The Department of Computer Science
The University of Auckland
New Zealand

A Constant Encoding Algorithm Which Tamper-proofs the CT-Watermark

Lei Wang

September 2006

Supervisors:

Clark Thomborson

A thesis submitted in partial fulfillment of the requirements of Doctor of Philosophy in Computer Science

Created: 5 July 2001

Last updated: 9 August 2001

Acknowledgement

Firstly, I would like to thank my supervisor, Professor Clark Thomborson. I could not have imagined having a better advisor and mentor for my thesis. I gratefully thank him for his guidance. Without his help, I could never finish my study. I would also like to thank all the members of Software System Group, especially to Dr Stephen Drape and Dr Jasvir Nagra, for their advices and friendship. I am also grateful to Barbara Thomborson for her help on grammars.

Contents

1 Introduction

1.1 Software Protection Techniques

With the emergence of giant software companies, software has become into a multi-billion business. However, a recent study has indicated that piracy is seriously cutting into their business — thirty-five percent of the software installed on personal computers world-wide was pirated in 2004 [1]. Besides pirating, reverse engineering and violating code integrity also jeopardize software properties. To fight against those illegal usages, computer scientists have developed some techniques to make software products hard to attack. Tamper-proofing, obfuscation and software watermarking are most commonly used tools to protect intellectual property of software[2].

Software tamper resistance is the art of crafting a program so that it can detect attacks from a potentially malicious attacker[3]. In this paper, tamper-proofing refers to the process to achieve tamper resistance. Tamper-proofing code must always detect modifications and activate the response mechanism. In fact, car alarms function very similarly to the tamper-proofing code. They detect touches on the vehicle and response by sounding alarms. A widely used tamper-proofing technique is to calculate a result from the protected software space(for example, using CRC check sum) and redo the calculation. If the result differs from the previous one, an attack is believed to have happened.

Obfuscation is the technique which transforms a program that is hard to understand but maintains its functional values [2]. A simple but non-trivial obfuscation method is to change names of variables into ones that contain less information, for instance, change "buttonA" into "d112".

As a relatively new tamper-proofing technique, software watermark was developed to prevent software piracy by proving ownership of the application.As a result, when an unauthorized use occurs the copyright holders of the software can prove ownership by extracting a secret message(such as a watermark) from an unauthorized copy[4].

1.2 Watermark basics

Software watermarks are categorized as static or dynamic. Static watermarks are embedded into the static representation of the program, i.e. into the data area or the text of the code of the program. Usually, static watermarks can be detected without running the program.

Dynamic watermarks are the watermarks that are built at run time. Specifically, with dynamic watermarking, we can embed the code which actually builds the watermark rather than the watermark itself into the program. The first dynamic watermark was proposed by Collberg and Thomborson at 1999 and they gave their initials to this watermark[5]. The name ”dynamic watermark” comes from the fact that these watermarked programs construct a special, recognizable, data structure representing a particular graph that serves as a watermark.

Several publications[6][2][7] have shown that the CT watermark has many advantages over the static ones, for instance: it can survive from most semantic-reserve transformations and it is hard to be removed by automatic attacks.

However, CT watermark alone cannot provide adequate defences against attacks because it also has vulnerabilities. The CT watermark can be distorted once the attacker locates the code that builds the watermark — simple changes of one statement could make the recognisor fail to return the correct value. Consequently, another layer of protection besides watermarking techniques is needed. Because tamper-proofing techniques are responsible for detecting and preventing code modifications, it is natural to consider applying them to protect watermarks.

1.3 Related Work

If consider published works that focuses on using tamper-proof techniques to protect watermarks, there are only a few publications:

In 1999 Collberg and Thomborson developed a watermark protection technique using the Java reflection mechanism[5]. This method verifies the intactness of the Java classes representing the watermark at run time. The authors pointed out that, this solution has a fairly obvious disadvantage in its lack of stealth. Subsequent research has addressed this weakness and advanced the stealth property of the CT watermark.

Palsberg introduced an approach to protect dynamic watermarks, by using opaque predicates to guard the watermark representation. Ideally, If the watermark representations are altered, the opaque predicates will switch the program flow into an error-making branch. Creating dependencies from the watermark structures to the original program(i.e., the one to be watermarked) is how Palsberg’s method tamperproofs watermarks.

In 2002, Yong He[8] came up with a method called the constant encoding algorithm to strengthen the resistance of the CT watermark. He presented a prototype design of the algorithm and successfully developed a codec which converts integers into PPCT structures and the other way around. In 2004 Thomborson etal[9] formalized the constant encoding algorithm and improved the resilience of the tamper-proofed program by introducing a real dependency between the watermark structure and the constants. The authors proved that the modified algorithm can make dynamic watermarks resilient to all static attacks and most dynamic attacks.

The two publications above have shown the constant encoding algorithm does provide the CT watermark considerable amount of protections. However, as there is no implementation of this tamper-proofing technique, it is hard to say that the proposed encoding algorithms can be fully realized as expected.In another word, the process of implementing different versions of this algorithm may reveal some interesting properties and underlying difficulties. Future research requires a prototype of the constant encoding algorithm. Such research should include considerations on the degree of resilience to dynamic attacks, the sensitivity of the response mechanism and so on. Without practical confirmations, most of the theoretical proofs are weak or unconvincing. Furthermore, practical experience with white-hat hackers serve as intelligent adversaries may reveal weaknesses we have not anticipated. Therefore, one of the objectives of this thesis is to build up a prototype that implements the algorithm.

During my research, while implementing a prototype, serval interesting and challenging problems have occurred. A serious amount of efforts has been made on exploring various related directions including feasibility analysis, adversary experiments and algorithm designing. I regretful report here that neither the time and my writing abilities allows to cover all the achievement in this thesis. As a result, it will mainly focus on introducing a tool called JMarkProtector which is the firsts implementation of constant encoding algorithms. This thesis also includes a careful discussion of underlying theories, usage direction of the tool, and a preliminary study on the usefulness of JMarkProtector, in aspect of tamper-proofing effectiveness.

1.4 Organisation

It is organised in the following way: Chapter2 covers CT watermark and Constant Encoding basics. chapter3 describes the key functions used in my prototype. chapter4 provides design objectives and an overview of the structure of the prototype ”JMarkprotector”. chapter 5 shows how to use JMarkprotector to tamper-proof java source code. At last, a simple evaluation is given at chapter 6.

2 CT Watermark and Constant Encoding algorithms

2.1 Introduction

This chapter introduces necessary software watermark knowledge and focuses on an implementation of the CT watermark algorithm. Following chapters will show that this information determines most of the design decisions of my tamper-proofing algorithm.

This chapter also discusses various constant encoding algorithms and describes an improved one that will be used in the prototype.

2.2 CT Watermark

As mentioned in the first chapter, in 1999,Collberg and Thomborson proposed a method of watermarking software which was hence referenced as the CT Watermarks[5]. The idea is to embed code which builds some special data structures representing a meaningful sign (the watermark) into the target program (the program to be watermarked). That means the CT watermark is formed along the execution path parts by parts. Therefore, The CT watermark is in the dynamic watermark category on strong contrast to those watermarks which are built before run time (Static Watermarks). This algorithm has a noticeable feature comparing to other watermarking techniques: it allows users to define a sequence of special input, for example, click on certain buttons and menus in certain order. The embedded watermark will not be built unless the correct input is given.

CT watermark algorithm uses a graph to represent a secret message. The graph can be built only when the predefined input sequence is given. To achieve the effect above, the algorithm has to handle the transformation from the secret message to the graph and vice visa. It also needs to generate the codes that actually builds the watermark graph and insert them into the right places.

2.2.1 Available Implementations

. In order to discuss properties of the CT watermark in depth, it is essential to put it in a context of a real implementation. More importantly, as the constant encoding algorithm aims to protect the CT watermarked program, we need an implementation to produce such an input. To the best of our knowledge, there are two implementations so far: JavaWiz and SandMark.

The CT Watermark Implementation in JavaWiz

Palsberg etc developed another researching tool called JavaWiz. In JavaWiz, the implementation of the CT algorithm does not have the Easter egg property which means the building of watermark tree structure does not depend on a predefined input sequence. As a result, the watermark will always be created in the heap. Because the extractor is distributed together with Javawiz, attackers do not even need to figure out a decode function to locate and retrieve the watermark. Instead, they may simply use the code provided in the extractor to test if an object in the heap is part of the watermark structure. Therefore, they can locate the watermark building code with minor efforts. Hence we believe that this implementation degrades the resilience of CT watermark against dynamic attacks. In this thesis, JavaWiz's CT watermark implementation will not be discussed and used any further.

The CT Watermark Implementation in Sandmark

SandMark is a tool that collects several software protection algorithms including code obfuscation, software watermarking and tamperproofing[10]. The rich set of these algorithms enables this tool to be useful for both doing research and practical use. Sandmark's implementation of CT watermark algorithm takes a jar file (the target program) as input and produces another jar file (the watermarked program) as output. There are four phases in watermarking/recognizing process.

In SandMark's implementation, there are four phases. Firstly, there is the annotation phase which is in charge of inserting annotation points. Annotation points are in fact simply function calls suck as CT.Watermark.mark().They are used to indicate to the watermark locations in the code where (part of) a watermark-building code can be inserted. Secondly, there is the tracing phase. As annotation points are used to indicate possible places, we need to run the code to pick up places that really are executed when the right input sequence is given. Thirdly is the embedding phase. In this stage, we first need to get the input (string or integer) as the watermark. Then it transfers the string into an integer. Then we can transfer the integer into a graph. What we are about to embed in the phase is the code that actually builds the watermark data structure. The code should be put into the place we found in the tracing phase.

Lastly is extracting phase. In this stage, this implementation uses debugging techniques intensively. It forces a breakpoint to the creation process of all instances. Then we can have a way to backtrack all the created instances so we need less effort to find out the data structure from the heap. It is important to note that only if the program runs through the same execution path with the one determined by the predefined input sequence, the watermark data structure can be built correctly.

Annotation

Firstly, there is an annotation phase which is in charge of inserting annotation points. Annotation points are in fact simply function calls such as CT.Watermark.mark. There are two forms of annotation points:

1. No parameters: CT.Watermark.mark; 2. Parameterized ones such as CT.Watermark.mark$inti$; CT.Watermark.mark$Strings$.

Other allowed parameters include char, byte, short or long value. They are overloading methods defined in the class

"sandmark.watermark.ct.trace.Annotator.sm". The reason for having various function declarations for the annotation points is to make the watermark forming and extracting more user input dependent. This will be explained with more details in the embedding and extracting phase. Annotation points are used to indicate to the watermark locations in the code where (part of) a watermark-building code can be inserted. In another word, the watermark building code will be inserted into the places where annotation points stay. However, it is possible for some places which are marked by some annotation points will be inserted watermark building code. That means annotation points are all the possible places where the code is going to be put while the code may not be inserted to some of the annotation points. This uncertainty will be reduced to 0 after the next phase. Therefore, annotation points are basically place holders for the watermarking code.

In this phase, the users are responsible for picking up the places to put annotation points. From my point of view, there are multiple reasons for involving human intelligence into this step: 1. It is very hard to select appropriate inserting places automatically "Appropriate" places mean (A). the places which are not performance critical. Apparently, any place inside a for-loop is not a good choice. (B).the places which are in an unpredictable execution path. For example:

```
if (Random() > 0.5) Insert annotation points.;
```

Because it is impossible to know whether or not the code in the if branch will be executed, the watermark will possibly not be generated correctly.

(C).the places which is of the owners' interest to protect. One example could be the place containing a smart algorithm. The owner may want to put a watermark there because firstly the watermark code itself could make the algorithm more obscure. Secondly and more importantly, it may still be used as an evidence to proof an infringement of copyright if the attacker ship the whole module into some other places if all the watermark building code happens to be in that module. Therefore, annotation points are very hard to decide.

Sandmark chooses the easiest way on this step: let the user take care of the selection of annotation points. That means the user is the one responsible for selecting desirable places for the annotation points. Because Sandmark also requires a Jar file as input in the following phase, in the step, the user should add annotation points into the source files, compile the code with Sandmark.jar in its class-path and pack all the compiled files and other resources into a jar file.

Based on similar reasons, The prototype I built for tamper-proofing also lets the user decide which constants to be encoded. It can be used as a tool to generate the annotated class files. Details will be shown in the next chapter.

Secondly, there is the tracing phase. As annotation points are used to indicate possible places, we still need to run the code in order to pick up places that are really executed and hence we can get a input sequence. So, In this step, Sandmark starts the input program as a sub process in debugging mode. In this mode, it is possible for the Sandmark to stop the execution at any break points, examine variables and step through the application. With this ability, in this phase, Sandmark runs the program and takes an input from user. For instance, the user may click on few buttons in some way. In reaction of this input, following the execution path determined, Sandmark keeps a record of all the annotation points that are right on the execution paths. Intuitively, this process is named "trace". Specifically, selected annotation points are stored in a file named "tracepoint.txt" which will be used in the embedding phase. To distinguish them from annotation points, I name them tracing points. Note: tracing points are always a subset of annotation points.

Four properties should be considered when selecting trace points from annotation points. 1. Uniqueness. 2. Reproducibility 3.Efficiency 4. Input dependency. Recall that there are several types of annotation points: differently parameterized function calls. Uniqueness means that pair of (Location l, Value v) should appear only once. Reproducibility refers to the property that every time the correct user input is given, the graphs representing the watermark will be built correctly. Apparently, reproducibility is essentially the correctness of the watermark building and extracting. As stated before, in the current implementation of Sandmark, this property might be affected by the user's selecting of annotation points. Selected mark places should not be in an execution path which will be executed multiple times. This concern denotes the property of efficiency. The selection of marked places should also give priority to those which are strongly determined by the user inputs. We call this property input dependency. In the tracing phase, Sandmark does not make any effort to assure the reproducibility, efficiency and input dependency.

Thirdly is the embedding phase. In this stage, Sandmark first gets the watermark(a string or integer) from the user.If the user enters an integer, we can transfer the integer into a graph using a codec. If the user enters a string, Sandmark needs to convert the string into an integer first then conducts the encoding. What we are about to embed in the phase is the code that actually builds the watermark data structure. The code will be put into the place we found in the tracing phase. The method encoding integer (or string) into graphs can also be used in the tamper-proofing algorithm and it is encouraged to use the same approach in order to get resembling code. In the embedding process, there are four kind of graphs can be chosen to encode constants. They are, radix graph, permutation graph,reducible permutation graph and planted plane cubic tree. This thesis will focus on the last one only. Users also have other controls on this embedding phase: they may chose a storage type for storing the reference of the root for each subgraph. They also can choose The number of subgraphs, the class representing the node and if using node cycling to protect the graph against node splitting attacks. Here we highlight the option of ”subgraphs”. During the embedding phase, Sandmark will by default split the watermark graph into 2 subgraphs of similar sizes. The purpose of splitting is to decrease density of the appearance of the ”new” operations(dynamic memory allocation command in java), so that the place where the watermark building code inserted becomes less obvious. The number of subgraphs actually determines the code insertion together with the tracing points.

For instance, if there are 5 tracing points found in the previous stage where the subgraph number is two, that means the watermark building code would be only put in the first two of the tracing points. However, if the subgraph number is 4 (set by the user) and the number of tracing points is 1, all the subgraph building code will be inserted at the same place sequentially. In this case, the purpose of spreading out the ”new” operations will not be satisfied and we need some extra code and space to merge the split graph into one graph. Therefore, the number of subgraphs should not be greater than the number of tracing points. If the number of subgraphs is much less than the number of tracing points, the dependency of the watermark building to the secret input sequence might be weaken or even totally ruin.

In the code blew, there are three trace points in the main method marked as trace point 1, trace point 2 and trace point 3. If all the three trace pints are filled the watermark graph building code, the whole watermark will depend on the input sequence, *56, * means an integer whose value is greater than 2. So, that means only if a input in the format of *56(such as 356) is given, the watermark tree will be built in the memory stack. However, if the subgraph number is set to be 1, only the trace point 1 will be filled by a watermark building function and there would be no dependency between the watermark building and the input sequence, because the trace point 1 is not determined by any input.

```
public static void main(String[] args) { trace point 1;
if(args.lenght != null &&(args[0]>2)) trace point 2;
if(args[1]!=null && args[1]==5)trace pint 2; if(args[3]!null &&
args[3]==6)trace pint 3;}
```

In [10], It is said that if the node class is not given, Sandmark will choose one class that most resembles the watermark class. By examining the code, I found that it simply picks the class with most number of new calls pointing to it. Here we highlight the point that by specifying the class that will serve as node in this step, we can make sure that the node class we use in the tamper-proofing process is the same one that we use in the watermarking process. Details will be given in the next chapter.

The last step is extracting. In this phase, Sandmark will start the watermarked jar file in the debug mode and monitor all the new object creating operation by inserting a breakpoint into the constructor of every class. Sandmark uses a circular buffer to store those new created objects. Based on the observation that all the root class for the watermark will be created last, to extract the watermark graph Sandmark tests each item in the circular buffer in reverse allocation order[10]. Speficically, By examine the source code, we found that in this step, Sandmark simply tries to apply all its decoding function to the last 200 objects stored in the buffer if the graph type is not given.

For a formal explanation and the ease of theoretical analysis in the later context, a mathematical representation of the above watermarking process is introduced here. This chapter will keep accordance with the notations in[11].

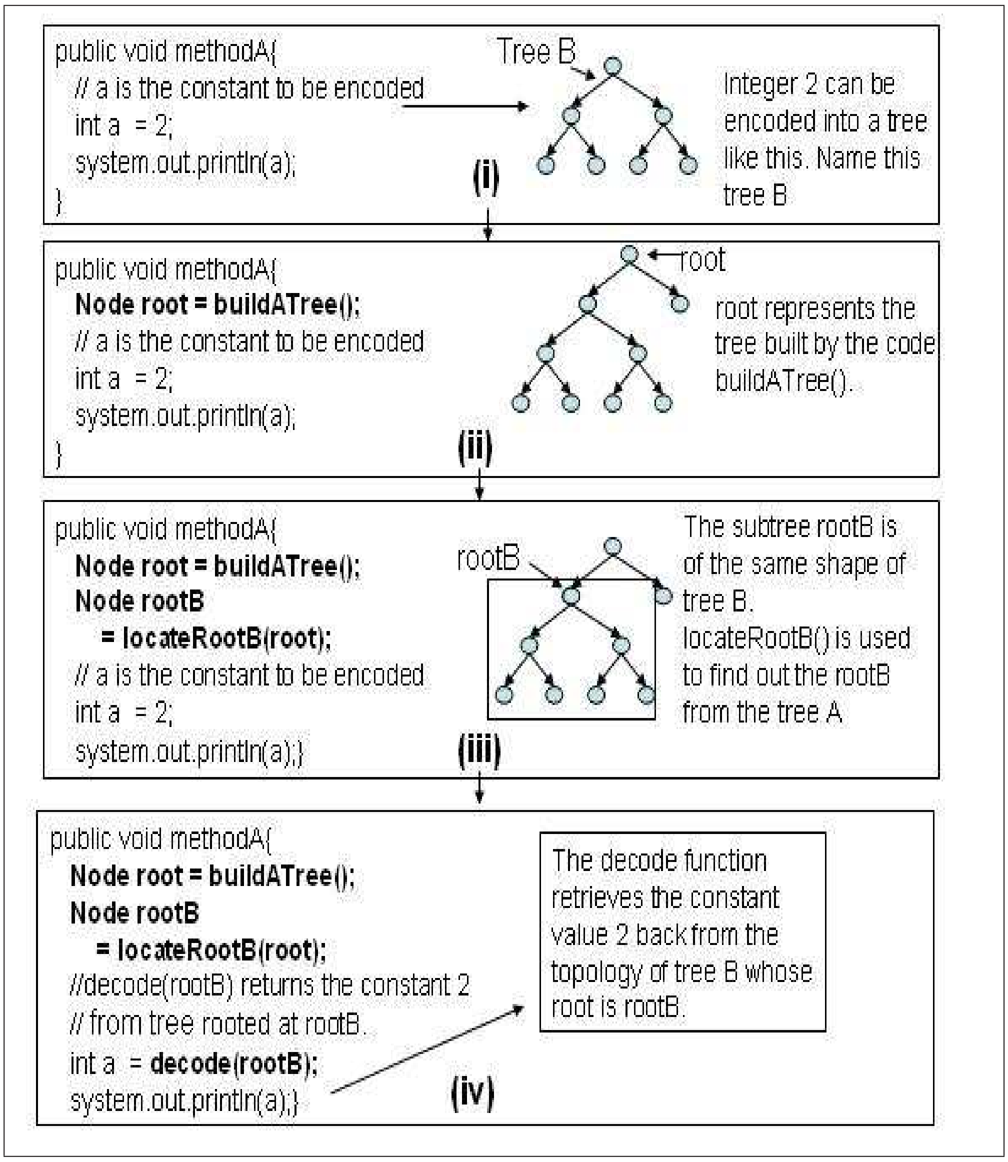
P is the set of all java programs. I is the set of all possible inputs for java programs. G is a set of graphs. Let w is a dynamic watermark which can be represented by a graph in G. Letter e refers to the encoder which maps N (the set of integers) onto G while d represents the decoder which is the inverse function of e. So, the function pair (e, d) is a graph codec. The watermark embedder E uses the encoder function e to compute the graph w = e(x) which corresponds to some desired watermark integer x. This graph is then embedded into the program P. More precisely, what is embedded in the program is some representation r(w) of the graphical watermark w, where r is a function r : G ! S that maps graphs in set G onto the set S of data structures that may be used by programs in P.

2.3 Constant Encoding Algorithm

2.3.1 Algorithm 1

In 2002, Yong He proposed a method called the constant encoding algorithm to tamper-proof the CT watermark[he02tamperproofing]. As the name indicates, The basic idea is to encode constants into some other forms and decode their value back when it is needed. It works in this way: constants are replaced with function calls. These functions first build graphs(the constants encoded to) and then decode the graph to return the constant value back. The idea can be shown in Figure 2.1.

Recall that in the embedding phase of watermarking, integer and string (the watermark) can be encoded into graphs. So, we can say that by using similar encoding functions we can make the graph representing the constant highly resemble the graph representing the watermark. As a result, if the attacker attacks all the code that looks like the watermark building code the constant value will be changed and hence the semantic of the software may be changed. It is worth noting that there is no real dependency between the constant value and the watermark graph. All the difficulties that this tamper-proofing technique may bring to the attacker entirely lie on an assumption that the attacker cannot confidentially distinguish the constant tree and the watermark tree. Hence the watermark code is independent to the temper-proofing process described here, the algorithm does not require that the input application has to be watermarked. It allows to "tamper-proof" the watermark that has not been inserted into the application and then let the watermark



```
public void methodA{
  // a is the constant to be encoded
  int a = 2;
  system.out.println(a);
}
```

```
public void methodA{
  Node root = buildATree();
  // a is the constant to be encoded
  int a = 2;
  system.out.println(a);
}
```

```
public void methodA{
  Node root = buildATree();
  Node rootB
    = locateRootB(root);
  // a is the constant to be encoded
  int a = 2;
  system.out.println(a);}
```

```
public void methodA{
  Node root = buildATree();
  Node rootB
    = locateRootB(root);
  //decode(rootB) returns the constant 2
  // from tree rooted at rootB.
  int a = decode(rootB);
  system.out.println(a);}
```

Integer 2 can be encoded into a tree like this. Name this tree B

root represents the tree built by the code buildATree().

The subtree rootB is of the same shape of tree B. locateRootB() is used to find out the rootB from the tree A

The decode function retrieves the constant value 2 back from the topology of tree B whose root is rootB.

Figure 2.1: A simplified work process of Algorithm 1

system produce the tamper-proofed, watermarked code.

We call Yong's method Constant Encoding Algorithm 1, or algorithm 1 if there is no confusion.

He assumed there is a Watermarking System(like SandMark) that can search for encodable constants from a source file, insert generated code into right places and watermark the final code. He claimed that attackers can not successfully distinguish the tree structure embedded during the tamper-proofing process and the watermark tree by using simple attacks.

2.3.2 Work Flow of Yong's algorithm

Step1. Accepts a list of constants. Small integers are preferable because they are easy to transfer to PPCT graphs.
Step2.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 the input constants. According to a statistical study in[8], a 100 leaves PPCT tree is usually enough for most of natural numbers.

Step3. For each constant, if it is too large to be encoded into a compact graph structure, the encoder will turn it into a suitable format for encoding. For example, X = a*b, if a and b are similarly big, this will decrease the complexity in a square scale. We can encode both a and b now with a much less cost. An alternative solution is to simply avoid encoding such constants.

Step4.The encoder generates codes to invert the process of step3, so that the original constant can be obtained at run time. For example decode that a and b back and get the multiple which equals to X in step3.

Step5. Encode the constant into the data structure used to build the watermark (PPCT in the example).

Step6.The encoder generates code to invert the process of step5, so that the value of the constant can be decoded from the graph structure at run time.

Step7.The encoder searches for a constant substructure in the constant tree, of the same shape of the constant graph. If the substructure is found, encoder records the location of the root of the constant structure, then starts to process the rest of the integers.

Step8 If a matching constant substructure is not found in step7, the encoder adds nodes to the constant tree so that it contains the required shape.

Step9. The encoder generates code to derive a reference of the substructure s. This reference is called , which supplies enough information to find the root of the substructure found in step8. This reference to the root is feed into the code generated in step6, so that the constant can be retrieved at run time.

Step10. The encoder generates code to form a final decoding function which firstly calls the code generated in step4, step6 and step9. This method takes constant tree ct and the as the input and outputs the constant value. As expected, this step finally produces the decode function.

Step11. The encoder exports the decoding functions for all constants and the code for gene rating the constant tree.

2.3.3 Properties of Algorithm1

1. The input program does not necessarily need to be watermarked. This algorithm simply encodes some constants in the input program. So the watermarking process can be done after the encoding.

2. This algorithm can be implemented on source code level or byte code(class file) level. All the steps: Chose constants, replace constants with function call and code insertion can be done on source code or byte code.

3. As there is no obvious constrains on the size of constant trees, this algorithm can handle a large range of constants.

4. There is no dependency between the watermark building code and the rest of the application even after the tamper-proofing. The effectiveness of the protection depends on the assumption that the attacker can not distinguish the watermark building code and the constant tree building code.

2.3.4 algorithm 2

In a later paper published in 2004, Clark and etc[9] proposed another algorithm which creates a real dependency between the constant value and the watermark tree structure. Instead of creating new tree, this algorithm uses the watermark tree as a base to search for subtrees that match the shape of constants which we want to encode. As a result, even if the attacker successfully locates the watermark, he will not able to remove or modify it without risking changing the constant values. This technique also has been patented at [12]

When I tried to implement this algorithm, I discovered some properties worth noting : 1. The range of encodable constants depend on the watermark value. For instance, the tree representing a integer 6 cannot be found in the watermark tree when the inserted watermark is 2, as shown in the pic2.2 On contrast to algorithm 1, the watermark tree can not be modified for matching some conditions, as any changes on the watermark tree might change the watermark value. For Algorithm 1, we can do whatever is necessary to the constant tree as it is irrelevant to the watermark value or the any other part of the application, which means there is much less constraints of encoding constants comparing to algorithm2. 2. This algorithm can only be done in byte code level. As this algorithm requests a watermarked application as input and the output CT-watermarked program by Sandmark is a jar file which contains a batch of class files, it is easier to work on byte code level. Actually, from the watermarking process described before we know that Watermarking process requires to run the input application and monitor and manipulates the runtime statues of the application in the tracing and embedding phase. These operations cannot be done in source code. So, we can say that even using some other CT-watermark implementations the output application will still be compiled files(byte code). As there is no guarantee that all the compiled files can be decompiled into java source code, even though it is relatively easier to do so in java, we can conclude that the algorithm 2 can only be implemented on byte code level. 3.There are real dependencies with the watermark and the constants. Attacks on the watermark building code may result in changes or errors in the decoding function.

2.3.5 My algorithm

As shown in the previous section, algorithm 1 does not create dependencies between the watermark and the constants. Once the attacker figures out the location of the watermark building code, this tamper-proofed application will be no saver than untamper-proofed application. Algorithm 2's main disadvantage is that if the watermark tree does not have a subtree which is of the same shape of a constant tree, the constant cannot be encoded.

Here I bring a new methodology for the constant encoding technique: use a combination of algorithm1 and algorithm2. The idea is to maintain the advantages of the two algorithms while decrease the disadvantages each one.

The new algorithm takes a java source code as input. Let the user specify which constants to encode and what the watermark value will be. Then the algorithm builds the watermark tree and constant trees.For each constant tree, the algorithm tests if the watermark tree contains a substructure which is of the same shape of the constant tree. If it is true, store the constant in an array. For all of these constants that cannot be encoded by using the watermark tree, apply algorithm one to them. After this step, we will get an encoded java source file.

Then the source file needs to be watermarked. As introduced in the watermarking section, we need to annotate the source file, compile it and then make a jar file which is the input to the Sandmark . The Sandmark system could take the jar file and outputs a partially-encoded watermarked jar file. Then apply algorithm 2 to the jar file to encode those constants that we know can be encoded. Without practical proof, we can see that this new algorithm can create real dependencies between the watermark and the constants while still has a large range of encodable constants. Besides this, we can see that this algorithm uses source code transferring(apply algorithm 1 on source code) as well as byte code manipulation (algorithm 2 works on a watermarked jar file). That means even the attacker could locate the constant decoding functions by searching for bytecode(Bytecode is the compiled format for Java programs[13])which are not generated by standard java compiler, there would be still some inserted functions that cannot be detected as they are generated from compiling a java source file.

It is relatively easy to see that the new algorithm contains two independent parts. The first one takes a java source code and outputs encoded and compiled files. The second part takes as input the watermarked and partially encoded jar file. Because I have not completed the implementation of the second part of the algorithm, this thesis will focus on the implementation of the first part. The difficulties and unexpected problems during the implementation of part2 will not be covered as well, also because of the time limit.

3 Algorithm

3.1 Introduction

This chapter describes some key functions used in my prototyping program. Before getting into details , it is essential to define and explain some terms uniformly. Most of these definitions keep in accordance with Yong's work[8].

Encoding: As defined in [8]:

"Depending on the context, encoding has two meanings . In restricted contexts,it refers

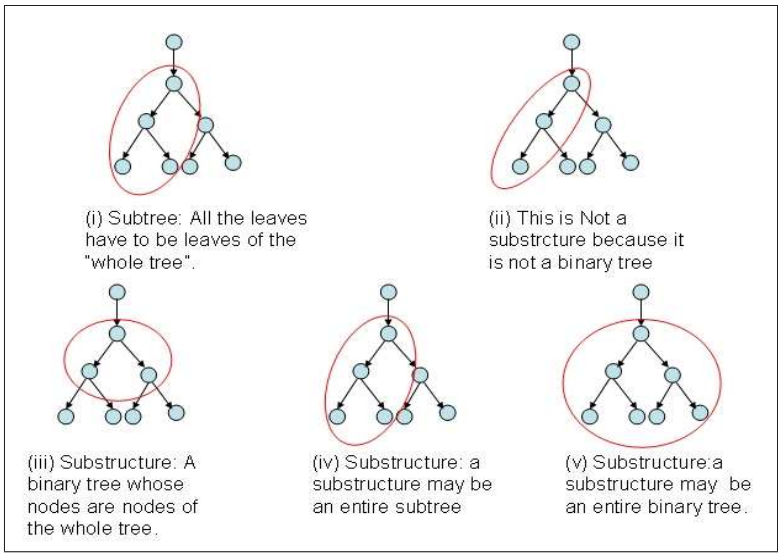

(i) Subtree: All the leaves have to be leaves of the "whole tree".
(ii) This is Not a substrcture because it is not a binary tree
(iii) Substructure: A binary tree whose nodes are nodes of the whole tree.
(iv) Substructure: a substructure may be an entire subtree
(v) Substructure:a substructure may be an entire binary tree.

Figure 3.1: Examples of binary tree, subtree and substructures

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."

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

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

sub-graph: sub-graph refers to a graph whose nodes and edges belong to another graph. Subtree: Subtree is a binary tree which is a part of another tree(the whole tree) and all its leaf nodes have to be the leaf nodes of the whole tree. In another word, it contains a complete branch of another tree. See 3.2(i) for an example. Substructure: refers to any binary tree that is a sub-graph of another binary tree. Note that substructure can be a subtree but not necessarily a subtree. So, in the figure3.2, (iii)(iv)(v) are all legal substructures. (ii) is not a substructure because such a sub-graph is not a binary tree.

Constant Substructure: if a substructure can be encoded from a constant, it is a constant substructure. Constant Tree: Constant tree refers to the "whole tree" which contains all the constant substructures. E-Constant: a constant chosen to be encoded from a candidate program

The complete constant encoding algorithm should provide methods to handle at least following problems:

1.For every E-constant, generate the Constant Graph which can be used to retrieve the constant. 2. Generate a Constant Tree 3. For every Constant Graph built in 1. Search the Constant Tree to see if the it contains a substructure or a subtree that matches the constant graph. If true, generate and record information that locates the substructure/-subtree. If false, change the Constant Tree or give up the constants. 4. Generate code that builds the Constant Tree. 5. For every constant whose corresponding graph has been found in the Constant Tree during step 3, generate code that retrieves the constant using information recorded in 3.

For question one, a codec is all it needs. In combinatorics, there are several classical methods to enumerate graphs especially tree structures efficiently[14]. Collberg and Thomborson summarized four enumerations of graphs. They are: (1) Planted Plane Cubic Plane Cubic Tree encoding. In the thesis, for the abbreviation "PPCT" is used for convenience.

In 1983, Goulden and Jackson gave a description of PPCT: (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. In the thesis, origin refers to the node that is distinguished (the root node in the description above) while root refers to the right child of origin. In figure3.2, the origin is the node in black. Please note that the PPCT structure we used in this thesis differs the definition given by Goulden and Jackson at one point: the root is not monovalent. see (i), all the leaf nodes have its right pointers point to themselves. The origin may have another outgoing edge points to another vertex, such as the right-most leaf, for some calculation convenience. There are situations as shown in graph 2 in 3.2(ii) that leaf nodes have only one child. We still call these two PPCTs.

Encoding and Decoding. Yong mentioned three types of codecs and focused on the Catalan Leaf-Oriented Conversion. The original idea came from[15][7] which shows that PPCT graphs can be effectively enumerated by the Catalan numbers Cn. Cn can be

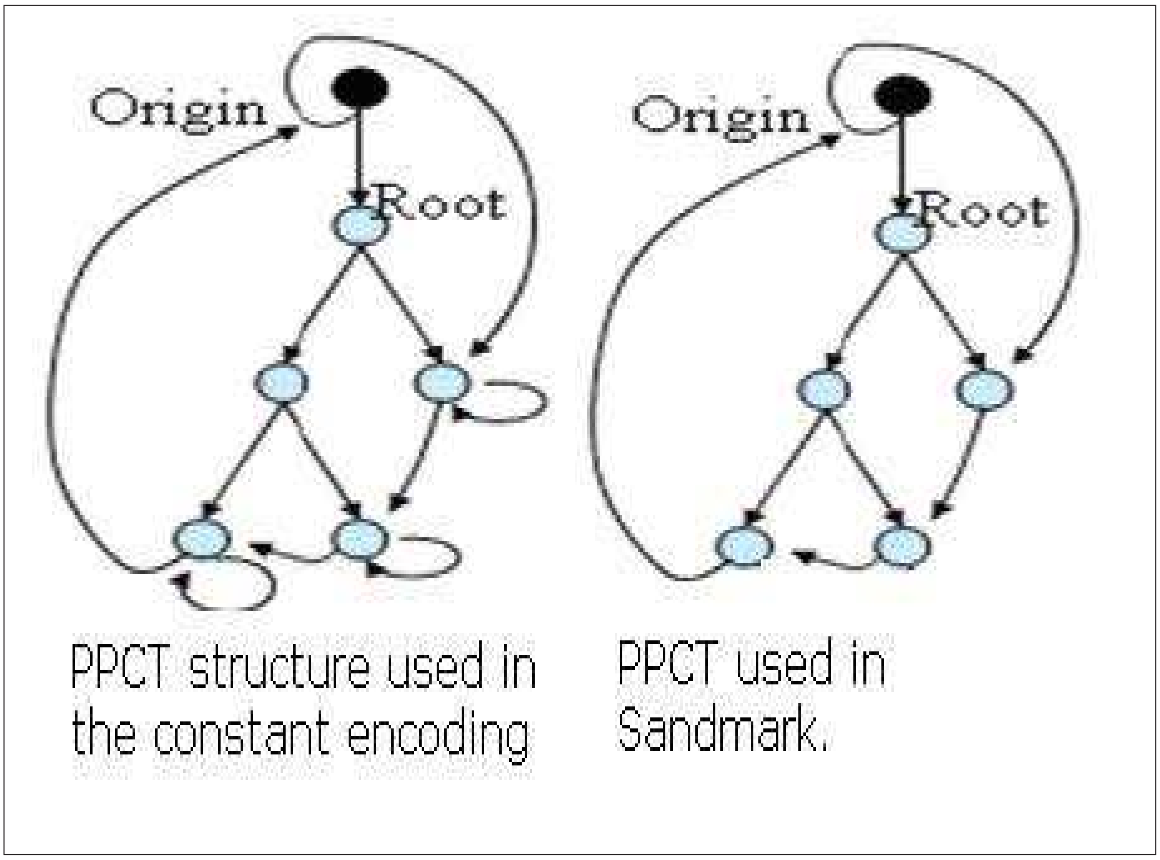


Figure 3.2: Examples of binary tree, subtree and substructures

calculated by the following formula: For example, the following sequence refers to the 1st to the 15th Catalan numbers [1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440]

Comparing to Yong's work, I have provided several extensions and completions during the development. All of the functions are placed in the util package. Although I do not all of them in the prototype, it is useful to put them together so other researchers can simply import them as a single library. All of the functions metitioned in this chapter are tested over large test cases.

Extension1.

We can see that the encoding function needs the number of leaves as an input parameter. Given the integer that to be encoded, we can calculate the minimum number of leaves the encoded PPCT tree may have. This can be easily done by this function called findMinLeafNum:

```
findMinLeafNum(int i){
    int n ;
    for(n = 1; i>c(n)&&i<c(n+1);n++);
    return n;
}
```

The encoding function then can be changed to:

```
encode(int i ){
    return encode(findMinLeafNum(i), i);
}
```

Sandmark uses a similar technique to make the encoding function a one to one map from the integer set to the PPCT set. By using this method, One interesting observation is that the decode(PPCTree origin) function does not need to know the number of leaves as it can be easily calculated by the given tree.

Extension 2. Using boundary to separate a substructure from a constant tree. He Yong provided a method called masking to achieve the same goal. Boundary refers to a set of nodes which belong to a constant tree but not belong to the constant substructure and are all pointed by one of the leaf nodes of the constant substructure. A huge advantage of boundary method is saving of the space.

Extension 3. Using union operations to generate the constant tree. In He Yong's algorithm, the constant tree is generated randomly while I provide another way to generate the constant tree.

The advantage of my way is that every node in the generated constant tree belongs to at least one constant substructure. So, any change to the constant tree will cause changes to at least one constant value. However, this feature cannot prove this way is better than randomly generated constant tree as the later one may create more difficulties to the attacker to locate the constant substructure. But, it is good to have options.

Extension 4. Provide options to generate a constant tree for every constant. This extension has no much practical meaning because generating single constant tree for each constant wastes space and reduces the difficulty for attacking. It may let the attack have a chance to replace every constant value back one by one. The reason of keeping this extension is that it helps first time user understand how this algorithm works.

Completion 1.Tree searching algorithm. Although mentioned more than once, Yong never gave implementation to the tree searching algorithm. In fact, all people with average combinatorics knowledge can finish this algorithm, but it still takes time to test the correctness. My searching algorithm also supports the boundary technique. The complete algorithm is listed in Appendix A. It can automatically compute the boundary node lists if a matching substructure is found in the constant tree.

Completion 2. Constant finder function. In order to reduce the difficulty of finding appropriate constants, I leave this responsibility to the user. The user needs to mark the line which contains at least one constant value. So, my constant finder function will take every marked line to search for constants. It uses regular expression to locate the constants. Because we need to replace constants with function calls, the places of constants have to be recorded.

4

Overview of the prototype JMarkProtector

4.1 Introduction

This chapter gives a high level description of the prototype which is named ”JMarkProtector”. Firstly, design objectives are discussed with detail, which is then followed by an overview of the structure of the prototype. This chapter ends up with introducing the

Codec used in JMarkProtector, and its difference to the one used in Sandmark.

4.2 Design Objectives

Design objectives listed below basically determines all the design decisions that had been made during the implementation process. It does not only serve as the explanations for the choice of ways to build the prototype, but also highlights different options that other implementations may choose in the future.

While building JMarkProtector, there are three objectives need to be considered carefully during the whole development process. 1. Correctness 2. Effectiveness. 3. Short developing time

1. Correctness. Correctness has two standards.

A. First, it refers to a desired property which is after applying the transfer (adding the code to encode/decode constants), the output application should be semantically equivalent to the original one. Specifically, it can also be divided into two sub-objectives. First, the code manipulation should have no error. This is a necessary condition of correctness because, in an extreme case, if the resulting application cannot run, it makes no sense to discuss the correctness.Second, the return value of the decoding function has to be correct. This condition should be satisfied as long as the algorithm is correctly implemented in the prototype. Otherwise, there is no way to assure the resulted application is functional equivalent to the original one as Values of constants could be critical to the program behavior.

B. The watermark should be able to be retrieved after the tamper-proofing process.

2. Effectiveness. Effectiveness means that the Prototype should provide a certain amount of protections to the watermarks embedded in the input application. This objective partially comes from the observation that quit a few protection techniques fail immediately on the second round of attack. This prototype should provide some protections without jeopardizing the good features of CT watermarks. In another word, it should not introduce new threats to the watermarks.

3. Short developing time. Besides the two, there is another concern: the time needed to implement the prototype should not be longer than three months. For a roughly 12 months' Masters Program, the suggested time allocated to writing is at least Four months. The research in the literature and the attempts of using different approaches of implementation takes no less than two months. Because one of the major goals of the thesis is to evaluate the algorithm by conducting experiments using the prototype, the experimenting has to be allocated a fair large amount of time, two months. Therefore, three months developing time (including testing) is a constraint that I have to follow.

4.3 High Level Representation

JMarkProtector takes java source code as input and produces encoded, compiled and annotated .class files or source files based on user's choice. The produced files can be easily used to zip into a jar file which the Sandmark system can take as input. As shown in the figure below. There are seven packages in this prototype, which are: embeder, embeder.codeGenerator, ui, util, util.codec, viewer and the default package.

The embeder package is responsible for supplying four tasks: 1.find the constants to be encoded, 2.get a number sequence(such as 021034345) representing the constant tree, 3.generate the code according to the binary sequence and 4.embed the code into the application to be tamper-proofed. There are four classes in this package: ClassDefinition, ConstantFinder, Embeder, SequenceGenerator and two classes in the internal package embeder.codeGenerator: CodeGenerator and StringInt. The internal package embeder.codeGenerator is in charge of task 3. The class StringInt simply encodes strings into Big Integers.

The ui package provides the user interface. It uses the other packages a lot.

The util package contains 11 class files, 7 of which also included in its internal package util.codec. It provides common utilities which may be used by all the other packages or classes. The reason of creating this package is put together commonly used classes (such as node,nodeFactory) and common functions like codec functions, tree operations(tree search, union, intersection). Doing so can reduce coupling between this package and all the other packages. Moreover, if some other researchers want to use those code as a external library, they can simply importing a single package instead of the whole project.

The viewer package takes care of the tree visualization functionality. It draws PPCTs according to encoded integers/strings. This package contains five class files and they all came from He Yong's project with quit a few modifications.

The default package contains only a single file named JMarkProtector. This file does nothing but create a process starting the whole program. It is the file which contains the main method.

5

JMarkProtector in practice

5.1 Introduction

JMarkProtector is the tool which implements the constant encoding algorithm described in 2.3.5 at page 20. JMarkProtector is writhen entirely in java and can only be compiled by the JDK whose version is higher than 1.5.0. This chapter shows its user interface and how to use it by a real example.

5.2 User Interface

JMarkProtector has a carefully designed GUI to facilitate the use of constant encoding algorithm. Specifically, it enables the user to choose constants they want to encode, decide where in the source code to put function definitions and class definitions, insert annotation points for the watermark embedding in the future, compile and generate class files which is encoded and annotated and run the compiled code to examine the correctness of the output code. Here is a snapshot of graphic interface of JMarkProtector:

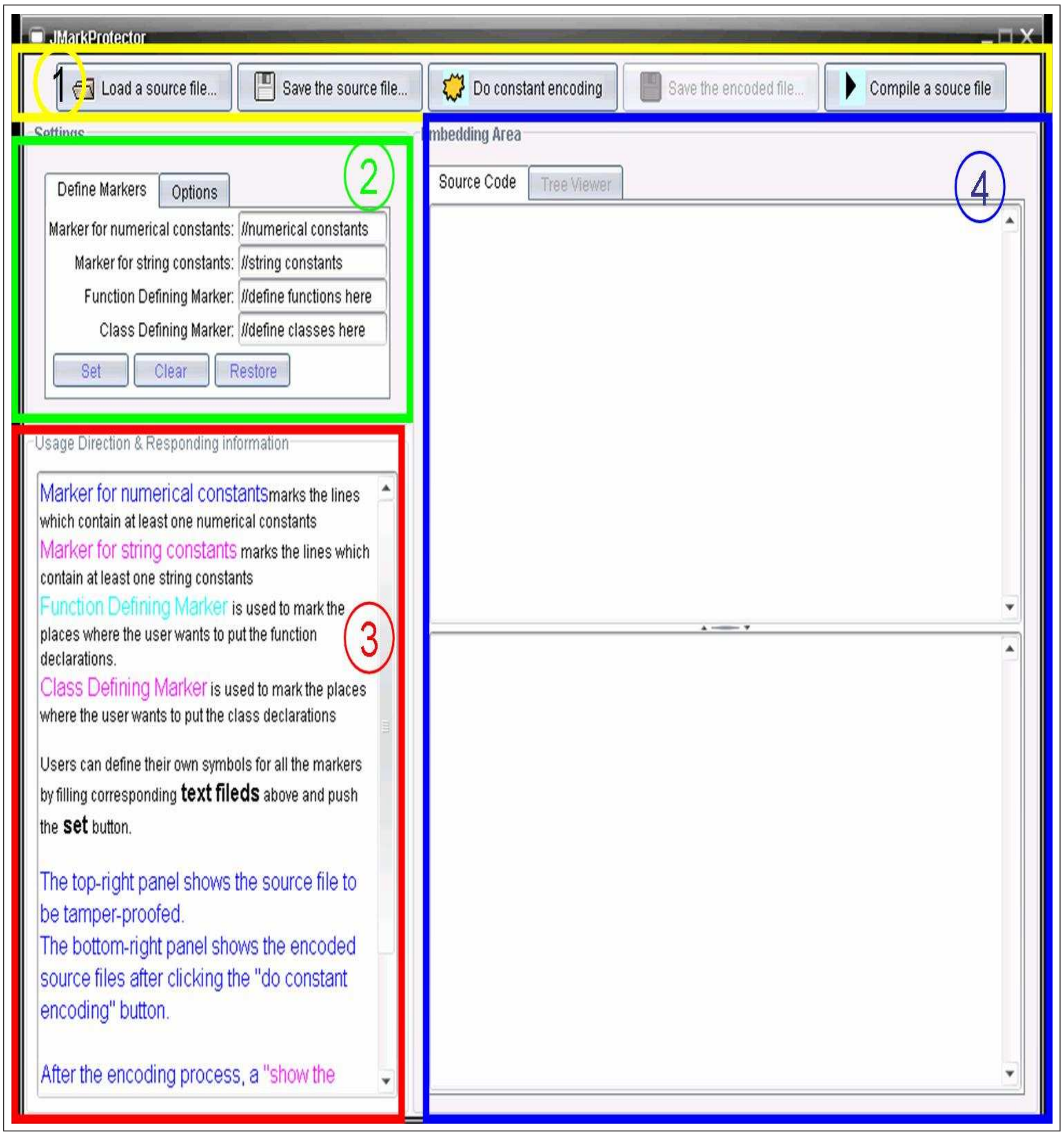
The whole user interface is comprised with 4 parts as shown in 5.1: 1. tool bar 2. setting panel 3. "embedding area" 4. usage direction and responding information panel.

1. The tool bar. The tool bar of JMarkProtector is composed of five buttons. These buttons provide most of the functionalities. As shown in the 5.2, these buttons are:

"Load source file", "Save the source file", "Do constant Encoding", "Save the encoded file" and "Compile a source file". As the name indicates, the first button allows the user to choose a java source file and load it into the source code text area in the "Embedding area" as shown in the figure5.1 labeled number 4. If the source file is modified, the user can save the changes made to the source file by pressing the "Save the source file" button. Button "Do constant Encoding" and "Save the encoded file" cannot be pressed (disabled) until a source file is loaded. "Do constant Encoding" button is used to tell JMarkProtector to perform the encoding algorithm to the source file loaded. The encoded source file will be shown in the text area at the bottom of "Embedding Area" once this button is pressed. The user can compile a source file by using the last button. The class path for the compile operation can be set in the "options" panel which will be introduced next.

The setting panel allows user to set a few options which control the encoding process. It has two panels: "Definition markers setting" panel(see figure5.3) and the "options" panel(see figure5.4).

The markers are used to inform JMarkProtector that where to insert the new generated code such as the constant retrieving function and some new class definitions. There are four kinds of markers: 1. The numerical constant marker By default, the marker is a



Marker for numerical constants: //numerical constants
Marker for string constants: //string constants
Function Defining Marker: //define functions here
Class Defining Marker: //define classes here

Marker for numerical constants marks the lines which contain at least one numerical constants
Marker for string constants marks the lines which contain at least one string constants
Function Defining Marker is used to mark the places where the user wants to put the function declarations.
Class Defining Marker is used to mark the places where the user wants to put the class declarations

Users can define their own symbols for all the markers by filling corresponding **text fileds** above and push the **set** button.

The top-right panel shows the source file to be tamper-proofed.
The bottom-right panel shows the encoded source files after clicking the "do constant encoding" button.

After the encoding process, a "show the

Figure 5.1: GUI of JMarkProtector

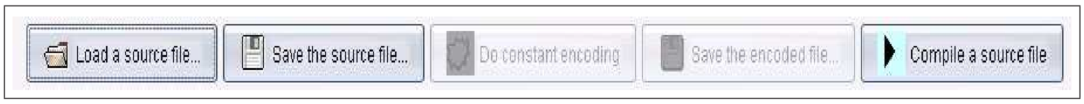
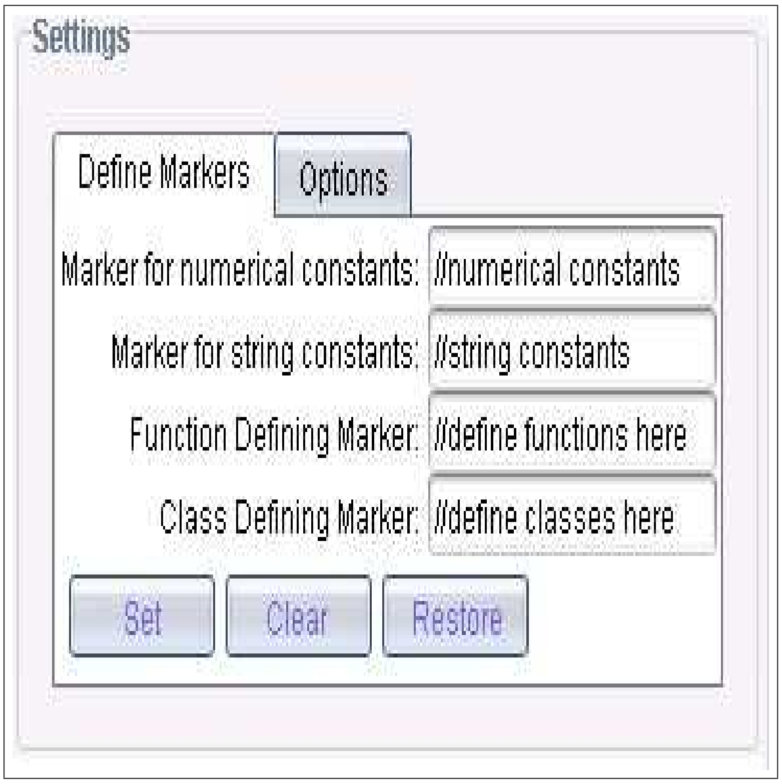


Figure 5.2: The tool bar

string "//numerical constants here". The user should write this string at the end of the statement which contains at least one numerical constant. For example, if the source code to be watermarked has a statement:


Settings

Marker for numerical constants: //numerical constants
Marker for string constants: //string constants
Function Defining Marker: //define functions here
Class Defining Marker: //define classes here

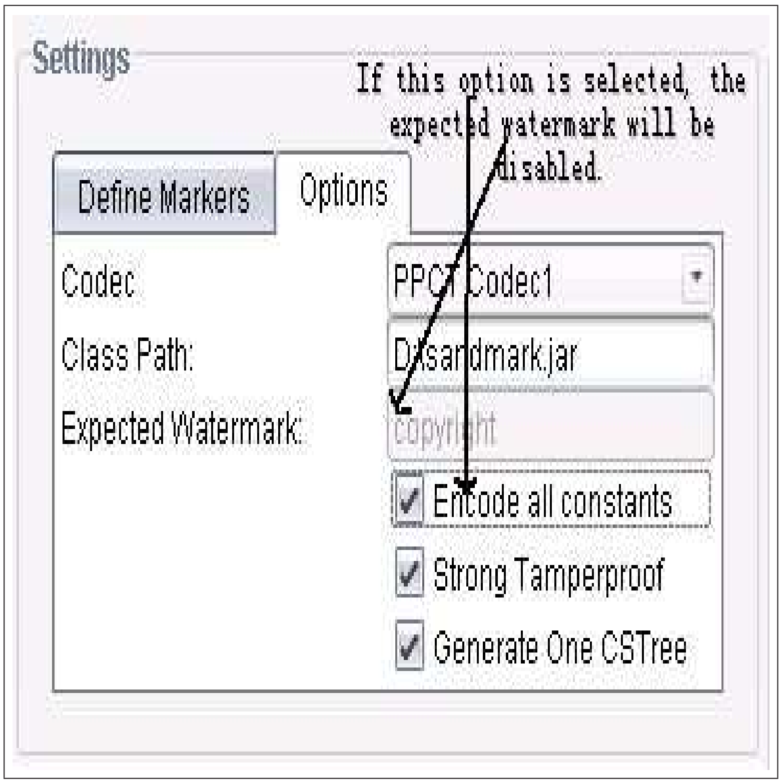
Figure 5.3: Setting Panel–tabbed panel ”define markers”

```
int counter = 100;
```

The user may wants to encode the integer constant 100. All he needs to do is to copy and paste the numerical marker at the end of this statement. So, it becomes:

```
int counter = 100; //numerical constants
```

Now the JMarkProtector knows which statement to search for E-constants. 2. The string constant marker Similar to numerical constant marker, string constant marker is used to inform JMarkProtector which statement contains string constants that need to be encoded. By default the string constant marker is ”//String constants here.” 3. Function defining marker This marker serves just as a place holder. The user should copy and paste the marker in the place where the new generated functions should be inserted. If the marker is placed in some inappropriate places such as in the middle of a keyword, or in another function, the output file will definitely not be able to pass the compiler. The generated functions are decoding functions:


Settings
If this option is selected, the expected watermark will be disabled.
Define Markers
Options

Codec: PPCT Codec1
Class Path: D:\sandmark.jar
Expected Watermark: copyright
☑ Encode all constants
☑ Strong Tamperproof
☑ Generate One CSTree

Figure 5.4: Setting Panel–tabbed panel "options"

```
private static int returnValue1(BogusWatermark[] ns) {
        BogusWatermark[] bounds = {ns[1], ns[2]};
        return new WMRetriever().getWmNum(ns[0], bounds).intValue();
}
```

BogusWatermark is the class used as the basic "node" in order to generate the whole tree structure. WMRetriever is another class that needs to be inserted in the source code. It does nothing but decoding the constant value back from a PPCT rooted at the node ns[0] as shown in the code above. The default value of this marker is "//define functions here" 4. Class definition marker: Class definition marker works similar to the function definition marker. The user is responsible to insert this marker in an appropriate place (for example, at the end of the source file, after the last curly brackets). The classes to be inserted differs if the user chooses different encoding options in the. Details will be given later.

The tabbed panel named "options" is shown in figure5.4:

There are 5 options the user can give different settings. 1. Combo box "Codec". By choosing different codec, the encoding process will use different encoding and decoding methods and hence the generated code will be different. The default codec is called PPCT Codec1 which has been described in the previous section. 2. Text filed "Class path". The user can specify the class path which is used to compile and run the output file after the encoding. Legal class path is comprised of a list of directories separated by a semicolon. For example: C:
myProject;c:
bins is a legitimate class path in Microsoft Windows systems. The counterpart of the same class path in Linux and Mac machines will be c:/myProject; c:/bins. In the figure above, the class path is set to be "D:
sandmark.jar". By adding this sandmark.jar file into the class path, JMarkProtector will be able to compile annotated source files. That means, if the user wants to produce annotated bytecode file, the path has to contain the sandmark.jar file. 3. Text filed named "Expected Watermark" is used to inform the JMarkProtector that what the watermark will be when the user watermarks this application. This filed will be disabled if the check box ("Encode all constants") under it is selected. So, if encode all constants is unchecked, and the expected watermark filed is filled, during the encoding process JMarkProtector will ignore these constants which can be encoded by the watermark tree. 4. Check box "Encode all constants". If it is checked, JMarkProtector will first encode the watermark value (specified in the "Expected watermark" filed) into a PPCT tree using the codec chosen from combo box "Codec". Then it gets all the constants from all of the marked statements. Encode each constant into a PPCT tree and search the watermark tree for a substructure which is of the same shape of each constant. If the substructure can be found, that means the corresponding constant can be encoded from the watermark tree. As mentioned in the previous chapter, this constant is better to be encoded after the watermarking process in "part 2" in order to create a real dependency between the watermark and the constant. Therefore, JMarkProtector will not encode these constants. For those constants which can be encoded into some substructures belonging to the watermark

tree, JMarkProtector will encode each of them. 5. Check box ”Strong Tamperproof”. If this check box is not selected, JMarkProtector will generate straightforward code. If it is checked, JMarkProtector will try to generate code which resembles the watermark building code. A typical watermark building code looks like:

```
// Watermarked by Sandmark, using PPCT codec, 1 subgraph, no cycled graph
public class Watermark {
    //This method builds up a PPCT tree which represents the watermark –
    public static void Create_12693857(Watermark sm$array[])
    {
        Watermark n7 = new Watermark();
        Watermark watermark = new Watermark();
        watermark.x$0 = n7;
        Watermark watermark1 = new Watermark();
        watermark1.x$0 = watermark;
        Watermark watermark2 = new Watermark();
        watermark2.x$0 = watermark1;
        Watermark watermark3 = new Watermark();
        watermark3.x$0 = n7;
        watermark3.x$1 = watermark;
        Watermark watermark4 = new Watermark();
        watermark4.x$0 = watermark3;
        watermark4.x$1 = watermark1;
        Watermark watermark5 = new Watermark();
        watermark5.x$0 = watermark2;
        watermark5.x$1 = watermark4;
        Watermark watermark6 = new Watermark();
        sm$array[7] = watermark6;
        watermark6.x$0 = watermark2;
        watermark6.x$1 = watermark5;
        n7.x$0 = watermark6;
    }

    public static Watermark[] CreateStorage_sm$array()
    {
        Watermark awatermark[] = new Watermark[8];
        return awatermark;
    }
    public Watermark x$0;
    public Watermark x$1;
}
```

If the user does not choose using strong tamperproof, if the tree building code for an integer 2 will be:

```
PPCTTreeNode[] nodes = new PPCTTreeNode[8];

for(int i = 0; i <nodes.length; i++){
    nodes[i] = new PPCTTreeNode();
}   nodes[0].left = nodes[3];
    nodes[0].right = nodes[1];
    nodes[1].left= nodes[2];
    nodes[1].right = nodes[3];
    nodes[2].left = nodes[4];
    nodes[2].right = nodes[5];
    nodes[4].left = nodes[0];
    nodes[4].right = nodes[4];
    nodes[5].left = nodes[6];
    nodes[5].right = nodes[7];
    nodes[6].left = nodes[4];
    nodes[6].right = nodes[6];
    nodes[7].left = nodes[6];
    nodes[7].right = nodes[7];
    nodes[3].left = nodes[7];
    nodes[3].right = nodes[3];
```

Apparently, these two tree building codes are different from each other in many places. Such as the watermark tree creates new node when it needs while the constant tree create all the needed nodes at one time (allocate memory for all the items in the nodes array).

If this strong tamperproof option is selected, the code which builds the constant tree encoding integer 2 will be:

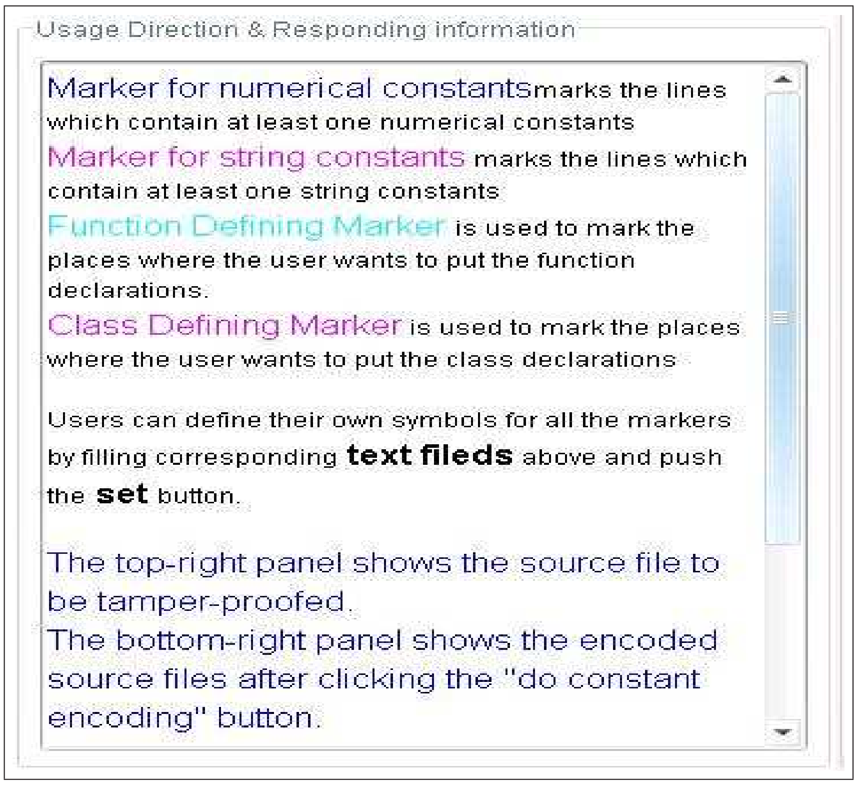
```
public static void Create_1(BogusWatermark sm$array[]){
    BogusWatermark watermark0= new BogusWatermark(); BogusWatermark
    watermark1= new BogusWatermark();
    watermark0.x$0 = watermark1;
    BogusWatermark watermark2= new BogusWatermark();
    watermark1.x$1 = watermark2;
    BogusWatermark watermark3= new BogusWatermark();
    watermark1.x$0 = watermark3;
    BogusWatermark watermark4= new BogusWatermark();
    watermark2.x$1 = watermark4;
    BogusWatermark watermark5= new BogusWatermark();
    watermark2.x$0 = watermark5;
    watermark4.x$1 = watermark0;
    watermark4.x$0 = watermark4;
    watermark5.x$1 = watermark4;
    watermark5.x$0 = watermark5;
BogusWatermark watermark6= new BogusWatermark();
watermark3.x$1 = watermark6;
sm$array[0] = watermark0;

BogusWatermark watermark7= new BogusWatermark();
watermark0.x$1 = watermark7;
watermark3.x$0 = watermark7;
watermark6.x$1 = watermark5;
watermark6.x$0 = watermark6;
watermark7.x$1 = watermark6;
watermark7.x$0 = watermark7; }
```

Apparently, the code here is more similar to the watermark building code. As the effectiveness of this tamper-proofing algorithm purely depends on the degree of similarity of the two pieces of code, we tend to switch on this strong tamper-proof option, although this method needs a little bit more space (30 bytes in this case) than the other. 5. Check box "Generate one CStree". This check box is to control whether or not generate the constant tree and decode all the constants from that tree. If it is not selected, JMarkProtector will encode each constant into a separate PPCT. If it is selected, algorithm 1 mentioned in the previous chapter will be applied: generate a constant tree and use its substructures to decode all of the encoded constants.

3. The usage direction and responding information panel is in charge of providing help information for the users to use this tool. The contents in this panel will be updated along with the user's operations. Basically, this panel has four statuses. The basic state shows usage information about the markers setting panel and the basic structure of JMarkProtector. See figure5.5.

When the user clicks on the encoding button in the toolbar, this panel will change its content into second state as shown in 5.6. In this status, the usage direction and responding information panel reports how many constants have been encoded and provides

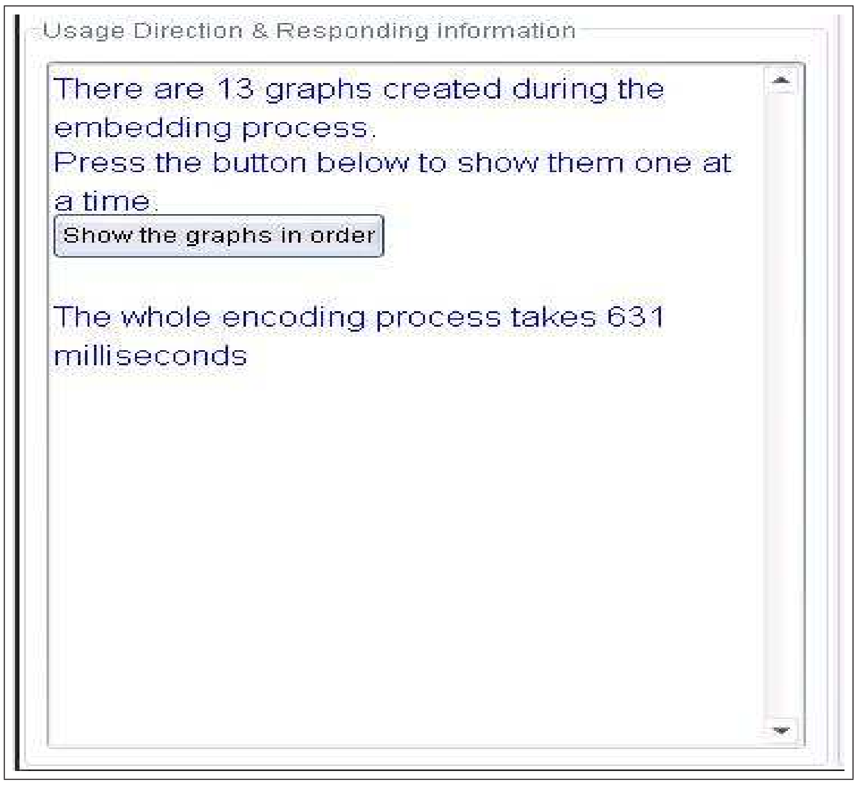


Marker for numerical constantsmarks the lines which contain at least one numerical constants
Marker for string constants marks the lines which contain at least one string constants
Function Defining Marker is used to mark the places where the user wants to put the function declarations.
Class Defining Marker is used to mark the places where the user wants to put the class declarations

Users can define their own symbols for all the markers by filling corresponding **text fileds** above and push the **set** button.

The top-right panel shows the source file to be tamper-proofed.
The bottom-right panel shows the encoded source files after clicking the "do constant encoding" button.

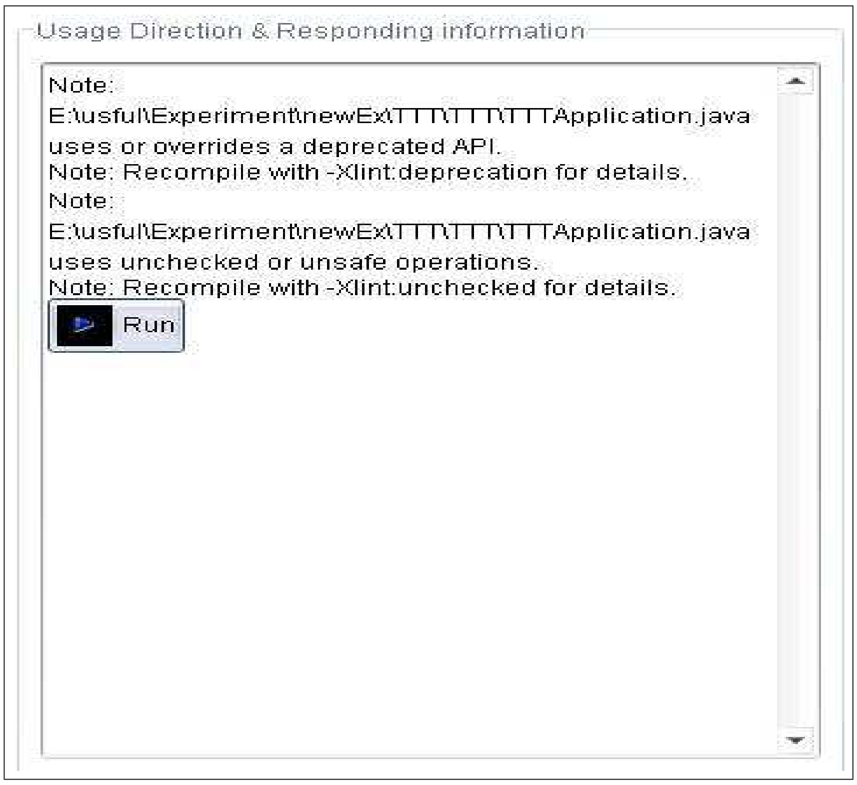
Figure 5.5: State 1 of the usage panel



There are 13 graphs created during the embedding process.
Press the button below to show them one at a time.

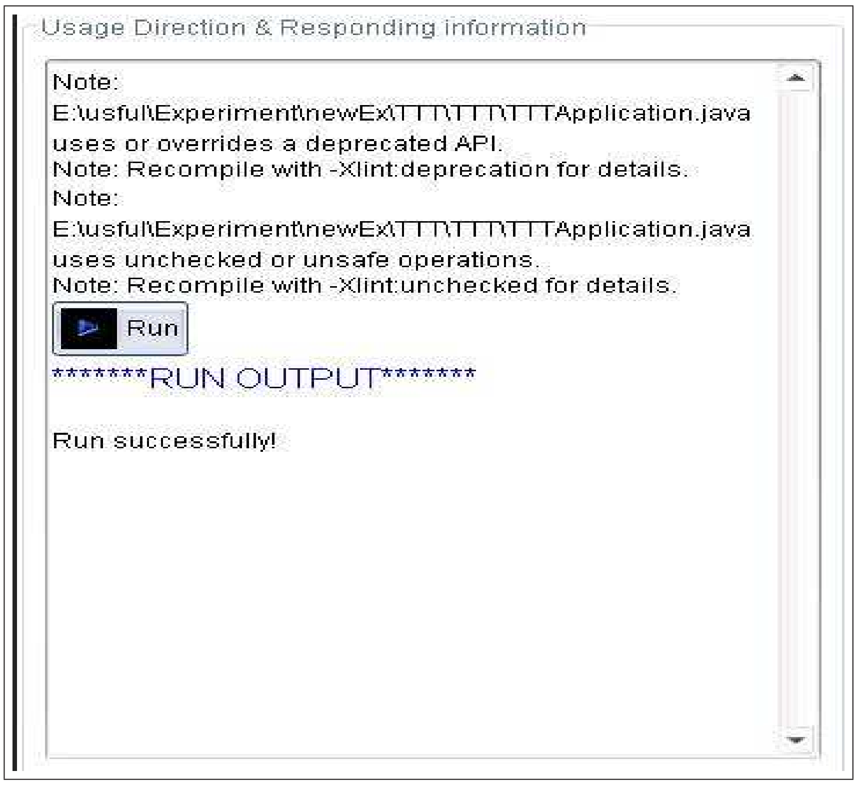
The whole encoding process takes 631 milliseconds

Figure 5.6: State 2 of the usage panel



```
Note:
E:\usful\Experiment\newEx\TTT\TTT\TTTApplication.java
uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
Note:
E:\usful\Experiment\newEx\TTT\TTT\TTTApplication.java
uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Figure 5.7: State 3 of the usage panel



```
Note:
E:\usful\Experiment\newEx\TTT\TTT\TTTApplication.java
uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
Note:
E:\usful\Experiment\newEx\TTT\TTT\TTTApplication.java
uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
********RUN OUTPUT********

Run successfully!
```

Figure 5.8: State 4 of the usage panel

a button which can be used to show each of the constant graph in the order of appearance in the source code. The graph of trees will be shown in the embedding area, tabbed panel "Tree Viewer", as shown in 5.10. The time cost of the encoding process is also given at the end of this panel. This panel goes into its third status once the user clicks on the compile button in the toolbar as shown in figure5.7. In this stage, its content has two kinds of information: 1. Compile output. The information in fact is the output message of the standard java compiler. The panel redirects this information from the standard output. The compile process will get the class path from the options panel as stated in (2). 2. A "run" button. By pressing this button, the usage direction and responding information panel will go into its last status, as shown in figure5.8. The run output information will be appended to the statement "***********RUN OUTPUT**********". GUI applications usually do not generate messages to the console, but it will show its graphic interface instead.

5. Embedding area. This area has two tabbed panels: Source code panel and Tree viewer panel. Their screenshoots can be seen at figure5.9 and figure5.10.

The tree viewer panel will be activated only after the "show graphs in order" button in the usage direction and responding information panel gets pressed. The source code panel contains two text area separated by a spin in the middle. The top text area is used to contain the source file which is going to be tamper-proofed. The bottom text area is used to show the output source file. Both of the two text areas are editable. All the markers should be inserted into appropriate places in this source code text area. For example, in the figure5.9, there are two string constants markers and one integer constant marker. These markers are either typed in by the user or copy and paste from the "Markers Setting" panel. At the same time, if the user wants to generate annotated files for the watermarking process, he may want to insert annotation points by using the source code text area. The bottom text area takes advantages of the styled document class in order to give different colours and fonts to distinguish different types of inserted code and the original code. The black colour and regular size font (12) applies to the original source code. The inserted functions which replace the constant value are in the colour of magenta and in the size of 16. Catalan number calculation class uses magenta and 16 points font as well. Cyan is used on the decoding functions while all the tree building code and the entire node class are in blue and 16 points font.

The tree view panel is responsible for drawing all PPCTs encoded from numerical or string constants. As shown in figure5.10, the string constant "abc" is first encoded into an integer 23159395 which then is encoded into the PPCT drawn below. The node in red refers to the origin of the PPCT. It is worth noting that this PPCT ignores a few edges in order to get a clear view of the topology. These edges include: all the edges which are from leaf nodes, the left outgoing edge which goes from the origin. The number under each node means the constant value which can be decoded from a PPCT rooted at that node. For example, the subtree rooted in the blue node shown in figure5.10 can be used to decode an integer constant 21784.

5.3 Tamper-proof Tic-tac-toe

Here I will show how to use JMarkProtector to constant-encode a simple java game called Tic-tac-toe. Tic-tac-toe is also called noughts and crosses. It is a game between two players, O and X, who alternate in marking the spaces in a 33 board as shown in the figure5.11.

A player wins by getting three of their own marks in a horizontal, vertical or diagonal row []. The source file is called TTTApplication. The output (tamper-proofed)source code is listed in AppendixB.

To tamper-proof this application, first, we need to load the source file TTTApplication.java into the embedding area. Then we type in(or copy and paste)markers to tell the system which line contains encoding constants. In this example, I use the default markers, namely, using "//numerical constants" to mark the lines which contain integer constants, using "//string constants" to mark the lines which contain string contains, using "//define functions here" to mark the place where new generated methods are going to inserted and using "//define classes here" to mark the line where the new generated class definitions will be inserted. There are two constants that I want to encode: integer grid and string "X". So, mark the code in this way:

```
int grid = 3//numerical constants; pan.setLayout(new
GridLayout(grid,grid));
...
...
if(ply == 1)
    sq[i].setLabel("X");//string constants
```

The integer constant grid is used to control the layout of the game board. For example, the line setlayout(new GridLayout(grid,grid)) means to generate a 3x3 game board as shown in figure5.11. The string constant "X" refers to the symbol X that appears in figure5.11. The code "if(ply == 1) sq[i].setLabel("X")" means the player who goes first will use the symbol X to mark his "territory".

We put the function definition marker in the line before the last curely bracket and the classes definition marker at the end of the file(after the last curely bracket).

Before we apply the constant encoding algorithm, we need to set up all the parameters in the options panel. If we use the default setting(see figure5.4),we can do the encoding immediately by pressing the "Do constant encoding" button in the toolbar. Once the button is pressed, the integer 3 will be picked up by the constantFinder class and passed to the codec to encode to a PPCT(see figure5.12). The string constant "X" will first encoded (by the class StringInt in the package embeder.codeGenerator)into a integer and then map to a PPCT. SequenceGenerator will encode the PPCT into a node sequence using a traverse in the order of (root, left child, right child). For the integer 3, the node sequence will be: [0, 3, 1, 1, 2, 3, 2, 4, 5, 4, 0, 4, 5, 6, 7, 6, 4, 6, 7, 6, 7, 3, 7, 3]. As PPCT is full binary tree, we know that it is an one-one mapping between the node sequences and the PPCTs. Please note, as mentioned in chapter 2, the rule I used to construct the PPCT is every node(including the root node and leaf nodes) has two outgoing edges as shown in figure5.12. The left edge of the root(node 0) always points to the rightmost leaf node(node 3). All the leaf nodes' right edges point to themselves and left edges point to another leaf node which is the first leaf node in the left side.

The string integer will be mapped into a integer 601 and then encoded into a PPCT with 18 nodes. The corresponding node sequence is [0, 9, 1, 1, 2, 3, 2, 4, 5, 4, 0, 4, 5, 6, 7, 6, 4, 6, 7, 6, 7, 3, 8, 9, 8, 10, 11, 10, 12, 13, 12, 14, 15, 14, 7, 14, 15, 16, 17, 16, 14, 16, 17, 16, 17, 13, 17, 13, 11, 13, 11, 9, 11, 9]. Then the Embeder class will call the Search class to test if one PPCT contains the other. By applying the tree search algorithm, it finds that the PPCT representing integer 3 can be found in the PPCT representing string "X" (or say the integer 601). At the mean time, the search function will record the boundary of the substructure which represents a integer 3(see figure5.13).

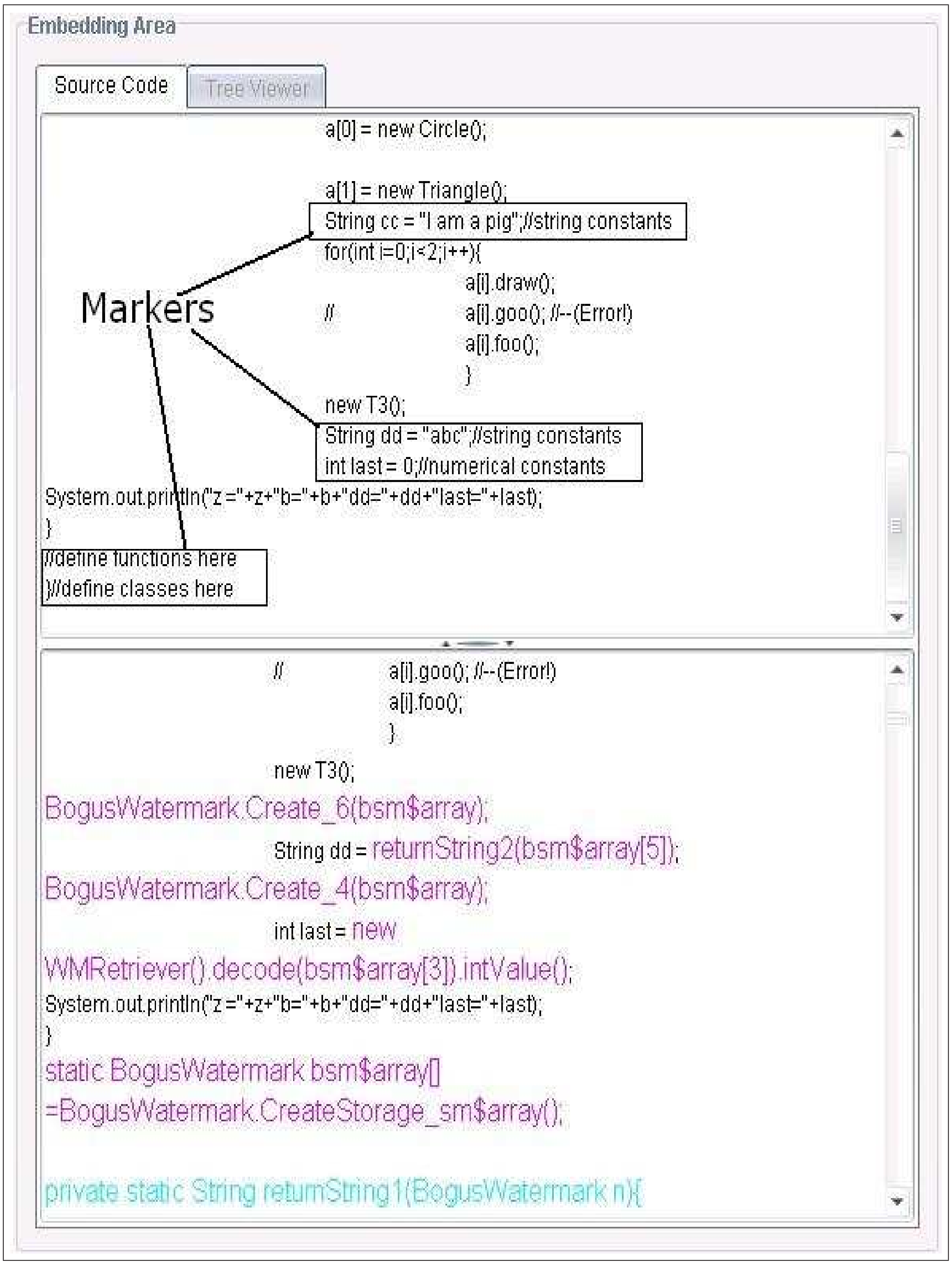
According to the generated node sequence, the embeder package uses class CodeGenerator to generate functions(see appendixB for real code)to be inserted. Please note that the boundary node lists is stored in the array awatermark in class BogusWatermark. The class ClassDefinitions will output the string of definitions of classes such as the decode class, the skeleton of bogus class and the stringInt class etc. After the encoding, the constant loading code will be:

```
int grid = returnValue1(bsm\$array); pan.setLayout(new
GridLayout(grid,grid)); ... ... if(ply == 1)
      sq[i].setLabel(returnString1(bsm\$array));//string constants
```

The bsm$array in the above code is a array which contains the root of the constant tree and the boundary list used to retrieve encoded values back. In this example, the boundary for the substructure representing integer 3 is 8,9 and for "X" is (namely, empty set). So, bsm$array is node0,node8,node9. Please refer to the index for more details.

After the encoding process, users are allowed to compile and run the output code. If the class path is set correct in the option panel, the compile should be run without troubles. However, if the user wants to annotate the source code for preparing input to the watermarking process, he needs to put the path of Sandmark.jar in the classpath as well.

The code given in appendxB has already been annotated: There are four annotation points which are all in the form of "Annotator.sm$mark();". In order to get this source code compiled, I changed the class path to be "D:
sandmark
sandmark.jar". Now after pressing the compile button in the toolbar, I got all five class files. Simply jar them into a .jar file, it is ready to pass into the Sandmark to generate tamper-proofed watermarked code.



```
a[0] = new Circle();

a[1] = new Triangle();
String cc = "I am a pig";//string constants
for(int i=0;i<2;i++){
                    a[i].draw();
//                  a[i].goo(); //--(Error!)
                    a[i].foo();
                    }
new T3();
String dd = "abc";//string constants
int last = 0;//numerical constants
System.out.println("z ="+z+"b="+b+"dd="+dd+"last="+last);
}
//define functions here
}//define classes here
```

```
//              a[i].goo(); //--(Error!)
                a[i].foo();
                }
new T3();
BogusWatermark.Create_6(bsm$array);
String dd = returnString2(bsm$array[5]);
BogusWatermark.Create_4(bsm$array);
int last = new
WMRetriever().decode(bsm$array[3]).intValue();
System.out.println("z ="+z+"b="+b+"dd="+dd+"last="+last);
}
static BogusWatermark bsm$array[]
=BogusWatermark.CreateStorage_sm$array();

private static String returnString1(BogusWatermark n){
```

Figure 5.9: The source code panel in the embedding area

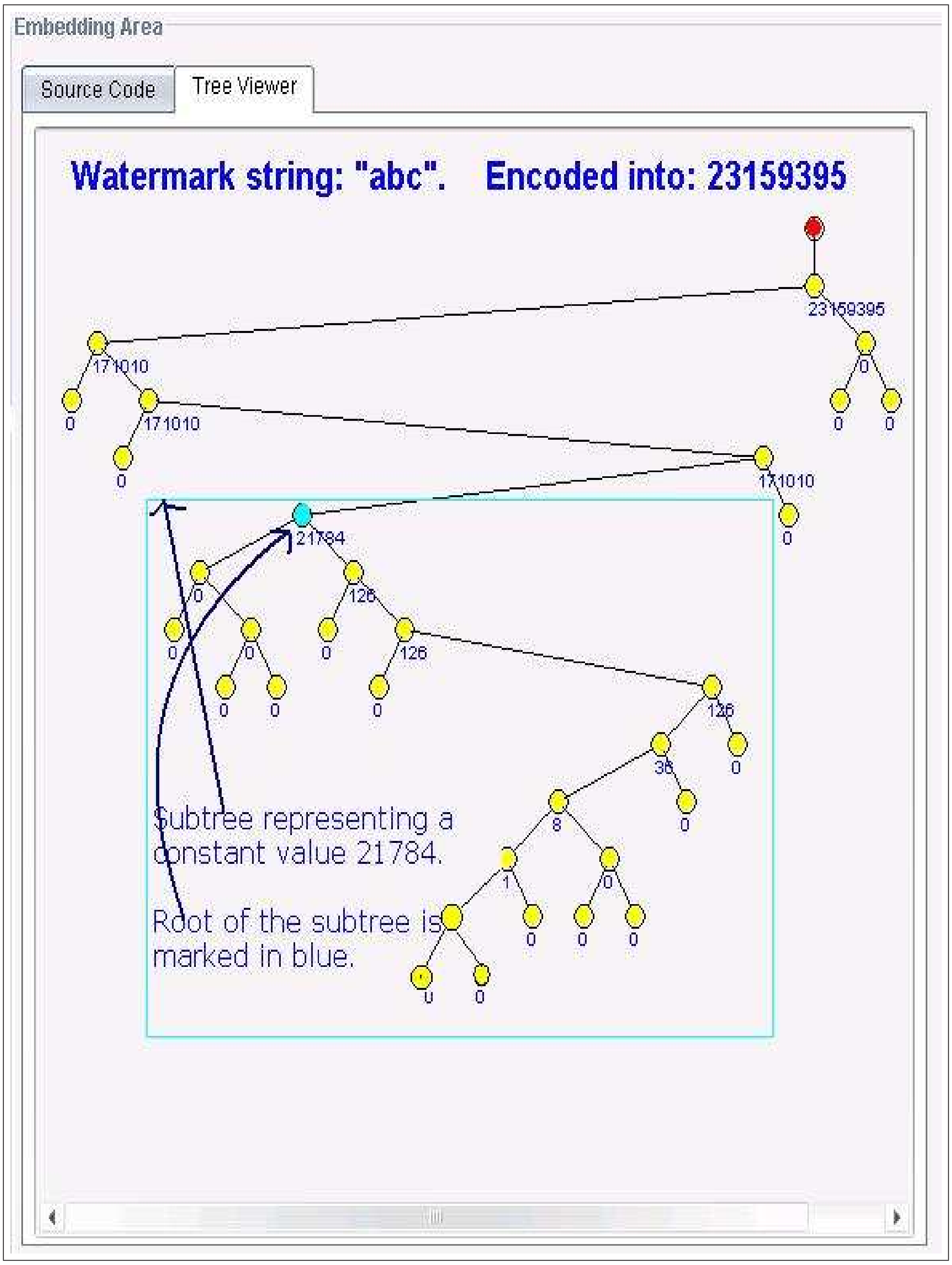

Embedding Area
Source Code
Tree Viewer
Watermark string: "abc". Encoded into: 23159395
23159395
171010
0
171010
0
0
0
0
171010
0
21784
0
126
0
0
0
0
126
0
0
126
36
0
0
8
1
0
0
0
0
0
0
0
Subtree representing a constant value 21784.
Root of the subtree is marked in blue.

Figure 5.10: Tree Viewer shows a PPCT tree which encodes from a string "abc"

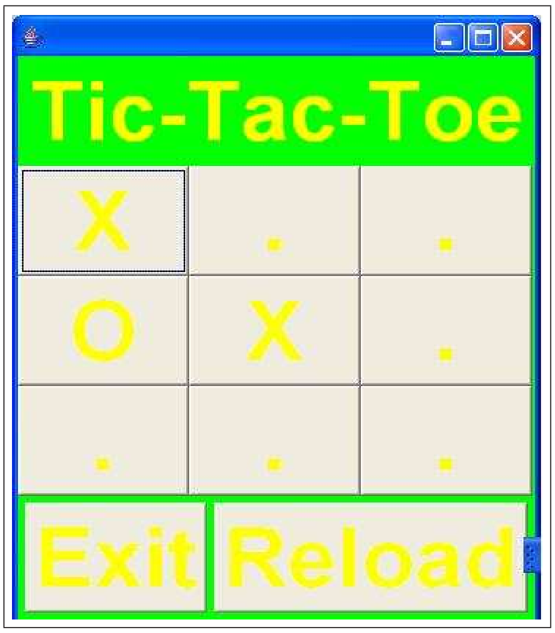
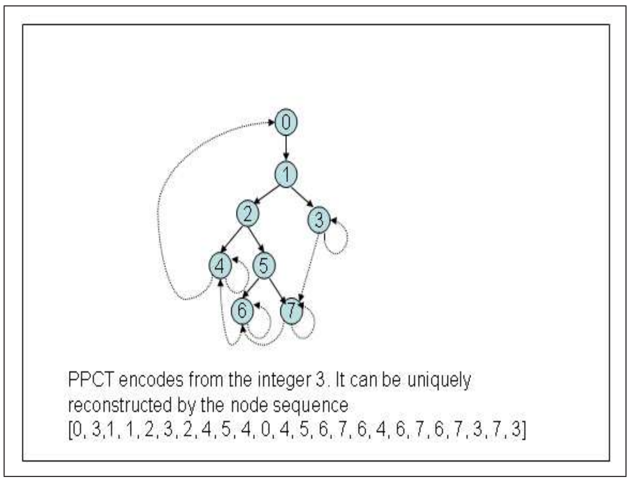


Figure 5.11: Game Tic-tac-toe



PPCT encodes from the integer 3. It can be uniquely reconstructed by the node sequence [0, 3,1, 1, 2, 3, 2, 4, 5, 4, 0, 4, 5, 6, 7, 6, 4, 6, 7, 6, 7, 3, 7, 3]

Figure 5.12: PPCT encoded from 3.

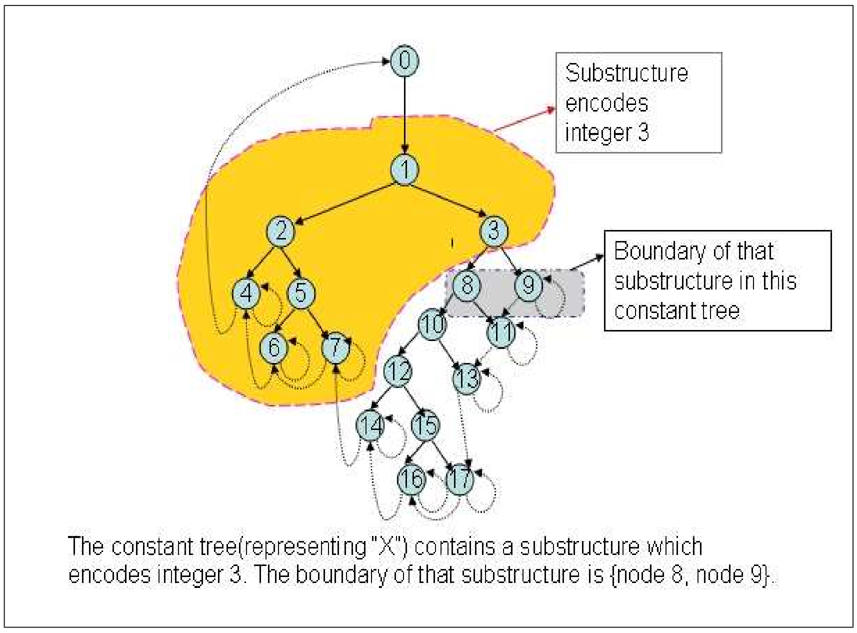

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
Substructure encodes integer 3
Boundary of that substructure in this constant tree
The constant tree(representing "X") contains a substructure which encodes integer 3. The boundary of that substructure is {node 8, node 9}.

Figure 5.13: The constant tree, substructure and its boundary

6 Conclusion

6.1 Evaluation

There are mainly two objectives that need to be achieved by JMarkProtector. First, the correctness of the output code. Second, the effectiveness of the temper-proofing mechanism.

6.1.1 Correctness

As mentioned in chapter 3. The correctness I defined here means functional correctness(semantically equivalent to the original program) and the property that the output code will not affect the watermarking process.

Because JMarkprotector needs users control at several levels (for instant, choose constants to be encoded, input different markers into right places), it is relatively hard to generate automatic correctness tests. Moveover, I am not aware of any methods to test if two programs are semantically equivalent to each other. However, by manually applying the prototype to 10 applications, I found all of the generated code are at least runable and I did not observe any obvious differences between the temper-proofed code and the original one. On a contrast, all the code in the util package has been tested over a large set of test cases. I applied these tree algorithms to randomly generated PPCTs with node number from 3 to 200. All of them can retrieve the constants back from the PPCTs. So, I have strong confidence on the correctness of the tree operation classes.

If considering the space cost, using the default setting the generated code will at least 4700Bytes larger than the original one. This is calculated by computing the difference between two applications after only encoding a integer 0. If "strong tamper-proof" option is unselected, the minimum increase of size will be 4200Bytes. It will be less noticeable if the original program is resonantly large (say, 5 MB). For the time cost, it is easy to collect the difference. However, the results are very vague and hence very hard to mark conclusion. Here is what I did to measure the time increase: I added two lines of code into the retrieving functions to record the time cost, for instance: adding long start = System.currentTimeMillis();

into the first line of the returnValue function, then adding

long timeCost = System.currentTimeMillis()-start;

to compute the increased time. I ran this test 100 times and found the time cost is really unstable: there were 81 times that the increased time is 0.0 milliseconds! 11 times that the result is approximately 200 milliseconds and 8 times some values near to 0 milliseconds. I think those results indicate that the run time performance strongly depends on some things other than the algorithm itself. Therefore, it is hard to say if the temper-proofed code runs obviously slowly than the original one.

I think I can conclude from the above observations that sometimes the tamper-proofed code run noticeably slower than the original code due to unknown reasons, although most of the time there would be no much differences on the aspect of time cost.

All in all,

6.1.2 Effectiveness

In aspect of effectiveness, JMarkprotector only takes care of the first half tamper-proofing algorithm as described in chapter 2. However, under the assumption that the attacker can not confidently distinguish the watermark code and the tamper-proofing code, JMarkprotector will still add a reasonable amount of strength to protect the embedded CT watermarks. Here I gave a preliminary experiment to prove this conclusion.

Using the code listed in AppendixB, we can get the .jar very easily(just need to compile and compress). Name this jar file TTTApplication.jar. Now I can use Sandmark to watermark the TTTApplication.jar file. As it has been annotated, we only need to do the tracing and embedding. In the tracing phase, I defined the input sequence by pressing three buttons as shown in the figure5.11. By Choosing the graph type "PPCT graph" and unchecking the option "use cycle graph" and leaving all the other settings unchanged, I applied the watermark embedding process to embed a string "Copyright" into the tamper-proofed annotated jar file.

I used "javap" to reverse engineer the Watermark.class file contained in the new generated watermarked TTTApplication.jar file and the BogusWatermark file which is generated after compiling the tamper-proofed code. Here is the first few lines of byte code of Watermark.class:

```
0    0:new                  2    <Class  Watermark>
       1      3:dup
```

```
 4:invokespecial   11  <Method void Watermark()>
 7:astore_1
 8:new             2   <Class Watermark>
11:dup
12:invokespecial   11  <Method void Watermark()>
15:astore_2
16:new             2   <Class Watermark>
19:dup
20:invokespecial   11  <Method void Watermark()>
23:astore_3
24:new             2   <Class Watermark>
27:dup
28:invokespecial   11  <Method void Watermark()>
31:astore          4
33:aload           4
35:aload_3
36:putfield        13  <Field Watermark x\$1>
39:new             2   <Class Watermark>
42:dup
```

And corresponding lines of BogusWatermark are:

```
0:new 2<Class BogusWatermark>
 3:dup
 4:invokespecial   3   <Method void BogusWatermark()>
 7:astore_0
 8:new             2   <Class BogusWatermark>
11:dup
12:invokespecial   3   <Method void BogusWatermark()>
```

```
15:astore_1
16:aload_0
17:aload_1
18:putfield          4    <Field BogusWatermark x\$1>
21:new               2    <Class BogusWatermark>
24:dup
25:invokespecial     3    <Method void BogusWatermark()>
28:astore_2
29:new               2    <Class BogusWatermark>
```

I assume the attacker can not distinguish which pieces of code belong to the watermark and which belong to the constant retrieving functions, then the attacker has to change all of the suspicious code. The attacker may think the intensive appearance of new operations is suspicious, then he will need to attach on the code listed above.

For most of the code in the first few lines, if deleted or altered, the program will fail to compile whereas if the code in the middle part is changed, it will often cause interesting problems, for example, after reordering the code in line number 321-333 in both of the two class files, the TTTApplication's appearance was changed, see figure6.1. Apparently, the string retrieving function failed to return the correct value "X", instead, it returned another symbol as shown in the picture.

If the attacker successfully located the tamper-proofing code, there is no need to figure out what the correct values should be. The attacker may simply ignore the code there and focus on attacking the watermark code. The difficulty of locating the tamper-proofing code may increase if the chosen constants are not very critical to the application's behavior or are less sensitive to changes. For example, if the constant a in the following code is encoded, it may take very long for the attacker to locate the constant tree building code.

```
int a = 100; if(-500<a<500) doSometing(); if(a%4==0) doSomething();
```

However, if the attacker has gained enough information about what the tamper-proofing code may look like, a simple pattern match can be good enough to find the constant

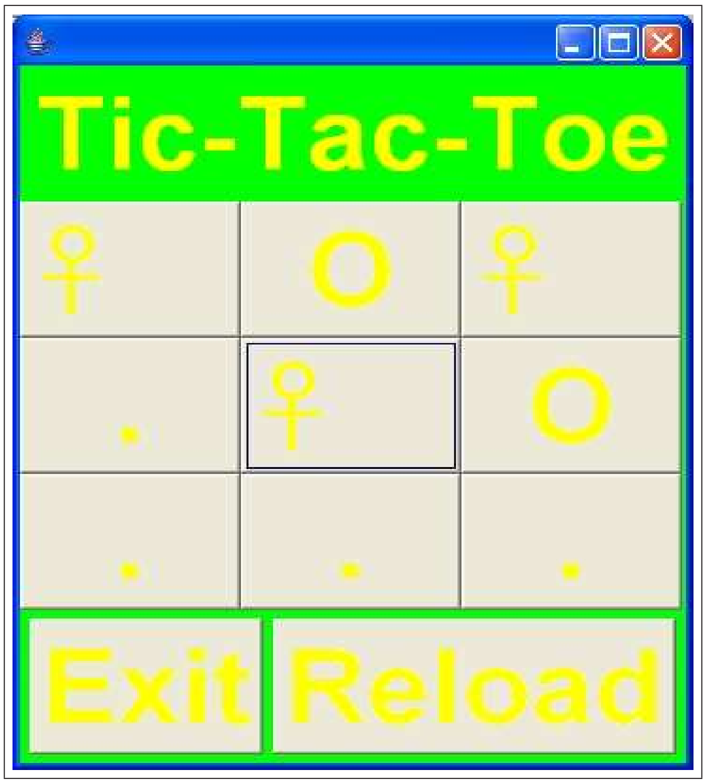


Figure 6.1: Appearance changed

encoding code. For example, They may search for the code sequence which contains many objects which point to themselves because in the constant tree every leaf node has a edge that points to itself while the watermark tree does not have this property.

So, we can conclude here, under the assumption, the temper-proofed code can provide certain amount of protection to simple attacks. But the effectiveness is greatly affected by the selection of constants. We believe that if the dependency between the watermark tree and the constant tree is built, the attacker will have to replace all the constant loading code with original constants in order to make changes to the watermark code, which will certainly increase the difficulty of attacking to a great extent.

6.1.3 Conclusion

JMarkProtector is a useful tool to research constant-encoding algorithms. It provides a rich set of PPCT tree operations which can be used to develop or experiment new applications. JMarkProtector can only provide limited protections to the embedded CT watermark. However, if JMarkProtector can be used in combination with another prototype which can create a real dependency between the CT watermark building code and the constant tree building code, it will give a very powerful protection to the CT watermark.

Appendices

Appendix A: Source code of the searching algorithm

```
package util;

import util.codec.PPCTCodec1;
import java.math.BigInteger;
import java.util.ArrayList;
import java.util.Vector;
import java.util.Stack;
import embeder.SequenceGenerator;


public class Search {

    /** Search for a substructure of the same shape of a PPCT
     *rooted at originB from the PPCT rooted at originA.
     *Return true if founded, otherwise return false.
     *
     * @param originA The origin of a PPCT where the search starts
     * @param originB The origin of a PPCT that to be searched
     * @param fatherOfRootBinA If the search returns true,
     * fatherOfRootBinA's right pointer
     * will point to the root of the substructure which matches
     * the shape of PPCT originB
     * @param boundary If found, the arrayList boundary will hold
     * a list of nodes which belong to
     */

    public static boolean search(Node originA, Node originB
        ,Node fatherOfRootBinA, ArrayList<Node> boundary){
        if(originA==null||originB==null) {
            System.out.println("Error!␣Origin␣is␣null");
            return false;
        }
        Node rootA = originA.right;
        Node rootB = originB.right;
        return searchRoot(rootA,rootB,fatherOfRootBinA,boundary);
    }
    public static boolean searchRoot(Node rootA, Node rootB
        , Node fatherOfRootBinA, ArrayList<Node> boundary){


        if(rootA==null||rootB==null){
            System.out.println("Error!␣Not␣suitable␣for␣PPCTCodec1␣!");
            return false;
        }
        if(ifConstains(rootA,rootB,boundary)){
            fatherOfRootBinA.right = rootA;
            return true;
        }
        else{
            if(rootA.right == rootA)return false;
            else return
              searchRoot(rootA.left,rootB,fatherOfRootBinA,boundary)||
                    searchRoot(rootA.right,rootB,fatherOfRootBinA,boundary);
        }
    }
    public static boolean ifConstains(Node rootA, Node rootB
                ,ArrayList<Node> boundary ){
            if(rootA==null||rootB==null){
                System.out.println("Error!");
                return false;
            }
            if(rootA.right==rootA&&rootB.right!=rootB)return false;
            if(rootA.right!=rootA&&rootB.right==rootB){
                boundary.add(rootA.left);
                boundary.add(rootA.right);
                return true;
            }
            if(rootA.right==rootA&&rootB.right==rootB)return true;
            if(rootA.right!=rootA&&rootB.right!=rootB)
                return(ifConstains(rootA.left,rootB.left,boundary)&&
                        ifConstains(rootA.right,rootB.right,boundary));
        return false;
    }

    /**
     * Try to find aNodeInA from tree originA, if found return
     * true and generate a path to get to aNodeInA from originA,
     * else return false. 0 means go left while 1 means go right.
     * @param root the root of A tree
     * @param node A node that should be in that tree .
     * @param path A stack used to store information about how to
     * find aNodeInA, 0 means go left, 1 means
     *  go right. The size of path refers to the depth of such a node.
     * The initial stack should be empty.
     * @return a stack, that contains a sequence of 0 and 1s.
     */
    public static Stack<Integer> getPath(Node root, Node node,
                                            Stack<Integer> path) {

        if (root.right==root)
        {
            System.out.println("reach␣a␣leaf!");
            return null;
        }

        if (root==node) {
            System.out.println("Found!");
            return path;}
        path.push(0);
        if (getPath(root.left, node, path) != null) return path;
        path.pop();
        path.push(1);
        return getPath(root.right, node, path);
    }

    private static boolean search(Node origin, Node node,
                                        Stack<Integer> path){
```

```
        if(origin == null)return false;

        Node root = origin.right;

        return search(root, node, path, 1);


    }


    private static boolean search(Node root, Node node,

                        Stack<Integer> path, int direction){

        path.push(direction);

        if(root == node)return true;

        if(root.right != root)

        {

            if(!search(root.left,node,path,0)) {

                path.pop();

                if(!search(root.right, node,path,1)){

                    path.pop();

                    return false;

                }

                else{

                    return true;

                }

            }

            else {

                return true;

            }


        }

        return false;

    }

    public static int[] sequence2Int(Stack<Integer> path){

        int[] info = new int[2];

        info[0] = path.size();

        System.out.println("path.size␣=␣"+path.size());

        info[1] = 1; //set info[1] = 2 to the power of 0, namly 1.

        for(int i = 1; i<info[0]; i++){

            System.out.println("path.pop␣=␣"+path.peek());
            info[1]+=path.pop()*(2<<(i-1)); // 2^n+2^(n-1)......
        }
        return info;
    }

    public static Node locateRoot(Node origin, Stack<Integer> path){
        if(origin==null||origin.right==null||origin.left==null){
            System.out.println("Error!");
            return null;
        }
        Node temp = origin;
        for(Integer direction :path){
            if(direction==0)temp = temp.left;
            else if(direction==1)temp = temp.right;
            else System.out.println("Error!Path␣can␣only␣contain␣01.");
        }
        return temp;
    }
}
```

Appendix B:Tamperproofed source code Tic-tac-toe

```
import java.awt.*;

import java.awt.event.*;

import sandmark.watermark.ct.trace.Annotator;


public class TTTApplication extends Frame

{


    public TTTApplication()

    {

        lab = new Label("Tic-Tac-Toe", 1);

        pan = new Panel();

        sq = new Button[9];

        south = new Panel();

        quit = new Button("Exit");

        reload = new Button("Reload");

        ply = 2;

    }


    public void start()

    {

        for(int i = 0; i < 9; i++)

        {

            sq[i] = new Button(".");

            sq[i].setActionCommand(Integer.toString(i));

            sq[i].addActionListener(new ActionListener() {


                public void actionPerformed(ActionEvent actionevent)

                {

                    int j = (new Integer

                        (actionevent.getActionCommand())).intValue();

                    move(j);

                }


            });
            pan.add(sq[i]);
        }

    }

    public void init()
    {
        Annotator.sm$mark();
        setBackground(Color.green);
        setForeground(Color.yellow);
        setFont(new Font("SansSerif", 1, 60));
        setSize(360, 360);
        setLayout(new BorderLayout());
        addWindowListener(new WindowAdapter() {

            public void windowClosing(WindowEvent windowevent)
            {
                dispose();
                System.exit(0);
            }

        });
        add("North", lab);
        add("Center", pan);

        //integer grid is the constant to be encoded!
        //Original value is 3, which means the layout
        //is a 3x3 chessboard.
        int grid = returnValue1(bsm$array);
        pan.setLayout(new GridLayout(grid,grid));
        Annotator.sm$mark();
        south.setLayout(new FlowLayout());
        quit.addActionListener(new ActionListener() {

            public void actionPerformed(ActionEvent actionevent)
            {
                dispose();
                System.exit(0);
            }

        });
        reload.addActionListener(new ActionListener() {

            public void actionPerformed(ActionEvent actionevent)
            {
                clear();
            }

        });
        add("South", south);
        south.add(quit);
        south.add(reload);
    }

    public static void main(String args[])
    {
        TTTApplication tttapplication = new TTTApplication();
        tttapplication.init();
        tttapplication.start();
        tttapplication.pack();
        tttapplication.show();
    }

    public void clear()
    {
        setForeground(Color.yellow);
        for(int i = 0; i < 9; i++)
            sq[i].setLabel(".");

        Annotator.sm$mark();
        lab.setText("Tic-Tac-Toe");
    }
```

```
public void move(int i)
{
    if(!hit() && free())
    {
        ply = ply != 2 ? 2 : 1;
        Annotator.sm$mark(ply);
        mark(i);
        if(hit())
        {
            setForeground(Color.blue);
            lab.setText("Player " + ply + " won!");
        }
    } else
    {
        if(free())
            setForeground(Color.magenta);
        else
            setForeground(Color.red);
        lab.setText("Reload game!");
    }
}

public boolean free()
{
    boolean flag = false;
    for(int i = 0; i < 9; i++)
        if(sq[i].getLabel() == ".")
            flag = true;

    return flag;
}

public boolean hit()
{
    boolean flag = singleHit(0, 4, 8) | singleHit(2, 4, 6);
```

```
            for(int i = 0; i < 3; i++)
                flag |= singleHit(i, i + 3, i + 6);

            for(int j = 0; j < 9; j += 3)
                flag |= singleHit(j, j + 1, j + 2);

            return flag;
        }

        public void mark(int i)
        {
            if(sq[i].getLabel() == ".")
            {
                if(ply == 1)//to be marked
                    sq[i].setLabel(returnString1(bsm$array));
                else
                    sq[i].setLabel("O");
                Annotator.sm$mark(i);
            }
        }

        private boolean singleHit(int i, int j, int k)
        {
            return sq[i].getLabel() != "." &&
            sq[i].getLabel() == sq[j].getLabel()
            && sq[i].getLabel() == sq[k].getLabel();
        }
    static BogusWatermark bsm$array[] =
        BogusWatermark.CreateStorage_sm$array();

    private static String returnString1(BogusWatermark[] n){
    WMRetriever wmr = new WMRetriever();
    byte[] b = wmr.getWmNum(n[0]).toByteArray();
    int len = b.length - 1;
    b[len] = (byte) ((b[len] & 0xFE) | (b[0] & 1));
    return new String(b, 1, len);
```

```
}

private static int returnValue1(BogusWatermark[] ns){
BogusWatermark[] bounds = {ns[1],ns[2]};
return new WMRetriever().getWmNum(ns[0], bounds).intValue();
    }



    Label lab;
    Panel pan;
    Button sq[];
    Panel south;
    Button quit;
    Button reload;
    int ply;
}

class WMRetriever {

    BogusWatermark base;
    int num = 0;


    public java.math.BigInteger getWmNum(BogusWatermark top) {
        while (top.x$0 != top) {
            top = top.x$1;
        }
        do {
            top = top.x$1;
        } while (top.x$0 == top);
        base = top.x$0;
        return cNum(top.x$0);
    }

    public java.math.BigInteger getSubTreeWmNumber(BogusWatermark top) {
        BogusWatermark current = top;
    do {
        current = current.x$0;
    } while (current.x$0 != current);
    if (top.x$1 == current) {
        return cNum(top.x$0);
    }
    return cNum(top);
}



private java.math.BigInteger cNum(BogusWatermark top) {
    if (top.x$0 == top)
        return java.math.BigInteger.ZERO;
    return cNum(top.x$1).multiply(
            (java.math.BigInteger)CatalanNumCalc.
                getCat().elementAt(getRightLeafNum(top)))
            .add(cNum(top.x$0)).add(
            minInt(getLeftLeafNum(top), getRightLeafNum(top)));
}

private java.math.BigInteger minInt(int x$1, int x$0) {
    if (x$1 == 1)
        return java.math.BigInteger.ZERO;
    return minInt(x$1 - 1, x$0 + 1).add(
            ((java.math.BigInteger)CatalanNumCalc.
                        getCat().elementAt(x$1 - 1))
                    .multiply((java.math.BigInteger)
                        CatalanNumCalc.getCat().elementAt(
                    x$0 + 1)));
}
public int getLeafNum(BogusWatermark top) {
    if (top.x$0 == top)
        return 0;
    BogusWatermark x$1, x$0;
    int num = 0;
    x$1 = top;
```

```
        x$0 = top;
        while (x$0.x$0 != x$0)
            x$0 = x$0.x$0;
        while (x$1.x$0 != x$1)
            x$1 = x$1.x$1;
        while (x$1 != x$0) {
            x$0 = x$0.x$1;
            num++;
        }
        return num + 1;
    }
    private int getLeftLeafNum(BogusWatermark top) {
        if (top.x$0 == top)
            return 0;
        if (top.x$1.x$0 == top.x$1)
            return 1;
        return getLeafNum(top.x$1);
    }



    private int getRightLeafNum(BogusWatermark top) {
        if (top.x$0 == top)
            return 0;
        if (top.x$0.x$0 == top.x$0)
            return 1;
        return getLeafNum(top.x$0);
    }
    public java.math.BigInteger getWmNum(BogusWatermark top
                        , BogusWatermark[] nodes) {
        //top has to be the origin of the constant tree!

        if(nodes==null)return getWmNum(top);
        if (nodes.length == 0) return getWmNum(top);
        return cNum(top.x$0, nodes);
    }
```

```
//
private java.math.BigInteger cNum(BogusWatermark top, BogusWatermark[] nodes) {
    if (top.x$0 == top || ifMatchBoundary(top, nodes))
        return java.math.BigInteger.ZERO;
    return cNum(top.x$1, nodes).multiply(
            (java.math.BigInteger) CatalanNumCalc.getCat().
                elementAt(getRightLeafNum(top, nodes)))
            .add(cNum(top.x$0, nodes)).add(
            minInt(getLeftLeafNum(top, nodes), getRightLeafNum(top, nodes)));
}



private boolean ifMatchBoundary(BogusWatermark node, BogusWatermark[] nodes) {
    if(nodes==null)return false;
    for (BogusWatermark node1 : nodes) {
        if (node.x$1 == node1) return true;
    }
    for (BogusWatermark node2 : nodes) {
        if (node.x$0 == node2) return true;
    }
    return false;
}

public int getLeafNum(BogusWatermark top, BogusWatermark[] nodes) {
    if(top.x$0 ==top||ifMatchBoundary(top,nodes))
   return 1;
    else return getLeafNum(top.x$1,nodes)+getLeafNum(top.x$0,nodes);
}

public int getLeftLeafNum(BogusWatermark top, BogusWatermark[] nodes) {
    if (top.x$0 == top || ifMatchBoundary(top, nodes))
        return 0;
    if (top.x$1.x$0 == top.x$1 || ifMatchBoundary(top.x$1, nodes))
        return 1;
    return getLeafNum(top.x$1, nodes);
}
```

```
    public int getRightLeafNum(BogusWatermark top, BogusWatermark[] nodes) {
        if (top.x$0 == top || ifMatchBoundary(top, nodes))
            return 0;
        if (top.x$0.x$0 == top.x$0 || ifMatchBoundary(top.x$0, nodes))
            return 1;
        return getLeafNum(top.x$0, nodes);
    }

}

//Declaration of Class CatalanNumCalc is here:
class CatalanNumCalc{
private static java.util.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;
public CatalanNumCalc(){
setupVec();
}
public static int getNodeNo(){
return nodeNo++;
}
public static void resetNodeNo(){
nodeNo = 0;
}
public static void setupVec(){
long t1 = System.currentTimeMillis();
cat = new java.util.Vector();
cat.setSize(MAXLEAF + 1);
cat.setElementAt(java.math.BigInteger.ZERO, 0);
cat.setElementAt(java.math.BigInteger.ONE, 1);
for(int i = 2; i <= MAXLEAF; i++)
{java.math.BigInteger biginteger = ((java.math.BigInteger)cat.elementAt(i - 1)).
    multiply(java.math.BigInteger.valueOf(2 * (2 * i - 3))).
```

```
            divide(java.math.BigInteger.valueOf(i));
cat.setElementAt(biginteger, i);
}
mid = new java.util.Vector();
mid.setSize(MAXLEAF + 1);
mid.setElementAt(new java.util.Vector(), 0);
mid.setElementAt(new java.util.Vector(), 1);
for(int j = 2;j <= MAXLEAF;j++)
{java.util.Vector temp = new java.util.Vector();
temp.setSize(j);
temp.setElementAt(java.math.BigInteger.ZERO, 0);
for(int k = 1;k < j;k++)
{java.math.BigInteger biginteger =
    (((java.math.BigInteger)cat.elementAt(k)).
    multiply((java.math.BigInteger)cat.elementAt(j - k))).
        add((java.math.BigInteger)temp.elementAt(k - 1));
temp.setElementAt(biginteger, k);
}
mid.setElementAt(temp, j);
}
}
public static java.util.Vector getCat(){
if (cat==null) setupVec();
return cat;
}
public static java.util.Vector getMid(){
if (mid==null) setupVec();
return mid;
}
}
class BogusWatermark
{



public static BogusWatermark[] CreateStorage_sm$array()
{
```

```
BogusWatermark watermark0= new BogusWatermark();


BogusWatermark n1304467= new BogusWatermark();
watermark0.x$1 = n1304467;
BogusWatermark watermark1= new BogusWatermark();
BogusWatermark n63224399= new BogusWatermark();
BogusWatermark watermark9= new BogusWatermark();
watermark0.x$0 = n63224399;
watermark0.x$1 = watermark9;
watermark0.x$0 = watermark1;
BogusWatermark n6820846= new BogusWatermark();
watermark1.x$1 = n6820846;
BogusWatermark watermark3= new BogusWatermark();
BogusWatermark n26454177= new BogusWatermark();
BogusWatermark watermark2= new BogusWatermark();
watermark1.x$0 = n26454177;
watermark1.x$1 = watermark2;
watermark1.x$0 = watermark3;
BogusWatermark n57222250= new BogusWatermark();
watermark2.x$1 = n57222250;
BogusWatermark watermark5= new BogusWatermark();
BogusWatermark n45176859= new BogusWatermark();
BogusWatermark watermark4= new BogusWatermark();
watermark2.x$0 = n45176859;
watermark2.x$1 = watermark4;
watermark2.x$0 = watermark5;
BogusWatermark n86951229= new BogusWatermark();
watermark4.x$1 = n86951229;
BogusWatermark n56832723= new BogusWatermark();
watermark4.x$0 = n56832723;
watermark4.x$1 = watermark0;
watermark4.x$0 = watermark4;
BogusWatermark n99714489= new BogusWatermark();
watermark5.x$1 = n99714489;
BogusWatermark watermark7= new BogusWatermark();
BogusWatermark n68491535= new BogusWatermark();
BogusWatermark watermark6= new BogusWatermark();
watermark5.x$0 = n68491535;
watermark5.x$1 = watermark6;
watermark5.x$0 = watermark7;
BogusWatermark n91491604= new BogusWatermark();
watermark6.x$1 = n91491604;
BogusWatermark n90540869= new BogusWatermark();
watermark6.x$0 = n90540869;
watermark6.x$1 = watermark4;
watermark6.x$0 = watermark6;
BogusWatermark n73908699= new BogusWatermark();
watermark7.x$1 = n73908699;
BogusWatermark n33521202= new BogusWatermark();
watermark7.x$0 = n33521202;
watermark7.x$1 = watermark6;
watermark7.x$0 = watermark7;
BogusWatermark n88698896= new BogusWatermark();
watermark3.x$1 = n88698896;
BogusWatermark n49273271= new BogusWatermark();
BogusWatermark watermark8= new BogusWatermark();
watermark3.x$0 = n49273271;
watermark3.x$1 = watermark8;
watermark3.x$0 = watermark9;
BogusWatermark n65859620= new BogusWatermark();
watermark8.x$1 = n65859620;
BogusWatermark watermark11= new BogusWatermark();
BogusWatermark n65478347= new BogusWatermark();
BogusWatermark watermark10= new BogusWatermark();
watermark8.x$0 = n65478347;
watermark8.x$1 = watermark10;
watermark8.x$0 = watermark11;
BogusWatermark n50396313= new BogusWatermark();
watermark10.x$1 = n50396313;
BogusWatermark watermark13= new BogusWatermark();
BogusWatermark n3475276= new BogusWatermark();
BogusWatermark watermark12= new BogusWatermark();
```

```
watermark10.x$0 = n3475276;

watermark10.x$1 = watermark12;

watermark10.x$0 = watermark13;

BogusWatermark n62149723= new BogusWatermark();

watermark12.x$1 = n62149723;

BogusWatermark watermark15= new BogusWatermark();

BogusWatermark n166489= new BogusWatermark();

BogusWatermark watermark14= new BogusWatermark();

watermark12.x$0 = n166489;

watermark12.x$1 = watermark14;

watermark12.x$0 = watermark15;

BogusWatermark n30615287= new BogusWatermark();

watermark14.x$1 = n30615287;

BogusWatermark n59363363= new BogusWatermark();

watermark14.x$0 = n59363363;

watermark14.x$1 = watermark7;

watermark14.x$0 = watermark14;

BogusWatermark n60683424= new BogusWatermark();

watermark15.x$1 = n60683424;

BogusWatermark watermark17= new BogusWatermark();

BogusWatermark n43557319= new BogusWatermark();

BogusWatermark watermark16= new BogusWatermark();

watermark15.x$0 = n43557319;

watermark15.x$1 = watermark16;

watermark15.x$0 = watermark17;

BogusWatermark n38980439= new BogusWatermark();

watermark16.x$1 = n38980439;

BogusWatermark n64374457= new BogusWatermark();

watermark16.x$0 = n64374457;

watermark16.x$1 = watermark14;

watermark16.x$0 = watermark16;

BogusWatermark n25448753= new BogusWatermark();

watermark17.x$1 = n25448753;

BogusWatermark n46463421= new BogusWatermark();

watermark17.x$0 = n46463421;

watermark17.x$1 = watermark16;
```

```
watermark17.x$0 = watermark17;
BogusWatermark n69060568= new BogusWatermark();
watermark13.x$1 = n69060568;
BogusWatermark n85581193= new BogusWatermark();
watermark13.x$0 = n85581193;
watermark13.x$1 = watermark17;
watermark13.x$0 = watermark13;
BogusWatermark n32285410= new BogusWatermark();
watermark11.x$1 = n32285410;
BogusWatermark n94303721= new BogusWatermark();
watermark11.x$0 = n94303721;
watermark11.x$1 = watermark13;
watermark11.x$0 = watermark11;
BogusWatermark n69945311= new BogusWatermark();
watermark9.x$1 = n69945311;
BogusWatermark n56351377= new BogusWatermark();
watermark9.x$0 = n56351377;
watermark9.x$1 = watermark11;
watermark9.x$0 = watermark9;

BogusWatermark awatermark[] = new BogusWatermark[4];

awatermark[1] = watermark0;
awatermark[2] = watermark8;
awatermark[3] = watermark9;

return awatermark;
}
public BogusWatermark x$1;
public BogusWatermark x$0;
}
```

References

Bibliography

[1] D. Mayster, *One Third of All Software in Use Still Pirated*, Online, 2005, Available from http://www.bsa.org/usa/press/newsreleases/Global-Piracy-Study-05-18-2005.cfm.

[2] C. S. Collberg and C. Thomborson, *Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection*, in *IEEE Transactions on Software Engineering*, volume 28, pages 735–746, 2002.

[3] D. Aucsmith, *Tamper Resistant Software: An Implementation*, in *Proceedings of the First International Workshop on Information Hiding*, pages 317–333, London, UK, 1996, Springer-Verlag.

[4] W. Zhu, C. D. Thomborson, and F.-Y. Wang, *A Survey of Software Watermarking*, in *Intelligence and Security Informatics, IEEE International Conference on Intelligence and Security Informatics, ISI 2005, Atlanta, GA, USA, May 19-20, 2005, Proceedings*, edited by P. B. Kantor, G. Muresan, F. Roberts, D. D. Zeng, F.-Y. Wang, H. Chen, and R. C. Merkle, volume 3495 of *Lecture Notes in Computer Science*, pages 454–458, Springer, 2005.

[5] C. Collberg and C. Thomborson, *Software Watermarking: Models and Dynamic Embeddings*, in *Proceedings of Symposium on Principles of Programming Languages,*

POPL'99, pages 311–324, 1999.

[6] C. Collberg, C. Thomborson, and G. M. Townsend, *Dynamic Graph-Based Software Watermarking*, Technical Report TR04-08, The Department of Computer Science, University of Arizona, 2004.

[7] J. Palsberg, S. Krishnaswamy, M. Kwon, D. Ma, and Q. Shao, *Experience with Software Watermarking*, in *16th Annual Computer Security Applications Conference*, 2000.

[8] Y. He, *Tamperproofing a Software Watermark by Encoding Constants*, Master's thesis, University of Auckland, 2002.

[9] R. S. Clark Thomborson, Jasvir Nagra and C. He, *Tamper-proofing Software Watermarks*, in *Proc. Second Australasian Information Security Workshop (AISW2004)*, volume 32, pages 27–36, 2004.

[10] C. Collberg and G. Townsend, *SandMark: Software Watermarking for Java*, 2001.

[11] J. Nagra, C. Thomborson, and C. Collberg, *A Functional Taxonomy for Software Watermarking*, in *Twenty-Fifth Australasian Computer Science Conference (ACSC2002)*, edited by M. J. Oudshoorn, Melbourne, Australia, 2002, ACS.

[12] R. S. Clark Thomborson, Yong He and J. Nagra, *tamper proofing watermarked computer programs*, US Patent 10/850,195, 2005.

[13] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification, Third Edition*, The Java Series, Addison-Wesley, Boston, Mass., 2005.

[14] M. Drmota, *Combinatorics and Asymptotics on Trees*, 2004.

[15] I. P. Goulden and D. M. Jackson, *Combinatorial enumeration*, Wiley-Interscience Series in Discrete Mathematics, John Wiley & Sons, Chichester-New York-Brisbane-Toronto-Singapore, 1983.