Watermarking Java Application Programs using the WaterRpg Dynamic Model

Ioannis Chionis   Maria Chroni   Stavros D. Nikolopoulos

Abstract: *We have recently presented an efficient codec system for encoding a watermark number w as a reducible permutation graph $F[\pi^*]$, through the use of self-inverting permutations π^*, and proposed a dynamic watermarking model, which we named WaterRpg, for embedding the watermark graph $F[\pi^*]$ into an application program P. In this paper, we implement our watermarking model WaterRpg in real application programs, taken from a game database, and evaluate its functionality under various watermarking issues supported by our WaterRpg model. More precisely, we selected a number of Java application programs and watermark them using two main approaches. First, we show in detail a straightforward or naive approach for watermarking a given program P which is based only on the well-defined call patterns of our model, and then we prove structural and programming properties of the call patterns based on which we can watermark the program P in a more stealthy way. The experimental results show the efficient functionality of all the programs P^* watermarked under the naive-case and all the stealthy-cases. The size and the time overhead of the propose watermarking are very low.*

Key words: *software watermarking, self-inverting permutations, reducible permutation graphs, graph embedding, call-graphs, codec algorithms, implementation.*

INTRODUCTION

Software watermarking is a technique that is currently being studied to prevent or discourage software piracy and copyright infringement. The idea is similar to digital (or, media) watermarking where a unique identifier, which is called *watermark*, is embedded in image, audio, or video data through the introduction of errors not detectable by human perception [6, 10].

The *software watermarking problem* can be described as the problem of embedding a structure w into a program P such that w can be reliably located and extracted from P even after P has been subjected to code transformations such as translation, optimization and obfuscation [13].

Although digital watermarking has made considerable progress and become a popular technique for copyright protection of multimedia information [10], research on software watermarking has recently received sufficient attention; the first software watermarking algorithm was presented in 1996 by Davidson and Myhrvold [12]; it is a static algorithm and embeds the watermark by reordering the basic blocks of a control flow-graph. The major software watermarking algorithms currently available are based on several techniques, among which the register allocation [18], spread-spectrum [15], opaque predicate [4], abstract interpretation [11], dynamic path techniques [5], code re-orderings [17]; see also, Collberg and Nagra [6] and [16, 17] for an exposition of the main results. It is worth mentioning that many algorithmic techniques on software watermarking have been also patented [2, 12, 1, 3].

CompSysTech'13, June 28–29, 2013, Ruse, Bulgaria.
Copyright©2013 ACM 000-0-0000-0000-0/00/00...$00.00.

Several software watermarking algorithms have been appeared in the literature that encode watermarks as graph structures [12, 14, 7, 8]. The second and third authors of this paper have extended the class of graphs which can be efficiently used in a software watermarking system by proposing several efficient codec algorithms that embed watermark values w into a type of reducible permutation graphs $F[\pi^*]$ through the use of self-inverting permutations π^* (or, for short, SiP). Recently, they proposed a dynamic watermarking model for embedding the watermark graph $F[\pi^*]$ into an application program P. The main idea behind the proposed watermarking model is a systematic use of appropriate calls of specific functions of the program P. We point out that, the first dynamic watermarking algorithm $(\mathtt{CT})$ was proposed by Collberg and Thomborson [9]; it embeds the watermark through a graph structure which is built on a heap at runtime.

In this paper, we implement our watermarking model WaterRpg in real application programs, taken from a game database, and evaluate its functionality under various watermarking issues supported by our WaterRpg model. More precisely, 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 programs P^* watermarked under the naive-case and all the stealthy-cases. We also experimentally measure the size and the time overhead of the propose watermarking.

THE DYNAMIC WATERMARKING MODEL

We next briefly describe the main operations and components of our dynamic watermarking model which we call WaterRpg. Based on this model we can efficiently watermark 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^* (see, authors' paper).

(I) Model Operations

The main operations performed by the dynamic watermarking 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 the outputs $O(P, I) = O(P^*, I)$ 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 1 shows the dynamic call-graph $G(P, I_{key})$ of an application program P, the reducible permutation graph $F[\pi^*]$ which encodes the number $w = 4$ and the dynamic call-graph $G(P^*, I_{key})$ of the watermarked program P^*.

(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

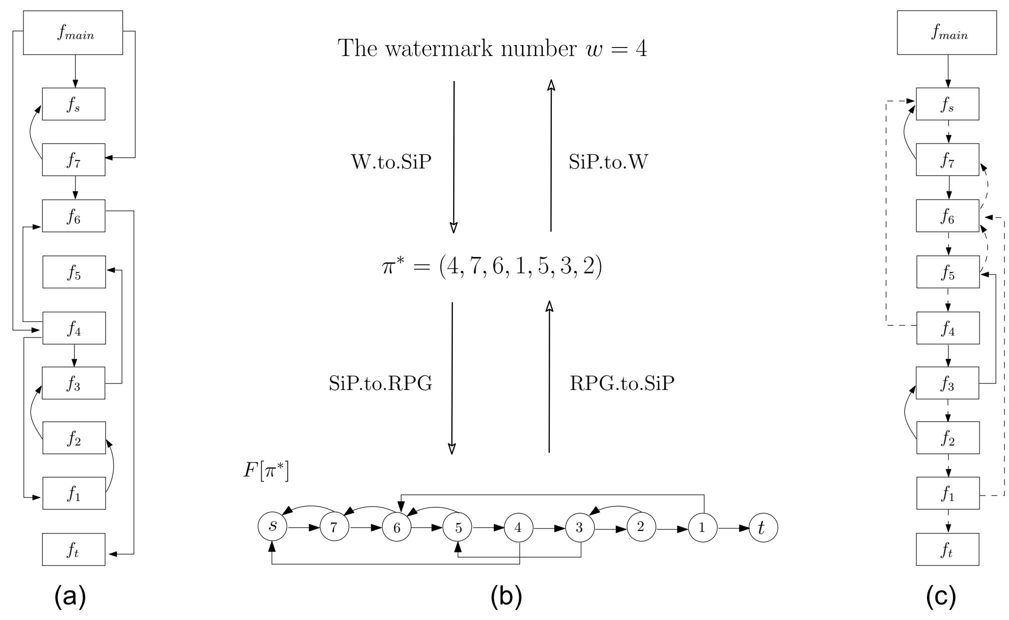

Figure 1: (a) The dynamic call-graph $G(P, I_{key})$ of an application program P. (b) The reducible permutation graph $F[\pi^*]$. (c) The dynamic call-graph $G(P^*, I_{key})$ of the watermarked program P^*.

statements which control the execution of real and water functions.

(II.a) *The Dynamic Call-graph* $G(P^*, I_{key})$*:* Let $F[\pi^*]$ be a watermark-graph on $n+2$ nodes and $G(P, I_{key})$ be the dynamic call-graph of a program P on $n+3$ nodes $f_{main}, f_s, f_1, \ldots, f_n, f_t$ taken after running the program P with the input I_{key}. We assign the $n + 2$ nodes $f_s = f_{n+1}, f_n, \ldots, f_1, f_0 = f_t$ of the call-graph $G(P, I_{key})$ to $n+2$ nodes $s = u_{n+1}, u_n, \ldots, u_1, u_0 = t$ of $F[\pi^*]$ into 1-1 correspondence; the main function f_{main} do not correspond to any node of $F[\pi^*]$. The dynamic call-graph $G(P^*, I_{key})$ is constructed as follows:

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

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. 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 which we present in a graphical way in Figure 2.

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

- if (f_i, f_j) is a water edge we add the statement `call`(f_j) in the function f_i, while
- if (f_i, f_j) is a real edge we do nothing since the statement `call`(f_j) exists in f_i.

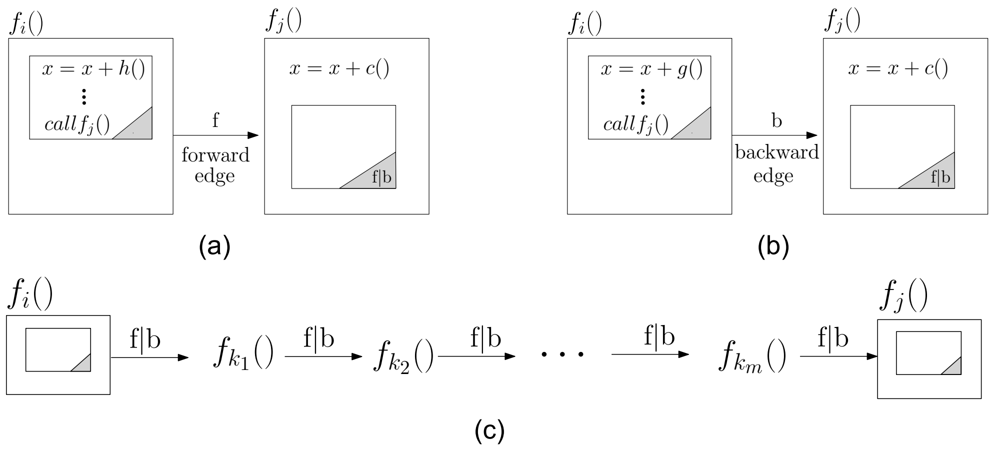

Figure 2: (a) The forward call pattern *f-call*; (b) The backward call pattern *b-call*; (c) The path call pattern *p-call*.

- 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 the function 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 the function f_j,

where x is a variable of type `A` and $h()$, $g()$ and $c()$ functions of the same type.

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}), ..., (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 the f-call and b-call patterns and include it in a specific control statement s causing thus an "appropriate execution flow" of the functions of the call-graph $G(P^*, I_{key})$; with the term "appropriate execution flow" we mean that the execution flow of the functions of the call-graph $G(P^*, I_{key})$ is such that $O(P, I) = O(P^*, I)$ for every input I.

Our model incorporates a mechanism which ensures an appropriate execution flow of the functions of $G(P^*, I_{key})$ through the altering of the execution flow of the functions of the program P by modifying or adding some specific control statements. In fact, the mechanism actually modifies the conditions or expressions of these control statements by adding opaque predicates [4, 13].

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 WaterRpg model the functions $f_s = f_{n+1}, f_n, \ldots, f_1, f_0 = f_t$ of the dynamic 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 reducible permutation graph $F[\pi^*]$; recall that, $F[\pi^*]$ encodes the watermark number w.

We next present the watermarking process performed by our WaterRpg model on the function $f_i = \texttt{up}()$ of the program `Laser`. First we show a naive case of the watermarking process of function $f_i = \texttt{up}()$ and, then, we proceed with stealthy cases. The Java code of the function $f_i = \texttt{up}()$ is the following:

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

Our model uses the 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 $f_j^2 = \texttt{health}()$.
Before we proceed to watermarking the function f_i, we divide the callee functions of f_i into the following three categories:

A: contains the functions f_j^1 and f_j^2 which correspond to forward node u_j^1 and backward node u_j^2 of graph $F[\pi^*]$, respectively;

B: contains the functions f_j^* of the dynamic call-graph $G(P, I_{key})$ except of f_j^1 and f_j^2;

C: contains the functions f_j^{**} which are not executed with the input I_{key}.

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 case, we use the `if-then-else` statement and add opaque predicates of the form `x==value`; see, statement `if (x==271 && down==false){...}` of Figure 3.

Then, we handle the functions f_j^1 and f_j^2 of categories A; 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; in our implementation $f_j^1 = \texttt{down}()$ and $h() = 3$; 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 $x = x + h()$; in our implementation $f_j^2 = \texttt{health}()$ and $g() = 2$; we call such a statement *b-statement*.

(II) In the case where the function f_i does not contains 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.

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

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].bgr() ...
        && x==268){
            hlth--;
    }
    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 $G(\texttt{Laser}, I_{key})$.

(III) We handle in a similar way the case where the function f_i does not contains 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 functions f_j^* of category B, that is, the callee functions of f_i that are functions of the call-graph $G(P, I_{key})$ except of f_j^1 and f_j^2. For every direct call (f_i, f_j^*) we compute the sequence $(f_i, f_{k_1}, \ldots, f_j^*)$ which corresponds to the shortest path $(u_i, u_{k_1}, \ldots, u_j^*)$ from u_i to u_j^* in graph $F[\pi^*]$; then, we remove the statement $\texttt{call}(f_j^*)$ from f_i and add either the statements $x = x + h()$ and $\texttt{call}(f_j^1)$ if (u_i, u_{k_1}) is a forward edge or the statements $x = x + g()$ and $\texttt{call}(f_j^1)$ if (u_i, u_{k_1}) is a backward edge in $F[\pi^*]$; in any case, we include the statements into a control statement with conditions that hold opaque predicates using the variable x; we call such a statement *p-statement*.

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

Stealthy-watermarking

Having described the naive case of the watermarking process of function $f_i = \texttt{up}()$, we next show properties and modification rules of the model's call patterns based on which we can stealthily watermark a Java application program P. The main modification cases, which we call stealthy cases, supported by the WaterRpg model are the following:

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

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

St.3 *Removing control statements*: We can remove the control statement that includes the statement $x = x + c()$ of a function $f_i = \texttt{up}()$ (Phase 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 are not modify the value of the cf-variable x.

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

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

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

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

CONCLUDING REMARKS

An interesting open question is whether the watermarking approaches and techniques provided by the WaterRpg model can help develop efficient watermarked Java programs with respect to resilience, size, time, space, or other watermarking metrics; we leave the experimental evaluation of our WaterRpg model as a problem for future investigation.

REFERENCES

[1] W.G. Horne, U. Maheshwari, R.E. Tarjan, J.J. Horning, W.O. Silbert, L. R. Matheson, A.K. Wright, and S.S. Owicki, "Systems and methods for watermarking software and other media," US Patent 8.140.850, 2012.

[2] C.S. Collberg, C.D. Thomborson, J.J. Horning, W.O. Silbert, L. R. Matheson, A.K. Wright, and S.S. Owicki, "Software watermarking techniques," US Patent 2011/0214188, 2011.

[3] T.F. Rodriguez, B.T. MacIntosh, and A.E. Gustafson, "Software watermarking," US Patent 2010/0095376, 2010.

[4] G. Arboit, "A method for watermarking Java programs via opaque predicates," 5th International Conference on Electronic Commerce Research (ICECR-5), 2002.

[5] C. Collberg, E. Carter, S. Debray, A. Huntwork, J. Kececioglu, C. Linn, and M. Stepp, "Dynamic path-based software watermarking," Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, SIGPLAN 39, 2004, pp. 107–118.

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

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

[8] 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 (2009) 56–67.

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

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

[11] P. Cousot and R. Cousot, "An abstract interpretation-based framework for software watermarking," Proc. 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'04), 2004, pp. 173–185.

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

[13] G. Myles and C. Collberg, "Software watermarking via opaque predicates: Implementation, analysis, and attacks," Electronic Commerce Research 6 (2006) 155–171.

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

[15] X. Zhang, Z. Zhang, and C. Zhang, "Spread spectrum-based fragile software watermarking", 15th North-East Asia Symposium on Nano, Information Technology and Reliability (NASNIT), 2011, pp. 150–154.

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

[17] B.K. Sharma, R.P. Agarwal, and R. Singh, "An efficient software watermark by equation reordering and FDOS", Proc. Int'l Conference on Soft Computing for Problem Solving (SocProS'11), AISC 131, 2011, pp. 735–745.

[18] L.X. Cheng and C. Zhiming, "Software watermarking algorithm based on register allocation", Proc. 9th Int'l Symposium on Distributed Computing and Applications to Business Engineering and Science (DCABES), 2010, pp. 539–543.

ABOUT THE AUTHORS

Ioannis Chionis, BSc, MSc Candidate, Department of Computer Science & Engineering, University of Ioannina, Phone: +30-265-100-8831, e-mail: `ichionis@cs.uoi.gr`

Maria Chroni, MSc, PhD Candidate, Department of Computer Science & Engineering, University of Ioannina, Phone: +30-265-100-8901, e-mail: `mchroni@cs.uoi.gr`

Stavros D. Nikolopoulos, PhD, Professor, Department of Computer Science & Engineering, University of Ioannina, Phone: +30-265-100-8801, e-mail: `stavros@cs.uoi.gr`