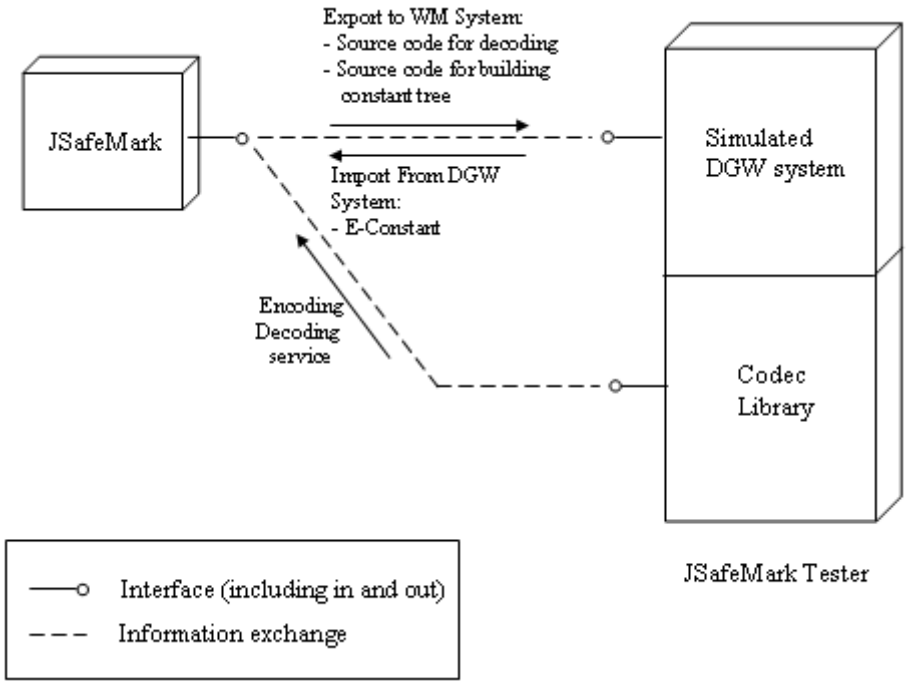


Figure 4-2 Implemented system structure in prototyping.

4.3 Information Exchanging

We introduced JSafeMark encoder structure in Section 4.2. We mentioned that the JSafeMark have two interfaces to interact and communicate with other components: the import interface and the export interface. In this section, we will discuss about the interfaces and we concentrate on the variable definition for the information exchanging between the JSafeMark encoder and other parts of the system. These will be discussed in the following two steps: what to exchange and how to exchange.

(1) What needs to be exchanged

Since the codec library is built in the tester, we ignore the information exchanging between the codec library and the tester. From the discussion in previous chapters, we realized the following operations will happen between the JSafeMark encoder and the DGW system.

(i) A set of E-constants needs to be input into the JSafeMark encoder for constant encoding process. This requires the import interface of the encoder can accept a set of constants.

(ii) The encoder needs to export the constant tree structure.

(iii) The encoder needs to export the source code of the decoding functions and the

reference information $Info_{CS}$, which includes *path*, *depth* and *mask* structure as discussed in Section 3.4.6, for each constant.

And the following operations will happen between the JSafeMark encoder and the codec library.

(i) The integer-to-PPCT encoding is done in the codec library. This requires the codec accept an integer and export a PPCT structure to the encoder[1]. The codec needs to return the source code for decoding method to the encoder.

(ii) There are many encoding methods to choose. This is done in a random manner except pre-selected by the user.

Considering the above operations, the information exchanged between the encoder, the DGW system and the codec library can be described as in Table 4-1.

From	To	Information
DGW system	JSafeMark	- a set of constants
JSafeMark	DGW system	- constant tree structure - a series of decoding functions and reference information $Info_{CS}$
JSafeMark	Codec	- an integer
Codec	JSafeMark	- a PPCT structure for the integer - decoding function for decoding the PPCT into the imported integer

Table 4-1 Information exchanged for encoding

(2) Data Structures of exchanging information

(i) Wrap everything in a Vector

When the data exchanged is complex, we put them in a Vector. All the information in Table 4-1 are exchanged within a Vector except the third row of the table, which contains only an integer in primitive type.

[1] In order to trace and analyse the system operations in the program debugging time, an encoding/decoding method may be selected for a particular constant by the user. For simplicity, this function is hard-coded in our prototyping.

(ii) Data Structure for exchanging a set of constants from the DGW system to the JSafeMark

The DGW system may import different kinds of constants into the encoder. All these constants are wrapped in a Vector. Since only objects can be added into a Vector in Java [Kamin, Mickunas, et al. 1998], a primitive type constant needs to convert into object type before adding to a Vector. For example, an integer need to convert into an Integer object first (please refer to Figure 3-7 for the data types in Java). It is difficult for the encoder to distinguish between an integer and an Integer object. However, this is not a serious problem. We design our encoder to handle only primitive types when this ambiguity happens. Thus, the DGW system is required to export primitive types, such as int, double, instead of the object type, such as Integer, Double.

(iii)Data Structure for exchanging PPCT

Please consider the following fact. From the DGW Watermarking technique discussed in Chapter 2, we know that a user may choose a class (or a similar data structure) to be the node structure for constructing watermark trees. This class may be chosen from a candidate program (See Section 2.3.3).

Since the constant tree, which is constructed in the encoder, and the watermark tree, which is constructed in the DGW system, must have the same data structure for the tree nodes, we need to have the same data structure in both the encoder and the DGW system. However, if the node class is chosen from the candidate program, the node class is not easy to be imported to the encoder because the encoder is independent from the DGW system. Thus, if an encoder needs to be independent of the DGW system it connected to, it is better not to exchange node class between the DGW system and the encoder.

A better way is to rebuild the structure in the encoder using local available resources, so that DGW watermark systems are free to use any data structure for watermark building.

A possible answer for exchanging the tree structure is to turn a PPCT structure into a two-dimensional array, which contains only the relationships between the tree nodes, but without any node information. For example, a PPCT structure in Figure 4-3 can be turned into an array as follows.

a[][] = {{3,1}, {2,3}, {4,5}, {5,3}, {0,4}, {4,5}}

In this array, the first dimension describes the indexes of the tree nodes, and the second dimension shows which two nodes this node is pointing to. Since each of the PPCT nodes contains two outgoing pointers, a two dimensional array is enough for holding the relationships.

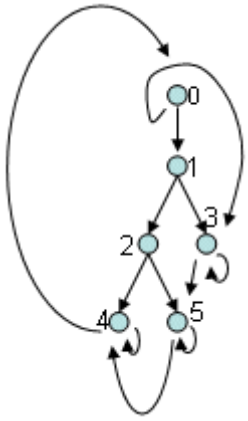


Figure 4-3 Sample tree structure for structure passing.

After this information is passed into the encoder, the encoder will convert it back to a tree structure using local available resources for further manipulation. This technique is used for all structure passing in the constant encoding process including the *mask* structure mentioned above.

Thus, we expand the Table 4-1 to include the data type for information exchanging as follows.

From	To	Information	Type for exchanging
DGW system	JSafeMark	- a series of constants	Vector
JSafeMark	DGW system	- constant tree structure - a series of decoding functions and reference information $Info_{CS}$	Vector
JSafeMark	Codec	- an integer	integer
Codec	JSafeMark	- a PPCT structure for the integer - decoding function for decoding the PPCT into the imported integer	Vector

Figure 4-4 Data Type for information exchanging

4.4 Summary

In this chapter, we discussed the ideal system structure for encoding constants, and showed the system structure in our prototyping. After that, we discussed the interfaces for information exchanging by analyzing the data type among the different parts of the system. In the next two chapters, we start to introduce the design and the implementation of our prototyping.

Chapter 5

Developing the JSafeMark Tester and Codec Library

5.1 Introduction

In the previous chapter, we introduced a high-level system structure of our prototyping. We mentioned that we designed a JSafeMark tester to simulate the functionality of the DGW system. In order to understand fully the workflow of our encoder, we first briefly introduce how the tester is designed and how it works. As indicated in the previous chapter, the codec library is built in the tester. Thus, in this chapter, we will also introduce how the codec is designed,

5.2 Building JSafeMark Tester

A DGW System PPCT simulator is designed mainly for three reasons. First, we need to know how this Encoder can work and communicate with a DGW watermark system. Second, while building and testing the JSafeMark encoder, many PPCT structures need to be fed into the encoder and it is not a wise idea to create them manually. Creating a simulator can help us work efficiently. Finally, we need to understand how a PPCT structure works and how the

structure may change while watermark number changes. This will help us with studying and understanding the PPCT structure, also can help us thinking about how the JSafeMark structure need to be, so that the encoder can be connected to a real DGW watermarking system. Now let us have a look how this simulator is built.

5.2.1 Overview of the JSafeMark Tester

The tester has two main functions: working as a DGW system to exchange information with the encoder; visually displaying tree structure of a given integer. Apart from the above, the tester can automatically increase a watermark number to show how the PPCT structure changes when the integer changes.

In order to communicate with the JSafeMark encoder, the tester includes a function to convert a PPCT structure into a two-dimensional array, which contains the relationships among the nodes as described in Section 4.3. It does not parse the source code of a program. Instead, it accepts some input, which can be used as the information from a candidate program.

The tester is built as a Java swing application, with a JScrollPane to display watermark structures. It is used as a client to request constant encoding service from the JSafeMark encoder.

5.2.2 JSafeMark Tester Structure

A JSafeMark consists of three parts (Figure 5-1): the Controller, the TreeViewer, and the Codec.

The Controller is the entry point for all the functions. It controls which function of the tester will be launched, also controls what information need to exchange with the encoder. The Controller also offers the array-PPCT converting function that builds a PPCT structure from a two dimensional array and also convert a two dimensional array into the corresponding PPCT structure. The array will only hold node indexes and relationships in order to pass into the JSafeMark encoder as discussed in Section 4.3.

The TreeViewer is used to display tree structures. Tree structures will be displayed in a Java Graphic User Interface. The user can set different parameters for convenient viewing. The TreeViewer is not a basic function in the DGW System, but acts as a monitor when a DGW system is simulated, so that the user may get visual information about PPCT structures. The TreeViewer can also be used together with the Codec for studying the PPCT structures.

The Codec is used to convert between an integer and a graph representation of the integer. It is a shared function that offering service to both the encoder and the tester. This will be discussed more fully in Section 5.3.

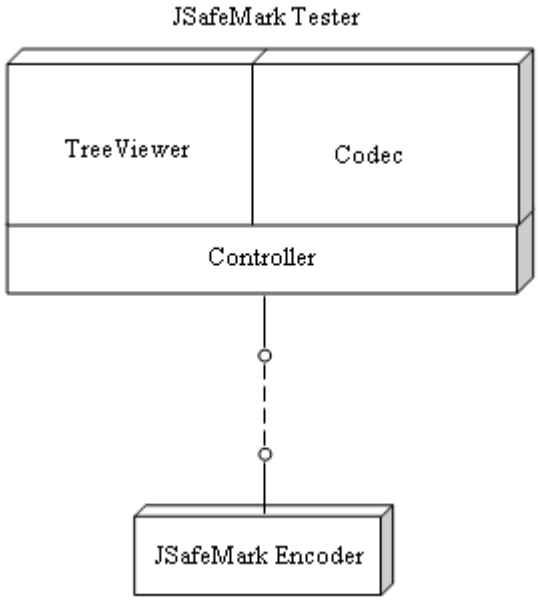


Figure 5-1 The JSafeMark tester structure

5.2.3 Using JSafeMark Tester

The tester is built on a Windows platform. Because it is written in Java, it will not be difficult to migrate it to other platforms.

The tester is started from a command line. It can be invoked by typing the following command at Windows command line window.

java JSTester – option

where option is one of the following:

-s DGW system simulation.

-d display PPCT structures.

With specific parameter settings, the simulator will launch one or the other of the following two different sessions.

(1) DGW System Simulator.

Command: java JSTester -s

This command will run the JSafeMark tester in command line mode. The user will be

prompted to input constants for encoding. The source code for constructing the constant tree and for the decoding functions will be generated by the JSafeMark encoder, and output to a file by the tester. This option will be further described when we demonstrate constant encoding.

(2) PPCT Structure viewer:

Command: java JSTester -d

This command will launch a Java Swing graphic user interface. The user can type in a watermark number, and the watermark structure for that number will be displayed inside the Java GUI panel. The encoding uses the Catalan Leaf-Oriented Conversion (CLOC), which is discussed in Section 3.4.4 in page 49. Meanwhile, every node inside the displayed tree structure will have an integer beside it. This is the watermark number for the subtree rooted at that node using CLOC encoding. These watermark numbers are useful information for understanding distribution of the watermark integers inside a PPCT.

The user can also specify one of the following two ways to display the PPCT structures.

1) Manual Select mode, and

2) Auto Increase mode.

Please see Figure 5-2 for a snapshot of the JSafeMark tester interface, in which a PPCT structure for watermark number 0 with 5 leaves is displayed.

When running in the Manual Select mode, The "*Delay*" option is disabled. The user can type an integer into the "Watermark number to draw" field, and specify the "*Leaf number*". In this mode, the tester encode integers using CLOC encoding. Thus a leaf number is required.

Option "*x-distance*" is used to specify the space between two adjacent leaves and "*y-distance*" is used to specify the vertical space between a node and its parent node, if there is one.

Option "*Normal display*" specifies the tree structure displayed in the panel as shown in Figure 5-2, in which the horizontal space of two adjacent child nodes is slightly smaller than the space between their parent node and the adjacent node beside it. The tree structures in "normal display" have a better look. However, if the watermark number is too big, the displayed structure will take a lot of space. To conserve space, "*Compact*

display" mode can be used, in which the space is decreased.

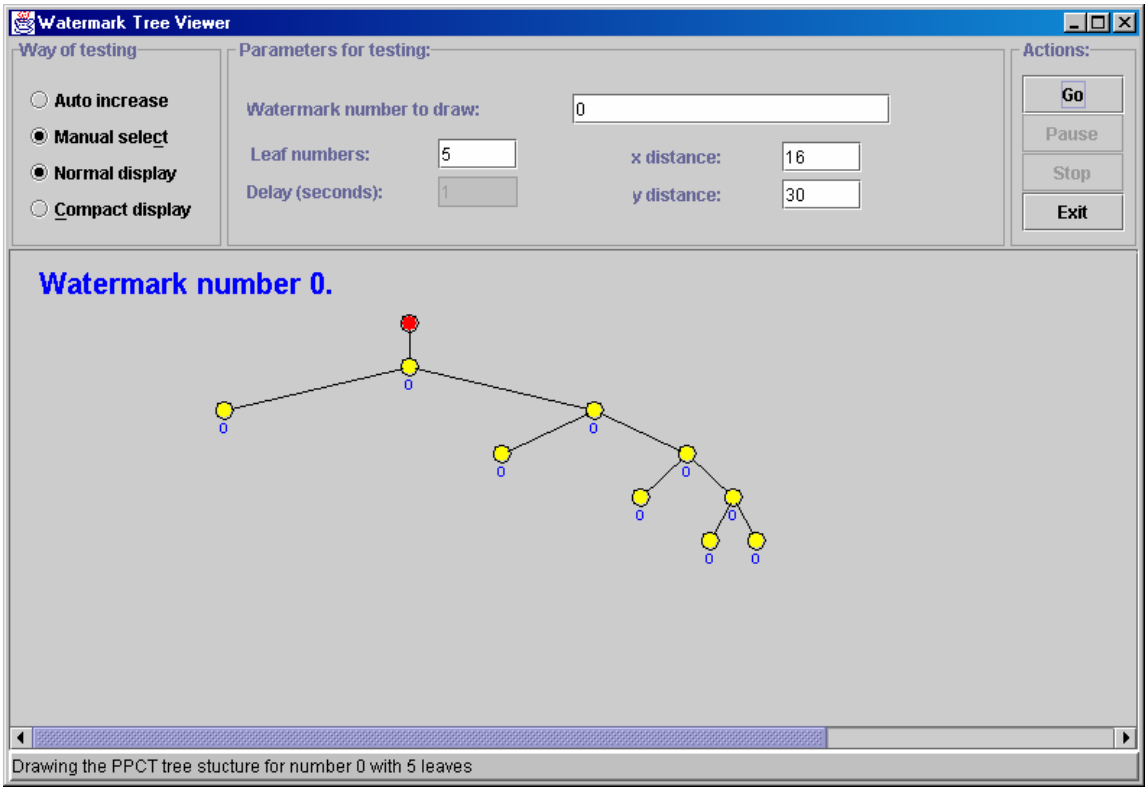


Figure 5-2 The JSafeMark tester running in "manual" and "normal display" mode, showing a PPCT structure for number 0 with 5 leaves.

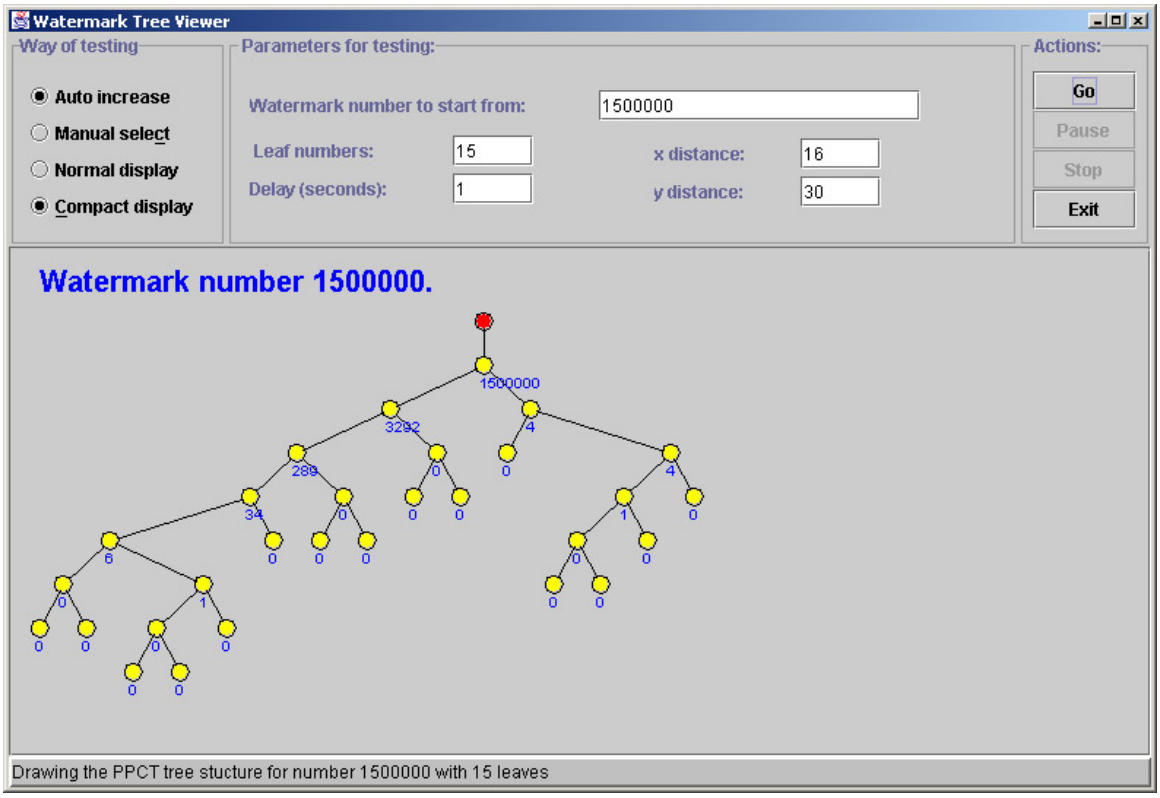


Figure 5-3 The JSafemark tester in "auto increase" and "compact display" mode, showing a tree structure for number 1500000 with 15 leaves.

When running in the "*Auto increase*" mode, the tester will increment the input watermark number automatically. In this mode, the "*Delay*" option is enabled, which can be used to specify how long each tree structure will be displayed. A tree structure for each watermark number will be displayed for the specified period of time until all the possible watermark numbers are displayed. The range of the watermark numbers to be displayed is from the *input-watermark-number* to *c(input-Leaf-Number) – 1*, where *c( )* is the Catalan number introduced in page 49.

5.3 Building the PPCT Codec Library

We have discussed system structure for the JSafeMark encoder in Section 4.2. As the structure shows in Figure 4-1, Codec becomes an individual part of the system, offering service to both the encoder and the DGW system. In fact, a codec contains the function for integer encoding and decoding, which is a very important part for the encoder. Thus, it is worth discussing in our implementation.

As shown in Figure 5-1, a codec library is built in the JSafeMark tester. Each codec in the Codec Library includes two parts: a TreeBuilder and a WMRetriever. The structure of codec is shown in Figure 5-4.

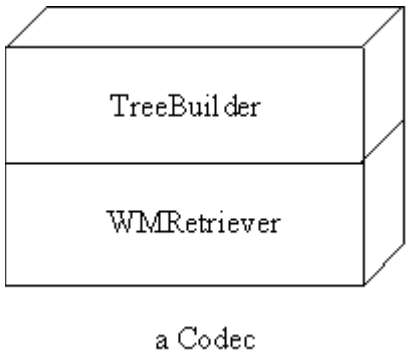


Figure 5-4 Internal structure for a codec

The TreeBuilder is the encoder of the codec, in charge of building PPCT structures for an integer according to a particular encoding algorithm, such as CLOC, CUIC etc. as discussed in Section 3.4.4. In the Java language, an integer can be in the type of *int, short, long* or *BigInteger*. In our implementation, our encoder will only accept *BigIntegers*.

The WMRetriever is used to decode a PPCT structure into its integer representation. The WMRetriever has two functions: *getWmNum* function converts a whole PPCT tree into a single integer. This is normally how a watermarking process works. The other function,

getSubTreeWmNumber function, is to retrieve an integer out of a subtree of a whole PPCT. The latter function is mainly for the purposes of analysis.

As we know, there should be many codecs in a codec library, and there should also be many encoding/decoding methods for each codec, so as to avoid pattern matching attacks. In order to expand the system in the future, organizing these codecs and methods in an appropriate way is needed. Thus, we write a class for each algorithm, and put them into a package called *CodecLibrary*, and all of the classes in the package need to extend an abstract class. This abstract class defines the basic format of the codec classes. It also defines the interface that the codec classes used to interact with the other parts of the system. In the future, if there are more algorithms needed to add into the encoder, we simply need to write a new codec class, which extends the abstract class, and put it in the encoder package. Figure 5-5 describes the structure of the PPCT codec package. It shows 3 PPCT codecs in the structure. All of them extend an abstract class *PPCTCodecLib*.

Apparently, codecs for other data structures can be easily added into the system by specifying new abstract classes such as RadixKCodecLib, which is similar to the PPCTCodecLib shown in Figure 5-5.

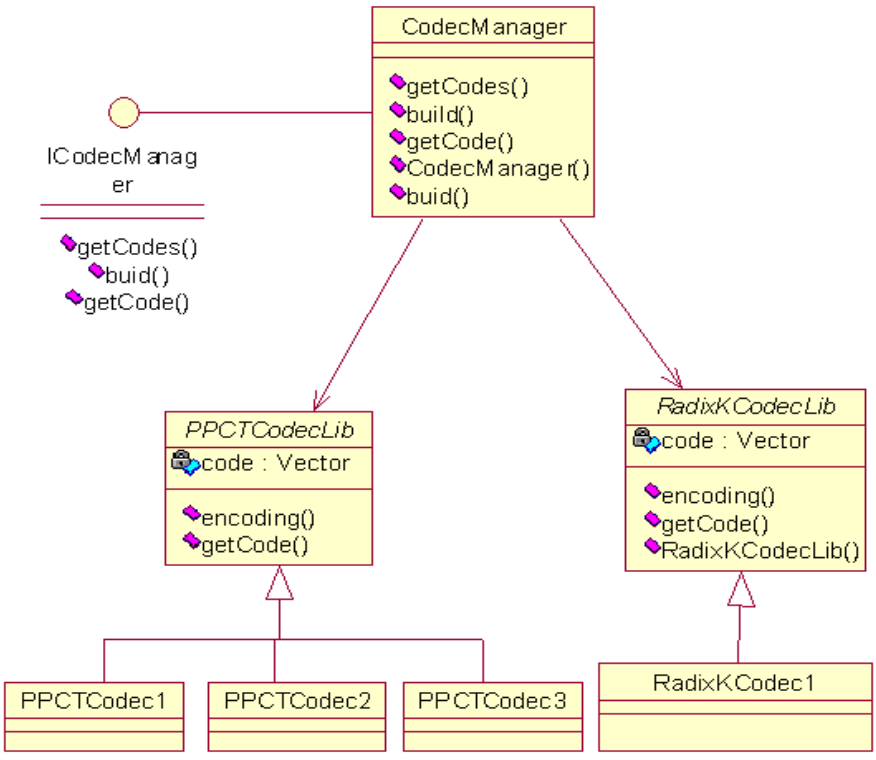


Figure 5-5 PPCT codec class diagram

5.4 Summary

In this chapter, we discussed the structure and the usage of the JSafeMark tester. We also discussed the structure of the codec library and how the codec library is designed in our prototyping. In the next chapter, we start to discuss the design of our JSafeMark encoder.

Chapter 6

Developing a JSafeMark Encoder

6.1 Introduction

In Chapter 4, we introduced the high-level concept of the JSafeMark encoder, including the system structure for the encoder. In Chapter 5, we discussed the structure and the design of the JSafeMark tester and the codec library, which are used to interact with the JSafeMark encoder for encoding constants. Now that we have the overall concept of the system environment, and in this chapter, we will discuss the analysis, design, and implementation issues of our JSafeMark encoder. This will cover the system structure, and the detailed technology used for implementation. The development of the JSafeMark encoder follows the procedure of software engineering [Sommerville 1996], and most of the analysis and design work uses The Unified Modeling Language (UML) [Martin and Kendall 2000]. The CASE tool [Hoffer, George, et al. 1999] used in this chapter is Rational Rose [Rational Software].

Different watermark graph structures need to work with different kinds of JSafeMark encoders. Therefore, there should be different JSafeMark encoders in the encoder library, such as a JSafeMark PPCT Encoder for PPCT constant encoding, a Radix-k Encoder for Radix-k tree constant encoding as we discussed in the previous chapter. Although different Encoders are based on different technologies, the implementation procedures for them are

quite similar. In this chapter, our analysis, design and implementation are based on the JSafeMark PPCT Encoder.

6.2 Analysis

Requirement analysis is the first thing to do when developing a system [Heineman and Councill 2001]. Most of the requirements for the JSafeMark PPCT encoder have been discussed in the previous chapters. Now from the point view of the software engineering, we give a review and outline these requirements as follows.

6.2.1 Functional requirements:

The functional requirement analysis describes the functionality of the system. Since we have fully discussed the functions of the encoder in Chapter 3, we will not repeat them in this chapter, please see Section 3.4 especially 3.4.1 for the tasks that the encoder should perform.

6.2.2 Non-functional Requirements

(1) Human factors: Users of the encoder should be moderately experienced. Configuration knowledge is needed when setting up the system.

(2) Documentation: Required to explain the algorithms of the encoding methods, and explain the operation procedure of the encoding.

(3) Hardware: same as DGW System.

(4) Software: JDK 1.3.

(5) Performance: the performance of the encoding process is not critical. However, the generated source code cannot noticeably slow down the performance of the candidate program.

(6) Error Handling: An Error log has to be generated for any error occurred.

6.2.3 Use Case View: Requirement Description

Conceptual static structure of a JSafeMark encoder is described by the UML Use Case Diagrams. The use case diagram is a high-level description about the activities happened within the encoder and the interactions between the encoder and the external systems.

The most important function the JSafeMark encoder should have is encoding constants

into constant graphs; and for each constant, searching for a substructure, which matches the constant graph for that constant, in a constant tree. Then, it should generate a decoding function for the process of retrieving constant, and output the result (that is, the Encode Constant use case). Another important function associating with the Encode Constant use case is that an encoder needs to generate a constant tree at the beginning of the encoding process (the Build Constant Tree use case). The constant tree is built in a pseudo-random manner as described in Section 3.4.4. Besides, the user needs to configure the encoder when the JSafeMark encoder is first connected to the DGW Watermarking System (the Configure Encoder use case[1]). These activities are described in Following Use Case Diagram (Figure 6-1).

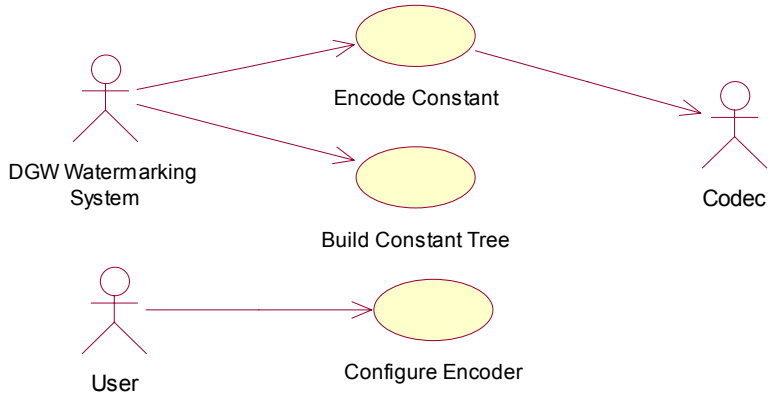


Figure 6-1 Use Case Diagram

Use case description

Use Case Description briefly shows the purposes and high-level overview of the functions of a use case, describes the activities the system needs to perform. A description of the use case Encode Constant is shown below.

Brief description of the Encode Constant use case:

> This use case is started by the DGW watermarking system the JSafeMark PPCT encoder (*JPE*) is connected to. It provides the capability to process a series of constants. For each constant, the *JPE* converts it into a PPCT constant graph, finds a substructure, which matches the constant graph, in a constant tree, and generates a decoding function for the constant that can be used to retrieve the value of the

[1] This function is not implemented in our prototyping.

constant from the constant tree.

Use Case Event Flow

A brief description of the Event Flows for the Encode Constant use case is listed below. An Event Flow describes the interactions that the use case captured between the encoder and the actors (the person or system outside the encoder that interacts with the encoder, in this case, DGW watermarking systems and codec library are the actors) [Martin and Kendall 2000].

Event Flow for the use case "Encoding Constant":

1. Preconditions: the Build Constant Tree use case has been executed, a constant tree already exists.
2. Used by DGW watermarking system.
3. Event Flows
   3.1. DGW System input a set of constants.
   3.2. Repeat until all constants have been processed.
      For each constant, a decoding function is generated, and used to get the value of the constant out of a substructure in a constant tree. If constant input error, goto 3.3.
   3.3. An invalid constant is encountered: error message is displayed showing constant invalid. Goto 3.1

6.2.4 Sequence Diagrams: Conceptual Dynamic Behavior

A sequence diagram describes the dynamic behaviors of a system. It is displayed according to time sequence and shows the interactions among objects. Figure 6-2 shows the sequence diagram for the Encode Constant use case in operation.

Brief explanation of Sequence diagram for Encoding function:

This activity starts from the DGW System calling encodeConst() method in the interface class *IConstantEncoder* in the JSafeMark. *IConstantEncoder* class then passes the call to its implementation class *ConstantEncoderImp*. *ConstantEcoderImp* starts the real process by finding a particular method (calling *selectEncodingMethod()* method), building constant graph for the given constant (calling *buildConstantGraph()* in *GraphManager* class), then

searching for a substructure, which matches the constant graph , in the constant tree structure (*search()* method in *GraphManager* class), finally, calling *generateDecodingCode()* method and get the decoding method source code built. Then the source decoding method can be returned to the DGW system to modify the source code. During the process, if a searching does not find any substructure, which matches the given constant graph, then *GraphManager* will add additional nodes to the constant tree. So that the substructure required can be found inside the constant tree. After the source code for the decoding method is generated, the code will be returned to the DGW system together with the constant tree structure.

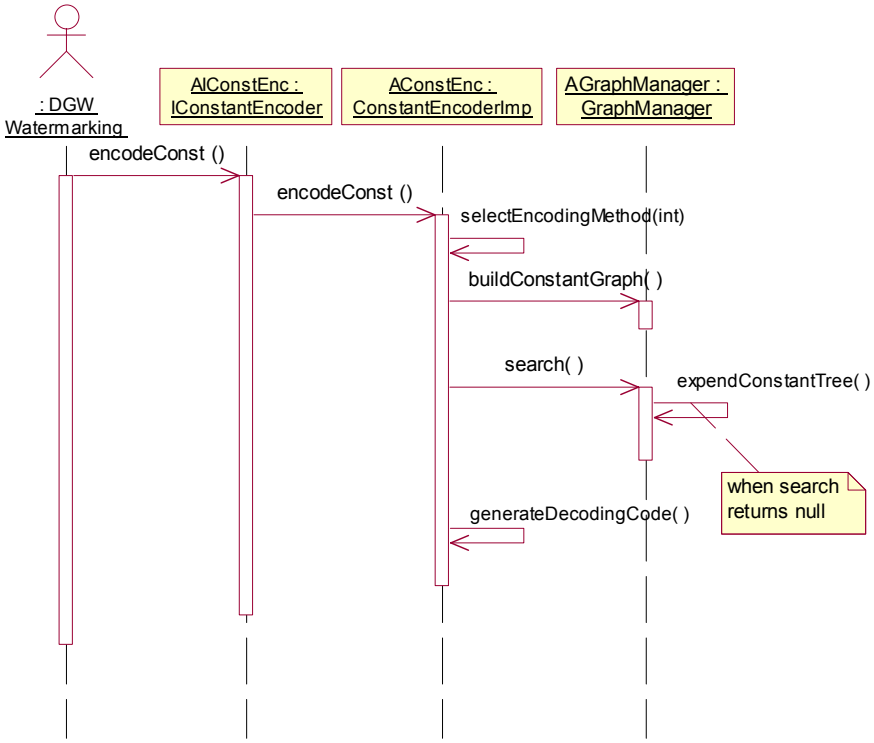


Figure 6-2 Conceptual sequence diagram for the encoding function.

6.2.5 Class Diagrams: Conceptual Static Structure

A class diagram describes the conceptual static structure of the system. It is composed of interrelated classes together with their relationships. The classes need further development in design phase. More classes may need to be added later as well. Figure 6-3 shows the conceptual classes for the JSafeMark encoder.

Brief description of classes:

(1) *IConstantEncoder*. This is the interface class for the JSafeMark encoder. It is in charge of the interaction with the DGW Watermarking System.

(2) *ConstantEncoderImp*. This is the Control class for the JSafeMark encoder controlling the workflow and the activities inside the JSafeMark encoder. It manages the functions of encoding, building PPCTs, and finding existing encoding methods. It is also responsible for generating decoding method, generating and display error message should an error occur.

(3) *GraphManager*. This is used for building the constant tree, building constant graphs, expanding constant tree, as well as searching a PPCT substructure inside the PPCT constant tree.

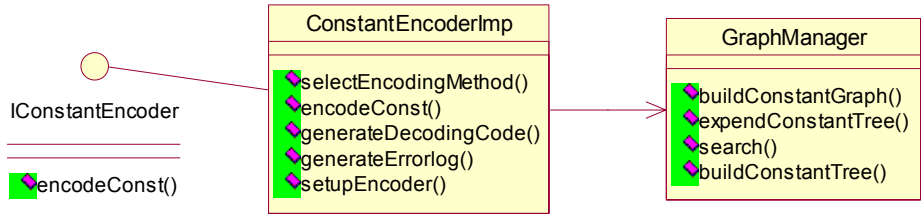


Figure 6-3 Conceptual class diagram for JSafeMark encoder

6.3 Design

After we have defined the functionality, conceptual structure and objects in the JSafeMark encoder, now we need to further develop them to define how they can be implemented. Because of the simple functionality of the JSafeMark, the design phase is relatively simple. We will introduce this process from two aspects: one is static design by further implementing the conceptual class diagrams, and the other is dynamic design by further specifying the workflow using sequence diagrams.

6.3.1 Static Design: Design Level Class Diagrams

In this section, we will refine the conceptual level classes into design level classes. The design level class constants solid information that can be used directly in system implementation.

One thing we need to consider is to introduce a node class into the system because we need to rebuild PPCT structures in the JSafeMark using local available resources. As we said before, the structures of PPCT are passed into the JSafeMark in the format of a two-dimensional array, and only the relations between nodes are passed, without any node information (see Section 4.3). Thus, we add a node class *GraphNode* into the JSafeMark encoder.

We have also split the function, *generatDecodingCode*, out of the *ConstantEncoderImp* class, because generating the decoding function is too complex to be a method in a control class. Meanwhile, an *ErrorLog* class is added into the model, for recording errors generated.

The search methods and constant converting methods are packaged separately. *ConstantHandler* and *Searcher* class are in charge of there operations. The classes are shown in Figure 6-4.

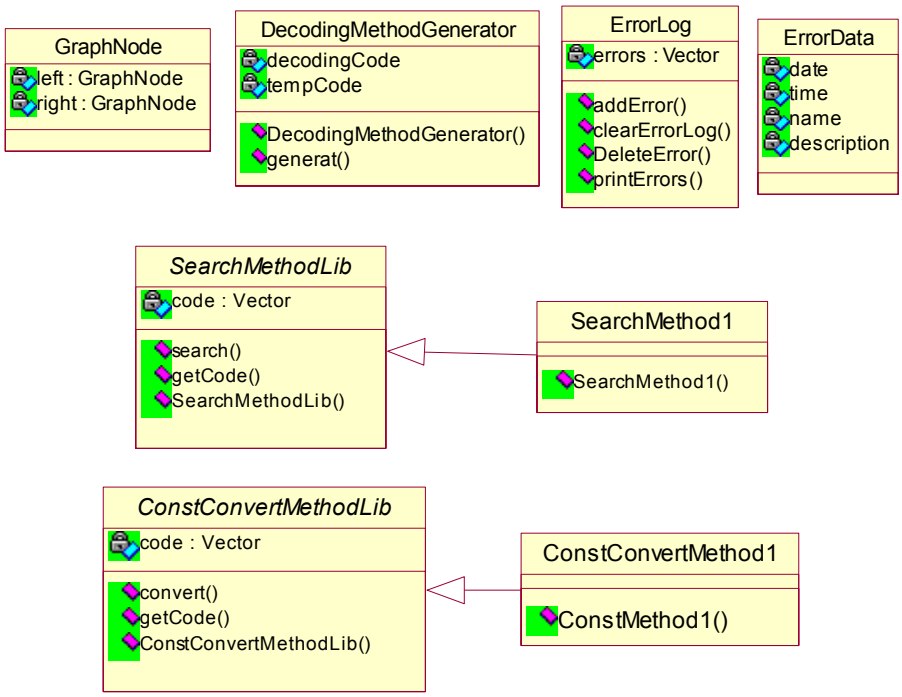


Figure 6-4 GraphNode class, DecodingMethodGenerator class, Error Classes and searchingMethod package constantConvertingMethod package example classes

After above discussion, we draw the design level class diagrams in Figure 6-5.

Description of selected design level classes in class diagram:

ConstantEncoderImp. It controls the workflow in the JSafeMark encoder, including setting up the encoder, encoding constant, building tree structures, searching, generating decoding method and error log.

GraphNode. This is the node structure for build a PPCT. It is used by *GraphManager* class to build all local PPCT structures.

DecodingMethodGenerator. This is used for generating source code of the decoding functions for a constant.

ConstantHandler. It handles constant to integer conversion, and offers source code for the converting constant value back from the integers.

Searcher: It searches a constant tree for a substructure that matches a constant graph, generates the source code for defining the location and boundary of the substructure.

ErrorData: contains error information.

ErrorLog: controls error generated.

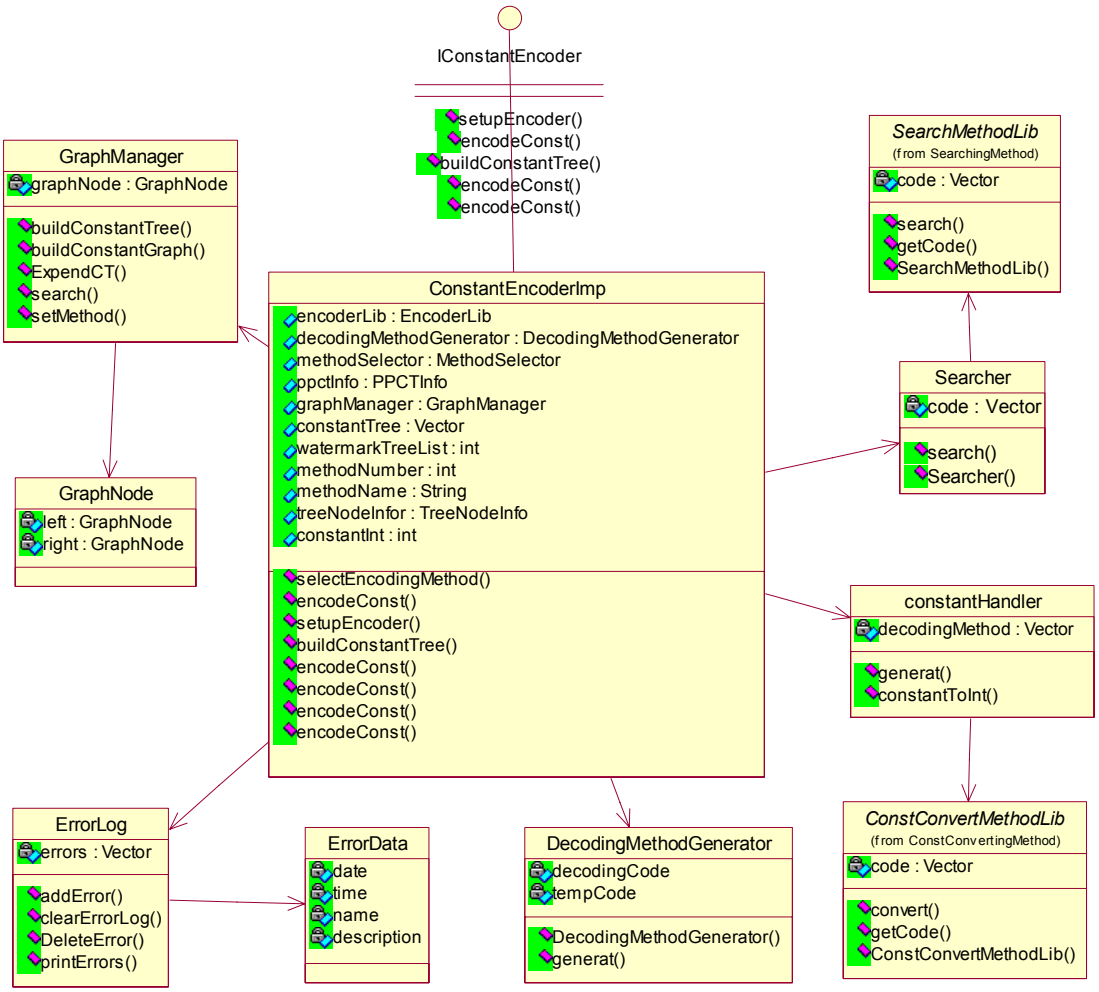


Figure 6-5 Design level class diagram

6.3.2 Dynamic Design: Design Level Sequence Diagrams

The Sequence Diagrams in design level not only describe the dynamic behavior of the encoder, but also examine if the defined activities for the encoder can work properly. They display more details than the ones in the conceptual level. Figure 6-6 illustrates the design level sequence diagram for Encode Constant use case.

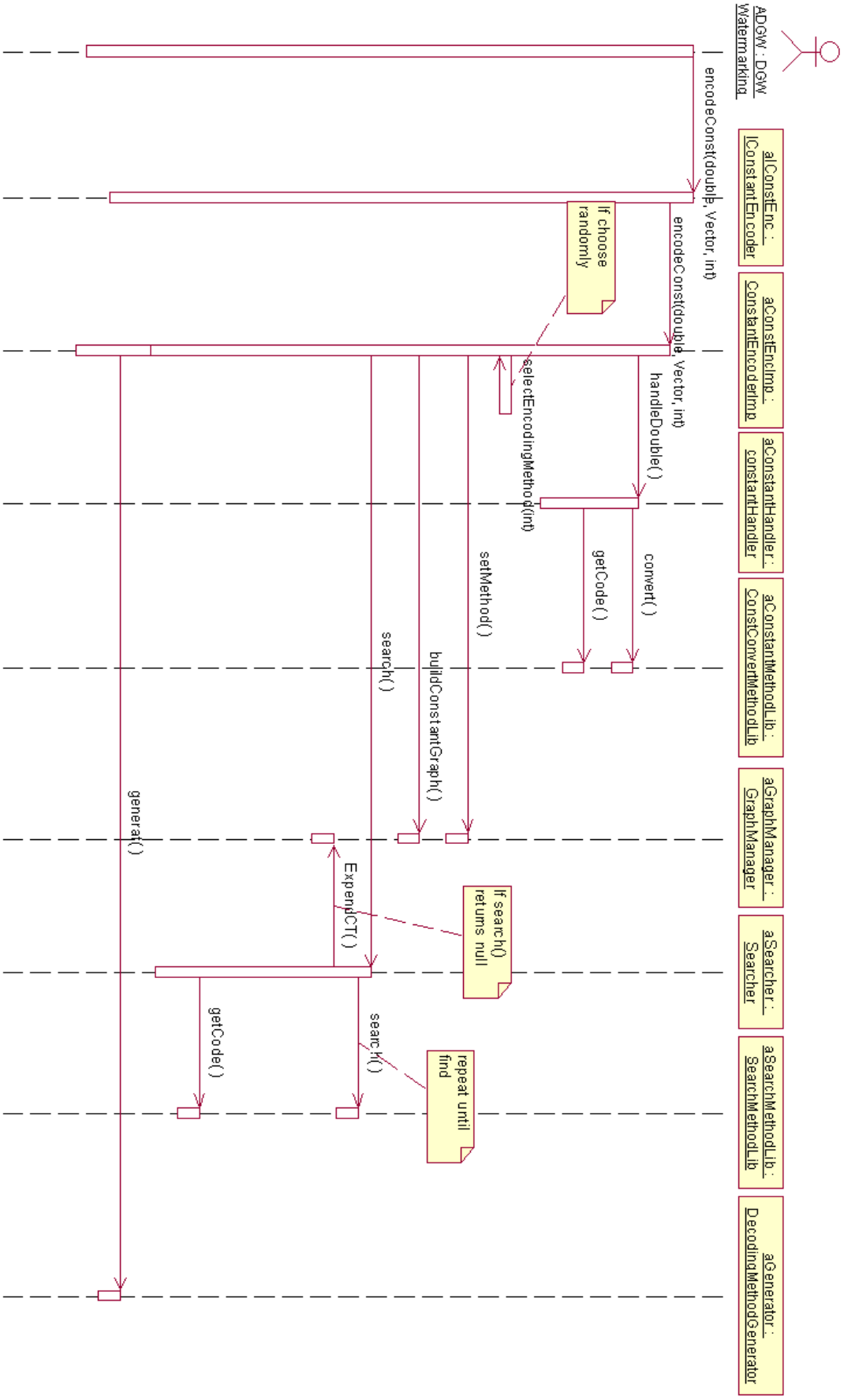


Figure 6-6 Design level sequence diagram for constant encoding

Explanation of design level Converting sequence diagram:

(1) encodeConst(double, Vector, int) method in *IConstantEncoder* class and *ConstantEncoderImp* class will start an operation to encode a double variable and get the source code back, using an algorithm from *SearchMethodLib*.

(2) *handleDouble*() will convert a double constant into integers, and generate a function for converting it back.

(3) *selectEncodingMethod*() will decide which method to use if the JSafeMark needs to choose randomly.

(4) *setMethod*() tells which method need to use, to construct the constant graph.

(5) *buildConstantGraph()* method builds a constant graph with the method defined by the method index. The actual building process is done in the Codec. *GraphManager* will call the building method in the Codec and restore the constant graph structure returned by the Codec.

(6) *search()* tells *Searcher()* to find a substructure that matches the constant graph, using a selected algorithm from *searchingMethodLib*.

(7) If search() doesn't find any substructure required, a expandCT() process is required to add additional nodes into the constant tree, so that the substructure will appear in the constant tree.

(8) *generate()* method will generate the source code to retrieve the double value out of the substructure which is inside the constant tree.

6.4 Implementation

In the last two sections, we analyzed the requirements and conceptual functionalities of the encoder, and we also gave the software structures in design level. All the analysis and design is in UML style.

Our implementation of the encoder is developed using JDK 1.3, part of the source code is listed in Appendix A.

In our implementation, the codecs use the CLOC and CUIC technology introduced in Section 3.4.4 in page 49. From the discussion in Section 3.4.4, we notice that the calculation of Catalan numbers is heavily involved. Thus, we pre-calculate all Catalan numbers needed.

Moreover, the *min_int(L, R)*'s we defined in Section 3.4.4 are also pre-calculated because its calculation also involves large amount of Catalan number processing. By pre-calculate the Catalan numbers and min_int(L, R)'s, we can make the program execution much more efficient.

In our implementation, this is done in Util.java class.

The best way a codec can use is storing all Catalan numbers in a Vector, instead of calculating at runtime. Unfortunately, this approach cannot be easily used in run time retrieval, because a list of unusual numbers will be a great target for attackers to start from.

6.5 Summary

In this chapter, we developed a prototyping system for constant encoding. We did the analysis and design using UML, and showed the detailed activities of some important part of the system. In next chapter, we will show how our JSafeMark dealing with the constant encoding with an example.

Chapter 7

JSafeMark in Practice

7.1 Introduction

In Chapter 4, we overviewed the concept of the JSafeMark encoder, and in Chapter 6, we discussed the detailed design and implementation issues mostly using the UML notation following the software engineering process. Although UML can describe the system clearly including what a class or even a method should look like, we still have no experience in how our encoder works in real world. In this chapter, we will show a solid example to see how this constant encoding algorithm works.

7.2 Example of Using JSafeMark encoder

In this section, we follow the workflow of the constant encoding described in Section 3.4 (page 40) to demonstrate how a constant is encoded. In the testing example, we use a double constant 1.4 as the E-constant and encode it in a small constant tree with five leaves. By using such a small number, and by choosing a small constant tree, we can easily track the status of the constant tree structures during execution.

We ignore the finding E-constants process, which is carried out in the DGW system (This is the encoding preparation phase in Section 3.4), because our simulator does not perform

these tasks. In the following testing, all of the intermediate data are obtained by tracing the runtime status of the encoder, and intercepting the intermediate output of the encoding process.

7.2.1 Import Constants (WF1)

Our workflow of Figure 3-5 in page 43 begins with step WF1, which is that the encoder accepts a list of constant. This is shown as follows in the test.

The test is launch by execute the tester in the DGW system simulator mode, as described in Section 5.2.3 by the following command:

```
java JSTester –s -1.4
```

The option –s tells the JSafeMark tester to launch the DGW system simulator session, and the parameter 1.4 tells the tester to export 1.4 to the encoder as the constant value.

We trace the program method calls and get the following output.

```
Entering ConstEncoder: encodeConst(Vector).

The imported constant is double 1.4.
```

This indicate that the JSafeMark tester called the *encodeConst()* method in the *ConstantEncoderImp* class and found the constant is a double value 1.4.

7.2.2 Building PPCT Constant Tree (WF2)

The next in the output is:

```
Entering GraphManager: buildConstantTree().

Constant tree 8 with leaf 5.
```

This indicates the building constant tree process is executed and a constant tree of index 8 with 5 leaves is constructed. This is the step WF2 in Figure 3-5.

In the testing, in order to simplify the testing, we chose 5 to be the leaf number of the random tree, and hardcoded in the buildConstantTree() method.

We launch another session of the JSafeMark tester in the “PPCT Structure Viewer” mode by typing the following command:

```
java JSTester -d
```

A graphic user interface is shown. We choose “manual select” and “compact display” mode.

Then display the PPCT structure of index 8 with 5 leaves. A screenshot is shown below.

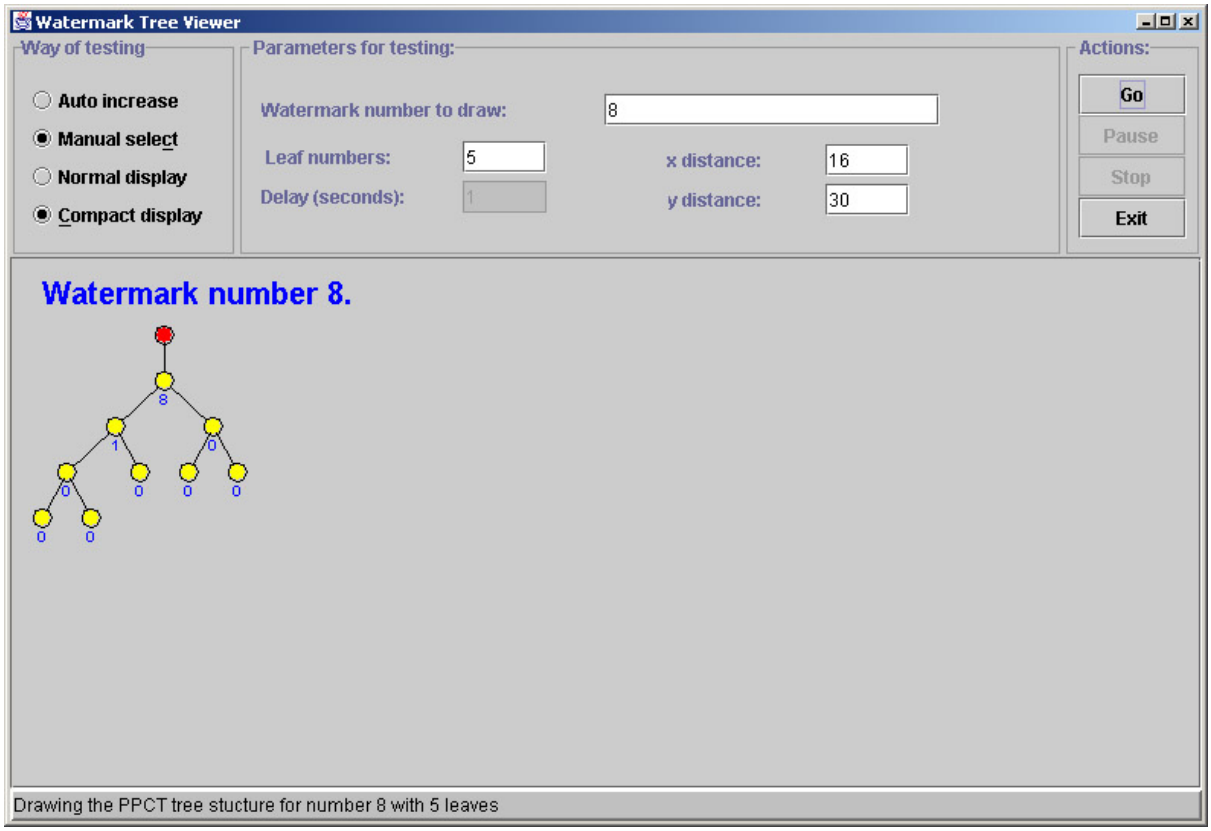


Figure 7-1 The random constant tree of index 8 with 5 leaves

7.2.3 Convert Constant to Integer (WF3)

The next step in Figure 3-5 is WF3, splitting constant into integers. In the testing, this step can be illustrated by the following output of the encoder.

Entering ConstantHandler: constantToInt(double).

double 1.4 >>>> integers 1, 4

The above output shows that the encoder called the *constantToInt*() method in the *ConstantHandler* class. The double constant 1.4 has been converted into integers 1 and 4.

For the detailed algorithm of method *constantToInt*(), please refer to page 48.

7.2.4 Generate Source Code to Reverse WF3 (WF4)

This is the step WF4 in Figure 3-5. In this step, the following output is intercepted, which is the source code stored in a Vector.

```
public double convertTwoDigitsToDouble(int firstInt, int secondInt){
    return firstInt + secondInt * 0.1;
```

```
}
```

The above code is used to convert two single digit integers into a double value, which is introduced in Section 3.4.6 in page 61 .

7.2.5 Convert Integer to Graph Structure (WF5)

The next step in Figure 3-5 is WF5, building a constant graph. We use Catalan Unique Indexed Conversion (CUIC in page 56) to be the conversion algorithm for converting the integers into the graph structures. This is hard coded in the encoder source code. The reason we choose CUIC encoding is, by using CUIC encoding, an integer can be converted into exactly one PPCT structure.

The following output shows the result of the conversion.

```
1 >>> {{3,1},{2,3},{0,2},{2,3}}

4 >>> {{7,1},{2,3},{0,2},{4,5},{2,4},{6,7},{4,6},{6,7}}
```

The above two two-dimensional arrays represent the corresponding integer, as we discussed in Section 4.3. We can manually draw the following structures according to the two dimensional arrays as shown in Figure 7-2.

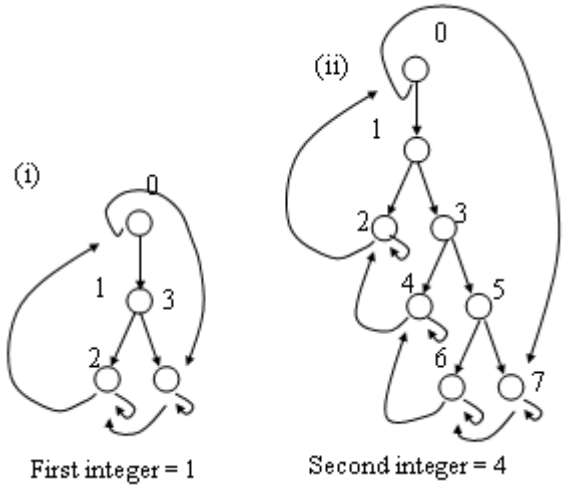


Figure 7-2 PPCT structures for first integer = 1 and second integer = 4

Please have a look at Figure 3-12 in page 57. The index 0 of CLOC encoding with 2 leaves in Figure 3-12, which is also the index 1 of CUIC encoding, is exactly the same as the first integer 1 in Figure 7-2 above. The index 4 of CUIC encoding in Figure 3-12, which is also the index 0 of CLOC encoding with 4 leaves, is the same as the second integer 4 in Figure 7-2 above.

7.2.6 Generate Source code to Reverse WF5 (WF6)

When the integers 1 and 4 are converted into PPCT structures, the source code for decoding the structures into integers has also been generated. We intercept the generated source code from the returned Vector, and get the following output.

```
public int decodingMethod(Node node){

}
```

This function is base on Catalan Unique Indexed Conversion (CUIC). Because of the complexity of the CUIC decoding function, we do not list it here. Please see Appendix C in page 133 for the decoding function source code. It is similar to the source code of the watermark retrieving function attached in Appendix A, which is based on Catalan Leaf Oriented Conversion (CLOC).

7.2.7 Searching (WF7)

This is the step WF7 (including WF7-1 and WF7-2) in Figure 3-5. As described in Section 3.4.5 (page 57), this step will search exhaustively through the constant tree to find the constant substructure for an integer. If a specified substructure cannot be found inside the constant tree, we need to expand the constant tree to include that substructure (WF8, discussed in the following section).

The following output shows the result of this step.

```
1 >>> found, node 4

4 >>> not found
```

This result shows that the search for integer 1 succeeded. However, a substructure for integer 4 was not found.

In fact, when an integer is found, the node index, at which the integer is found, will also be recorded as described in WF7-2 in Figure 3-5. This information was also output after an integer is found (“node 4” after “1>>>4”). However, after the constant tree is fully expanded later on, the indices of the tree nodes will be sorted again. Thus, we do not output where the integer is found at this stage. It will be traced again after the constant tree is fully expanded.

7.2.8 Expand the Constant Tree (WF8)

Once a substructure for an integer is not found in the constant tree, according to the

workflow in Figure 3-5, the constant tree will be expanded (WF8).

From the previous output, listed in Section 7.2.7, we know that the substructure for integer 4 was not found. Thus, we get the following output.

Entering GraphManager: expandCT(int)

4 >>> found, at node 3

The above output shows that the expandCT method in the GraphManager class is called. Thus the constant tree is expanded.

The output also shows that after the WF8, which expand constant tree is performed, the step WF7-1 in the Figure 3-5 is performed again, and a substructure is found at node with index 3.

After this process, all the integers are found. The node indices of the constant tree are sorted and output the following information.

Entering GraphManager: sortIndex()

CT >>> {{13,1},{2,3},{4,5},{6,7},{8,9},{9,5},{5,6},{10,11},{0,8},{8,9},{6,10},

{12,13},{10,12},{12,13}}

1 >>> sorted at node 4

4 >>> sorted at node 3

The above information is output for debugging. The first two lines contain the information of the fully built constant tree. The next line indicate that after the indices of the constant tree are sorted, the substructure for the integer 1 is rooted at node 4, and the substructure for integer 4 is rooted at node with index 3.

According to these output, we manually draw the constant tree as shown in Figure 7-3.

In Figure 7-3, the PPCT structure on the left shows the fully built constant tree, which is drawn according to the output two-dimensional array. The diagram on the right shows two substructures. The substructure rooted at the node 4 is for the integer 1 and the substructure rooted at the node 3 is for the integer 4.

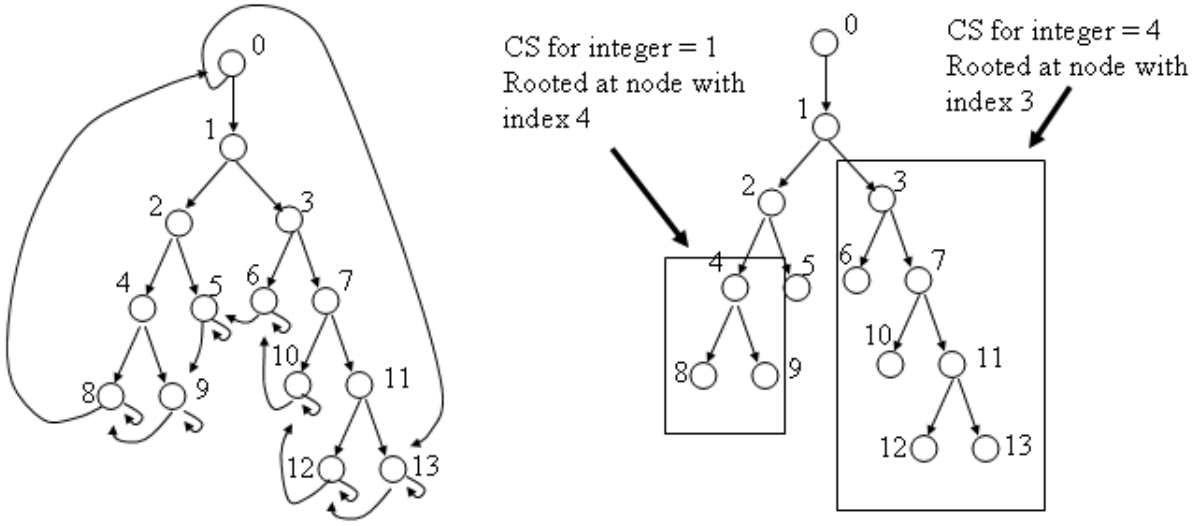


Figure 7-3 Constant tree and the substructure for integers 1 and 4

7.2.9 Generate Referencing information (WF9)

In the workflow in Figure 3-5, the next step is WF9, generate referencing information. According to the discussion in Section 3.4.6 in page 62, our implementation uses three parameters to find the root of the substructure at runtime. They are *path*, *depth* and *mask*. The following output shows the values of the three parameters for each of the integers.

1 >>> path 0 depth 2 mask {{7,1},{2,3},{4,5},{6,7},{0,4},{4,5},{5,6},{6,7}}

4 >>> path 1 depth 1 mask {{15,1},{2,3},{4,5},{6,7},{0,4},{4,5},{8,9},{10,11},

{5,8},{8,9},{12,13},{14,15},{9,12},{12,13},{13,14},{14,15}}

Figure 7-4 shows how to use path = 0 and depth = 2 to find the root of the substructure for the first integer = 1, and use path = 1 and depth = 1 to find the root of the substructure for the second integer = 4.

According to the output information about the mask structures, we draw the mask trees on the left side of Figure 7-5. Referring to the substructure information in the diagram on the right side of Figure 7-3, we mark the overlapping part of the mask tree and the constant tree in black, as shown in Figure 7-5.

From the analysis, we know that the output value of the path, depth, and mask information is correct. Now we have a look at the generated source code for runtime referencing.

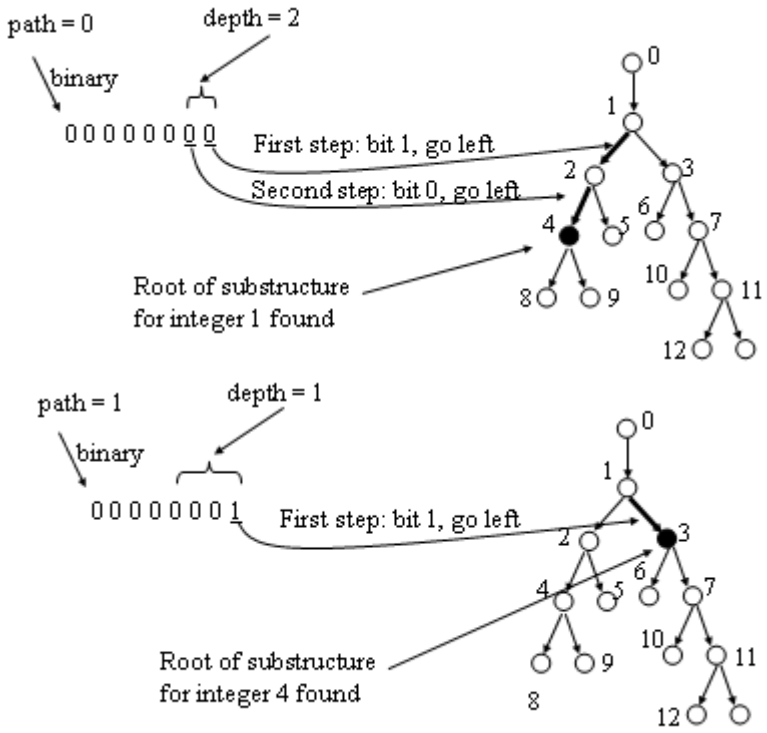


Figure 7-4 Referencing to the substructures for integers 1 and 4

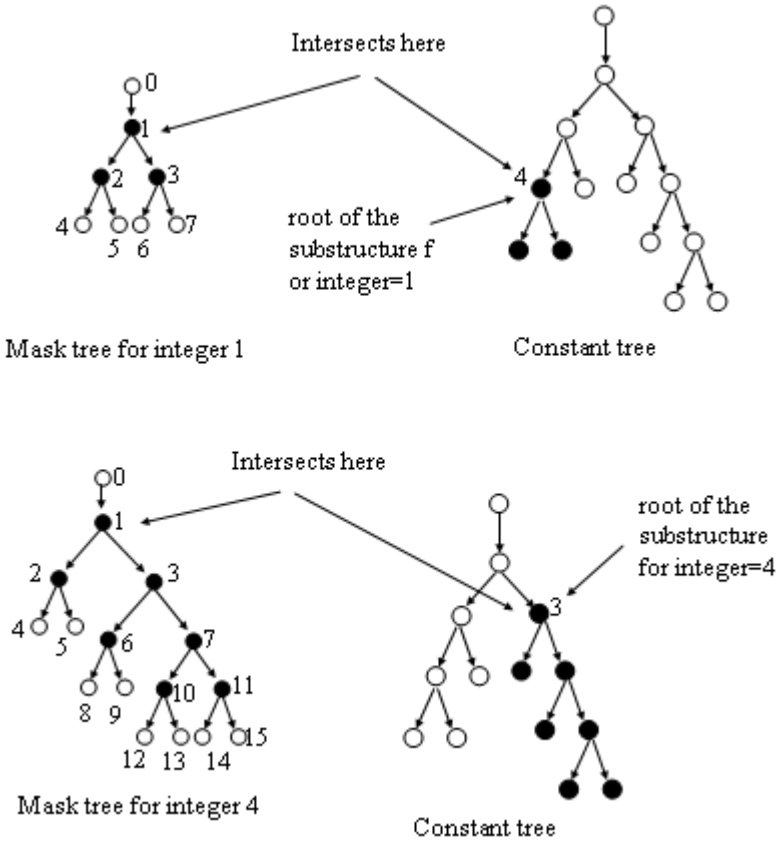


Figure 7-5 intersecting mask and CT to find the boundary information for integers 1 and 4

Part of the source code for runtime referencing is output as follows.

```
public Node getSubstructure(Node RCT, int path, int depth, Node mask){
        Node RCS = getSubstructureRoot(RCT.right, path,  depth);
        return getCS (RCS,  mask);
}
public Node getSubstructureRoot(Node RCT, int path, int depth){}
public Node getCS(Node RCS, Node mask){}
```

The source code generation has been discussed in Section 3.4.6. Please also refer to Appendix C in page 133 for the full functional source code.

7.2.10 Generate and Decoding Source Code (WF10, 11)

Now that we have finished the source code generation for different steps in the workflow in Figure 3-5, the next step is combine the generated code together to make it work (WF10) and out export the source code to the tester.

We test this by examine the final output of the encoder.

In the Vector that contains the output, we find the following information.

Two-dimensional int arrays:

```
{{13,1},{2,3},{4,5},{6,7},{8,9},{9,5},{5,6},{10,11},{0,8},{8,9},{6,10},{12,13},
        {10,12},{12,13}}
{{7,1},{2,3},{4,5},{6,7},{0,4},{4,5},{5,6},{6,7}}
{{15,1},{2,3},{4,5},{6,7},{0,4},{4,5},{8,9},{10,11},{5,8},{8,9},{12,13},{14,15
        },{9,12},{12,13},{13,14},{14,15}}
```

Expression used to replace the E-constant in the candidate program.

```
getConstant(rct, 0,2,m1,1,1,m2))
```

Source code of the definitions of the functions.

The source code is listed in the Appendix C. Please see page 133 for detailed source code. For ease of understanding, the variable names in the Appendix C have been modified to make them meaningful.

The first line in Appendix C is the expression used to replace the E-constant in the candidate program, as discussed above. The rest parts are the function definitions. Simple comments are added into the source code to make it understandable.

We copied these output and tested it in a Java program by

System.out.println("The constant is "+getConstant(rct, 0,2,m1,1,1,m2));

and get the following output.

The constant is 1.4

7.3 Discussion

The testing performed in this Chapter is successful. However, since the encoder is partly prototyped, it can only handle one constant at this stage.

The decoding function generated during the testing is about 3.9KB, and the runtime decoding process cost 10ms.

The performance of the decoding process in the testing is acceptable. However, since the source code is generated for a single constant, we believe the code size will increase when encoding for more constants with more algorithms.

7.4 Summary

In this chapter, we demonstrated some practical aspects of the encoder, and tested it with a simple double constant. In the next chapter, we will point out some good and bad points about our JSafeMark, and also discuss some future work.

Chapter 8

Conclusion

8.1 Pros and Cons of JSafeMark

The JSafeMark algorithm is a new technology that can be used to give additional protection to the dynamic graph watermarks. By encoding constants into a constant tree, which is very similar to the watermark structure, the watermark source code is protected from removal by automated attacks based on dependency analysis. An attacker might hope that a static dependency analysis would reveal that the watermark tree is independent of the program computation, and thus that it could be removed without affecting program function. However, a static analysis on watermark node will reveal only that they may have the same dependencies as the nodes in the constant tree, and many portions of the watermarked code will depend on these nodes for correct evaluation of program constants. Thus, the attacker will be unable to remove the watermark tree safely, on the basis of a static dependency analysis.

Our constant encoding method will be protected against pattern-matching attacks because that in a fully developed encoder, there are many different algorithms that can be chosen from. Each algorithm uses different set of methods to decode the values of E-constants at runtime. Moreover, the source code generated from the same algorithm can be much different when working with different candidate programs. Thus, it is hard for the attacker to find out a

common pattern to work on.

As we discussed in Section 3.5.1, the possibility of finding a constant in a watermarked program is satisfying. As shown in Table 3-1 in page 69, in a randomly selected program, there are thousands of constant loading operations. In the encoding process, we expect that the DGW system can find some small integers in the candidate program for encoding. This will make the decoding process much simpler, because the steps WF3 and WF4 in Figure 3-5 in page 43 are not needed for the simple integers that can be encoded into constant tree directly.

However, since our technology is not yet mature, it has some obvious weaknesses. For example, our algorithm is based on finding constants inside a candidate program. It is possible that there are not any constants in a small program. In this case, our algorithm cannot be applied unless, perhaps, suitable constants are introduced in an obfuscation step. Another weakness is that the code generated for the decoding function seems a little bit complicated. For example, in our testing, 3.9KB code is required to encode a double constant 1.4 using CUIC encoding algorithm. This complexity can be reduced by carefully choosing simpler algorithms and encoding simple integers instead of complex constants.

8.2 Future Work

Given the time constraints of our Master's thesis, we were not able to fully specify, implement, test, and document the JSafeMark encoder. We believe that many improvements to the encoder are still needed, both in the algorithm and in the implementation before we could convincingly demonstrate its effectiveness at protecting the DGW watermarks against a wide variety of attacks. Now we indicate some ideas for improvement below.

8.2.1 Protect the Return Value of the Decoding Function

There is a weakness in JSafeMark algorithm, namely, if an attacker can monitor the returned value of a decoding function, he will realize that the returned value will always be the same. Although this weakness is protected to some extent in our implementation by using multiple parameters in the decoding functions, and by applying obfuscation techniques, we are not completely satisfied.

We would like to implement a more secure algorithm for the encoding. For example, if we can merge the decoding function directly with the statement accessing this constant, then

the returned value will not be so obvious, and it may be much harder to analyze. For example, in the following statement, if x is an integer variable,

int a = *x* + 2;

if we replace 2 with a decoding function call *decode()*,

int a = *x* + *decode()*;

the method *decode()* will always evaluate to 2. However, if we can use the following format,

int a = *decode(x)*;

then the attacker will not be able to see the weakness. Because the result of the entire decoding function is partially depending on the value of *x*.

However, if a true dependency is not created, the constant value 2 still needs to be protected. Thus, a true dependency is what we expected. However, it is hard to create, because a variable cannot be easily encoded into a constant structure, which is worth further research.

8.2.2 Changing Constant Tree at Runtime

Currently, the constant tree is built at the beginning of the program execution, and it is unchanged during the program execution. This is because decoding function might return an inappropriate value if the constant tree structure changes at runtime. Please look at Figure 8-1 and Figure 8-2.

In Figure 8-1, a program is executed with certain statement sequence. During this sequence, statement s1 and s2 are executed, and a constant tree CT is built The statement s3 includes the decoding functions *decode(CT)*. Suppose the decoding function is built on this structure, it will work well.

Figure 8-2 shows the same program is executed with a different statement sequence. During the execution, statement s4 is executed, and the constant tree is built in a different shape. When the program executes the same statement s3 as in Figure 8-1, the decoding function will work on a different constant structure. This will cause an error in decoding.

However, if the execution can be carefully analyzed, it may be possible to change portions of the constant tree at runtime. If this is possible, constant encoding will be more secure, because a changing structure is harder to analyze than a constant structure.

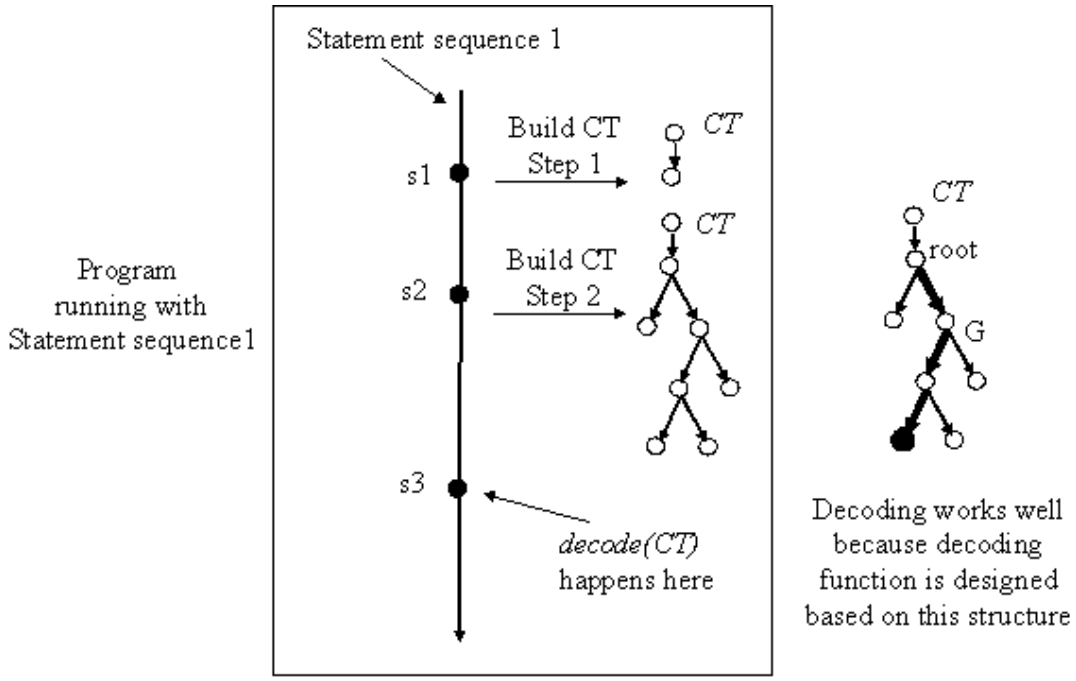


Figure 8-1 Changing constant tree at runtime. Constant tree is changed when executing a sequence of statements. Decoding function works well.

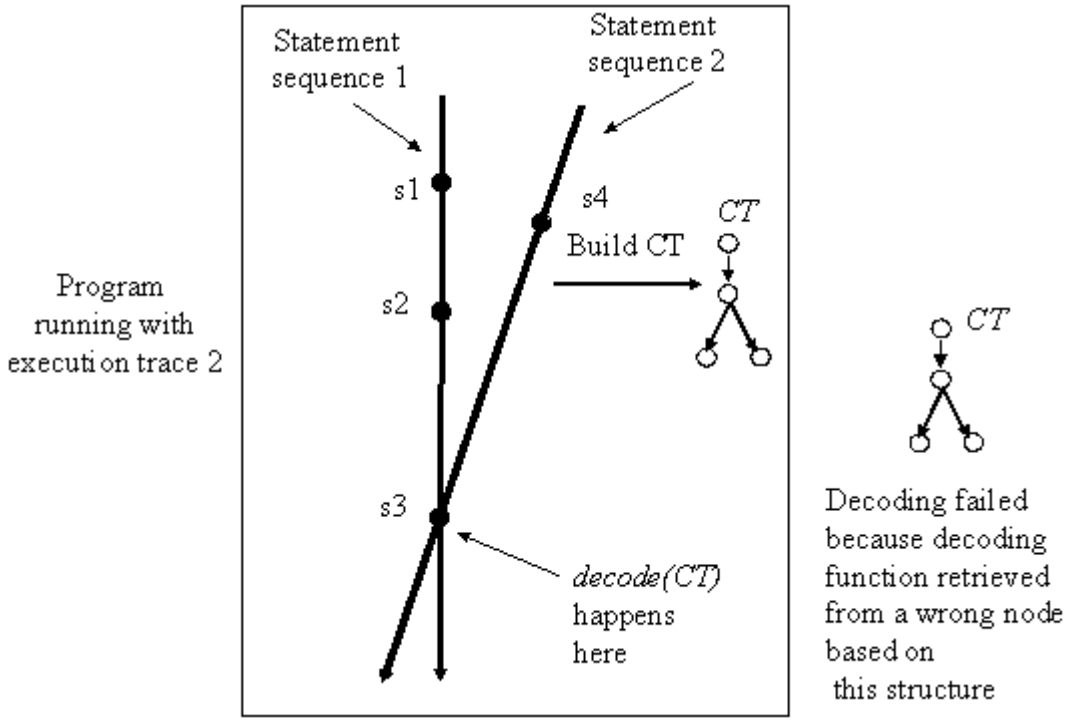


Figure 8-2 Changing the constant tree can cause errors.

8.2.3 Encode Constant into Watermark Structures

When we encode the constant into a constant tree, we create a false dependency for watermark tamperproofing. That is, the watermarked program is protected by the constant

tree, because the attacker cannot reliably distinguish the watermark structure from the constant tree. Thus, an attacker who tries to remove or alter the watermark tree is in danger of altering the constant tree.

However, we can create a true dependency from the watermarked program to the watermark structure if we encode E-constants in the watermark structure. If this portion of the watermark structure is removed, then watermarked program is damaged.

This approach must consider that the watermark structure will change at runtime. In Section 2.3.3, we indicated that the Sandmark system would create different parts of the watermark structure at different times and different places of a program that is presented with a special input sequence. Thus, the watermark structure embedded by Sandmark will change during the execution of the program. For some program inputs, no nodes might be allocated for the watermark structure. This degree of variability will bring great difficulties if we try to encode constant into Sandmark watermark structure.

8.2.4 Using Code that builds Watermarks to build a Constant Tree

We could build a constant tree at runtime, at least partially by executing the same source methods that will be used to construct a watermark structure if the special input sequence is presented. By executing the source code of the watermark structure, we create a constant tree that is essentially a new instance of the watermark structure (or part of the watermark structure). This new instance of the watermark structure will not be used for the retrieval of the watermark number. Instead, we will use it as a substructure of the constant tree. What we expect is, if an attacker modifies or removes the source code for constructing the watermark structure, the constant tree will not be built correctly, and the correct value of the encoded constant will not be retrieved.

8.2.5 Using a Set of Constant Trees

Another idea is to build a set of constant trees for different integers in different time slices, or in different statement sequences, of the execution of program execution. According to this idea, a large number of small constant trees are built, and each of the small constant trees may contain only a subset of the whole constant tree, but there is no large the constant tree. To adopt this algorithm, we must redesign our encoder, so that it can decode an integer from multiple substructures.

8.2.6 Using Constant Subtrees as Masks

With one miner modification to our encoding process of Figure 3-5, we could search for subtrees of the constant tree that would serve as masks during the decoding of constant. The "boundary information" field of a decoding function could then specify a mask by a pair (*mPath, mDepth*) of integers, defining the location of the mask in the constant tree. By applying this modification, the amount of export information will be reduced. More importantly, because the mask structure and the constant structure are both obtained from the constant tree, the degree of protection is increased.

8.3 Summary

This thesis introduced a new idea for protecting DGW watermarks, gave an outline of an algorithm for encoding constants, and examined the possibility of using our constant encoding algorithm. We gained some practical experience of our constant encoding technology. We concluded that there is still a lot of work to be done, to improve the algorithm and the implementation of our encoder to demonstrate an efficient and practical method for protecting dynamic graph watermarks.

Appendix A.

CLOC Encoding Implementation

In this appendix, we attached part of the source code of the Catalan Leaf Oriented Conversion codec. There are three classes in the source code. The WMRetriever class is used to convert a PPCT graph structure into an integer, and the TreeBuilder class is used to convert an integer into a graph structure. Both of the classes will use the information in the Util class, which is used to generate Catalan Numbers and min_int() values to speed up the encoding/decoding process.

```
/**
 * This class is used to retrieve watermarks from a dynamic watermark tree structure.
 * The tree structure needs to be a PPCT style, with only public left and right leaves,
 */
public class WMRetriever {
        PPCTTreeNode base;
   /**
    * Constructor of WMRetriever, create this object, doesn't do anything.
    */
        public WMRetriever(){
        }
   /**
    * Retrieve watermark number for whole tree structure.
    * @param top  PPCTTreeNode
    *        top can be any node inside the tree structure,
    *        doesn't need to be the root or origin of PPCT.
```

```
 * @return  a BigInteger representation of watermark number.
 */
      public BigInteger getWmNum(PPCTTreeNode top){
            while(top.right!=top){top = top.left;}
            do{     top = top.left;}
            while(top.right == top );
            base = top.right;
            return cNum(top.right);
      }
/**
 * Retrieve watermark number for a given subtree of a PPCT.
 *
 * @param top  PPCTTreeNode
 *       top must be the root of subtree
 * @return  a BigInteger representation of watermark number.
 */
      public  BigInteger  getSubTreeWmNumber(PPCTTreeNode top){
            PPCTTreeNode current=top;
            do {current = current.right;}
            while(current.right!=current);
            if(top.left == current){ return cNum(top.right);}
            return cNum(top);
      }

/**
 * Actual operation for retrieving a watermark number.
 * Used by both getWmNum() and getSubTreeWmNumber().
 * @param top  PPCTTreeNode
 *       top must be the root of a subtree
 * @return  a BigInteger representation of watermark number.
 */
      private BigInteger cNum(PPCTTreeNode top){
            if(top.right == top) return BigInteger.ZERO;
            return cNum(top.left).multiply ((BigInteger)Util.getCat().
                  elementAt(getRightLeafNum(top))).
```

```
                add(cNum(top.right)).add(minInt(getLeftLeafNum(top),
                getRightLeafNum(top)));
    }

/**
 * Method used to find the minimum number according to the left and
 * right leaf numbers of a given subtree.
 * @param left  the leaf number of the left subtree.
 * @param left  the leaf number of the right subtree.
 * @return  a BigInteger representation of the minimum number.
 */
    private BigInteger minInt(int left, int right){
        if(left==1) return BigInteger.ZERO;
        return minInt(left-1,right+1).add(((BigInteger)
                Util.getCat().elementAt(left-1)).
                multiply((BigInteger)Util.getCat().elementAt(right+1)));
    }

/**
 * Method used to get the leaf number of a given subtree.
 * @param top  PPCTTreeNode
 *       top must be the root of subtree
 * @return  a integer representation of the leaf number.
 */
    private int getLeafNum(PPCTTreeNode top){
        if (top.right == top) return 0;
        PPCTTreeNode left,right;
        int num=0;
        left = top;
        right = top;
        while(right.right!=right) right = right.right;
        while(left.right !=left) left = left.left;
        while(left!=right) {
                right=right.left;
```

```
                num++;
            }
            return num+1;
        }

    /**
     * Method used to get the leaf number of the left subtree
     * @param top  PPCTTreeNode
     *        top must be the root of subtree
     * @return  a integer representation of the leaf number.
     */
        private int getLeftLeafNum(PPCTTreeNode top){
            if (top.right == top) return 0;
            if(top.left.right == top.left) return 1;
            return getLeafNum(top.left);
        }

    /**
     * Method used to get the leaf number of the right subtree
     * @param top  PPCTTreeNode
     *        top must be the root of subtree
     * @return  a integer representation of the leaf number.
     */
        private int getRightLeafNum(PPCTTreeNode top){
            if (top.right == top) return 0;
            if(top.right.right == top.right) return 1;
            return getLeafNum(top.right);
        }
}
```

```
/**
 * This class is used for creating some static variables and also for debugging.
 */

public class Util{
        private static Vector cat,mid;
        public static int MAXLEAF = 150;
        public static final boolean DBG = false;
        public static final boolean DBG1 = false;
        public static int nodeNo=0;

    /**
     * Create a new Util object, meanwhile setup some variables
     * to be used to speed up operation.
     */
        public Util(){
                setupVec();
        }

    /**
     * Setup Catalan number vector and min_int vector for calculating
     * to be used in other class to speed up progress
     */
        public static void setupVec(){
                long t1 = System.currentTimeMillis();

                cat = new Vector();
                cat.setSize(MAXLEAF + 1);
                cat.setElementAt(BigInteger.ZERO, 0);
                cat.setElementAt(BigInteger.ONE, 1);

                for(int i = 2; i <= MAXLEAF; i++)
                {
                        BigInteger biginteger = ((BigInteger)cat.
                                elementAt(i - 1)).multiply(BigInteger.valueOf
```

```
                        (2 * (2 * i - 3))).divide(BigInteger.valueOf(i));
                cat.setElementAt(biginteger, i);
            }

            mid = new Vector();
            mid.setSize(MAXLEAF + 1);
            mid.setElementAt(new Vector(), 0);
            mid.setElementAt(new Vector(), 1);

            for(int j = 2; j <= MAXLEAF; j++)
            {
                Vector temp = new Vector();
                temp.setSize(j);
                temp.setElementAt(BigInteger.ZERO, 0);
                for(int k = 1; k < j; k++)
                {
                    BigInteger biginteger = (((BigInteger)cat.elementAt(k)).
                        multiply((BigInteger)cat.elementAt(j - k))).
                        add((BigInteger)temp.elementAt(k - 1));
                    temp.setElementAt(biginteger, k);
                }
                mid.setElementAt(temp, j);
            }
        }

    /**
     * method used to get catalan number vector
     * @return  a Vector representation of catalan number.
     */
        public static Vector getCat(){
            if (cat==null) setupVec();
            return cat;
        }

    /**
```

```
    * method used to get minimum integer vector
    * @return  a Vector mid.
    */
        public static Vector getMid(){
                if (mid==null) setupVec();
                return mid;
        }
}
```

```
/**
 * This class is used to create a dynamic watermark tree structure. Based on a given
 * watermark number and given leaf number.
 * Because the value range of integers or longs is relatively small, BigInteger is used.
 */
public class TreeBuilder
{
   public Leaf origin; // the origin of a watermark tree

   /**
    * Create a watermark tree structure according to given leaf number
    * and given watermark number.
    * If the wm is not too large to encode in a tree with nodeNo leaves,
    * then a tree that encoding wm is built at origin.
    *
    * @param nodeNo  an integer
    *        nodeNo is the number of leaves in the tree to be built
    * @param wm  a BigInteger
    *        wm is the watermark index of the tree to be built
    */

   public void createTree(int nodeNo, BigInteger wm)
   {
         if(wm.compareTo((BigInteger)Util.getCat().elementAt(nodeNo)) > 0){
            System.out.println("The watermark number is out of range.");
         }else{
         origin = buildTree(nodeNo,wm);
         }
   }
   /**
    * Actual process to create a watermark structure according to given leaf number
    * and given watermark number
    * @param i   an integer
    *        i is the number of leaves for building the tree or subtree.
    * @param wm  a BigInteger
```

```
*       wm is the watermark number for building the tree or subtree.
* @return  a Leaf representation of the origin of PPCT subtree.
*/
private Leaf buildTree(int i, BigInteger biginteger)
{
   if(i == 1 && biginteger.compareTo(BigInteger.ZERO) == 0){
                        Leaf tree = new Leaf();
                        Leaf leaf = new Leaf();
                        tree.left = leaf;
                        tree.right = leaf;
                        leaf.left = tree;
                        leaf.right = leaf;
                        return tree;
               }else if(i == 2 && biginteger.compareTo(BigInteger.ZERO) == 0){
                        Leaf tree = new Leaf();
                        Leaf leaf1 = new Leaf();
                        Leaf leaf2 = new Leaf();
                        Leaf leaf3 = new Leaf();
                        tree.left = leaf3;
                        tree.right = leaf1;
                        leaf1.left = leaf2;
                        leaf1.right = leaf3;
                        leaf2.left = tree;
                        leaf2.right = leaf2;
                        leaf3.left = leaf2;
                        leaf3.right = leaf3;
                        return tree;
               }else if(i < 3){
                       return null;
   }

   int k = 1;
   Vector temp= (Vector)Util.getMid().elementAt(i);
   while(biginteger.compareTo((BigInteger)temp.elementAt(k))>=0) {
                        if(k == temp.size()) {
```

```
                    return null;
                }
                k++;
            }
        biginteger = biginteger.subtract((BigInteger)temp.elementAt(k - 1));
        Leaf leaf1 = buildTree(k, (biginteger.divide((BigInteger)
                Util.getCat().elementAt(i - k))));
        Leaf leaf2 = buildTree(i - k, (biginteger.remainder(
                (BigInteger)Util.getCat().elementAt(i - k))));
        return merge(leaf1, leaf2);
    }

    /**
     * Process to merge two PPCT into one.
     * @param tree1  a Leaf to use for the origin of the first subtree.
     * @param tree2  a Leaf to use for the origin of the second subtree.
     * @return  a Leaf representation of the origin of merged PPCT subtree.
     */
    private Leaf merge(Leaf tree1, Leaf tree2)
    {
        if(tree1 == null || tree2 == null) return null;
        Leaf left = tree2.right;
        while(left.right!=left) left = left.left;
        left.left=tree1.left;
        tree1.left = tree2.left;
        tree2.left = tree1.right;
        tree1.right = tree2;
        return tree1;
    }
}
```

Appendix B.

Experimental Data of Integer Frequency in PPCT Structures

The data listed in this appendix are part of our experimental result showing the probabilities of integers that can be found in PPCTs with 50, 100, 200, 300, 500 nodes. The "Number of nodes in PPCT" indicates the size of the PPCTs. The integer values on the left-side shows the integers that can be found in the PPCTs. The experimental result is shown in the range of 0 to 100, where 100 mean the estimated probability is 100%.

integer	Number of nodes in PPCT					integer	Number of nodes in PPCT				
	50	100	200	300	500		50	100	200	300	500
0	100	100	100	100	100	12	91	100	100	100	100
1	100	100	100	100	100	13	99	100	100	100	100
2	100	100	100	100	100	14	97	100	100	100	100
3	100	100	100	100	100	15	94	100	100	100	100
4	100	100	100	100	100	16	95	100	100	100	100
5	100	100	100	100	100	17	99	100	100	100	100
6	100	100	100	100	100	18	100	100	100	100	100
7	100	100	100	100	100	19	93	100	100	100	100
8	100	100	100	100	100	20	99	100	100	100	100
9	82	100	100	100	100	21	98	100	100	100	100
10	96	100	100	100	100	22	93	100	100	100	100
11	96	100	100	100	100	23	53	77	93	98	100

integer	Number of nodes in PPCT					integer	Number of nodes in PPCT				
	50	100	200	300	500		50	100	200	300	500
24	60	94	99	100	100	56	82	97	100	100	100
25	60	97	100	100	100	57	68	97	100	100	100
26	67	94	100	100	100	58	71	93	100	100	100
27	75	97	100	100	100	59	77	95	99	100	100
28	63	95	100	100	100	60	77	100	100	100	100
29	72	96	99	100	100	61	72	98	100	100	100
30	63	98	100	100	100	62	72	95	99	100	100
31	78	100	100	100	100	63	74	95	99	100	100
32	76	97	100	100	100	64	51	90	98	100	100
33	67	92	100	100	100	65	21	44	77	95	99
34	71	97	100	100	100	66	30	59	85	98	100
35	76	98	100	100	100	67	27	62	88	97	100
36	83	100	100	100	100	68	34	64	95	98	100
37	65	97	99	100	100	69	39	75	97	99	100
38	74	98	100	100	100	70	37	63	87	98	100
39	70	96	100	100	100	71	31	78	87	98	100
40	60	99	100	100	100	72	29	73	93	100	100
41	74	93	100	100	100	73	39	74	92	100	100
42	76	97	100	100	100	74	40	71	92	98	100
43	75	97	100	100	100	75	34	71	96	100	100
44	78	97	100	100	100	76	44	76	94	99	100
45	75	97	99	100	100	77	40	78	98	98	100
46	77	99	99	100	100	78	51	83	99	100	100
47	71	96	100	100	100	79	31	58	87	96	100
48	72	94	100	100	100	80	27	74	93	100	100
49	79	98	100	100	100	81	37	75	93	99	100
50	73	91	100	100	100	82	30	74	93	98	100
51	70	100	100	100	100	83	42	72	91	100	100
52	78	98	100	100	100	84	31	77	95	98	100
53	68	95	100	100	100	85	36	80	89	99	100
54	66	96	100	100	100	86	40	69	96	100	100
55	71	97	100	100	100	87	36	80	92	99	100

integer	Number of nodes in PPCT					integer	Number of nodes in PPCT				
	50	100	200	300	500		50	100	200	300	500
88	42	75	92	99	100	200	14	36	64	82	97
89	33	72	94	99	100	210	19	52	86	89	99
90	37	80	97	100	100	220	18	40	67	84	98
91	43	80	93	100	100	230	28	46	69	88	98
92	43	78	98	100	100	240	14	37	74	86	93
93	32	74	94	99	100	250	18	39	72	84	98
94	39	77	96	98	100	260	14	44	70	94	97
95	39	69	96	99	100	270	17	47	76	93	98
96	35	68	95	99	100	280	17	51	72	96	99
97	36	75	95	100	100	290	18	44	73	89	98
98	45	75	94	99	100	300	14	48	76	93	96
99	41	73	94	98	100	310	22	41	74	90	100
100	44	70	95	100	100	320	26	53	73	93	98
105	44	84	97	100	100	330	16	35	71	94	95
110	33	74	97	100	100	340	18	49	64	92	100
115	47	66	89	99	100	350	19	42	69	86	96
120	51	78	96	99	100	360	21	42	78	89	98
125	38	68	98	98	100	370	27	53	80	88	97
130	47	70	93	100	100	380	18	43	80	87	99
135	44	75	95	100	100	390	16	53	70	87	98
140	44	72	93	98	100	400	13	41	73	92	100
145	42	77	100	98	100	410	23	58	74	88	98
150	37	81	94	100	100	420	20	50	72	86	98
155	44	70	92	98	100	430	18	48	79	91	98
160	37	80	99	100	100	440	24	48	68	87	94
165	39	65	97	98	100	450	16	41	71	90	100
170	50	83	93	100	100	460	21	44	83	90	97
175	43	70	100	99	100	470	17	45	77	92	99
180	43	69	93	100	100	480	12	45	78	91	98
185	38	81	97	98	100	490	16	46	81	91	98
190	35	71	90	96	100	500	19	54	68	90	98
195	40	73	90	99	100	550	22	56	79	88	98

integer	Number of nodes in PPCT				
	50	100	200	300	500
600	16	48	67	93	99
650	5	21	37	64	84
700	10	23	47	64	80
750	9	27	64	72	91
800	5	21	44	66	74
850	5	25	45	68	78
900	10	25	50	70	87
1000	15	21	46	61	91

Appendix C.

The output of the generated Source Code in testing

```
getConstant(rct, 0,2,m1,1,1,m2))

public double getConstant(Node RCT, int path1, int depth1, Node mask1, int path2,
int depth2, Node mask2){
//The entry point of the decoding process.
        int firstInt, secondInt;
        Node CS1 = getSubstructure(RCT, path1, depth1, mask1);
        Node CS2 = getSubstructure(RCT, path2, depth2, mask2);
        firstInt = decodeMethod(CS1);
        secondInt = decodeMethod(CS2);
        return convertTwoDigitsToDouble(firstInt, secondInt);
}

public Node getSubstructure(Node RCT, int path, int depth, Node mask){
        //control the process of getting the substructure
        //return the substructure that can be used to retrieve integer value
        Node RCS = getSubstructureRoot(RCT.right, path,  depth);
        return getCS (RCS,  mask);
}

public Node getSubstructureRoot(Node RCT, int path, int depth){
        // return the root of the substructure in the constant tree
        Node result = RCT;
```

```
        int temp = path; // create a local copy of integer path
        for(int i = 0; i<depth; i++){
                int least = temp&1; //get the least significant bit
                if (least == 1) result = result.right; //goto the right child
                else result = result.left; //goto the left child
                temp = temp>>1; //shift right
        }
        return result;
}

public Node getCS(Node RCS, Node mask){
        //return the substructure when the root of the substructure and
        //the mask tree is known.
        // RCS: root of the constant substructure
        Node tempCS = mask;
                // create a local copy of mask to be the initial structure of CS
        Node current = tempCS.right;
                //create a reference to the root of tempCS.
                //because node tempCS is the origin of the PPCT.
        checkNode(RCS, current);
        return tempCS;
}

public void checkNode(Node RCS, Node current){
        // modify the tempCS structure to be the CS (constant substructure) that
        // can be used to retrieve integer
        if(RCS.right!=RCS && current.right !=current){
                checkNode(RCS.right, current.right);
                        //recursively check and modify the right child
                checkNode(RCS.left, current.left);
                        //recursively check and modify the left child
        }else if (RCS.right==RCS){
                Node temp = current;
                while (temp.right!=temp) temp = temp.right;
                Node temp2 = temp;
```

```
                while (temp2.left !=temp) temp2 = temp2.left;
                temp2.left = current;
                temp = current;
                while (temp.right !=temp) temp = temp.left;
                current.left =temp.left;
                current.right = current;
                //cut off the child of current    when RCS is leaf, current node should
                //be corresponding to the leaf of the CS.
        }
        // when current is leaf, do not need to do anything
}

public double convertTwoDigitsToDouble(int firstInt, int secondInt){
        return firstInt + secondInt * 0.1;
}


//the methods below are used to decode a PPCT to an integer representation.
public int decodeMethod(Node top){
        int l = getLeafNum(top.right);
        int sum = 0;
        for (int j = 1; j< l; j++){
                sum+=((Integer)getCat().elementAt(j)).intValue();
        }
        return cNum(top.right)+sum;
}

private int cNum(Node top){
        if(top.right == top) return 0;
        return cNum(top.left)*((Integer)getCat().elementAt(
                getRightLeafNum(top))).intValue()+(cNum(top.right))+(
                minInt(getLeftLeafNum(top),getRightLeafNum(top)));
}

private int minInt(int left, int right){
```

```
        if(left==1) return 0;
        return minInt(left-1,right+1)+(((Integer)getCat().elementAt(
                left-1)).intValue()*((Integer)getCat().elementAt(right+1)).intValue());
}

private int getLeafNum(Node top){
        if (top.right == top) return 0;
        Node left,right;
        int num=0;
        left = top;
        right = top;
        while(right.right!=right) right = right.right;
        while(left.right !=left) left = left.left;
        while(left!=right) {
                right=right.left;
                num++;
        }
        return num+1;
}
private int getLeftLeafNum(Node top){
        if (top.right == top) return 0;
        if(top.left.right == top.left) return 1;
        return getLeafNum(top.left);
}
private int getRightLeafNum(Node top){
        if (top.right == top) return 0;
        if(top.right.right == top.right) return 1;
        return getLeafNum(top.right);
}

public static void setupVec(){
        cat = new Vector();
        cat.setSize(MAXLEAF + 1);
        cat.setElementAt(new Integer(0), 0);
        cat.setElementAt(new Integer(1), 1);
```

```
        for(int i = 2; i <= MAXLEAF; i++)
        {
                int b = ((Integer)cat.elementAt(i - 1)).intValue()*(2 * (2 * i - 3))/i;
                cat.setElementAt(new Integer(b), i);
        }


        mid = new Vector();
        mid.setSize(MAXLEAF + 1);
        mid.setElementAt(new Vector(), 0);
        mid.setElementAt(new Vector(), 1);

        for(int j = 2; j <= MAXLEAF; j++)
        {
                Vector temp = new Vector();
                temp.setSize(j);
                temp.setElementAt(new Integer(0), 0);
                for(int k = 1; k < j; k++)
                {
                        int b = (((Integer)cat.elementAt(k)).intValue()*
                                ((Integer)cat.elementAt(j - k)).intValue())+
                                ((Integer)temp.elementAt(k - 1)).intValue();
                        temp.setElementAt(new Integer(b), k);
                }

                mid.setElementAt(temp, j);
        }

}
public static Vector getCat(){
        if (cat==null) setupVec();
        return cat;
}
private static Vector cat,mid;
```

```
        public static int MAXLEAF = 20;

}
```

Bibliography

[Brown 2000] A. W. Brown, Large-Scale, Component-Based Development. Princes Hall, 2000.

[Burke, Carini, et al. 1999] M. Burke, P. Carini, J.-D. Choi, and M. Hind, "Interprocedural pointer alias analysis," ACM Transactions on Programming Languages & Systems, vol. 21 (4), pp. 848-94, Jul 1999.

[Business Software Alliance 2001] Business Software Alliance, "Sixth annual BSA global software piracy study," <http://www.bsa.org/resources/2001-05-21.55.pdf>, undated, (Accessed: 6 Nov 2001).

[Byrne 1992] E. J. Byrne, "A conceptual foundation for software re-engineering," presented at Conference on Software Maintenance, Los Alamitos, CA, USA, 1992.

[Chikofsky and Cross 1990] E. J. Chikofsky and J. H. Cross, "Reverse engineering and design recovery: A taxonomy," IEEE Software, vol. 7 (1), pp. 13-17 1990.

[Chuan-Fu and Wen-Shyong 2000] W. Chuan-Fu and H. Wen-Shyong, "Image refining technique using digital watermarking," IEEE Transactions on Consumer Electronics, vol. 46 (1), pp. 1-5 2000.

[Collberg Page] C. Collberg, "Sandmark homepage," <http://www.cs.arizona.edu/sandmark/>, undated, (Accessed: 20 Jan 2002).

[Collberg 2002] C. Collberg, "Hacking sandmark," private communication, 10 Jan 2002.

[Collberg and Thomborson 1999] C. Collberg and C. Thomborson, "Software watermarking: models and dynamic embeddings," presented at POPL '99. 26th ACM SIGPLAN-SIGACT. Symposium on Principles of Programming Languages. ACM, New York, NY, USA, 1999.

[Collberg and Thomborson 2000] C. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation - tools for software protection," Computer Science Department Technical Report 2000-03, University of Arizona(USA), 2000.

[Collberg, Thomborson, et al. 1997] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Computer Science Department Technical Report 148, University of Auckland(New Zealand), Jul 1997.

[Collberg, Thomborson, et al. 1998a] C. Collberg, C. Thomborson, and D. Low, "Breaking abstractions and unstructuring data structures," presented at IEEE International Conference on Computer Languages, Chicago, May 1998.

[Collberg, Thomborson, et al. 1998b] C. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in Conference Record of POPL '98: 25th ACM SIGPLAN-SIGACT. ACM, New York, NY, USA, 1998.

[Collberg and Townsend 2001] C. Collberg and G. Townsend, "Sandmark system source code," <http://www.cs.arizona.edu/sandmark/sandmark-0.2-src.tgz>, undated, (Accessed: 10 Nov 2001).

[Collberg, Townsend, et al. 2001] C. Collberg, G. Townsend, and C. Thomborson, "SandMark - a tool for the study of software protection," private communication, Oct 2001.

[Curtis 1994] D. Curtis, "Software piracy and copyright protection," presented at Wescon/94: Idea/Microelectronics, New York, NY, USA, 1994.

[Davidson and Myhrvold 1996] R. L. Davidson and N. Myhrvold, "Method and system for generating and auditing a signature for a computer program," US Patent 5,559,884, Microsoft Corporation, Sep 1996.

[Dershowitz and Zaks 1980] N. Dershowitz and S. Zaks, "Enumerations of ordered trees," Discrete Math, vol. 31 (1), pp. 9-28 1980.

[EEGGS] EEGGS.COM, "The Easter Egg Archive," <http://www.eeggs.com>, undated, (Accessed: 6 Nov 2001).

[Gosling, Joy, et al. 2000] J. Gosling, B. Joy, and G. Steele, The Java Language Specification. Second ed: Addison-Wesley, 2000.

[Goulden and Jackson 1983] I. P. Goulden and D. M. Jackson, Combinatorial Enumeration. New York: Wiley, 1983.

[Hartung and Girod 1998] F. Hartung and B. Girod, "Digital watermarking of uncompressed and compressed video," Signal Processing, vol. 66 (3), pp. 283-301, May 1998.

[Heineman and Councill 2001] G. T. Heineman and W. T. Councill, "Component-based software engineering : putting the pieces together," Addison-Wesley, 2001.

[Hoffer, George, et al. 1999] J. A. Hoffer, J. George, and J. Valacich, Modern Systems Analysis and Design. Second ed: Addison-Wesley, 1999.

[Holmes 1994] K. Holmes, "Computer software protection," US Patent 5,287,407, International Business Machines, Feb 1994.

[Jha 2002] S. Jha, private communication, 18 Jan 2002.

[Jokipii] E. Jokipii, "Jobe - The Java obfuscator," <http://www.meurrens.org/ip-Links/Java/CodeEngineering/jobe-doc.html>, undated, (Accessed: 20 Feb 2002).

[Kahn 1996] D. Kahn, "The history of steganography," presented at Information Hiding. First International Workshop Proceedings. Springer-Verlag, Berlin, Germany, 1996.

[Kamin, Mickunas, et al. 1998] Samuel N. Kamin, M. Dennis Mickunas and Edward M. Reingold, An introduction to computer science using Java. WCB/McGraw-Hill, 1998.

[Kirovski and Malvar 2001] D. Kirovski and H. Malvar, "Robust spread-spectrum audio watermarking," in 2001 IEEE International Conference on Acoustics, Speech, and Signal Processing, Piscataway, NJ, USA, 2001.

[Klassmaster] Zelix Pty Ltd, "It's a 2nd Generation Java Obfuscator," <http://www.zelix.com/klassmaster/feature3.html>, 2001, (Accessed: 20 Feb 2002).

[Kouznetsov] P. Kouznetsov, "Jad - The fast Java decompiler," <http://kpdus.tripod.com/jad.html>, 20 Feb 2002, (Accessed: 20 Feb 2002).

[Lindholm and Yellin 1997] T. Lindholm and F. Yellin, The Java Virtual Machine Specification. Addison-Wesley, 1997.

[Low] D. Low, "Protecting Java code via code obfuscation," <http://www.acm.org/crossroads/xrds4-3/codeob.html>, 25 Jan 2001, (Accessed: 28 Jan 2001).

[Mann and Borusan 2001] S. Mann and A. Borusan, "Towards a component concept for continuous software engineering," <http://cis.cs.tu-berlin.de/Lehre/WS-0102/Sonstiges/esi-pages/Encoder_conception_tr5500.pdf>, undated, (Accessed: 20 Dec 2001).

[Martin and Kendall 2000] F. Martin and S. Kendall, UML Distilled. Second ed: Addison-Wesley, 2000.

[Moskowitz and Cooperman 1996] S. A. Moskowitz and M. Cooperman, "Method for stegacipher protection of computer code," US Patent 5,745,569, The Dice Company, Jan 1996.

[Palsberg Source] J. Palsberg, "JavaWiz source code," < http://www.cs.purdue.edu/homes/madi/wm/JavaWiz.tar.gz>, undated, (Accessed: 20 Jan 2002).

[Palsberg SWM] J. Palsberg, "Software watermarking," <http://www.cs.purdue.edu/homes/madi/wm/>, undated, (Accessed: 20 Oct 2001).

[Palsberg, Krishnaswamy, et al. 2000] J. Palsberg, S. Krishnaswamy, K. Minseok, D. Ma, S. Qiuyun, and Y. Zhang, "Experience with software watermarking," presented at 16th Annual Computer Security Applications Conference (ACSAC'00). IEEE Comput. Soc., Los Alamitos, CA, USA, 2000.

[Ping Wah and Memon 2001] W. Ping Wah and N. Memon, "Secret and public key image watermarking schemes for image authentication and ownership verification," IEEE Transactions of Image Processing, vol. 10 (10), pp. 1593-601, Oct 2001.

[Proebsting and Watterson 1997] T. A. Proebsting and S. A. Watterson, "Krakatoa: decompilation in Java (does bytecode reveal source?)," presented at Third USENIX Conference on Object-Oriented Technologies and Systems (COOTS). USENIX Assoc, Berkeley, CA, USA, 1997.

[Rational Software] Rational Software Corporation, "Rational Rose," <http://www.rational.com/products/rose/>, undated, (Accessed: 31 Jan 2002).

[Samson 1994] P. R. Samson, "Apparatus and method for serializing and validating copies of computer software," US Patent 5,287,408, Autodesk Inc., Feb 1994.

[Sommerville 1996] I. Sommerville, Software Engineering. Fifth ed: Addison-Wesley, 1996.

[Sun Microsystems REF] Sun Microsystems, "Reflection," <http://java.sun.com/j2se/1.3/docs/guide/reflection/>, undated, (Accessed: 20 Feb 2002).

[Tip 1995] F. Tip, "A survey of program slicing techniques," Journal of Programming Languages, vol. 3 (3), pp. 121-89, Sep 1995.

[Venkatesan, Vazirani, et al. 2001] R. Venkatesan, V. Vazirani, and S. Sinha, "A graph theoretic approach to software watermarking," presented at 4th International Information Hiding Workshop, Pittsburgh, PA, USA, Apr 2001.

[Wang 2000] C. Wang, "A security architecture for survivability mechanisms," The Faculty of the School of Engineering, The University of Virginia(USA), 2000.

[Webopedia Codec] webopedia.lycos.com, "The lycos tech glossary definition - CODEC" <http://www.webopedia.com/TERM/c/codec.html>, 25 Jan 1998, (Accessed: 15 Feb 2002).

[Webopedia DW] webopedia.lycos.com, "The lycos tech glossary definition - digital watermark" <http://webopedia.lycos.com/TERM/d/digital_watermark.html>, 25 Jan 1998, (Accessed: 15 Feb 2002).