A Survey Of Graph Based Software Watermarking

James Hamilton and Sebastian Danicic
Department of Computing
Goldsmiths, University of London
United Kingdom
james.hamilton@gold.ac.uk, s.danicic@gold.ac.uk

***Abstract*—Software watermarking involves embedding a unique identifier within a piece of software, to discourage software theft. The global revenue loss due to software piracy was estimated to be more than $50 billion in 2008. We survey the proposed software watermarking algorithms based on graph encoding.**

Graph based watermarking schemes, like other watermarking schemes, can be divided into two groups: static and dynamic. Static graph watermarks are embedding in a control-flow graph within a program whereas dynamic graph watermarks are embedding in a graph data-structure built at run-time.

We describe the proposed static and dynamic graph watermarking schemes, highlighting strengths and weaknesses and concluding with a proposal for future work with dynamic graph watermarking.

***Index Terms*—software watermarking; program transformation;java; bytecode;**

I. Introduction

Software theft, also known as software piracy, is the act of copying a legitimate application and illegally distributing that software, either free or for profit. Legal methods to protect software producers such as copyright laws, patents and license agreements [17] do not always dissuade people from stealing software, especially in emerging markets where the price of software is high and incomes are low. Ethical arguments, such as fair compensation for producers, by software manufacturers, law enforcement agencies and industry lobbyists also do little to counter software piracy. The global revenue loss due to software piracy was estimated to be more than $50 billion in 2008 [3].

Technical measures have been introduced to protect digital media and software, due to the ease of copying computer files. Some software protection techniques, of varying degrees of success, can be used to protect intellectual property contained within Java class-files. Java bytecode is higher level than machine code and is relatively easy to decompile with only a few problems to overcome [19].

Software watermarking involves embedding a unique identifier within a piece of software. It does not prevent theft but instead discourages software thieves by providing a means to identify the owner of a piece of software and/or the origin of the stolen software [33]. The hidden watermark can be used, at a later date, to prove ownership of stolen software. It is also possible to embed a unique customer identifier in each copy of the software distributed which allows the software company to identify the individual that copied the software.

In this paper, we examine the currently proposed static and dynamic graph watermarking schemes. Static graph watermarks are embedding in a control-flow graph within a program whereas dynamic graph watermarks are embedding in a graph data-structure built at run-time. We report previous findings, describe some recent additions and conclude by suggesting a direction for future work.

II. Software Watermarking

Software watermarks can be broadly divided into two categories: static and dynamic [10]. The former is embedded in the data and/or code of the program, while the latter is embedded in a data structure built at runtime.

A watermark is embedded into a computer program through the use of an *embedder*; it can then be extracted by an *extractor* or verified by a *recogniser*. The former extracts the original watermark, while the latter merely confirms the presence of a watermark [51]. A watermark recognition or extraction algorithm may also be classified as *blind*, where the original program and watermark is unavailable, or *informed*, where the original program and/or watermark is available [50].

Watermarks should be resilient to semantics preserving transformations and ideally it should be possible to recognise a watermark from a partial program. Semantics preserving transformations, by definition, result in programs which are syntactically different from the original, but whose behaviour is the same. The attacker can attempt, by performing such transformations, to produce a semantically equivalent program with the watermark removed [20]. Redundancy and recognition with a probability threshold may help with these problems [32].

The runtime cost of a program with an embedded watermark should not differ significantly from the original program but some transformations applied by the watermark could have an effect on size and execution time [35].

III. Encoding Watermarks in Graphs

Collberg et al. [13] describe several techniques for encoding watermark integers in graph structures. The algorithms in this paper rely on the fact that graph-generating code is difficult to analyse due to aliasing effects [18] which, in general, is known to be un-decidable [40].

An ideal class of watermarking graph should have the following properties [16]:

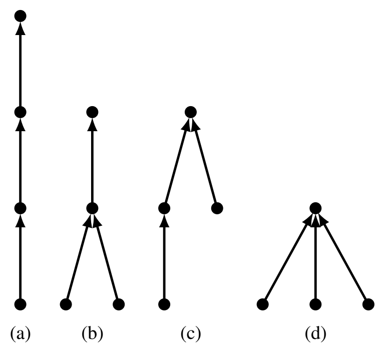
Fig. 1: Enumerations of a directed graph with 4 indistinguishable vertices.

- ability to efficiently encode a watermark integer; and be efficiently decodable to a watermark integer
- a root node from which all other nodes are reachable
- a high data-rate
- a low outdegree to resemble common data structures such as lists and trees
- error correcting properties to allow detection after transformation attacks
- tamper-proofing abilities
- have some computationally feasible algorithms for graph isomorphism, for use during recognition

A. *Graph Enumeration*

The family of graph encoding is based on a branch of graph theory known as graphical enumerations [21]. An integer watermark n is encoded as the n^{th} enumeration of a given graph.

1) Directed Parent-Pointer Trees: This family of graphs contains a single edge between a vertex and it's parent. For example, there are 4 possible enumerations of DPPTs having four indistinguishable vertices [28], as shown in figure 1. We choose the n^{th} enumerated graph to embed the watermark number n.

how is it ordered?

Implementing a parent-pointer tree data-structure is space efficient because each node has just one pointer field referencing it's parent. However, the data-structure is fragile and an adversary could add a single node or edge to distort the watermark.

2) Planted Planar Cubic Trees: Planted planar cubic trees (PPCT) are binary trees where every interior vertex, except the root, has two children (see figure 2). These are easily enumerated by using a Catalan recurrence [4, 29]. Figure 4 shows PPCT enumerations with 1 to 4 leaves; the Catalan number $c(n)$ gives the number of unique trees for each n. An integer is encoded as one of the trees, for example, a 0 could be encoded as any of the trees in the first column.

The Catalan number is given by the formula:

$$c(n) = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!n!} \text{ for } n \geq 0 \qquad (1)$$

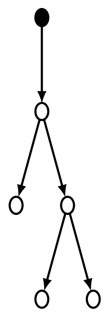
Fig. 2: Planted Planar Cubic Tree

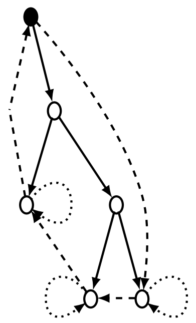
Fig. 3: Enhanced Planted Planar Cubic Tree, with leaf self-pointers (dotted) and an outer cycle (dashed).

PPCTs can be made more resilient to attacks by a) marking each leaf with a self-pointer, and b) creating an outer cycle from the root to itself through all the leaves [9] (see figure 3). This allows single edge and node insertions to be detected; however, multiple changes cannot be detected or corrected. The PPCT graphs have a lower bit-rate than other graph families but are more resilient to attacks.

B. *Radix Graphs*

Radix graphs add an extra pointer field in each vertex of a circular linked list of length k to encode a base-k digit. It is possible to encode watermark digits where a self-pointer represents 0, a pointer to the next node 1, and so on. Figure 5 shows the radix-5 expansion of 365_{10} (24305_5) encoded in a linked list structure.

These graphs give the highest data-rate for encoding watermarks however they are fragile as the watermark could easily be distorted. We can add redundancy to these watermarks by restricting the indegree to two and outdegree to two and using a permutation graph [13].

Several papers [5, 39, 48, 47, 26, 49] describe a technique which combines radix graphs with the error-correcting properties of PPCTs by converting a radix graph into a PPCT-like structure.

C. *Permutation Graphs*

Permutation based graphs (as defined by Collberg et al. [13]) use the same basic singly linked circular list structure as the radix graphs but have error-correcting properties. In this encoding scheme a permutation $P = \{p_1, p_2, \ldots, p_n\}$ is

different from wikipedia & Spinrad [41]

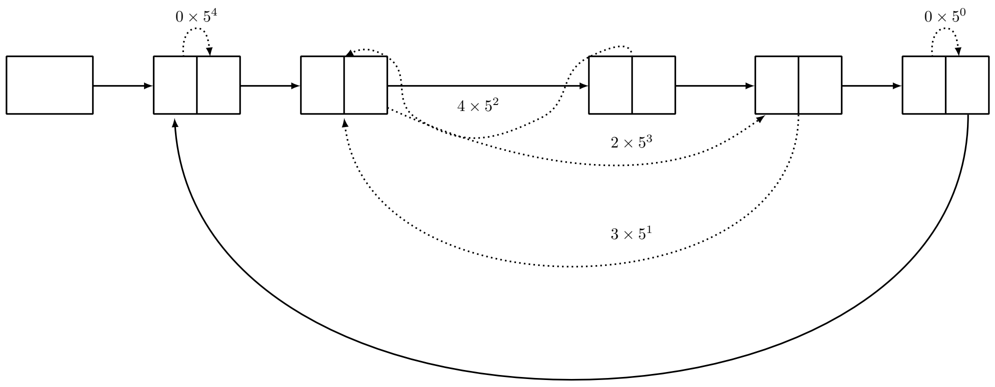

Fig. 5: Radix-5 expansion of the watermark 365. The spine is black and edges representing radix-k are dotted.

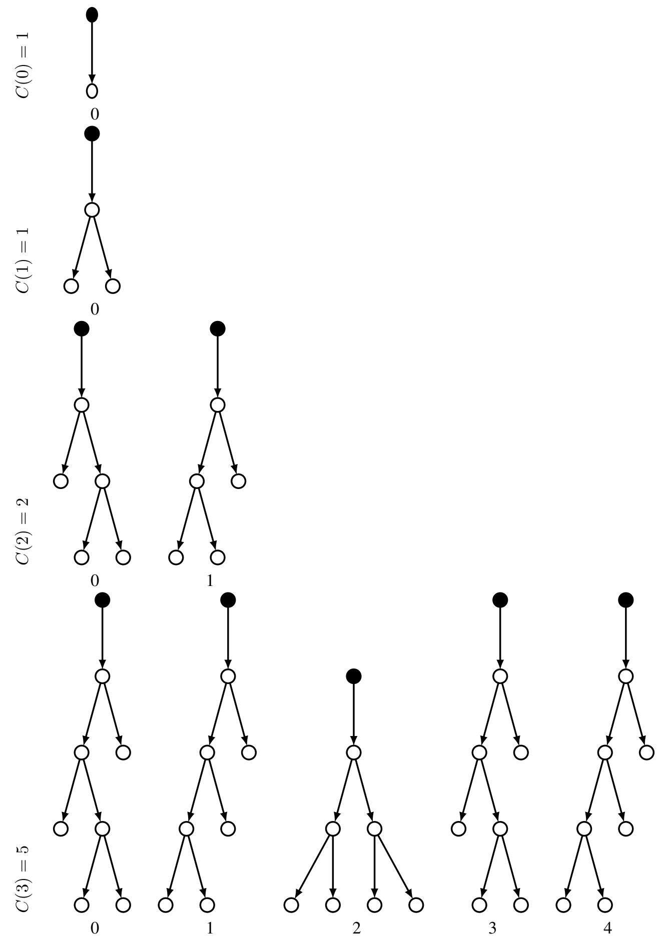

Fig. 4: Enumerations of PPCTs with 1 to 4 leaves, showing the index below.

derived from the watermark integer n; the permutation is then encoded in the graph by adding edges between vertices i and p_i.

Collberg and Nagra [9] give algorithms for encoding and decoding an integer as a permutation, shown here as algorithm 1 and 2. For example, the integer 97_{10} is encoded as the permutation $\{3, 1, 0, 2, 4\}$ according to algorithm 1. We then encode this in a graph as shown in figure 6.

If an adversary changes one of the non-spine pointers to point to a different vertex we will discover an error because the permutation graph encodes a unique permutation - each vertex must have indegree two. However, if the adversary swaps two pointers we will lose the watermark.

Algorithm 1 int2perm [9]

```
Input: integer $V$, length of permutation $len$
Output: permutation encoding of $V$
$perm = (0, 1, 2, \ldots, len - 1)$
for $r = 2$; $r < len$; $r++$ do
    swap perm[$r - 1$] and perm[$V$ mod $r$]
    $V = V \div r$
end for
return perm
```

D. Reducible Permutation Graphs

Reducible permutation graphs (RPG) [45, 44] are very similar to permutation graphs but they closely resemble control-flow graphs as they are reducible-flow graphs [24, 23].

Definition 1. *Reducible flow graph [1]: A flow graph G is reducible if and only if we can partition the edges into two disjoint groups, often called* forward *edges and* back *edges, with the following two properties:*

1) *The forward edges from an acyclic graph in which every node can be reached from the intial node of G.*
2) *The back edges consist only of edges whose heads dominate their tails*

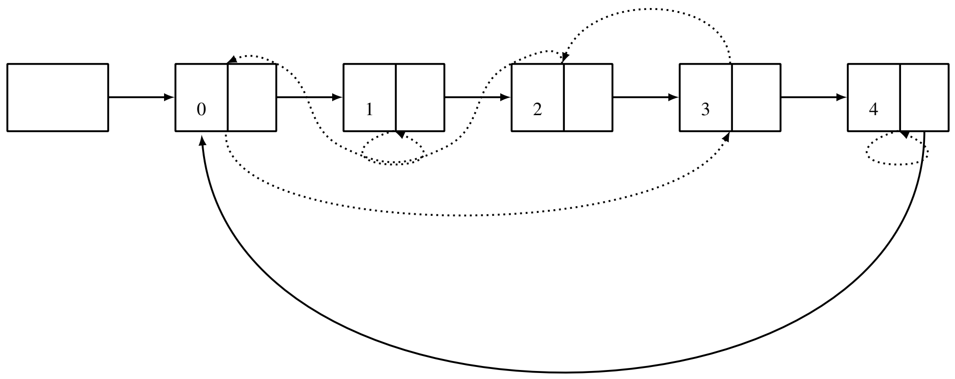

Fig. 6: Permutation graph encoding the integer 97_{10} using algorithm 1 to generate the permutation $\{3, 1, 0, 2, 4\}$.

Algorithm 2 perm2int [9]

```
Input: a permutation $perm$
Output: encoded integer $V$
$V = 0$
$f = 0$
for $r = length(perm)$; $r \geq 2$; $r - -$ do
  for $s = 0$; $s < r$; $s + +$ do
    if $perm[s] == r - 1$ then
      $f = s$
      break
    end if
  end for
  swap perm[$r - 1$] and perm[$f$]
  $V = f + r \times V$
end for
return $V$
```

The reducibility of this family of graphs means that they resemble control-flow graphs constructed from programming constructs such as *if*, *while* etc. [9].

RPGs, like CFGs, contain a unique entry node and a unique exit node, a preamble which contains zero or more nodes from which all other nodes can be reached and a body which encodes a watermarking using a self-inverting permutation [6].

Definition 2. *Inverse permutation [6]: an inverse permutation of* $\{p_0, p_1, \ldots, p_n\}$ *is the permutation* $\{q_0, q_1, \ldots, q_n\}$ *where* $q_{p_i} = p_{q_i} = i$.

hmm...

Definition 3. *Self-inverting permutation [6]: a permutation that is it's own inverse, i.e.* $P_{p_i} = i$.

For example, the following permutation $Q = \{0, 5, 2, 10, 4, 1, 9, 7, 8, 6, 3\}$ is a self-inverting permutation of $P = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$.

This family of graphs is resistant to edge-flip attacks, where an attacker inverts the condition of conditional jumps in a program.

draw graph example

IV. Static Graph Watermarking

Venkatesan et al. [45] proposed the first static graph watermarking scheme, Graph Theoretic Watermarking (GTW), which encodes a value in the topology of a program's control-flow graph [1]. The idea was later patented by Venkatesan and Vazirani [44] for Microsoft. The basic concept is to encode a watermark value in a reducible permutation graph and convert it into a control flow graph; it is then merged with the program control flow graph by adding control flow edges between the two.

Figure 7(a) shows the control flow graph for the Java method in listing 1 and figure 7(b) shows our watermark graph - this graph encodes our watermark (it's not important, for this example, what the watermark is; just that 7(b) encodes our watermark).

The watermark graph in figure 7(b) is also the control flow graph for the Java method in listing 2 - we embed our watermark graph in a program by converting a graph watermark into a control flow graph using programming constructs from the programming language we are using.

The watermark control flow graph can be integrated with the original program control flow graph as shown in figure 7(c) and code listing 3.

The algorithm adds bogus control flow edges between random pairs of vertices in the program CFG and watermark CFG in order to protect against static analysis attacks looking for sparse-cuts [2] in the control-flow graph. A sparse-cut would indicate a possible joining point of the original program CFG and the watermark CFG where the attacker could split the program with as few edges broken as possible. For example, we could insert a call to the `sum`, or any other method in the original program, in the watermark method. The GTW algorithm also inserts bogus method calls in between parts of the original program so that the location of bogus calls cannot be used to point out the location of the watermark.

In order to recognise a GTW we must know which basic blocks, from the program's CFG, are part of the watermark graph; we must therefore mark the watermark basic blocks in

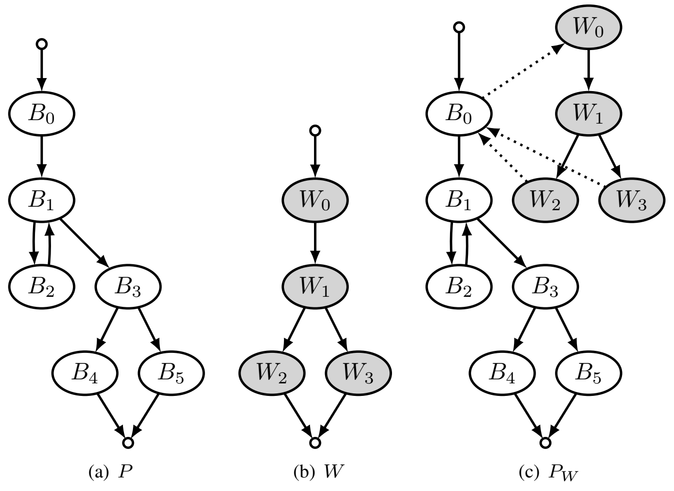

Fig. 7: Graph theoretic watermarking

Listing 1: Example Java Method - Sum

```
public static void sum(int[] numbers) {
  int total = 0;

  for(int i = 0; i < numbers.length; i++) {
    total += numbers[i];
  }

  if(total < 0) {
    System.out.println("total is less than zero
        :(");
  }else{
    System.out.println("total is " + total);
  }
}
```

Listing 2: Example Static Graph Watermark Method - Sum

```
public static boolean wm(int i) {
  if(i < 0)
    return true;
  else
    return false;
}
```

some way in order to identify them. We could, for example, re-order instructions such that they are in lexicographic order, or insert bogus instructions to identify a watermark basic block [9]. However, these techniques are not resilient to semantics-preserving transformations and an attacker could remove the marks so that we cannot mark our watermark blocks.

Collberg et al. [7] implemented a version GTW_{SM} of GTW in Sandmark [8] in order to evaluate the algorithm. They measured the size and time overhead of watermarking and evaluated the algorithm against a variety of attacks. They also introduce two methods (Partial Sum splitting and Generalised Chinese Remainder Theorem [30] splitting) for splitting a watermark integer into redundant pieces so that a large integer can be stored in several smaller CFGs. They found that stealth is a big problem; for example, the basic blocks of the generated watermark method consisted of 20% arithmetic instructions compared to just 1% for standard Java methods [14]. Watermarks of up to 150 bits increased program size by between 40% and 75%, while performance decreased by between 0% and 36% [7].

Listing 3: Example Static Graph Watermark Java Method - Sum

```
public static void sum(int[] numbers) {
  int total = 0;
  wm(total);
  for(int i = 0; i < numbers.length; i++) {
    total += numbers[i];
  }

  if(total < 0) {
    System.out.println("total is less than zero
        :(");
  }else{
    System.out.println("total is " + total);
  }
}

public static boolean wm(int i) {
  if(i < 0)
    return true;
  else
    return false;
}
```

V. Dynamic Graph Watermarking

Collberg and Thomborson [10] proposed the first dynamic graph based watermarking scheme CT to overcome problems with static watermarking schemes; most notably, static watermarks are highly fragile and therefore susceptible to semantics-preserving transformation attacks [20].

Listing 4: Example Dynamic Graph Watermarked Java Method - Sum

```
class Node { public List<Node> children = new
    ArrayList<Node>(); }

public static void sum(int[] numbers) {
  int total = 0;

  if(numbers.length > 5) {
    Node root = new Node();
    Node n1 = new Node();
    root.children.add(n1);
    Node n2 = new Node();
    Node n3 = new Node();
    n1.children.add(n2);
    n1.children.add(n3);
  }

  for(int i = 0; i < numbers.length; i++) {
    total += numbers[i];
  }

  if(total < 0) {
    System.out.println("total is less than zero
        :(");
  }else{
    System.out.println("total is " + total);
  }
}
```

Dynamic graph watermarking schemes are similar to static graph watermarking except the graph structures are built at run-time. CT can use any of the previously described graph encoding schemes to store the watermark.

Figure 4 demonstrates how we could, trivially, embed our example watermark graph from figure 7(b) dynamically into the example Java program from listing 1. We only execute the watermark code when the `numbers` array is longer than 5 - this serves as our secret input. We would inspect the Java heap to retrieve our watermark when the program is executed with the secret input.

The first implementation [38] of the CT algorithm CT_{JW}, implemented in a system called JavaWizz [37], was for Java bytecode using PPCT graphs to encode the watermark integer. A class is chosen from the program to be watermarked and converted into a *node class* by adding additional fields which contain references to other nodes.

The graph building code is including in the program by sub-classing a class and putting the graph building code in it's constructor. For example, in listing 5 the class $A1$ is a sub-class of A; to build the graph an execution of $new\ A()$ only has to be replaced by $new\ A1()$.

Palsberg et al. [38] discuss a simple implementation which only resists attacks against dead-code removal; they suggest that other software protection techniques are applied after the program has been watermarked. Dependencies are added between the watermark generating code and the original program by replacing a statement S with a statement of the form $if(x \neq y)\ S$ where x and y are distinct nodes in the watermark graph.

Listing 5: CT watermark implementation.

```
class A {

  A() {
    ....
  }
  ....
}

class A1 extends A {

  A1() {
    super();
    // graph building
    // code goes here
  }

}
```

In order to retrieve the watermark, the watermarked program is executed and the Java heap is accessed by dumping an image using the `-Xhprof` heap profiling JVM option [36].

The following steps are performed to retrieve the watermark:

1) Extracting potential node classes
2) Exracting potential node objects
3) Determining potential edges
4) Searching for the watermark graph

In general, searching for the graph would be an NP-complete problem however we know that the graph is a PPCT graph of a certain size and can prune away unnecessary nodes. Palsberg et al. [38] evaluated their implementation and found that, although it is possibly to retrieve a watermark, it is time-consuming — taking from 0.6 minutes up to 8.9 minutes on their set of test programs. It was shown that the CT algorithm is a good watermarking scheme because it is stealthy and resilient to semantics-preserving transformation attacks but it must be combined with other software protection techniques such as obfuscation [11] and tamper-proofing [15].

Collberg and Thomborson [16] implemented a version of the CT algorithm, CT_{SM}, in Sandmark [8] and compared it to JavaWiz [37]. The CT_{SM} implementation differs from the JavaWiz in the following ways:

- CT_{SM} requires a key to encode and decode the watermark, requiring user annotation of the program to be watermarked
- CT_{SM} can encode arbitrary strings whereas CT_{JW} requires an integer watermark
- CT_{JW} uses PPCTs to encode the watermark, whereas CT_{SM} offers the choice of Radix, PPCT, Permutation or Reducible Permutation graphs; also offering a cycled graph option to protect against node-splitting attacks.
- CT_{JW}'s graph building code is concentrated in one location whereas CT_{SM} embeds code fragments at sereval user-specified locations
- CT_{JW} uses an existing class for graph nodes whereas CT_{SW} generates its own

- CT_{JW} uses the heap profiling option of the JVM whereas CT_{SM} uses the Java Debug Inteferace (JDI) API [42]

The CT_{SW} algorithm embedding algorithm consists of the following steps:

Annotation
: In this first step the programmer must insert calls to a method `mark()` which indicates to the watermarking system locations in which graph building code can be inserted. This manual annotation step allows a programmer to carefully choose the best locations for code insertion, maximising the stealthiness of the embedded watermark.

Tracing
: After the code has been annotated the program is run with a secret input during which one or more annotation points will be encountered. Some of these encountered locations are used to store the graph building code.

Embedding
: The watermark is encoded in a graph structure (strings watermarks are first converted to integers) and is converted into Java bytecode that builds the graph. The graph node class is generated from a similar class already in the program, or if no suitable class is found, classes from the Java library, such as `LinkedList` could be used. The graph is partitioned into several, equal sized, sub-graphs so that the code is more stealthy and the code is inserted at the marked locations discovered in the tracing step.

During extraction the program is run with the secret input; this will invoke the graph building code at the locations marked by the programmer. The graph structure will be on the Java heap and the Java debugging interface is used to retrieve it. The JDI is used to add breakpoints to every constructor in the program to build a circular linked buffer of the last 1000 objects allocated. The list is then searched, in reverse, for the watermark graph.

Collberg and Thomborson [16] evaluate CT_{SM} and conclude that it is efficient and resilient to semantics-preserving transformations. The greatest vulnerability is it's limited stealthiness and it is suggest that future work investigates combining a locally stealthy but not steganographically stealthy technique, such as describe by Nagra and Thomborson [34], with the CT algorithm would be helpful.

VI. Attacks against graph watermarks

Collberg et al. [16] describe 4 types of attacks against dynamic graph watermarking schemes:

1) adding extra graph edges
2) adding extra nodes to the graph
3) renaming and re-ordering instance variables
4) splitting nodes into several linked parts

However, an adversary will probably not know the location of the watermark in a large program and as such would have to apply transformations across the program resulting in a large performance decrease. The use of RPG or PPCT graphs prevent renaming and re-ordering attacks because these graph encodings do not rely on the order of the nodes.

The biggest problem is the addition of bogus nodes to a graph for which tamper-proofing techniques are required. Collberg et al. [16] suggest the use of Java reflection may help. However, they did not implement this because they conclude that the code would be unstealthy as this kind of code is unusual in most programs.

Luo et al. [31] describe an algorithm based on CT using the Chinese Remainder Theorem [30] to split the watermark .

Is this just plagiarised from collberg?

VII. Tamper-proofing by Constant Encoding

He [22], and later Thomborson et al. [43], developed a tamper-proofing technique for dynamic graph watermark called constant encoding. Constant encoding encodes some of the constants in a program into a tree structure similar to the watermark and decodes them at run-time. An attacker cannot distinguish between the watermark graph and the constant encoding graphs so they cannot change any graph structure as they may introduce bugs into the program.

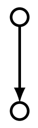
Fig. 8: A possible graph encoding of the constant 0.

For example, assuming that the constant 0 can be encoded as the graph in figure 8, we can replace the constant 0 in the example Java program from listing 1 with a method call which decodes the graph back into the constant 0, as shown in listing 6.

Listing 6: Example Constant Encoding Java Method - Sum

```
class Node { public List<Node> children = new
    ArrayList<Node>(); }

public static void sum(int[] numbers) {

  Node cg = buildConstantGraph();

  int total = decode(cg);

  for(int i = 0; i < numbers.length; i++) {
    total += numbers[i];
  }

  if(total < 0) {
    System.out.println("total is less than zero
        :(");
  }else{
    System.out.println("total is " + total);
  }
}
```

An attacker that manipulates the constant graph will change the encoding; so the decode function will return a different number and the program will be semantically incorrect.

A weakness of this system is that an attacker may be able to discover that certain program functions always return the same value, for every execution. He [22] suggests introducing dependencies, to make the code harder to analyse, by providing an input variable to the decoding function. A further suggestion, for future work, is to change the constant tree at runtime making the code harder to analyse.

Jian-qi et al. [25] propose using the constant encoding functions within the parameters of opaque predicates Collberg et al. [12] so that if an attacker manipulates a constant graph control will pass to the wrong clause of an *if* statement using the opaque predicate.

Thomborson et al. [43] combine the watermarking and tamper-proofing techniques further by using sub-graphs of the watermark graph to encode constants, rather than separate graphs. Thomborson et al. [43] also formally define the problem of tamper-proofing and show that commonly occurring constants in computer programs can be replaced automatically and that a long series of dynamic analyses would be required to remove the watermark. However, it is not clear if their constant encoding technique is resilient against pattern-matching attacks – a question which is left for future work.

Khiyal et al. [27] suggest splitting constants so that large constants can be encoded in small trees. They evaluated the constant encoding technique by comparing watermarked and tamper-proofed programs for efficiency, size and resilience. They found that the size of a program does increase when tamper-proofed and the execution time is slower; however, the tamper-proofing technique should be used with obfuscation to further protect a tamper-proofed program.

VIII. Conclusion

We have presented a survey of software watermarking schemes based on graph encoding using both static and dynamic embedding techniques. Static techniques, as always, are highly susceptible to semantics preserving transformation attacks and are therefore easily removed by an adversary.

Dynamic graph watermarking, however, is resilient to most semantics-preserving transformations if the right graph encoding is chosen. However, a choice between variable degrees of high stealth, resilience and bit-rate has to be made, as no algorithm has been developed which combines the best of all these properties.

Further research should continue into dnyamic graph watermarking, especially on improving the CT algorithm from attacks. We intend to investigate the use of program slicing [46] to attack dynamic watermarks and improve resilience to subtractive attacks.

References

[1] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition, August 2006. ISBN 0321486811.

[2] Sanjeev Arora, Satish Rao, and Umesh Vazirani. Expander flows, geometric embeddings and graph partitioning. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 222–231, Chicago, IL, USA, 2004. ACM. ISBN 1-58113-852-0.

[3] Business Software Alliance. Sixth annual BSA and IDC global software piracy study. Technical Report 6, Business Software Alliance, 2008.

[4] Eugène Catalan. Note extraite d'une lettre adressée à l'éditeur. *Journal för die reine und angewandte Mathematik*, 27:192, 1844.

[5] XiaoJiang Chen, DingYi Fang, JingBo Shen, Feng Chen, WenBo Wang, and Lu He. A dynamic graph watermark scheme of tamper resistance. In *Proceedings of the 2009 Fifth International Conference on Information Assurance and Security - Volume 01*, pages 3–6. IEEE Computer Society, 2009. ISBN 978-0-7695-3744-3.

[6] Maria Chroni and Stavros D. Nikolopoulos. Encoding watermark integers as self-inverting permutations. In *Proceedings of the 11th International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing on International Conference on Computer Systems and Technologies*, pages 125–130, Sofia, Bulgaria, 2010. ACM. ISBN 978-1-4503-0243-2.

[7] C. Collberg, A. Huntwork, E. Carter, and G. Townsend. Graph theoretic software watermarks: Implementation, analysis, and attacks. In *Workshop on Information Hiding*, 2004.

[8] Christian Collberg. Sandmark, August 2004. URL http://www.cs.arizona.edu/sandmark/.

[9] Christian Collberg and Jasvir Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, 2009. ISBN 0321549252, 9780321549259.

[10] Christian Collberg and Clark Thomborson. Software watermarking: Models and dynamic embeddings. In *Principles of Programming Languages 1999, POPL'99*, January 1999.

[11] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, July 1997.

[12] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Principles of Programming Languages 1998, POPL'98*, January 1998.

[13] Christian Collberg, Stephen Kobourov, Edward Carter, and Clark Thomborson. Error-Correcting graphs for software watermarking. In *Proceedings of the 29th Workshop on Graph Theoretic Concepts in Computer Science*, pages 156–167, 2003.

[14] Christian Collberg, Andrew Huntwork, Edward Carter, Gregg Townsend, and Michael Stepp. More on graph theoretic software watermarks: Implementation, analysis, and attacks. *Inf. Softw. Technol.*, 51(1):56–67, 2009.

[15] Christian S. Collberg and Clark Thomborson. Watermarking, Tamper-Proofing, and obfuscation - tools for software protection. In *IEEE Transactions on Software Engineering*, volume 28, page 735746, August 2002.

[16] Christian S. Collberg, Clark Thomborson, and Gregg M.

Townsend. Dynamic graph-based software fingerprinting. *ACM Trans. Program. Lang. Syst.*, 29(6):35, 2007.

[17] Gareth Cronin. A taxonomy of methods for software piracy prevention. Technical report, Department of Computer Science, University of Auckland, New Zealand, 2002.

[18] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a DAG, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–15, St. Petersburg Beach, Florida, United States, 1996. ACM. ISBN 0-89791-769-3.

[19] James Hamilton and Sebastian Danicic. An evaluation of current java bytecode decompilers. In *Ninth IEEE International Workshop on Source Code Analysis and Manipulation*, volume 0, pages 129–136, Edmonton, Alberta, Canada, 2009. IEEE Computer Society. doi: 10.1109/SCAM.2009.24.

[20] James Hamilton and Sebastian Danicic. An evaluation of static java bytecode watermarking. In *Proceedings of the International Conference on Computer Science and Applications (ICCSA'10)*, The World Congress on Engineering and Computer Science (WCECS'10), San Francisco, October 2010. ISBN 978-988-17012-0-6. To appear.

[21] Frank Harary and Edgar M Palmer. *Graphical Enumeration*. Academic Press, New York,, 1973. ISBN 0123242452.

[22] Yong He. *Tamperproofing a Software Watermark by Encoding Constants*. Masters thesis, University of Auckland, June 2002.

[23] M. S. Hecht and J. D. Ullman. Characterizations of reducible flow graphs. *J. ACM*, 21(3):367–375, 1974.

[24] Matthew S. Hecht and Jeffrey D. Ullman. Flow graph reducibility. In *Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 238–250, Denver, Colorado, United States, 1972. ACM.

[25] Zhu Jian-qi, Wang Ai-min, and Liu Yan-heng. Tamper-proofing software watermarking scheme based on constant encoding. *Education Technology and Computer Science, International Workshop on*, 1:129–132, 2010. doi: 10.1109/ETCS.2010.429.

[26] Zhu Jianqi, Liu YanHeng, and Yin KeXin. A novel dynamic graph software watermark scheme. In *Proceedings of the 2009 First International Workshop on Education Technology and Computer Science - Volume 03*, pages 775–780. IEEE Computer Society, 2009. ISBN 978-0-7695-3557-9.

[27] Malik Sikandar Hayat Khiyal, Aihab Khan, Sehrish Amjad, and M. Shahid Khalil. Evaluating effectiveness of tamper proofing on dynamic graph software watermarks. *CoRR*, abs/1001.1974, 2010.

[28] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd ed.): Fundamental Algorithms*, volume 1. Addison Wesley Longman Publishing Co., Inc., 3 edition, 1997. ISBN 0-201-89683-4.

[29] Thomas Koshy. *Catalan Numbers with Applications*. Oxford University Press, 2008. ISBN 9780195334548.

[30] Lay Lam. *Fleeting footsteps: tracing the conception of arithmetic and algebra in ancient China*. World Scientific, Singapore, River Edge NJ, rev. ed. edition, 2004. ISBN 9789812386960. English translation of ”The Mathematical Classic” by Sun Zi.

[31] Yang-Xia Luo, Jian-Hua Cheng, and Ding-Yi Fang. Dynamic graph watermark algorithm based on the threshold scheme. In *Proceedings of the 2008 International Symposium on Information Science and Engieering - Volume 02*, pages 689–693. IEEE Computer Society, 2008. ISBN 978-0-7695-3494-7.

[32] Anshuman Mishra, Rajeev Kumar, and P. P. Chakrabarti. A method-based Whole-Program watermarking scheme for java class files. 2008.

[33] Ginger Myles. Using software watermarking to discourage piracy. *Crossroads - The ACM Student Magazine*, 2004. URL http://www.acm.org/crossroads/xrds10-3/watermarking.html.

[34] Jasvir Nagra and Clark Thomborson. Threading software watermarks. In Jessica J. Fridrich, editor, *Information Hiding*, volume 3200 of *Lecture Notes in Computer Science*, pages 208–223. Springer, 2004. ISBN 3-540-24207-4.

[35] Jasvir Nagra, Clark Thomborson, and Christian Collberg. A functional taxonomy for software watermarking. In Michael J. Oudshoorn, editor, *Aust. Comput. Sci. Commun.*, pages 177–186, Melbourne, Australia, 2002. ACS.

[36] Kelly O'Hair. HPROF: a Heap/CPU profiling tool in J2SE 5.0. *Sun Developer Network*, Developer Technical Articles & Tips, November 2004. URL http://java.sun.com/developer/technicalArticles/Programming/HPROF.html.

[37] Jens Palsberg and Di Ma. Javawiz, 2000. URL http://www.cs.purdue.edu/homes/madi/wm/. No longer available.

[38] Jens Palsberg, S. Krishnaswamy, Minseok Kwon, D. Ma, Qiuyun Shao, and Y. Zhang. Experience with software watermarking. In *ACSAC*, pages 308–316, 2000.

[39] Zhou Ping, Chen Xi, and Yang Xu-Guang. The software watermarking for tamper resistant radix dynamic graph coding. *Inform. Technol. J.*, 9:1236–1240, June 2010. doi: 10.3923/itj.2010.1236.1240.

[40] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, 1994.

[41] Jeremy Spinrad. *Efficient graph representations*. Number 19 in Fields Institute Monographs. American Mathematical Society, Providence R.I., 2003. ISBN 9780821828151.

[42] Sun Microsystems, Inc. Java debug interface, 2005. URL http://java.sun.com/j2se/1.5.0/docs/guide/jpda/jdi/index.html.

[43] Clark Thomborson, Jasvir Nagra, Ram Somaraju, and Charles He. Tamper-proofing software watermarks. In

ACSW Frontiers '04: Proceedings of the second workshop on Australasian information security, Data Mining and Web Intelligence, and Software Internationalisation, page 2736, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[44] Ramarathnam Venkatesan and Vijay Vazirani. Technique for producing through watermarking highly tamper-resistant executable code and resulting watermarked code so formed, May 2006. Microsoft Corporation, US Patent: 7051208.

[45] Ramarathnam Venkatesan, Vijay Vazirani, and Saurabh Sinha. A graph theoretic approach to software watermarking. In *Proceedings of the 4th International Workshop on Information Hiding*, 2001.

[46] Mark Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, page 439449, Piscataway, NJ, USA, 1981. IEEE Press. ISBN 0-89791-146-6.

[47] Wang Yong and Yang Yixian. A software watermark database scheme based on PPCT. In *CIHW2004*, 2004.

[48] J. Zhu, Y. Liu, and K. Yin. A novel planar IPPCT tree structure and characteristics analysis. *Journal of Software*, 5(3):344, 2010.

[49] Jianqi Zhu, Kexin Yin, and Yanheng Liu. A novel DGW scheme based on 2D_PPCT and permutation. *Multimedia Information Networking and Security, International Conference on*, 2:109–113, 2009. doi: 10.1109/MINES.2009.177.

[50] William Zhu. Informed recognition in software watermarking. In *Proceedings of the 2007 Pacific Asia conference on Intelligence and security informatics*, pages 257–261, Chengdu, China, 2007. Springer-Verlag. ISBN 978-3-540-71548-1.

[51] William Feng Zhu. *Concepts and Techniques in Software Watermarking and Obfuscation*. PhD thesis, The University of Auckland, 2007.