Information and Software Technology 51 (2009) 56–67

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

More on graph theoretic software watermarks: Implementation, analysis, and attacks ☆

Christian Collberg [a,*], Andrew Huntwork [b], Edward Carter [c], Gregg Townsend [a], Michael Stepp [d]

[a] Department of Computer Science, The University of Arizona, Tucson, AZ 85721, United States
[b] Amazon.com, 1200 12th Ave S., Suite 1200, Seattle, WA 98144, United States
[c] Department of Mathematics, University of California, Berkeley, CA 94720, United States
[d] Department of Computer Science, University of California, San Diego, La Jolla, CA 92093, United States

ARTICLE INFO

Keywords:
Software watermarking
Software protection
Surreptitious software
Software piracy protection

ABSTRACT

This paper presents an implementation of the watermarking method proposed by Venkatesan et al. in their paper [R. Venkatesan, V. Vazirani, S. Sinha, A graph theoretic approach to software watermarking, in: Fourth International Information Hiding Workshop, Pittsburgh, PA, 2001]. An executable program is marked by the addition of code for which the topology of the control-flow graph encodes a watermark. We discuss issues that were identified during construction of an actual implementation that operates on Java bytecode. We present two algorithms for splitting a watermark number into a redundant set of pieces and an algorithm for turning a watermark number into a control-flow graph. We measure the size and time overhead of watermarking, and evaluate the algorithm against a variety of attacks.

1. Introduction

This paper builds upon and elaborates a software watermarking scheme proposed by Venkatesan et al. [3]. We will refer to that paper as *VVS* and to its watermarking scheme as *GTW*. The present paper contributes:

- The first public implementation of GTW.
- An implementation that operates on Java bytecode.
- An example of an error-correcting graph encoding.
- The generation of executable code from graphs.
- Several alternatives for marking basic blocks.
- Extraction (not just detection) of a watermark value.
- Empirical measurements of an actual GTW implementation.
- Experimental analysis of possible attacks.

Graph theoretic watermarking encodes a value in the topology of a *control-flow graph*, or CFG [4]. Each node of a CFG represents a *basic block* consisting of instructions with a single entry and a single exit. A directed edge connects two basic blocks if control can pass from one to the other during execution. The CFG itself also has a single entry and a single exit.

A watermark graph W is merged with a target program's graph P by adding extra control-flow edges between them. Basic blocks belonging to W are *marked* to distinguish them from the nodes of P. These marks are later used to extract W from $P+W$ during the recognition process. The *GTW* process is illustrated in Fig. 1.

The *VVS* paper hypothesizes that naïvely inserted watermark code is weakly connected to the original program and is therefore easily detected. Weakly connected graph components can be identified using standard graph algorithms and can then be manually inspected if they are few in number. Such inspection may reveal the watermark code at much lower cost than manual inspection of the full program.

The attack model of *VVS* considers an adversary who attempts to locate a *cut* between the watermark subgraph and the original CFG (dashed edges in Fig. 1). The *GTW* algorithm is designed to produce a strongly connected watermark so that such a cut cannot be identified. The *VVS* paper proves that such a separation is unlikely. More formally, the *GTW* algorithm adds edges between the program P and the watermark W in such a way that many other node divisions within P have the same size cut as the division between P and W.

We have implemented the *GTW* algorithm in the framework of SANDMARK [5], a tool for experimenting with algorithms that protect software from reverse engineering, piracy, and tampering. SANDMARK contains a large number of obfuscation and watermarking algorithms as well as tools for manual and automatic analysis and reverse engineering. SANDMARK operates on Java bytecode. It can be downloaded for experimentation from `sandmark.cs.arizona.edu`.

☆ This paper extends material previously published in Refs. [1,2].

* Corresponding author.
E-mail addresses: collberg@cs.arizona.edu (C. Collberg), ash@huntwork.net (A. Huntwork), ecarter@math.berkeley.edu (E. Carter), gmt@cs.arizona.edu (G. Townsend), mstepp@cs.ucsd.edu (M. Stepp).

0950-5849/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2008.09.016

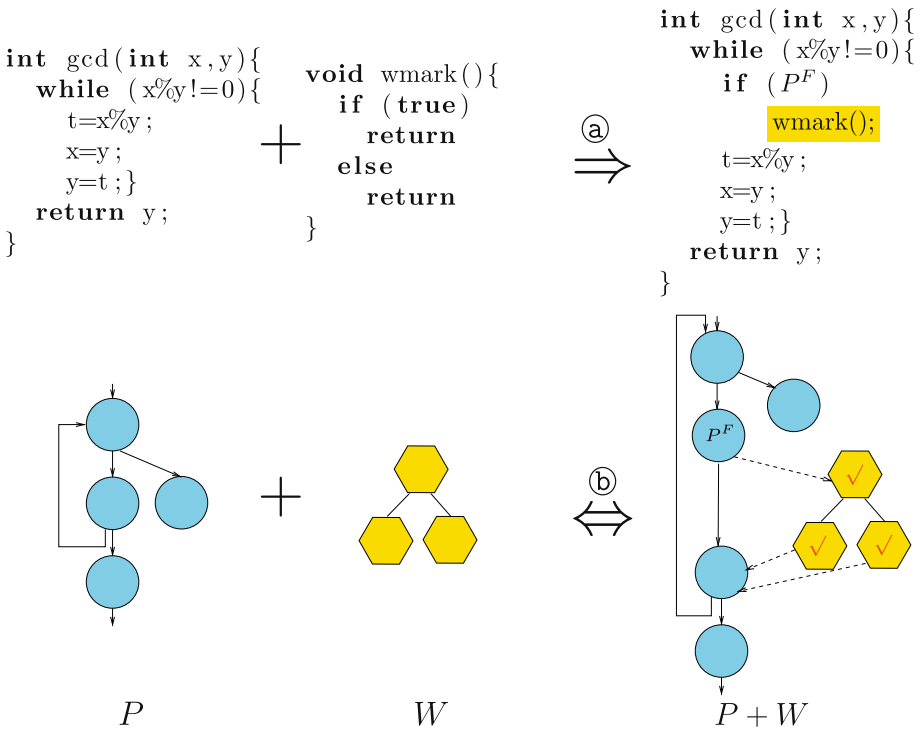

Fig. 1. Overview of graph theoretic watermarking. In ⓐ the code for watermark *W* is merged with the code for program graph *P*, by adding fake calls from *P* to *W*. In ⓑ the same process is shown using a control-flow graph notation. Part ⓑ also shows how the mark is later recovered by separating the marked (√) nodes of *W* from *P* with some tolerance for error.

Our implementation of *GTW*, which we will call GTW_{SM}, is the first publicly available implementation of the *GTW* algorithm. We have found that *GTW* can be implemented with minimal overhead, a high degree of stealthiness, and with relatively high bit-rate.

Error-correcting graph techniques make the algorithm resilient against edge-flip attacks, in which the basic blocks are reordered, but it remains vulnerable to a large number of other semantics-preserving code transformations. *GTW*'s crucial weakness is its reliance on the reliable recognition of marked basic blocks during watermark extraction. We are unaware of *any* block marking method that is invulnerable to simple attacks.

The remainder of this paper is organized as follows. Section 2 surveys related work. Section 3 presents an overview of our implementation, and Sections 4 and 5 describe the embedding and recognition algorithms in detail. Section 6 evaluates *GTW* with respect to resilience against attacks, bit-rate, and stealth. Section 7 discusses future work.

2. Related work

Davidson and Myhrvold [6] published the first software watermarking algorithm. A watermark is embedded by rearranging the order of the basic blocks in an executable. Like other order-based algorithms, this is easily defeated by a random reordering.

Qu and Potkonjak [7,8] encode a watermark in a program's register allocation. Like all algorithms based on renaming, this is very fragile. Watermarks typically do not survive even a decompilation/recompilation step. This algorithm also suffers from a low bit-rate.

Stern et al. [9] use a spread-spectrum technique to embed a watermark. The algorithm changes the frequencies of certain instruction sequences by replacing them with equivalent sequences. This algorithm can be defeated by obfuscations that modify data-structures or data-encodings and by many low-level optimizations.

Arboit's [10] algorithm embeds a watermark by adding special opaque predicates to a program. Opaque predicates are logical expressions that have a constant value, but not obviously so [11].

Watermarks are categorized as static or dynamic. The algorithms above are static markers, which embed watermarks directly within the program code or data. Collberg and Thomborson [12] proposed the first dynamic watermarking algorithm, in which the program's run-time behavior determines the watermark. Their algorithm embeds the watermark in the topology of a dynamically built graph structure constructed at runtime in a response to a particular key input sequence. This algorithm appears to be resilient to a large number of obfuscating and optimizing transformations.

Collberg et al. [1] describe another dynamic algorithm that embeds the watermark in the sequence of conditional branches taken by the program. Each taken branch represents a 1 and each untaken branch represents a 0. In this way, an execution path can spell out an arbitrary bitstring using contrived branch patterns. In addition, several techniques are described to increase the stealth of the placement of the watermark code and the contrived branch patterns.

3. An overview of GTW_{SM}

Our implementation of *GTW* operates on Java bytecode. Choosing Java lets us leverage the tools of the SANDMARK and BCEL [13] libraries, and lets us attack the results using SANDMARK's collection of obfuscators. Like every executable format, Java bytecode has some unique quirks, but the results should be generally applicable.

The *GTW* embedding algorithm takes as input application code *P*, watermark code *W*, secret keys ω_1 and ω_2, and integers *m* and *n*. GTW_{SM} uses a smaller and simpler set of parameters. Values of *m* and *n* are inferred from *P*, *W*, and ω_1. The *clustering* step (Section 4.4) is unkeyed, so ω_2 is unused. Thus, our implementation takes as input application code *P*, a secret key ω, and a watermark value.

Fig. 2 gives an overview of the GTW_{SM} embedding process which proceeds in the following steps:

1. The watermark value v is split into k values, $\{v_0, \ldots, v_{k-1}\}$ using a splitting algorithm. We present two such algorithms (Section 4.1).
2. The split values are encoded as directed graphs $\{G_0, \ldots, G_{k-1}\}$ (Section 4.2).
3. The generated graphs are converted into CFGs $\{W_0, \ldots, W_{k-1}\}$ by generating executable code for each basic block (Section 4.3).
4. Each basic block is marked to indicate whether it is part of the watermark (Section 4.6).
5. The application's clusters are identified (Section 4.4) and the watermark is merged with the application by adding bogus call-graph edges between the control-flow graphs (Section 4.5).

The recognition process described in *VVS* has three steps: detection of watermark nodes, sampling of subsets of the watermark nodes, and computation of robust properties of these subsets. The set of robust property values composes the watermark. The process is as follows: Fig. 3

1. Marked nodes of the program CFG are identified (Section 5.1).
2. The recognizer selects several subsets of the watermark nodes for decoding (Section 5.2).
3. Each subset is decoded to compute a value.
4. The individual values are combined to yield the watermark (Section 5.3).

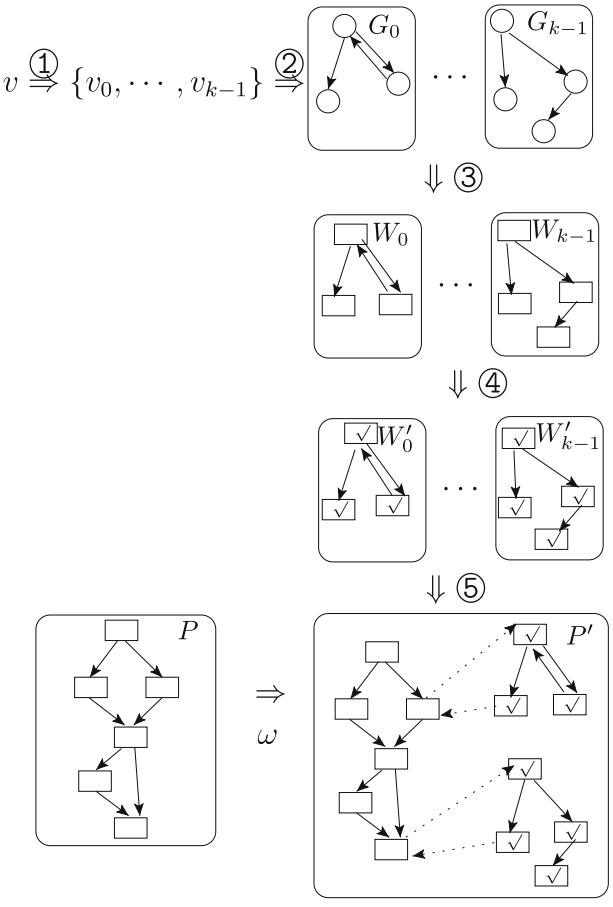

Fig. 2. An overview of the embedding process in GTW_{SM}.

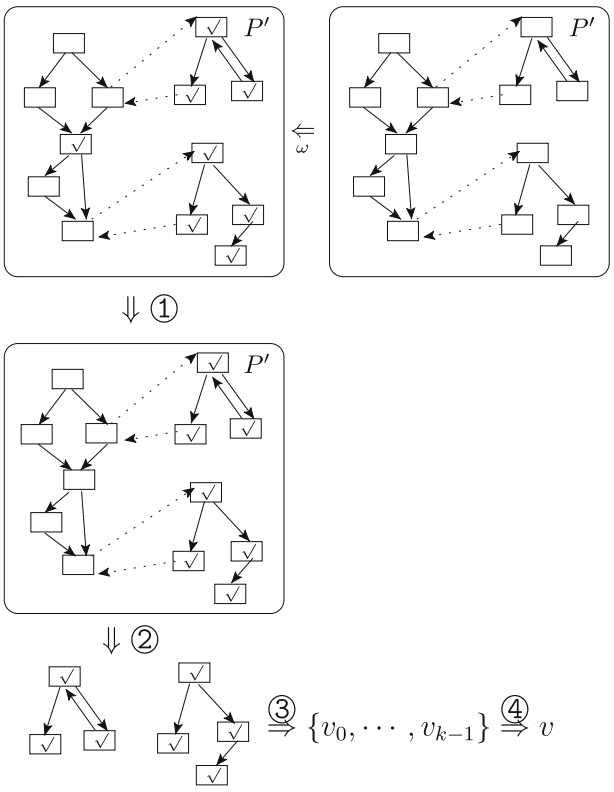

Fig. 3. An overview of the recognition process in GTW_{SM}.

3.1. Example

For a concrete example, we embed the watermark 31415926 into the Java code shown in Figs. 4 and 5, using the Partial Sum Splitter. A key of 42 seeds the random number generator.

The watermark value is split into a set of ten integers (2, 8, 14, 15, 22, 27, 33, 40, 46, and 47) using the algorithm described in Section 4.1.1. Each part is encoded as a graph using the algorithm of Section 4.2. The graph encoding of the integer 8 is shown in Fig. 8. The new methods are collected into a single new Java class and merged into the application.

A cluster graph is constructed next. In our implementation, each method is a cluster so the cluster graph is exactly the call-graph.

Random edges are now added to the cluster graph. In this trivial example, a single edge is added based on the original application's characteristic connectivity. Additional edges are added to prevent detection of the watermark through dead code analysis.

```
public class Factorial {
    public static int factorial(int n) {
        if(n < 0)
            throw new ArithmeticException();

        int result;
        for(result = 1 ; n > 1 ; result *= n--);
        return result;
    }
    public static void main(String argv[]) {
       for(int i = -1000 ; ; i++)
          try { factorial(i) ; break; }
          catch(ArithmeticException e) {}
    }
}
```

Fig. 4. Source code for a program that exits after finding a valid input to the factorial function by trying integers in ascending order starting with −1000.

```
Method int factorial(int)
   0 iload_0
   1 ifge 12                    // if (R0==0) goto 12
   4 new #2 <ArithmeticException>
   7 dup
   8 invokespecial #3
  11 athrow                     // throw new ArithmeticException();
  12 iconst_1
  13 istore_1                   // R1 = 1;
  14 goto 25
  17 iload_1
  18 iload_0
  19 dup
  20 iconst_1
  21 isub
  22 istore_0                   // R0--;
  23 imul
  24 istore_1                   // R1 *= R0
  25 iload_0
  26 iconst_1
  27 if_icmpgt 17               // if (R0>1) goto 17
  30 iload_1                    // return R1
  31 ireturn

Method void main(String[])
   0 sipush -1000
   3 istore_1                   // R1 = -1000
   4 goto 7
   7 iload_1
   8 invokestatic #4 <factorial>   // call factorial(R1)
  11 pop
  12 goto 25
  15 astore_2
  16 goto 19
  19 iinc 1 1                   // R1++
  22 goto 7
  25 return
Exception table:
   from   to  target type
     7    12    15   <Class ArithmeticException>
```

Fig. 5. Bytecode for the code shown in Fig. 4.

Finally, every block in the program is inspected, and altered if necessary, to indicate whether it is part of a watermark method; this inserts load/add/store sequences throughout the application (see Fig. 6). Section 4.6 describes the marking algorithm.

Recognition begins by detecting marked blocks, and consequently marked methods, as described in Section 5. For the example program, the watermark methods (`m0`,`m2`,...,`m9`) are found to be marked. The control-flow graph of each marked block is decoded as described in Section 4.2. The complete set of integers (2, 8, 14, 15, 22, 27, 33, 40, 46, and 47) is then combined to yield 31415926, the original watermark value.

4. Embedding

The construction of a watermark graph W is not discussed in *VVS*. In GTW_{SM} we accept an integer value for transformation into a watermark CFG. The recognition process performs the inverse transformation from CFG to integer.

The embedding process involves several steps: splitting the watermark value into small integers; constructing directed graphs that encode these values; generating code that corresponds to the graphs; and connecting the code to the program. We present two distinct methods of splitting watermarks into small integers, and compare their effectiveness in Section 6.

4.1. Watermark value splitting

We next present two algorithms for splitting a watermark integer.

4.1.1. Partial sum splitter

The partial sum splitter splits a watermark value v into a multiset S of k integers, $k \geqslant 2$. Empirically, we have determined that values of k between 5 and 15 produce watermark methods that are neither overly large nor overly numerous.

```
Method int factorial(int)
   0 nop
   1 iload_0
   2 getstatic #57 <dummy>
   5 iconst_1
   6 iadd
   7 putstatic #57 <dummy>      // dummy++;
  10 ifge 33                    // if (R0>=0) goto 33
  13 iconst_0
  14 invokestatic #35 <m2>      // call m2(0);
  17 pop
  18 nop
  19 iconst_0
  20 invokestatic #32 <m9>      // call m9(0)
  23 pop
  24 nop
  25 new #2 <ArithmeticException>
  28 dup
  29 invokespecial #3
  32 athrow                     // throw new ArithmeticException()
  ...

Method int m4(int)
   ...
   7 istore_0                   // R0 = (31>>R0)^R0;
   8 iload_0
   9 iconst_3
  10 ishr
  11 getstatic #29 <dummy>
  14 iconst_1
  15 iadd
  16 putstatic #29 <dummy>      // dummy++;
  19 istore_0                   // R0 = 3>>R0;
  20 iload_0
  21 iconst_4
  22 iushr
  23 istore_0                   // R0 = 4>>R0;
  24 iload_0
  25 getstatic #29 <dummy>
  28 iconst_3
  29 iadd
  30 putstatic #29 <dummy>      // dummy+=3;
  33 ifeq 47                    // if (R0==0) goto 47
```

Fig. 6. Disassembly of watermarked application code (excerpts).

A watermark value v is split as follows:

1. Compute the minimum exponent l such that v can be represented using $k-1$ digits of base 2^l.
2. Split the value v into digits $v_0, v_1, \ldots, v_{k-2}$ such that $0 \leqslant v_j < 2^l$ and $v = \sum_{j=0}^{k-2} 2^{jl} v_j$.
3. Encode the digits in the multiset $\{s_0, s_1, \ldots, s_{k-1}\}$ where $s_0 = l - 1$ and $s_i = s_{i-1} + v_{i-1}$.

For a concrete example, consider splitting a watermark value of 31415926 with $k = 10$. The minimum radix is 8, so $l = 3$. This produces a list v_i of 6, 6, 1, 7, 5, 6, 7, 6, 1 and finally the multiset {2, 8, 14, 15, 22, 27, 33, 40, 46, 47}.

4.1.2. GCRT splitter

This algorithm makes use of the *generalized Chinese remainder theorem*. The process of turning W into a set of values to be embedded into the program consists of several steps, illustrated with an example in Fig. 7 (description taken from [1]).

1. Let $p_1, \ldots, p_r$ be pairwise relatively prime, where $W < \prod_{k=1}^{r} p_k$. W is split into up to $\frac{r(r-1)}{2}$ pieces, each piece being of the form $W \equiv x_k \bmod p_{i_k} p_{j_k}$, where $0 \leqslant x_k < p_{i_k} p_{j_k}$, $i_k < j_k$. This is step Ⓐ in Fig. 7. The generalized Chinese remainder theorem [14] allows W to be reconstructed from these statements in a straightforward manner, since the p's are pairwise relatively prime.

$$
\begin{array}{lll}
W = 17 & \xRightarrow{\text{Ⓐ}} & \begin{array}{rcl} W & \equiv & 5 \bmod p_1p_2 \\ W & \equiv & 7 \bmod p_1p_3 \\ W & \equiv & 2 \bmod p_2p_3 \end{array} \\
& \xRightarrow{\text{Ⓑ}} & \begin{array}{rcl} 5 & = & 5 \\ p_1p_2 + 7 & = & 13 \\ p_1p_2 + p_1p_3 + 2 & = & 18 \end{array} \\
& \xRightarrow{\text{Ⓒ}} & \text{insert into program}
\end{array}
$$

Fig. 7. Splitting the watermark value $W = 17$ via the generalized Chinese remainder theorem, with $p_1 = 2, p_2 = 3, p_3 = 5$ (diagram taken from [1]).

2. Each statement $W \equiv x_k \bmod p_{i_k}p_{j_k}$ is turned into a single integer by an enumeration scheme. In our scheme,

$$w_k = x_k + \sum_{n=1}^{i_k-1} \sum_{m=n+1}^{r} p_n p_m + \sum_{m=1}^{j_k-1} p_{i_k} p_m.$$

   This is step Ⓑ.
3. In step Ⓒ, each piece w_k is encrypted using a block cipher. This step enables us to make randomness assumptions about any corrupted data when decoding.

4.2. Encoding integers as graphs

Each integer is converted into a graph for embedding in the application. Several issues must be considered when choosing a graph encoding:

1. The graph must be a *digraph* (a directed graph) for use as a CFG.
2. The graph must have the structure of a valid CFG. It should have a header node with in-degree zero and out-degree one from which every node is reachable, and it should have a footer node with out-degree zero that is reachable from every node.
3. The graph should have a maximum out-degree of two. Basic block nodes with out-degrees of one or two are easily generated using standard control-structures such as if- and while-statements. Nodes with higher out-degree can only be built using switch-statements. These are relatively unusual in real code, and hence conspicuous.
4. The graph should be *reducible* [4], because true Java code produces only reducible graphs. Intuitively, a CFG is reducible if it is compiled from properly nested structured control constructs such as if- and while-statements. More formally, a reducible flow graph with root node r has edges that can be split into an acyclic component and a component of back-edges, where each back-edge (u, v) has the property that every path from r to u passes through v. In this case, v is said to *dominate* u.
5. The control-structures represented by the graph should not be deeply nested, because real programs seldom nest deeply.

In GTW_{SM} each part of the split watermark is encoded as a *reducible permutation graph*, or RPG [15]. These are reducible control-flow graphs with a maximum out-degree of two, mimicking real code. They are resilient against edge-flip attacks and can be correctly decoded even if an attacker rearranges the basic blocks of a method.

An RPG is a reducible flow graph with a Hamiltonian path consisting of four pieces (see Fig. 8):

(a) *A header node*:
The root node of the graph having out-degree one from which every other node in the graph is reachable. Every control-flow graph has such a node.

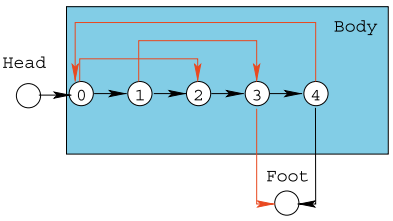

Fig. 8. Reducible permutation graph of the integer value 8.

(b) *The preamble*:
Zero or more additional initial nodes from which all later nodes are reachable. Any node in the body can have an edge to any node in the preamble while preserving reducibility.

(c) *The body*:
The set of nodes used to encode a value. Edges within the body, from the body to the preamble, and from the body to the footer node encode a permutation that is its own inverse.

(d) *A footer node*:
A node with out-degree zero that is reachable from every other node of the graph. This node represents the method exit.

There is a one-to-one correspondence between self-inverting permutations and isomorphism classes of RPGs, and this correspondence can be computed in polynomial time. An RPG encoding a permutation on n elements has a bitrate of at least $\frac{1}{4}\log n - 0.62$ bits per node [15].

For encoding integers we use only those permutations that are their own inverses, as this greatly reduces the need for a preamble. An integer n is encoded as the RPG corresponding to the nth self-inverting permutation, using the enumeration of Collberg et al. [15].

4.3. Generating code from a graph

A graph is embedded in an application by building a set of instructions that have a corresponding CFG. We want to generate code (in this case Java bytecode) that is stealthy, executable, and efficient. In *VVS* it is expected that watermark code be connected to the application by means of opaque predicates, and hence never executed. This leaves the watermarked application open to tracing attacks. In GTW_{SM}, we generate executable watermark code that has no semantic effect on the program.

Given a graph, our code generator produces a static method that accepts an integer argument and returns an integer result. Tiny basic blocks that operate on an integer are chosen randomly from a set of possibilities to form the nodes in the graph. The basic blocks are connected as directed by the graph, using conditional jumps and fall-through paths whenever possible. When used in combination with a graph encoder that mimics genuine program structures (such as our RPG encoder), the result is a synthetic function that is not obviously artificial.

If the graph has at least one leaf node (representing a return statement) then the generated function is guaranteed to reach it, so the function can safely be called. Furthermore, the generator can be instructed to guarantee a positive, negative, zero, or non-zero function result, allowing the function call to be used in an opaque predicate.

4.4. Clustering

GTW includes a clustering step before the edge addition step to increase the complexity of the graphs to which edges are added. If

edges are added directly to control-flow graphs, few original nodes will have more than two out-edges or a small number of in-edges, and high-degree nodes generated by edge adding will be conspicuous. The clustering step allows complex graphs to occur stealthily. *VVS* specifies a clustering step that proceeds by

> *Partition[ing] the graph G into n clusters using ω as a random seed, so that edges straddling across clusters are minimized (approximately).*

VVS also states that

> *The clustering step (2) must have a way to find different clusterings for different values of ω, so that the adversary does not have any knowledge about the clustering used.*

With Java bytecode, edges can be added only within methods or to entry points of other (accessible) methods. This constrains the usable clusterings. Fortunately, the natural clustering of basic blocks into Java methods is suitable for our needs. The proven difficulty of separating *W* from *P* does not rely on keyed clustering, so we have chosen in GTW_{SM} to simply treat each Java method as a cluster.

Each node in the cluster graph then represents an application or watermark method, and an edge between two nodes represents a method call. This clustering scheme is very likely to approximately minimize the number of edges between clusters, since (for typical programs) two basic blocks in the same method are much more likely to be connected than two basic blocks in different methods. This scheme also allows us to implement edge addition stealthily, efficiently, and easily. We were unable to identify any substantially different clustering scheme with both of these properties.

4.5. Adding control-flow edges

The *GTW* algorithm adds edges between clusters using a random walk, with non-uniform probabilities designed to merge the watermark code indistinguishably into the program. This process begins by choosing a random start node v_n, then repeatedly choosing another node v_l, creating an edge between v_n and v_l, and then repeating, this time with v_l as the start node. The process continues until *m* edges have been added between *P* and *W*. Or, more formally:

```
n ← rnd( );
while (fewer than m edges have been
       added between P and W) {
    l ←rnd( );
    add the edge v_n → v_l;
    n ← l;
}
```

To ensure that watermark code is not trivially detected as dead code, we then continue randomly adding edges until no watermark method has degree zero.

VVS does not address the issue of choosing *m*, the number of edges to be added between program methods and watermark methods. Our implementation chooses *m* so that the average degree of the watermark nodes is approximately the same as the average degree of the application nodes as follows.

At each step of the random walk, the next cluster visited is chosen from among all clusters, both program and watermark, other than the current cluster. Each cluster will be chosen with equal probability. Therefore, if *p* is the number of program clusters and *w* is the number of program clusters, then

$$q_p = \frac{p-1}{p+w-1},$$

is the probability that the next cluster chosen will be a program cluster given that the current cluster is from the program as well, and

$$q_w = \frac{w-1}{p+w-1},$$

is the probability that the next cluster will be from the watermark given that the current cluster is as well. Starting from a program cluster, the random walk will spend some number of steps exclusively in program clusters before traversing to a watermark cluster. Since the probability of the first watermark cluster coming after exactly *n* steps is $q_p^{n-1}(1-q_p)$, the expected number of steps required to reach a watermark cluster starting from a program cluster is

$$E_p = \sum_{n=1}^{\infty} n q_p^{n-1}(1-q_p) = \frac{1}{1-q_p}.$$

Similarly, the expected number of steps required to reach a program cluster starting from a watermark cluster is

$$E_w = \frac{1}{1-q_w}.$$

For each pair of edges between program and watermark clusters added by the random walk, we expect $1+E_p$ edges to be added pointing to program clusters and $1+E_w$ to be added pointing to watermark clusters. In our implementation, all watermark clusters have no edges either in or out at the beginning of the random walk. Therefore, the average number of edges we expect to be pointing to each watermark cluster at the end of the random walk is

$$\frac{m(1+E_w)}{2w},$$

and the expected average number pointing to each program cluster is

$$\frac{2e+m(1+E_p)}{2p}.$$

Setting these two quantities equal and solving for *m*, we obtain

$$m = \frac{4ew(1-q_w)(1-q_p)}{p(2-q_w)(1-q_p)-w(2-q_p)(1-q_w)}. \quad (1)$$

Because each method is a cluster, adding an edge from cluster *A* to cluster *B* means inserting code into method *A* that calls method *B*. The generated watermark methods are pure functions, so they can be executed without affecting program semantics. Therefore, the added method calls to watermark methods can actually be executed. However, application code may have arbitrary side effects, so the edge adding process must not change the number or order of executions of application methods. Therefore, added application method invocations are protected with opaquely false predicates to ensure that they are not actually executed. Additionally, application methods may be declared to throw checked exceptions. Preparing for and catching checked exceptions requires the addition to *A* of several blocks other than the method call block.

Also as a result of making each method a cluster, not every edge can be created. For example, private methods from different classes cannot call each other. In this case, the edge is simply not created and the process continues normally.

4.6. Marking basic blocks

Each basic block that corresponds to a node of the watermark must be individually marked for later recognition. The *VVS* paper does not provide an actual algorithm, but suggests that

> *one may store one or more bits at a node that flags when a node is in W by using some padded data after suitable keyed encryption and encoding.*

For marking purposes, the contents of a block can be changed as long as the modified code is functionally equivalent to the original. Here, are some examples of possible block markers:

1. Add code that accomplishes no function but just serves as a marker, for example by loading a value that is never used or writing to a value that has no effect on overall program behavior.
2. Count the number of instructions in a block, and use the parity as a mark. Add a no-op instruction, or make a more subtle change, to alter the mark.
3. Count accesses of static variables to determine a mark. Add variables and accesses as necessary to produce the desired results.
4. Compute a checksum of the instructions and use one or more bits of that as a mark. Alter the code as necessary to produce desired results.
5. Transform the instruction sequence in each block to a canonical form, then vary it systematically to encode marks.
6. Add marks in the meta-information associated with each block. For example, alter or create debugging information that associates code locations with source line numbers.

All of these marking methods are easily defeated if an adversary's goal is to disrupt the watermark without necessarily reading it. We are not aware of any robust block marking technique; this remains an unsolved problem.

For our implementation we have adopted the checksum technique, computing the MD5 digest [16] of each block. Only instruction bytes and immediate constant values, such as those in `bipush`, contribute to the digest value. This makes the digest insensitive to some simple changes such as reordering of the Java "constant pool".

A block is considered marked if the low-order two bits of the checksum are zero. We expect, then, to alter $\frac{3}{4}$ of the blocks in the watermark set but only $\frac{1}{4}$ of the other blocks to get the right results. A real application will have many more application blocks than watermark blocks, so this is a desirable imbalance.

Marking is keyed by concatenating a secret value to the instruction sequence before computing the MD5 digest. The set of marks cannot be read, nor can it be counterfeited, without knowing the key.

The process of marking a block *block* with key *key* can be summarized like this:

```
while (true) {
    b ← block||key;
    m ← MD5(b);
    l ← two low-order bits of m;
    if (block ∈ W ∧ l = 0) return block;
    if (block ∈ P ∧ l≠0) return block;
    make a small, random, semantics-preserving
    transformation of block;
}
```

5. Recognition

The recognition process in *VVS* has three steps: detection of watermark nodes, sampling of subsets of the watermark nodes, and computation of robust properties of these subsets. The set of robust property values composes the watermark.

5.1. Node detection

A basic block that is part of the watermark code can be detected by computing its MD5 digest, as described in Section 4.6. A digest value ending in two zero bits indicates a mark. Attacks on the watermarked program may change the digest value of some blocks, but our recognizer uses "majority logic" to recover from isolated errors. If 60% of the blocks in a method are marked, the recognizer treats all the blocks in that method as marked. If fewer than 40% of the blocks are marked, all are considered unmarked. If the number is between 40% and 60%, the recognizer tries both possibilities.

5.2. Subset sampling

GTW specifies that after the watermark nodes have been detected, several subsets of them should be sampled. GTW_{SM} uses method control-flow graphs as samples, and every watermark node is contained in exactly one sample set, in particular, the control-flow graph it belongs to.

5.3. Graph decoding

The recognition process attempts to decode each sampled method control-flow graph as a reducible permutation graph [15] that encodes an integer. A valid RPG can be decoded into a self-inverting permutation. The decoder proceeds by first computing the dominance hierarchy of the graph and, once the graph is verified to be reducible, finding the unique Hamiltonian path in the graph. This Hamiltonian path imposes an order on the vertices, after which decoding the graph into a self-inverting permutation is relatively straightforward, as laid out in [15].

Each graph's permutation is mapped back to an integer, using the same enumeration as in Section 4.2. The combined set of integers S is combined to produce single integer v, the watermark. The same splitting algorithm that formed the integers must be inverted to produce the original watermark value.

5.3.1. Partial sum recombination

For the partial sum splitter, the inverse function can be computed as follows:

1. Let $k = |S|$. Write S as $\{s_0, s_1, \ldots, s_{k-1}\}$, where $s_0 \leqslant s_1 \leqslant \cdots \leqslant s_{k-1}$.
2. Set $l = s_0 + 1$. For each $0 \leqslant j \leqslant k-2$, set $v_j = s_{j+1} - s_j$.
3. Then $v = \sum_{j=0}^{k-2} 2^{jl} v_j$.

5.3.2. GCRT recombination

The GCRT Splitting algorithm is more complicated, and requires more work to invert. It is composed of three steps, illustrated with an example in Fig. 9. The following description is taken from [1].

First, the bit-string $b_0 b_1, \ldots, b_n$ is split into a set of fixed-size blocks $B_0 = b_0, \ldots, b_{63}, B_1 = b_1, \ldots, b_{64}, \ldots$. These blocks are decrypted using the same cryptosystem as in the embedding process. Finally, the resulting 64-bit blocks $B_0^d = d_k(B_0), B_1^d = d_k(B_1) \ldots$ are passed to an algorithm that attempts to find a group of blocks that agree on the watermark.

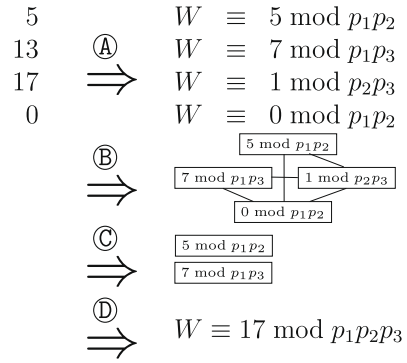

Fig. 9. Re-combining the watermark value $W = 17$, with $p_1 = 2, p_2 = 3, p_3 = 5$ (diagram taken from [1]).

In step Ⓐ of Fig. 9, we invert our enumeration scheme to turn these 64-bit blocks into statements about W of the form $W \equiv x_k \bmod p_{i_k}p_{j_k}$. Some of these blocks may have errors in them as a result of attacks (such as 18 in Fig. 7 being changed to 17 in Fig. 9), and there will be a very large number of blocks that have nothing to do with the watermark (such as, in the figure, the value 0). The remaining steps of the recombination algorithm attempt to determine which blocks to use for reconstructing the watermark.

Since the number of basic blocks can be very large, it is helpful to reduce the number of blocks to consider. To this end, we hold a vote on the value of $W \bmod p_i$ for each i. If there is a clear winner (which we define as the first-place vote-getter being strictly greater than twice second-place), this winner is presumed to be the value of $W \bmod p_i$, and all statements contradicting this are removed from consideration. This step has been empirically observed to greatly improve the average-case running time of the algorithm, while having a negligible effect on the probability of success.

Next, we consider pairs of statements of the form

$$W \equiv x_k \bmod p_{i_k}p_{j_k},$$
$$W \equiv x_m \bmod p_{i_m}p_{j_m}.$$

Such a pair is consistent if there exists an integer W satisfying both equations simultaneously and inconsistent otherwise. If $\gcd(p_{i_k}p_{j_k}, p_{i_m}p_{j_m}) = 1$, the pair is consistent by the Chinese remainder theorem. Otherwise the pair is consistent if and only if

$$x_k \equiv x_m \bmod \gcd(p_{i_k}p_{j_k}, p_{i_m}p_{j_m}).$$

The gist of the algorithm is that, if the p's are chosen so that $\gcd(p_{i_k}p_{j_k}, p_{i_m}p_{j_m})$ is large whenever it is not 1, it is unlikely for pairs of statements about W to be consistent in this second manner at random. Therefore, groups of statements that do agree in this second manner are likely to be ones that were inserted during the embedding of the watermark.

Let $V = \{v_0, v_1, \ldots, v_m\}$ be the set of statements on W we are given. In step Ⓑ, we construct two graphs, G and H, on V. (Fig. 9 only shows G). Two vertices are adjacent in G iff they are inconsistent. Two vertices are adjacent in H iff they are consistent and their moduli are not relatively prime. Any two vertices in V are adjacent in at most one out of G and H, but possibly neither. For step Ⓒ, we initialize $U := \emptyset$ and repeat the following steps until G has no edges:

1. Let v be some vertex in the set $V - U$ of maximum degree in H. This vertex is presumed to be a true statement about W.
2. Let S be the set of v's neighbors in G. Set $G := G - S$, $H := H - S$, and $U := U \cup \{v\}$.

Once G has no more edges, we have a set of statements about W that are consistent and can be combined by the generalized Chinese remainder theorem in step Ⓓ.

In order for this algorithm to succeed in reconstructing W, we need to know the value of $W \bmod p_i$ for each i. If each p_i is a node, we can think of each statement of the form $W \equiv x \bmod p_ip_j$ as an edge between p_i and p_j. Then the effect of attacks on the watermark can be modeled as edges being deleted at random. If q is the probability that each edge will be deleted and we start with the complete graph on n nodes, then the probability that each node will still have at least one incident edge is given by

$$\sum_{j=0}^{n}(-1)^j\binom{n}{j}q^{j\left(n-\frac{j+1}{2}\right)}. \tag{2}$$

This serves as a good approximation for the probability of W being successfully reconstructed. Fig. 12 shows the empirical probability of recovering a 768-bit value for W with a varying number of statements left intact versus the theoretical approximation of this probability as given by (2).

6. Evaluation

Most software watermarking research has focused on the discovery of novel embedding schemes. Little work has been done on their evaluation. A software watermarking algorithm can be evaluated using several criteria:

1. *Data rate:*
   What is the ratio of size of the watermark that can be embedded to the size of the program?
2. *Embedding overhead:*
   How much slower or larger is the watermarked application compared to the original?
3. *Resistance to detection (stealth):*
   Does the watermarked program have statistical properties that are different from typical programs? Can an adversary use these differences to locate and attack the watermark?
4. *Resilience against transformations:*
   Will the watermark survive semantics-preserving transformations such as code optimization and code obfuscation? If not, what is the overhead of obfuscating transformations? How much slower or larger is the application after enough transformations have been applied that the watermark can no longer be recognized?

6.1. Data rate and embedding overhead

A watermark of any size can be embedded in even the smallest of programs using this algorithm. Larger watermarks merely require larger watermark graphs, or a larger number of them, thus incurring larger overhead in terms of increased code size.

Using the partial sum splitter, there is little relationship between watermark size and code growth for non-trivial programs. This is illustrated in Fig. 10. Block marking and edge addition add code that is proportional to the size and complexity of the application, not the watermark. For watermarks up to 150 bits, size increases varying between 40% and 75% were measured. When using the GCRT Splitter the code size makes a jump approximately every 128 bits – when the splitter algorithm needs to add another watermark piece – as shown in Fig. 11.

CaffeineMark [17] benchmark results show the effect of watermarking on execution time. Some programs were not affected sig-

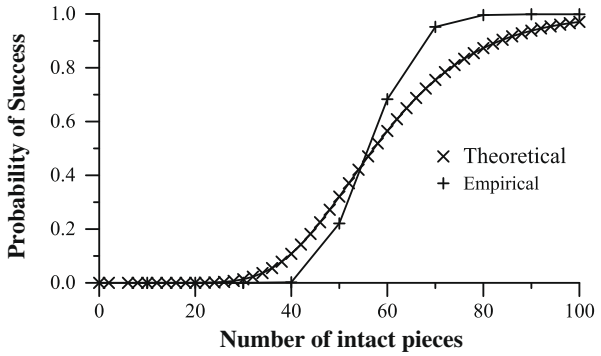

Fig. 10. Number of watermark pieces recovered intact versus the probability of successful watermark recovery.

nificantly, while others took up to 87% longer, as shown in Tables 1 and 2.

6.2. Stealth

Some common attacks against watermarking systems, such as manual attacks and subtractive attacks, begin by identifying the code composing the watermark. To resist such attacks, watermarking could should be stealthy: It should be indistinguishable from the host code. Two useful measures of stealth are the similarity of the watermark code to the host code and the similarity of the watermark code to general application code.

GTW_{SM} introduces several new artificially-generated methods to an application. These methods are not stealthy in two respects. First, these methods include a very high percentage of arithmetic operations. While general Java bytecode includes approximately 1% arithmetic instructions, the methods inserted by GTW_{SM} contain approximately 20% arithmetic instructions. Second, the control-flow graphs of the inserted methods are all reducible permutation graphs. While RPGs are designed to mimic the structure of real control-flow graphs, only 2 of 3236 methods in the SpecJVM benchmarking suite have control-flow graphs that are RPGs. Therefore, RPGs are not stealthy if an attacker is looking for them.

Table 1
CaffeineMark scores before and after embedding a watermark (partial sum splitter).

Category	Original	Watermarked	Slowdown (%)
Sieve	8676	6876	20.7
Loop	25636	16344	36.2
Logic	20635	13231	35.9
String	19481	20198	−3.6
Float	18657	18646	0
Method	19106	12783	33.1
Overall	17719	13816	22.0

Table 2
CaffeineMark scores before and after embedding a watermark (GCRT splitter).

Category	Original	Watermarked	Slowdown (%)
Sieve	16455	15489	5.9
Loop	55884	40171	28.1
Logic	46345	43274	6.6
String	28681	28473	0.7
Float	23857	3079	87.0
Method	20956	11709	44.1
Overall	29094	17388	40.2

GTW_{SM} currently introduces unstealthy code to implement edge addition between clusters. Edges between application methods are protected using the particularly conspicuous opaque predicate `if (null != null)`. Also, GTW_{SM} passes a constant for each argument to the called function; real code is more likely to compute at least one of its arguments.

As future work, we plan to improve the stealthiness of GTW_{SM} by using less conspicuous opaque predicates in the watermark code. We will also make the function call-site less conspicuous by passing opaque constants rather than transparent constants as the parameters. We also propose a method to increase the stealthiness of the watermark code itself. We will examine a large corpus of jar-files and determine a set of very commonly occurring basic

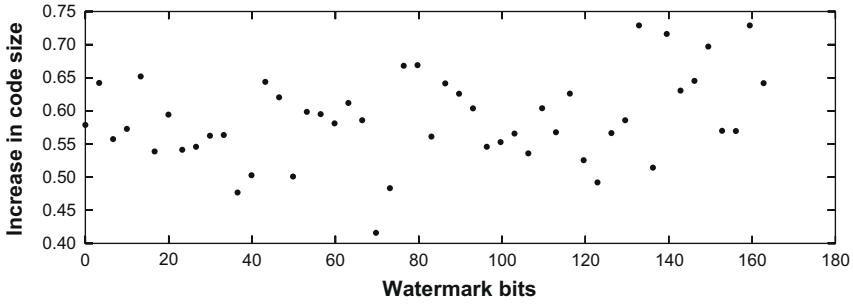

Fig. 11. Increase in code size for the machineSim program (partial sum splitter).

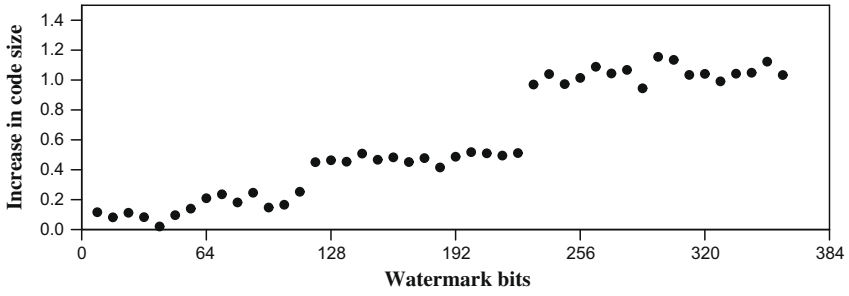

Fig. 12. Increase in code size for the machineSim program (GCRT splitter).

blocks. We will then adapt our watermark code so that it uses as many of these common basic blocks as possible. This will prevent a statistical analysis of the watermarked jar from revealing the location of the watermark, and increase the overall stealth of the embedding.

6.3. Semantics-preserving attacks

Automated attacks are the most serious threat to any watermark. Debray [18–20] has developed a family of tools that optimize and compress X86 and Alpha binaries. BLOAT [21] optimizes collections of Java class files. SANDMARK implements a collection of obfuscating code transformations that can be used to attack software watermarks.

We first tested the robustness of GTW_{SM} on a Java application *machineSim* which simulates a Von Neumann machine. Various SANDMARK obfuscations were applied to see if a watermark could survive. Using the partial sum splitter, the watermark was successfully recognized after inlining methods, register re-allocation, local variable merging (representing two 4-byte integers as a single 8-byte long integer), array splitting, class inheritance modification, local variable splitting (representing a single integer as two integers, whose XOR equals the original) and many others. It was destroyed by primitive boxing, basic block splitting, method merging, class file encryption, and code duplication. These types of transformations are described in [22,23,11]. For the GCRT Splitter, the watermark was successfully recognizable after applying all of the obfuscations implemented in SANDMARK except for class encryption and opaque predicate insertion.

Method merging makes such large changes to control-flow graphs that there is really no hope of recovering the watermark value. Primitive boxing changes the instructions in many basic blocks in a method, and thereby changes the marks on the blocks. Opaque predicate insertion is basically the addition of spurious code, but specifically it is the addition of if-statements. This will alter the control-flow graph of the method, thus complicating the recognition phase. Code duplication and basic block splitting add nodes to the control-flow graph of a method. While RPGs can survive some kinds of attacks on edges, they cannot survive node additions.

The attack model considered in *VVS* is a small number of random changes to the watermarked application. We have implemented an obfuscation that randomly modifies a parameterized fraction of blocks in a program. If fewer than about half of the blocks in a watermarked application are modified, the watermark survives. If more than that are modified, the watermark cannot be recovered.

6.4. False positive rates

For our implementation to detect a spurious watermark in an unmarked application, the application would have to have at least two methods with acceptable control-flow graphs in which the majority of basic blocks would produce MD5 digests with two low-order zero bits. The probability of finding a mark in a single basic block is only $\frac{1}{4}$. We examined a large group of methods from real programs and found the probability of a control-flow graph being a valid RPG to be 0.002. While there is a possibility of finding an RPG with only two or three nodes where all the nodes are marked in a real program, choosing watermark values from a sufficiently sparse set should be enough to prevent false positives.

7. Discussion and future work

Because our recognizer returns a specific watermark value, as opposed to just a success/failure flag, GTW_{SM} can be used for fingerprinting. This is a technique where each copy of an application program is distributed with its own unique watermark value, allowing pirated copies to be traced back to a specific original.

Our implementation of the *GTW* watermarking system is fully functional and reasonably efficient. It is resilient against a small number of random program modifications, in accordance with the threat model assumed by *VVS*.

The system is more vulnerable to pervasive changes, including several obfuscations implemented in the SANDMARK system. Such vulnerabilities stem from issues left unaddressed by the *VVS* paper. These and other areas provide opportunities for future work.

Static marking of basic blocks is the fundamental mutation applied by the watermarker. Development of a robust marking method, capable of withstanding simple program transformations, is still an unsolved problem.

Another area of great potential is the encoding of values as graph structures. In particular, the development of other error-correcting graphs, as postulated by *VVS*, would greatly increase the strength of a watermark.

More sophisticated generated code and opaque predicates would improve the stealthiness of a watermark. One such technique would be to examine a large corpus of basic blocks occurring in real code and adapt the watermark code to use these common blocks, whenever possible.

Implementations of *GTW* for other architectures besides Java would undoubtedly prove enlightening, because they would be likely to supply somewhat different challenges and opportunities.

One key feature of *GTW* is the algorithm for connecting new code representing a watermark into an existing application. This algorithm also adds branches within the pre-existing code and is interesting in its own right as a means of obfuscation. This also has potential for further research.

8. Summary

We have produced a working implementation of the graph theoretic watermark described by Venkatesan et al. [3]. The implementation is faithful to the paper within the constraints of Java bytecode, and includes necessary components that were left unspecified by the original paper. While the *GTW* design protects against detection, its fundamental dependence on static block marking leaves watermarked programs vulnerable to distortive attacks.

References

[1] C. Collberg, E. Carter, S. Debray, A. Huntwork, J. Kececioglu, C. Linn, M. Stepp, Dynamic path-based software watermarking, in: Programming Language Design and Implementation, PLDI'04, 2004.
[2] C. Collberg, A. Huntwork, E. Carter, G. Townsend, Graph theoretic software watermarks: implementation, analysis, and attacks, in: Workshop on Information Hiding, 2004, pp. 192–207.
[3] R. Venkatesan, V. Vazirani, S. Sinha, A graph theoretic approach to software watermarking, in: Fourth International Information Hiding Workshop, Pittsburgh, PA, 2001.
[4] A.V. Aho, R. Sethi, J.D. Ullman, Compilers, Principles, Techniques, and Tools, Addison-Wesley, 1986, ISBN 0-201-10088-6.
[5] C. Collberg, G. Myles, A. Huntwork, Sandmark – a tool for software protection research, IEEE Security and Privacy 1 (4) (2003) 40–49. <http://dx.doi.org/>, <http://dx.doi.org/10.1109/MSECP.2003.1219058>. doi:<http://dx.doi.org/10.1109/MSECP.2003.1219058>.
[6] R.L. Davidson, N. Myhrvold, Method and system for generating and auditing a signature for a computer program, US Patent 5,559,884, assignee: Microsoft Corporation, September 1996.
[7] G. Qu, M. Potkonjak, Analysis of watermarking techniques for graph coloring problem, in: IEEE/ACM International Conference on Computer Aided Design, 1998, pp. 190–193.
[8] G. Myles, C. Collberg, Software watermarking through register allocation: implementation, analysis, and attacks, in: International Conference on Information Security and Cryptology, 2003.
[9] J.P. Stern, G. Hachez, F. Koeune, J.-J. Quisquater, Robust object watermarking: application to code, in: Information Hiding, 1999, pp. 368–378.

[10] G. Arboit, A method for watermarking Java programs via opaque predicates, in: The Fifth International Conference on Electronic Commerce Research (ICECR-5), 2002.
[11] C. Collberg, C. Thomborson, D. Low, Manufacturing cheap, resilient, and stealthy opaque constructs, in: Principles of Programming Languages 1998, POPL'98, San Diego, CA, 1998.
[12] C. Collberg, C. Thomborson, Software watermarking: models and dynamic embeddings, in: Conference Record of POPL'99: The 26th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (January 1999), 1999.
[13] M. Dahm, Byte code engineering, in: The Scientific German Java Conference, 1999.
[14] D.E. Knuth, Seminumerical Algorithms, of the Art of Computer Programming, third ed., vol. 2, Addison-Wesley, Reading, MA, USA, 1997.
[15] C. Collberg, E. Carter, S. Kobourov, C. Thomborson, Error-correcting graphs, in: Workshop on Graphs in Computer Science (WG'2003), 2003.
[16] R. Rivest, The MD5 message-digest algorithm, the internet engineering task force RFC 1321, 1992. <http://www.ietf.org/rfc/rfc1321.txt>.
[17] P. Software, Caffeinemark 3.0, 1998. <http://www.p.software.com/pendragon/cm3/>.
[18] S. Debray, B. Schwarz, G. Andrews, M. Legendre, PLTO: a link-time optimizer for the Intel IA-32 architecture, in: Proceedings of the 2001 Workshop on Binary Rewriting (WBT-2001), 2001.
[19] S. Debray, R. Muth, S. Watterson, K.D. Bosschere, ALTO: a link-time optimizer for the compaq alpha, Software – Practice and Experience 31 (2001) 67–101.
[20] S. Debray, W. Evans, R. Muth, B.D. Sutter, Compiler techniques for code compaction, ACM Transactions on Programming Languages and Systems 22 (2) (2000) 378–415.
[21] N. Nystrom, Bloat – the bytecode-level optimizer and analysis tool, 1999. <http://www.cs.purdue.edu/homes/whitlock/bloat>.
[22] C. Collberg, C. Thomborson, D. Low, A taxonomy of obfuscating transformations, Tech. Rep. 148, Department of Computer Science, University of Auckland (Jul. 1997).
[23] C. Collberg, C. Thomborson, D. Low, Breaking abstractions and unstructuring data structures, in: IEEE International Conference on Computer Languages, ICCL'98, Chicago, IL, 1998.