Evaluating the WaterRpg Software Watermarking Model on Java Application Programs

Ioannis Chionis
Dept. of Computer Science
University of Ioannina
Ioannina, Greece
ichionis@cs.uoi.gr

Maria Chroni
Dept. of Computer Science
University of Ioannina
Ioannina, Greece
mchroni@cs.uoi.gr

Stavros D. Nikolopoulos
Dept. of Computer Science
University of Ioannina
Ioannina, Greece
stavros@cs.uoi.gr

ABSTRACT

Recently, we have presented a dynamic watermarking model, which we named WaterRpg, for embedding a reducible permutation graph $F[\pi^*]$ into an application program P. The main idea behind the proposed watermarking model is to modify the dynamic call-graph $G(P, I_{key})$ of the program P, taken by the specific input I_{key}, so that the dynamic call-graph $G(P^*, I_{key})$ of the resulting watermarked program P^* and the the reducible permutation graph $F[\pi^*]$ are isomorphic; within this idea the program P^* is produced by only altering appropriate calls of specific functions of the input application program P. Our model belongs to execution trace watermarks category. In this paper, we implement our WaterRpg watermarking model on several Java application programs and evaluate it under various criteria in order to gain information about its practical behavior. More precisely, we selected a number of Java application programs and watermark them using two main watermarking approaches supported by our WaterRpg model, namely naive and stealthy approachs. The experimental results show the stable functionality of all the Java programs P^* watermarked under both the naive and stealthy cases. The experiments also show that the watermarking approaches supported by our model can help develop efficient watermarked Java programs with respect to resilience, size, time, space, and other watermarking metrics.

Categories and Subject Descriptors

D.2 [**Software Engineering**]: Software Architectures, Metrics—*information hiding, performance measures*

General Terms

Legal Aspects, Performance, Security

Keywords

Software watermarking, Dynamic watermarking models, Self-inverting permutations, Reducible permutation graphs, Dynamic call-graphs, Graph embedding, Codec algorithms, Implementation, Experimental evaluation.

PCI '13 Thessaloniki, GREECE

1. INTRODUCTION

Digital watermarking is a technique for protecting the intellectual property of any digital content. The idea of digital watermarking is the embedding of a unique identifier, which we call watermark, into the digital image, audio, video, software or text through the introduction of errors not detectable by human perception [8].

Software Watermarking. The *software watermarking problem* can be described as the problem of embedding a structure w into a program P producing thus a new program P^* such that w can be reliably located and extracted from the program P^* even after P^* has been subjected to code transformations such as translation, optimization and obfuscation. More precisely, given a program P, a watermark w, and a key k, the software watermarking problem can be formally described by the following two functions: $\texttt{embed}(P, w, k) \rightarrow P^*$ and $\texttt{extract}(P^*, k) \rightarrow w$ [6].

There are two main categories of watermarking algorithms namely *static* and *dynamic* algorithms [7]. A static watermark is stored inside program code in a certain format, and it does not change during the program execution. A dynamic watermark is built during program execution, perhaps only after a particular sequence of input. It might be retrieved by analyzing the data structures built when watermarked program is running. In other cases, tracing the program execution may be required. Further discussion of static and/or dynamic watermarking issues can be found in [9, 11, 14].

Having designed a static or dynamic software watermarking algorithm for embedding a watermark w in to an application program P, it is very important to evaluate it under various criteria in order to gain information about its practical behavior. A software watermarking algorithm can be evaluated using several criteria [4]; we mention some of the most important:

(i) Embedding overhead,
(ii) Resistance to detection (stealth), and
(iii) Resilience against transformations.

Related Work. Although digital watermarking has made considerable progress and became a popular technique for copyright protection of multimedia information [8], research

on software watermarking has recently received sufficient attention. The major software watermarking algorithms currently available are based on several techniques, among which are the register allocation [15], spread-spectrum [16], code re-orderings [13], opaque predicate [12], dynamic path techniques [3]; see also, Collberg and Nagra [6] for an exposition of the main results.

In 1996, Davidson and Myhrvold [9] proposed the first software watermarking algorithm which is static and embeds the watermark by reordering the basic blocks of a control flow-graph. Based on this idea, Venkatesan, Vazirani and Sinha [14] proposed the first graph-based software watermarking algorithm which embeds the watermark by extending a method's control flow-graph through the insertion of a directed subgraph; it is also a static algorithm and is called VVS or GTW. In [14] the construction of a directed graph G (or, watermark graph G) is not discussed. Later, Collberg et al. [5] proposed an implementation of GTW, which they call GTW_{sm}; it is the first publicly available implementation of the algorithm GTW. In GTW_{sm} the watermark is encoded as a reducible permutation graph (RPG) [4], which is a reducible control flow-graph with maximum out-degree of two, mimicking real code. Note that, for encoding integers the GTW_{sm} method uses only those permutations that are self-inverting. The first dynamic watermarking algorithm (CT) was proposed by Collberg and Thomborson [7]; it embeds the watermark through a graph structure which is built on a heap at runtime.

Several software watermarking algorithms have been appeared in the literature that encode watermarks as graph structures [9, 14, 4, 5]. The authors of this paper extended the class of software watermarking algorithms and graph structures by proposing an efficient and easily implemented codec system for encoding watermark numbers as reducible permutation flow-graphs [1, 2]. They presented algorithms which encode a watermark number w as self-inverting permutation π^* and then encode π^* as a reducible permutation flow-graph $F[\pi^*]$.

Our Contribution. Recently, we have presented a dynamic watermarking model, which we called WaterRpg, for embedding a reducible permutation graph $F[\pi^*]$ into an application program P. The main idea behind the proposed watermarking model is to modify the dynamic call-graph $G(P, I_{key})$ of the program P, taken by the specific input I_{key}, so that the dynamic call-graph $G(P^*, I_{key})$ of the resulting watermarked program P^* is isomorphic to the reducible permutation graph $F[\pi^*]$; within this idea the program P^* is produced by only altering appropriate calls of specific functions of the input application program P.

In this paper, we first briefly describe the main operations and components of our dynamic watermarking model WaterRpg and show that it efficiently watermarks an application program P by embedding a reducible permutation graph $F[\pi^*]$, i.e., the graph $F[\pi^*]$ which encodes the watermark w, into P producing thus the watermarked program P^*. In fact, we present an implementation of our watermarking model WaterRpg on Java application programs downloaded from a free non commercial game database, and evaluate its functionality under various watermarking issues supported by our WaterRpg model. We selected a number of Java application programs and watermark them using two main approaches: (i) the straightforward or naive approach, and (ii) the stealthy approach. The naive approach watermarks a given program P using only the well-defined call patterns of our model, while the stealthy approach watermarks P using structural and programming properties of the call patterns.

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

2. BACKGROUND RESULTS

In this section, we present basic components and background results that are used in the design and implementation of our watermarking model WaterRpg.

2.1 Watermark Components

We consider finite graphs with no multiple edges. For a graph G, we denote by $V(G)$ and $E(G)$ the vertex set and edge set of G, respectively. We also consider permutations π^* over the set $N_n = \{1, 2, \ldots, n\}$ that are self-inverting (or, for short, SiP). Throughout the paper we denote a call-graph G of an application program P over the input I as $G(P, I)$.

A. Reducible Permutation Graphs (RPG)

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

Definition 2.2. A flow-graph is reducible when it does not have a strongly connected subgraph with two (or more) entries.

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

We have presented efficient codec algorithms for encoding an integer w as self-inverting permutation π^* and embedding it into a reducible permutation flow-graph $F[\pi^*]$ [1, 2]; see, Figure 1 and Subsection 2.2 for these results.

B. Dynamic Call-graphs

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

Definition 2.3. A dynamic call-graph G is a directed graph that includes invocations of caller–callee pairs, over an execution of the program P.

A dynamic call-graph can be considered as one instance of the corresponding static call-graph for a specific input

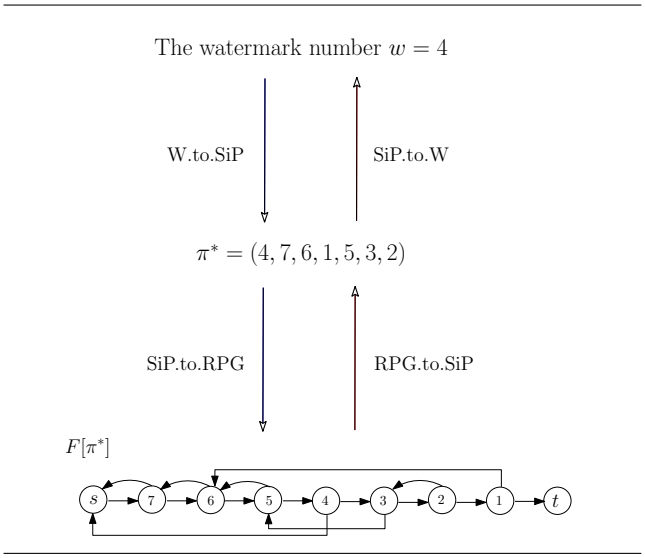

Figure 1: The main data components used by our codec algorithms (i.e., watermark w, SiP π^*, and RPG $F[\pi^*]$) and a flow of the process of encoding a watermark number w into the graph $F[\pi^*]$ and extracting it from $F[\pi^*]$.

sequence I. The call-graph G is the key data structure that dynamic optimizers use to analyze and optimize whole-program behavior. Such a graph can be extracted by a profiler. It is fair to mention that the construction of a dynamic call-graph G of a program P is not a time consuming process even if P is a large scale software.

Figures 2(a) depicts the structure of the dynamic call-graph $G(P, I_{key})$ of an application program P taken by the input I_{key}.

2.2 Encode Numbers as RPGs

In [1] we introduced the notion of *bitonic permutations* and we presented two algorithms, namely `Encode_W.to.SiP` and `Decode_SiP.to.W`, for encoding an integer w into an self-inverting permutation π^* and extracting it from π^*; see also [1]. We have actually proved the following results.

Theorem 2.2. *Let w be an integer and let $b_1 b_2 \cdots b_n$ be the binary representation of w. The algorithm Encode_W.to.SiP encodes the number w in a self-inverting permutation π^* of length $2n + 1$ in $O(n)$ time and space.*

Theorem 2.3. *Let π^* be a self-inverting permutation of length n which encodes an integer w using the algorithm Encode_W.to.SiP. The algorithm Decode_SiP.to.W correctly decodes the permutation π^* in $O(n)$ time and space.*

We have recently presented the algorithm `Encode_SiP.to.RPG` which encodes the self-inverting permutation π^* as a reducible permutation flow-graph $F[\pi^*]$ by exploiting domination relations on the elements of π^* and using an efficient DAG representation of π^* [2]. We also proposed the decoding algorithm `Decode_RPG.to.SIP`, which extracts π^* from $F[\pi^*]$ by converting first the graph $F[\pi^*]$ into a directed tree $T[\pi^*]$ and then applying DFS-search on $T[\pi^*]$. Our results presented in [2] are summarized in the following theorems.

Theorem 2.4. *Let π^* be a self-inverting permutation over the set N_n. The algorithm Encode_SiP.to.RPG encodes the permutation π^* into a reducible permutation graph $F[\pi^*]$ in $O(n)$ time and space.*

Theorem 2.5. *Let $F[\pi^*]$ be a reducible permutation graph of order $O(n)$ produced by the algorithm Encode_SiP.to.RPG. The algorithm Encode_RPG.to.SiP correctly extracts the permutation π^* from $F[\pi^*]$ in $O(n)$ time and space.*

Figure 1 depicts the main data components used by our codec algorithms, i.e., the watermark number w, the SiP π^*, and the RPG $F[\pi^*]$. The same figure shows a flow of the process of encoding a watermark number w into the graph $F[\pi^*]$ and extracting it from $F[\pi^*]$ through the use of self-inverting permutations.

2.3 The Watermarking Model

We next briefly describe the main operations and components of our watermarking dynamic model WaterRpg. It watermarks an application program P by first encoding a watermark number w as reducible permutation graph $F[\pi^*]$ and then embedding the graph $F[\pi^*]$ into P producing thus the watermarked program P^*. We point out that WaterRpg constructs execution trace watermarks.

(I) Model Operations

The main operations performed by the WaterRpg model can be outlined as follows: it first takes a specific input I_{key}, the dynamic call-graph $G(P, I_{key})$ of the original application program P, taken by the specific input I_{key}, and the graph $F[\pi^*]$, and produce the watermarked program P^* having the following key property: its dynamic call-graph $G(P^*, I_{key})$ is isomorphic to reducible permutation graph $F[\pi^*]$.

The call-graphs $G(P, I_{key})$ and $G(P^*, I_{key})$ dictate the execution flow of the original program P and the watermarked program P^*, respectively. Thus, since the call-graph $G(P, I_{key})$ is not isomorphic to $G(P^*, I_{key})$, the model controls the flow of selected function calls of P^* so that $O(P, I) = O(P^*, I)$ for every input I. Within this idea the program P^* is produced by only altering appropriate calls of specific functions of the input program P.

Figure 2 shows the dynamic call-graph $G(P, I_{key})$ of an application program P, the reducible permutation graph $F[\pi^*]$ which encodes the number $w = 4$ and the dynamic call-graph $G(P^*, I_{key})$ of the watermarked program P^*.

(II) Model Components

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

(II.a) *The Dynamic Call-graph $G(P^*, I_{key})$:* Let $G(P, I_{key})$ be the dynamic call-graph of a program P on $n + 3$ nodes $f_{main}, f_s, f_1, \ldots, f_n, f_t$ taken after running the program P with the input I_{key} and let $F[\pi^*]$ be a watermark-graph on $n + 2$ nodes. We assign the $n + 2$

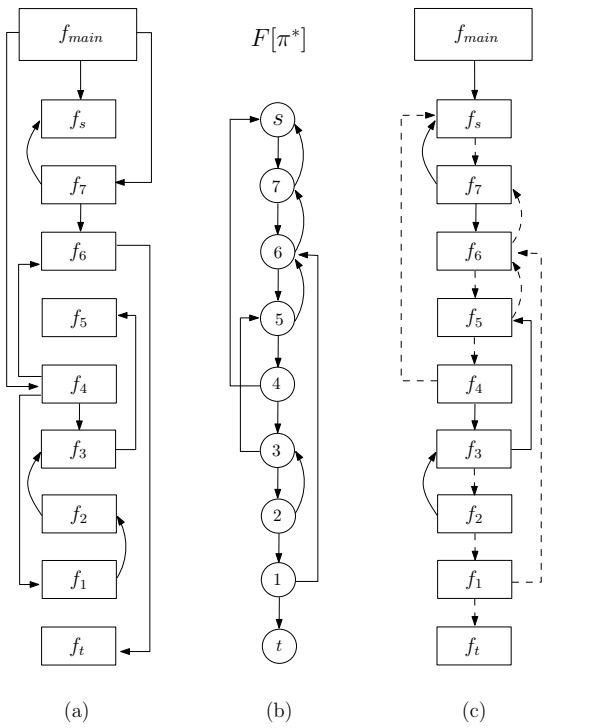

Figure 2: (a) The dynamic call-graph $G(P, I_{key})$ of an application program P. (b) The reducible permutation graph $F[\pi^*]$. (c) The dynamic call-graph $G(P^*, I_{key})$ of the watermarked program P^*.

nodes $f_s = f_{n+1}, f_n, \ldots, f_1, f_0 = f_t$ of the call-graph $G(P, I_{key})$ to $n+2$ nodes $s = u_{n+1}, u_n, \ldots, u_1, u_0 = t$ of $F[\pi^*]$ into 1-1 correspondence; the main function f_{main} do not correspond to any node of $F[\pi^*]$. The dynamic call-graph $G(P^*, I_{key})$ is constructed as follows:

- it has the same node set $V(G(P^*, I_{key}))$ as the graph $G(P, I_{key})$;
- (f_i, f_j) is an edge in $E(G(P^*, I_{key}))$ iff the corresponding (u_i, u_j) is an edge in $F[\pi^*]$.

An edge (f_i, f_j) of the call-graph $G(P^*, I_{key})$ is characterized as *real edge* if it is an edge in $G(P, I_{key})$ otherwise it is characterized as *water edge*. Moreover, if (u_i, u_j) is a forward (resp. backward) edge in the graph $F[\pi^*]$ we say that the corresponding edge (f_i, f_j) in graph $G(P, I_{key})$ is a forward (resp. backward) edge or calls. Thus, in our model the call (f_i, f_j) is either real, water, forward, or backward.

(II.b) *Call Patterns:* In the implementation phase, our model modifies the source code of program P using specific function call-patterns.

Let (f_i, f_j) be an edge of call-graph $G(P^*, I_{key})$ or, equivalently, an edge which we want to appear in graph $G(P^*, I_{key})$. Based on whether (f_i, f_j) is a real, water, forward, or backward edge, we do the following:

- if (f_i, f_j) is a *water edge* we add the statement `call`(f_j) in the function f_i, while
- if (f_i, f_j) is a *real edge* we do nothing since the statement `call`(f_j) exists in f_i.
- if (f_i, f_j) is a *forward edge* we add the statement $x = x + h()$ in function f_i before the call-point of f_j, and the statement $x = x + c()$ in f_j, while
- if (f_i, f_j) is a *backward edge* we add the statement $x = x + g()$ in function f_i before the call-site of f_j, and the statement $x = x + c()$ in f_j,

where x is a variable of type `A` and $h()$, $g()$ and $c()$ functions of the same type; we call it *cf-variable*.

Note that, in a call-graph of an application program we usually meet sequences of calls of the form $(f_i, f_{k_1}, f_{k_2}, \ldots, f_{k_m}, f_j)$. In this case, we actually have the direct calls $(f_i, f_{k_1}), (f_{k_1}, f_{k_2}), \ldots, (f_{k_m}, f_j)$ which are either forward or backward.

(II.c) *Control Statements:* In our watermarking model we use the values of the variable x of model's call patterns and include it in a specific control statement s causing thus an "appropriate execution flow" of the functions of the call-graph $G(P^*, I_{key})$; with the term "appropriate execution flow" we mean that the execution flow of the functions of the call-graph $G(P^*, I_{key})$ is such that $O(P, I) = O(P^*, I)$ for every input I.

Our model incorporates a mechanism which ensures an appropriate execution flow of the functions of the call-graph $G(P^*, I_{key})$; it alters the execution flow of the functions of the program P by modifying or adding some specific control statements. In fact, the mechanism actually modifies the conditions or expressions of these control statements by adding opaque predicates [12].

3. IMPLEMENTATION

Having described the main operations and components of our watermarking model WaterRpg, let us in this section present in detail the watermarking process of a Java application program P. In our implementation, P is a game program with market-name `Laser`; it has been downloaded from `www.java-gaming.org` web-site.

In our model the functions $f_s = f_{n+1}, f_n, \ldots, f_1, f_0 = f_t$ of the call-graph $G(P, I_{key})$ are into 1-1 correspondence with the nodes $s = u_{n+1}, u_n, \ldots, u_1, u_0 = t$ of the graph $F[\pi^*]$; recall that, $F[\pi^*]$ encodes the watermark number w.

We next present the watermarking process performed by our WaterRpg model on function $f_i =$ `up`() of the program `Laser`; the Java code of function f_i is the following:

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

First we show the naive watermarking of $f_i =$ `up`() and, then, we proceed with stealthy cases. Our model uses the

The Naive-watermarking

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

                hlth-;
        }

        b[cx+1][cy].bgr(black);
        ⋮
```

Two Case of a Stealthy-watermarking

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

    }
    b[cx+1][cy].bgr(black);
    ⋮
```

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

Figure 3: The function `up()` of the original program `Laser` watermarked with the naive approach and a stealthy approach; the functions `down()` and `health()` are both water functions and belong to category B, i.e., both are functions of $G(\texttt{Laser}, I_{key})$.

cf-variable x which increases its value by $h()$, $g()$, or $c()$; in our implementation, we take $h() = 3$, $g() = 2$, and $c() = 1$.

Naive-watermarking

Let u_i be the node of graph $F[\pi^*]$ which corresponds to $f_i = \texttt{up}()$, and let (u_i, u_j^1) and (u_i, u_j^2) be the forward and backward edges, respectively, which both are outgoing edges from node u_i, $1 \leq i \leq n$; note that $s = u_{n+1}$ has only one outgoing edge while $t = u_0$ has only one incoming edge. Let f_j^1 and f_j^2 be the two functions of $G(P, I_{key})$ which correspond to nodes u_j^1 and u_j^2, respectively; in our implementation, $f_j^1 = \texttt{down}()$ and $f_j^2 = \texttt{health}()$.

We next describe the modifications we make in function $f_i = \texttt{up}()$ according to the watermarking rules of our WaterRpg model. The watermarking process consists of the following phases:

(I) In the first phase, we include the body of the function f_i into a control statement with conditions that hold opaque predicates using the variable x; in our implementation of the naive-watermarking case, we use the `if-then-else` statement and add opaque predicates of the form `x==cf-value`; see, statement `if (x==271 && down==false){...}` of Figure 3.

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

- Statement $\texttt{call}(f_j^1)$: we add the statement $x = x + h()$ in a call-point before that of $\texttt{call}(f_j^1)$ and include both $x = x + h()$ and $\texttt{call}(f_j^1)$ into a control statement with opaque predicates using the variable x; we call such a statement *f-statement*.
- Statement $\texttt{call}(f_j^2)$: we similarly handle this statement but we add the statement $x = x + g()$ instead of $x = x + h()$; we call such a statement *b-statement*.

(II) In the case where the function f_i does not contain any statement $\texttt{call}(f_j^1)$, we locate a call-point before that of the control statement of Phase I and add the statements $x = x + h()$ and $\texttt{call}(f_j^1)$ in this order; then, we include both statements into a control statement with conditions that hold opaque predicates using the variable x; recall that, $h() = 3$; see, statement `if (x==271 && down==true){...}` of Figure 3.

(III) We handle in a similar way the case where the function f_i does not contain any statement $\texttt{call}(f_j^2)$; indeed, we locate a call-point before that of the control statement of Phase II, and add the statements $x = x + g()$ and $\texttt{call}(f_j^2)$ in this order; we also include both statements into a control statement as in Phase II; see, statement `if (x==268){...}` of Figure 3.

(IV) In this phase, we locate a call-point before that of the control statement of Phase III, add the statement $x = x + c()$ and include it into a `if-then-else` control statement with conditions that hold opaque predicates using the variable x; in our implementation $c() = 1$; we add this statement since in the program P^* there exists at least one function f_k such that (f_k, f_i), and thus according to our model we have to add the statement $x = x + c()$ in f_i; see, statement `if (x==267){...}` of Figure 3.

(V) In the last phase we handle all the callee functions f_j^* of f_i that are functions of the call-graph $G(P, I_{key})$ except of f_j^1 and f_j^2. For every direct call (f_i, f_j^*) we compute the sequence $(f_i, f_{k_1}, \ldots, f_j^*)$ which corresponds

to the shortest path $(u_i, u_{k_1}, \ldots, u_j^*)$ from u_i to u_j^* in graph $F[\pi^*]$; then, we remove the statement `call`(f_j^*) from f_i and add either the statements $x = x + h()$ and `call`(f_j^1) if (u_i, u_{k_1}) is a forward edge or the statements $x = x + g()$ and `call`(f_j^1) if (u_i, u_{k_1}) is a backward edge in $F[\pi^*]$; in any case, we include the statements into a control statement with conditions that hold opaque predicates using the variable x; we call such a statement *p-statement*.

All the rest callee functions f_j^{**} of the function $f_i =$ `up`() are ignored during the process of watermarking since they are not executed with the input I_{key}.

Stealthy-watermarking

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

(S.i) *Making nested patterns*: We can merge f-statements and b-statements in any way; for example, we can include the control b-statement `if (x==268){...}` inside the f-statement `if (x==271 && down==true){...}` after the statement `call`(f_j^1) = `up`(); we appropriately change their opaque predicates; see, middle code in Figure 3.

(S.ii) *Removing control statements*: We can remove the control statement that includes the statement $x = x + c()$ of a function f_i (Phase V); we can do that in the case where f_i is called by a function of category C; note that, functions of category C do not modify the value of the cf-variable x; see, stealthy codes in Figure 3.

(S.iii) *Merging control statements*: We can merge control statements that we added in program P^* with program's original control statements by appropriately merging their corresponding logical expressions; see, stealthy codes in Figure 3.

4. EXPERIMENTAL EVALUATION

In the literature, several criteria have been appeared and used for software watermarking evaluation purposes. It is a common belief that a good watermarking system must have the following properties [4]:

- High performance: the watermarking should not adversely affect the size and execution time of the watermarked program P^*;
- High data rate: the ratio of the number of bits encoded by the watermark w to the total size of the watermark should be high;
- High resiliency: a watermarking system must be resilient against a reasonable set of de-watermarking attacks;
- High stealth: both P and P^* should have similar statistical properties.

Table 1: Number of Calls in `Laser` and `Laser`*

Nodes in $F[\pi^*]$	Calls in `Laser`	Calls in `Laser`*	Real-calls `Laser`*	Water-calls `Laser`*
11	32	48	20	28
13	32	51	21	30
15	32	56	19	37

In order to gain information about the practical behavior of our WaterRpg watermarking model we implemented it on several Java application programs and experimentally evaluated it under various criteria.

More precisely, we selected a number of Java application programs, watermarked them using the two watermarking approaches supported by our WaterRpg model, i.e.,

(i) the Naive approach, and

(ii) the Stealthy approach,

and carried out an experimental study focusing on the following criteria:

(A) Time and space performance;

(B) Bytecode instructions overhead;

(C) Resilience and stealth.

The selected Java application programs are downloaded from a free non commercial game database; they have been downloaded from `www.java-gaming.org` web-site. All the programs are almost of the same size and are watermarked by embedding a graph $F[\pi^*]$ of three different sizes; in our implementation we use watermarking graphs $F[\pi^*]$ having number of nodes $n = 11$, $n = 13$, and $n = 15$. Indicatively, Table 1 shows the number of function calls in program `Laser` and its watermarked program `Laser`* for each of the three sizes of the watermarked graph $F[\pi^*]$.

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

(A) Time and Space Performance

In order to evaluate the performance of our WaterRpg model we choose the parameters (i) execution time, (ii) disk usage, and (iii) heap space usage.

We measure the execution time, the disk usage, and heap space usage of the selected Java application programs P and the corresponding watermarked programs P^* under both the naive and stealthy approaches. In the evaluation process, each program is executed "n" times with different inputs. The runtime of each tested program is computed by taking the the difference of the start-value and the end-value of the Java method `System.currentTimeMillis()`.

The execution time overhead is proportional to the size of the watermarking graph $F[\pi^*]$. The experimental results in Table 2 indicate that for a graph $F[\pi^*]$ on $n = 11$, $n = 13$ and $n = 15$ nodes the execution time of the naive watermarking causes a slight increase of 5.25%, 7.65% and 11.07%, respectively, while the corresponding increments for the stealthy case are even smaller.

Table 2: Execution Time (msec)

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

Table 3: Disk Usage (Kb)

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

Table 4: Heap Space Usage (Mb)

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

The storage requirements of programs P^* compared to P increases as the number of nodes of the graph $F[\pi^*]$ increases. Applying the stealthy approach a noteworthy amount of storage memory is saved because many of the control statements and opaque predicates that were not necessary to maintain proper functionality of the program P^* removed safely from the code. Table 3 illustrates the percentages incensement of disk demand for P_N^* and P_S^*, as well as the improvement caused by the stealth approach in comparison to the naive.

The experimental results show that our WareRpg watermarking model has a similar performance for the heap space usage; see, Table 4. The results for all the evaluating parameters are also sowed in a graphical form in Figure 4.

Table 5: Bytecode Instructions

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

Table 6: Indicative Bytecode Instructions

Bytecode	Original `Laser`	Naive `Laser`*	Stealthy `Laser`*
Control Statements			
`if_icmpne`	19	78	57
`ifne`	3	4	4
`goto`	43	45	45
Invocations			
`invokevirtual`	188	208	208
Assignments			
`iconst_1`	186	202	202
`getstatic`	368	614	529
`iadd`	84	132	132
`aload_0`	136	156	156
Rest Instructions			
`dup`	33	33	33
`ldc`	19	19	19

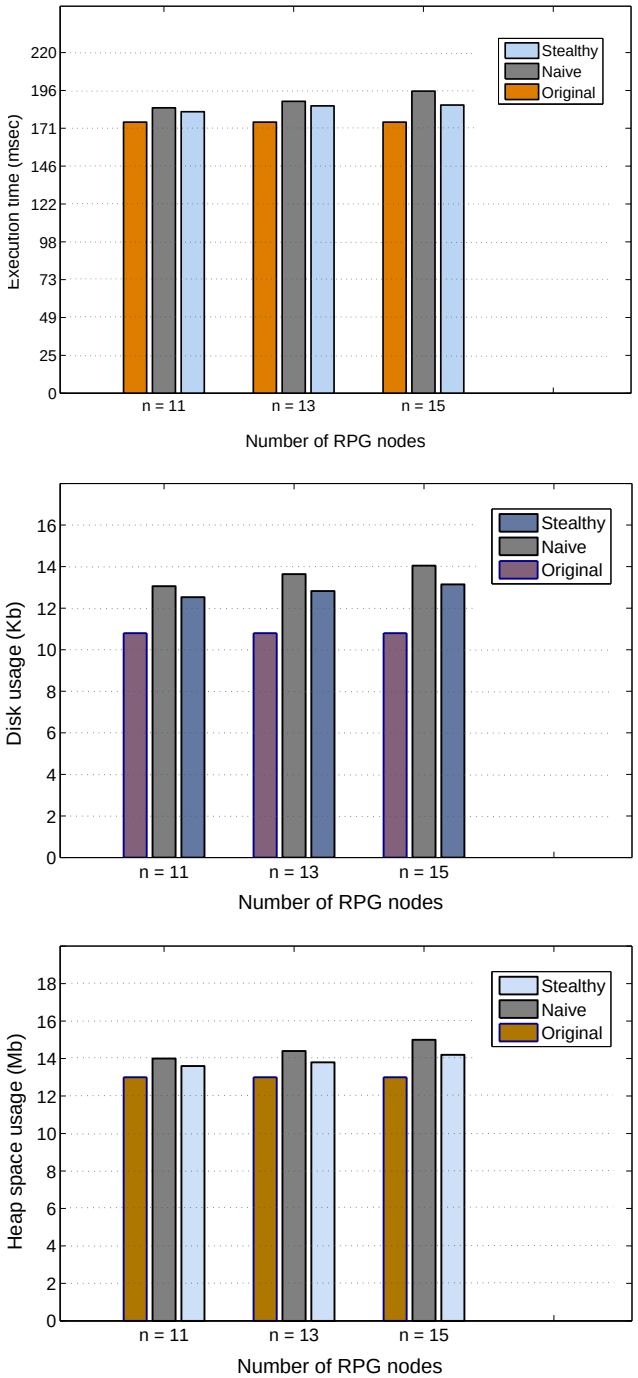

Figure 4: Graphical representation of the results for parameters Execution time, Disk usage and Heap space usage of P, Naive P^* and Stealthy P^*.

(B) Bytecode Instructions Overhead

Towards the evaluation of the data payload of our watermarking method we compute the the total amount of the

bytecode instructions added to watermarked program P^*. In particular, we compute the percentage of the increment resulted by adding control statements, functions calls and variable assignments to the program P. To this end, we count the bytecode instructions of watermarked programs P_N^* and P_S^* that belong to four main categories: (i) Control statements, (ii) Invocations, (iii) Assignments, and (iv) Rest instructions; see, Table 5. Note that the category (iv) contains all the bytecode instructions that remain unchanged after the watermarking process.

Table 6 shows some bytecode instructions of the application program `Laser`.

(C) Resilience and Stealth

Resiliency refers to the ability to recognize a watermark even after the watermarked program has been attacked or subjected to code transformations such as translation, optimization and obfuscation [12].

A watermarking system must be resilient against a reasonable set of de-compilation attacks. In our experimental study, we have also included the evaluation of our watermarking model WaterRpg against de-compilation attacks. Indeed, we tested our programs with Java De-compiler (JD-GUI) [10] and figured out that in all the cases WaterRpg successfully extracts the watermarking graph $F[\pi^*]$ from the watermarked programs P_N^* and P_S^*; indeed, in all the cases the dynamic call-graph $G(P^*, I_{key})$ taken by the input I_{key} were isomorphic to graph $F[\pi^*]$.

Moreover, the watermark code embedded to a program should be locally indistinguishable from the rest of the program so that it is hidden from malicious users. The code embedded to P by our watermarking model WaterRpg is not highly unusual, and thus it is quite difficult to locate and remove it from P^*. More precisely, in our work we do not add dead or dummy code but only programs' functions, and control statements and variable assignments, where in the stealthy case most of them are already used in the source code. The experimental results indicate that there is an increment from 10.6% up to 32.4% of function calls, control statements, and assignment in the stealthy case; see, Table 5.

5. CONCLUDING REMARKS

In this paper we presented a dynamic watermarking model for embedding a reducible permutation graph $F[\pi^*]$ into an Java application program P and evaluated it under several and broadly used watermarking criteria.

We point out that the number of nodes of the graph $F[\pi^*]$ affects the number of functions we use for embedding. Thus, it is possible to use less functions which would result in a graph $F[\pi^*]$ with fewer nodes. We consider that the selected graph sizes satisfy our evaluating criteria; note that, the graph $F[\pi^*]$ on $n = 2k+1$ nodes can encode a watermarking integer w of the range $[0, 2^{k-1} - 1]$; see, [2, 1].

The experimental results show the efficient functionality of all the Java programs P^* watermarked under both the naive and stealthy cases. The experiments also show that the watermarking approaches supported by our model can help develop efficient watermarked Java programs with respect to resilience, size, time, space, and other watermarking metrics.

6. REFERENCES

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

[2] M. Chroni and S. Nikolopoulos. An efficient graph codec system for software watermarking. In *36th IEEE Conference on Computers, Software, and Applications (COMPSAC'12)*, volume IEEE, pages 595–600, 2012.

[3] C. Collberg, E. Carter, S. Debray, A. Huntwork, J. Kececioglu, C. Linn, and M. Stepp. Dynamic path-based software watermarking. In *Proc. ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 107–118, 2004.

[4] C. Collberg, E. Carter, S. Kobourov, and C. Thomborson. Error-correcting graphs for software watermarking. In *Proc. 29th Workshop on Graphs in Computer Science (WG'03)*, volume LNCS 2880, pages 156–167, 2003.

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

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

[7] C. Collberg and C. Thomborson. Software watermarking: models and dynamic embeddings. In *Proc. 26th ACM SIGPLAN-SIGACT on Principles of Program. Languages (POPL'99)*, pages 311–324, 1999.

[8] I. Cox, J. Kilian, T. Leighton, and T. Shamoon. A secure, robust watermark for multimedia. In *Proc. 1st Int'l Workshop on Information Hiding*, volume LNCS 1174, pages 317–333, 1996.

[9] R. Davidson and N. Myhrvold. Method and system for generating and auditing a signature for a computer program. *US Patent*, 5.559.884, 1996.

[10] J. Decompiler. Retrieved December 2012, from http://java.decompiler.free.fr/, 2012.

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

[12] G. Myles and C. Collberg. Software watermarking via opaque predicates: implementation, analysis, and attacks. *Elect. Commerce Research*, 6:155–171, 2006.

[13] B. Sharma, R. Agarwal, and R. Singh. An efficient software watermark by equation reordering and fdos. In *SocProS (2)*, volume 131, pages 735–745, 2011.

[14] R. Venkatesan, V. Vazirani, and S. Sinha. A graph theoretic approach to software watermarking. In *Proc. 4th Int'l Workshop on Information Hiding (IH'01)*, volume LNCS 2137, pages 157–168, 2001.

[15] L. XiaoCheng and C. Zhiming. Software watermarking algorithm based on register allocation. In *Ninth Int'l Symposium on Distributed Computing and Applications to Business Engineering and Science (DCABES'10)*, pages 539–543, 2010.

[16] X. Zhang, Z. Zhang, and C. Zhang. Spread spectrum-based fragile software watermarking. In *15th North-East Asia Symposium on Inform. Technology and Reliability (NASNIT)*, pages 150–154, 2011.