A Graph Theoretic Approach to Software Watermarking

Ramarathnam Venkatesan
Microsoft Research

Vijay Vazirani
Georgia Tech

Saurabh Sinha
UW Seattle

March 23, 2000

Abstract

We present a graph theoretic approach for watermarking software in a robust fashion. While watermarking typical software that are small in size (e.g. a few kilobytes) may be infeasible through this approach, it seems to be a viable scheme for large applications. Our approach works with program flow graphs and uses some abstractions, approximate k-partitions, and a random walk method to embed the watermark, with the goal of minimizing and controlling the additions to be made for embedding, while keeping the estimated effort to undo the Watermark (WM) as high as possible. The watermarks are so embedded that small changes to the software or flow graphs are unlikely to disable detection by a probabilistic algorithm that has a secret. This is done by using some relatively robust graph properties, and with some error correcting codes.

Under some natural assumptions about the added code to embed the WM, locating the WM by an attacker is related to some graph approximation problems. Since hardness of typical instances of graph approximation problems has little theoretical work done so far, we present some heuristics to generate such hard instances and in a limited case present some heuristic analysis of how hard it is separate the WM in an information theoretic model. We describe some related experimental work. Our watermarking scheme has connections to software tamper resistance as well.

1 Motivation

The problem of software watermarking, at a very basic level, is to insert some data W (the watermark) into a program P, so that in the resulting program P' it is not easy to detect and remove the watermark. To motivate our approach, we look at some toy examples of watermarking schemes and possible attacks against each.

Scheme 1: Let W_k be a small piece of W, and let $\mathrm{cr}(W_k)$ be an encryption of W_k. Suppose we insert W_k in the form of an instruction like *move RegisterX, cr(W_k)* just before another instruction that writes *RegisterX*. We could insert all pieces W_k of W in this manner, distributed at different places in the program. However, a simple algorithm that does register flow analysis would discover that the instructions we inserted are dead code, and remove them altogether. Clearly, this scheme of inserting W is not safe to automated attacks. Note that even if W has been somehow encoded in the form of a dummy function $W(x)$ which is called at various places in the program, unless the values returned by $W(x)$ affects the program variables, a data flow analysis program would detect the redundancy of $W(x)$ and remove it.

Scheme 2: We now modify it so that actual program variables are in some way affected by $W(x)$. Suppose at some point in P, variables x and z are *live*, and suppose z is used in instruction $I(z)$. Replace $I(z)$ with the sequence of instructions:

```
y := W(x)
t := Encrypt(z,y) /* use y as a key */
z := Decrypt(t,y)
I(z)
```

Now, $W(x)$ can be seen to be redundant upon careful visual inspection, but it might be difficult for automated tools to discover this redundancy. Clearly one can think of may other ways of link W tightly with the program P. However, the link between P and W is still *weak* in the following sense: considering P as a graph, and W as a graph, the function call between P and W is a single-edge cut between the two subgraphs, and an algorithm that looks for such single-edge cuts between regions of the graph P' would be able to flag W as a possible candidate for removal.

In fact, there are graph algorithms that can efficiently separate regions of a graph that are *weakly connected*, where a weak connection may mean small cuts. Moreover, such a *graph based* attack on the watermark is effective against any scheme that inserts W in the code/data section, without ensuring that the subgraph W is not weakly connected to the rest of the graph. In other words, any method that attempts to place a WM in the code/data section must contend with attacks that use such automated tools to create a short list of suspected WM locations, which can be isolated with smaller amount of semantic information or human intervention. We propose an algorithm for inserting or *embedding* a watermark graph W into a program graph P such that the adversary is not at any advantage with automated tools of the type mentioned above, and is thus forced into excessive visual inspection and semantic inference. Note that an attacker can use the semantics by observing the execution and input-output behaviour and effectively re-write the program, removing any WM.

Another attractive feature of the solution, is that provides a tool for Software Tamper Resistance. The goal is to make an executable resist changes to the code (e.g. to remove a license check) without excessive tracing and use of semantics. We do not address the details here, but we beileve a good solution must address both WM and tamper resistance together, which other approaches do not seem to attempt. We mention the general principles but addressing criteria and actual generation of the code that is inserted to meet these principles is beyond the scope of this paper;in practice, this is a significant amount of work that is quite important. While making the modifications to the original code, it is important to preserve the performace, and we note that the usual profiling and optimizing approaches work with the graph of the programs as well.

1.1 Difficulties in designing a robust WM

Hiding and recovering the watermark The task of inserting a WM in such a way that it cannot be recovered efficiently calls for some sort of a one-way function on executables, or on their flow graphs. But unlike in cryptography, there are no known ways of defining plausible one-way functions on these domains. There are basic difficulties in accomplishing this. For instance, to design and reason about one-way functions, one must specify the distribution of instances. In cryptography, it is crucial that the instances are random, whereas in our case, the graphs are already generated by some specific development process, and they cannot be modified substantially without degradation of performance. An additional difficulty is that the attack algorithms need only be approximate, in a sense that will become clear later. Therefore, the flexiibility for generating hard instances for the watermark removal problem is constrained.

Our constructions are graph theoretically motivated and can be seen as a heuristic to hide a WM in a way that would require identifying a specific cut (or a close approximation to it) among an exponential

number of cuts with the same or nearly same graph theoretic parameters. To achieve this, we extract a random-looking graph from the orignal flow graphs, using a k-partition algorithm. In spririt, this is similar to Szemerdi's regularity lemma, which attempts to extract a random looking structures [deistel]. The implicit constants for the lemma are truly astronomical and no improvement is possible, although weaker versions of regularity are sufficient [KF]. Ours is a "poor mans version" of these. Empirically, it appears plausible that such an extraction can be done on relatively large programs. As noted earlier, contending with such cut-based attacks seems necessary for any WM method.

The above discussion indicates that we cannot hope to successfully watermark a small executable without significantly increasing its size artificially. Another problem to contend with is recovery of the WM from an attacked version of the program. We view this problem as designing a function that uses a secret key and returns the same value on a graph even if it is subject to malicious but small alterations.

Local Indistinguishability The added code W (or data) should not be distinguishable from the original payload code P by looking at local properties. For example, there may be more than usual randomized data in a segment, and this can be detected by using tools like [Shamir et al]. Alternately, unusual access patterns may be found and exploited in shortlisting suspected watermark locations. Moreover, addition of the WM may also cause local properties to be noticably different. We handle this problem heuristically and our way of adding to the executable is to minimize the possibility of inserting any noticeable anamoly.

Some available tools First we would like to point out the existence of tools that take a program binary as input, construct the corresponding control flow graph at some basic block level, and provide an interface that allows transformations to be made to this graph. Some are even available as disassemblers with well defined interfaces. Such tools, hereafter referred to as *graph analyzers*, give the adversary the ability to observe and make modifications without changing its functionality, and can be used successfully for reverse engineering. For example, the Machine-SUIF CFG library ([**?**]) provides a Control Flow Graph interface to programs, where nodes are lists of instructions. Similarly, OPTIMIX ([**?**]) is a tool that allows tranformations and optimizations in a program through a graph rewrite specification. Examples of poweful disassemblers are [Sourcer] and [Ursoft]. To restate what has been explained above, any watermarking scheme of the future has to be robust to attacks that make heavy use of graph analyzers. Secondly, the algorithms for partition, separator, and cut problems of various flavours seem to work quite well in practice for the typical inputs that occur here. Finally we focus on WM for any executable in this paper; knowledge of the domain and typical operations as well as the implementation details can be used to harness the WM and can be used in conjunction with this work.

2 Previous Work

A comprehensive survey and taxanomy appears in [Collberg et al]. *Static* schemes embed them in *code section (code watermarks)*, or in the *data section (data watermarks)*. While the latter may be relatively easy to recover and remove, code watermarks are more robust and may be encoded in the order of independent instructions, in register use patterns [BCS], or control flow layout (e.g., order of C-style case statements or basic block sequence in the program flow graph ([Davidson]). Many of the methods may be prone to *distortive attacks* by a graph analyzer, which shuffle the crucial order or pattern while maintaining the functionality of the program.

Dynamic schemes store the watermark in the program's excution state. See [Collberg et al] for a brief description of their variety, where some weaknesses and possible attacks are pointed out, and for a variant wherein the topology of a dynamically built graph (stored on the heap) encodes the watermark. See also [Anderson et al].

For software protection, [Sander et al] describes a scheme for *hiding* a function (implementing an arbitrary polynomial) in the program, where an execution yields encrypted result which a legal user needs to decrypt via a trusted authority; thus illegal users will be unable to proceed further and use the result. We wish to consider detection by an agent or device possessing some secret and no other interventions. We do not address the issue of such agents any further.

3 Goals and Assumptions

Below, the term *program* refers to the usual notion of a computer program running on a RAM mcahine. It includes programs in high-level languages such as C, and executable *binaries*, i.e., programs that are a sequence of machine-specific instructions. Also, two programs P and P' are said to be *functionally equivalent* if their output is the same for any user-input and the user-interface or the performance does not have any discernible difference; we allow minor differences such as in the exact instructions and their order in two programs.

3.1 Software Watermarking

A watermarking algorithm E takes as input a program P, a *watermark* object W, and a random secret key ω, and outputs a program P'; in short, $E(P, W, \omega) = P'$ such that P' is functionally equivalent to and not much larger than P and there exists an *efficient* algorithm e that can retrieve W from P' given a key, i.e., $e(P', K) = W$, where e is called an extractor for the watermark. The key $K = f(P, W, P', \omega)$ for some f. (e.g., $K = \omega$ could be a key).

Let A denote an adversary that modifies the program P' to produce $A(P')$. (We shall see shortly what we mean by an adversary, and in what ways it may modify a program.) A watermarking algorithm is said to be *secure* against A if $\exists$ an efficient extractor e such that $e(A(P'), K) = W$ if $P' = E(P, W, \omega)$ for some P, W, ω and $e(A(P'), K) = NULL$ otherwise.

3.2 The adversary

Now we consider relevant adversarial models. Based on how the adversary modifies P', we classify it as being either a *removing*, *extracting* or a *jamming* adversary. A *removing* adversary A_r is such that $A_r(P') = P''$, where P'' and P' are functionally equivalent and P'' in an information theoretic sense has no information about WM. For example, it could be a human agent who is assisted by a powerful tool and examines the entire program P', instruction by instruction, infers the semantics, and writes an equivalent P''. Such an adversary can undo any watermark. Our goal is to ensure that A_r is the only possible model of an effective adversary. For notational convenience, we define an *extracting* adversary A_e as $A_e(P') = W$. Finally, the *jamming* adversary A_j modifies the program P' so that it is difficult for the watermark extractor e to extract the watermark, even with the secret key K, i.e., $A_j(P') = P''$, where P' and P'' are functionally equivalent programs, yet $e(P'', K) \neq W$, for any *efficient* e. The jamming adversary is the most important model from our point of view, and A in the definition of a secure watermark in Section 3.1 is assumed to be such an adversary.

We shall assume that a successful attack may be probabilistic, in the sense that it could succeed in undoing the watermark with a high probability, or it may be approximate, i.e., it could undo a significant portion of the watermark.

4 Graph Theory and Watermarking

As noted earlier, we refer to programs by their corresponding flow graphs, where nodes correspond to basic blocks in the program and edges correspond to control flow (jumps and 'fall through's) and function calls. Let the flow

graphs of P, W and P' be G,W and H respectively. We may describe

the process of watermarking as $G + W \rightarrow H$. It *merges* G and W by adding edges. Addition of edges corresponds to automated ways of inserting code, data and control flow into the programs. We claim two necessary conditions for a good watermarking scheme in this framework:

1. W must be *locally indistinguishable* from G in H.
2. W must be *well connected* to G in H.

Condition (1) is explained in Section 1.1, while condition (2) is explained in the next section.

4.1 Separation Algorithms

$E(G,W)$, the set of edges between G and W, $E(G,W)$, should be properly chosen so that standard separation algorithms cannot locate it, even approximately.

The Lipton-Tarjan algorithm [Lipton-Tarjan], for example, can find (roughly) equal-sized partitions with a low *cut ratio* in planar graphs. Other efficient partitioning methods include spectral partitioning ([Spielman et al], [Guattery et al]), and multi-commodity flow based methods. Metis ([Metis]) is a family of programs that uses certain heuristics to finds low cut partitions very efficiently, making the problem of hiding the cut non-trivial.

4.2 Hiding a cut in a graph

Our ϵ-*separation* problem is: *Given H, partition its nodes into G' and W' such that at least $1-\epsilon$ fraction of nodes of W are in W' and at least $1-\epsilon$ fraction of the nodes of G are in G'.* The original G, W are not part of the input. Clearly, the separation problem is equivalent to finding the 'right' cut, within a small margin of error. $E(P,W,\omega)$ therefore must *hide* the cut so that it is hard to recover. We present a heuristic that hides a cut of size m in H such that it is hard to find, even approximately, from an information theoretic standpoint.

5 Embedding the watermark

In this section, we describe how to construct H such that the separation problem is *likely* to be hard on H. We shall make certain assumptions about G and W and defer the experimental justification of some of those assumptions till Section 7.

5.1 Algorithm

Given: Program P, watermarking code W, secret keys ω_1, ω_2 and ω_2, integer m.

1. *Graph step*: Compute *flow graph* G from P. As mentioned earlier, the flow graph has the basic blocks of P as nodes, and edges correspond to either control flow or to function calls. Similarly for W. G and W are both digraphs.

2. *Clustering step*: Partition the graph G into n clusters using

   ω_1 as random seed, so that edges straddling across clusters are minimized. Let G_c be the graph where each node corresponds to a cluster in G and there is an edge between two nodes if the corresponding clusters in G have an edge going across them. This step produces an undirected graph G_c of smaller order. Similarly, W yields W_c.

3. *Merging step*: Here we add edges to and between G_c and W_c using a random process. Recall that adding an edge between two nodes implies adding an edge between two basic blocks in the original flow graphs, which may be translated using tools into transformations that add code,data and control flow to the programs. The edges are added by a *random walk*: Assume we are at a node v in say G_c. We look at current values of d_{gg} and d_{gw}, the number of nodes adjacent to v in G_c and W_c respectively. Let $p_{gg} = \frac{d_{gg}}{d_{gg}+d_{gw}}$ and $p_{gw} = \frac{d_{gw}}{d_{gg}+d_{gw}}$. We visit next a random node in G_c with probability p_{gw} or a node in W_c with probability p_{gg}. The choices are made using ω_2. In this step we add an edge from v to u. We use an analogous method of choosing the next node if we are currently in W_c. We repeat this m times. Let H be the resultant graph.

4. *Recovery step*: Compute a *fingerprint* of W_c - use some robust property of W_c, compute its value and encrypt it using secret key ω_3. We shall discuss shortly which robust properties may be considered.

5.2 Discussion

The undirected graphs G_c and W_c obtained from Step 2 in the algorithm are found, empirically, to be very similar in structure to the random graph model $G_{n,p}$, with n nodes and edge probability p ([Bollobas]).

We next examine the effect of the merging step (3) on the graphs G_c and W_c. We assume that the edges of $E(G_c, W_c)$, added by the random process in Step 3, are such that we can bound the probability of each edge being present away from 0 and 1 by suitable constants ϵ_1 and ϵ_2.

The purpose of the random walk is to yield a better merged graph in terms of local indistinguishability for actual inputs in the presence of real world imperfections of a truly random graph.

Call an equi-partition $\{G', W'\}$ of H an *m-partition* if the cut size $|E(G'W')| = m$. Call an m-partition $\{G', W'\}$ *good* (for the adversary) if $|W' \cap W|/|W| \geq 1 - \epsilon$. The ratio of the expected number of *good* m-partitions to the expected number of all m-partitions (in H) is exponentially small in n, for some value of m.

Proof: Consider an arbitrary equi-partition of the nodes of H into G' and W'. Let $|G' - G_c| = x$. Therefore, $|W' - W_c| = x$. Now consider the following sequence of random variables: X_1 = number of edges between $G' \cap G_c$ and $W' \cap W_c$, X_2 = number of edges between $G' - G_c$ and $W' - W_c$, X_3 = number of edges between $G' \cap G_c$ and $G' - G_c$, X_4 = number of edges between $W' \cap W_c$ and $W' - W_c$.

Note that each of these random variables follows a *binomial distribution*: $X_1 \sim b((n-x)^2, p')$, $X_2 \sim b(x^2, p')$, $X_3 \sim b(x(n-x), p)$, $X_4 \sim b(x(n-x), p)$. Clearly, the partition $\{G', W'\}$ is an m-partition if and only if $\sum_i X_i = m$. Let $p_x = Pr[\sum X_i = m]$.

The ratio mentioned in the claim is equal to

$$\frac{\sum_{x=0}^{n/10} \binom{n}{x}^2 p_x}{\sum_{x=0}^{n} \binom{n}{x}^2 p_x} \tag{1}$$

Clearly, if we can prove that $p_x = \theta(1/n^c)$, for some c, then Claim 2 follows. Let $S_x = \sum X_i$ and $m_x = E(S_x) = \sum E(X_i) = (n^2 - 2nx + 2x^2)p' + 2(nx - x^2)p$. To be continued...

QED.

Thus, even with the knowledge of the correct m (which the adversary may randomly guess), it has little chance of finding a *good* partition.

Step 4 of the algorithm computes some *robust* function of W_c. Since W_c is assumed to be a large random graph, it has several properties that are resistant to minor changes in the graph. For example, if A is the adjacency matrix of W_c and d is its diameter, then $A^{\frac{d}{2}}$ is expected to be robust to a few changes in A. This has empirical evidence for it, which is presented in Section 7. Other properties of the graph, such as the all-pairs minimum s-t cut size, or the path length distribution may also be provably robust, and further investigation is needed on these aspects.

As we have stated earlier, this watermarking algorithm may be used in conjunction with any other scheme that exploits semantic or other specific knowledge about the programs.

5.3 Software Tamper Resistance

One important feature of our watermarking approach is its implicit yet strong connection to the problem of *software tamper resistance*. Informally, tamper resistance refers to the ability of fragments of programs to be resistant to modification by an active adversary. Such resistance is usually required in portions of code that make crucial checks pertaining to the validity of relevant software licenses. Since our watermarking approach is based on generating hard instances of the location problem, it may be used to embed crucial checks in the program such that it is hard to locate and hence tamper with them. We believe this to be a positive feature of our approach.

6 Watermark Recovery

7 Experiments

At the beginning of Section 5.2 we made the assumption that the graphs G_c and W_c obtained by clustering the corresponding flow graphs are similar in structure to the random graph $G_{n,p}$, a graph with n nodes where each of the $\binom{n}{2}$ edges is present with a probability p. This assumption is necessary for the analysis to go through, and in Section 7.1 we shall see why the assumption may be justified. Later, in Section 7.2, we present experimental evidence that suggests that the medium powers of the

adjacency matrix are robust to addition and deletion of a few edges in the graph. This fact is used in Step 4 of the algorithm in Section 5.1

7.1 Random Graph Tests

Let n = number of nodes in G_c and m = number of edges. We describe below some tests in support of the hypothesis that G_c is very similar to a random graph $G_{n,p}$, with $p = \frac{m}{\binom{n}{2}}$. The experimental steps are:

1. Obtain G_c corresponding to a large application binary (several Megabytes in size).
2. Pick k vertices of G_c at random and let the subgraph of G_c induced by V be called a *hyperedge*. Since each actual edge in this hyperedge is hypothesized to be present with probability p, the number of actual edges in the hyperedge should follow a binomial distribution $b(\binom{k}{2}, p)$. Randomly pick N hyperedges as described above, independently, and count the number of hyperedges that have c actual edges, for small values of c. Let the number of hyperedges with c actual edges be denoted by N_c, and let the corresponding random variable, which counts the hyperedges with c actual edges in a random graph $G_{n,p}$ be X_c. If $K = \binom{k}{2}$, we have Pr[number of edges in a hyperedge is c] $= \binom{K}{c} p^c (1-p)^{K-c} = p_c$ (say), and hence $E(X_c) = Np_c$. Thus this step gives us one sample (N_c) for a random variable that under the hypothsis has an expectation of $E(X_c) = Np_c$.
3. Repeat Step 1 a large number of times, say T times, to get T samples of the random variable X_c. Compute the observed mean $\bar{N_c}$ and compare it with the expectation $E(X_c)$.

We performed the above steps for $N = 10000$, $T = 1000$, $c = \{0, 1, 2\}$ and $k = \{3, 4, 5, 6\}$. The following tables summarize the results:

	$c=0$		$c=1$		$c=2$	
k	Expected	Observed	Expected	Observed	Expected	Observed
3	9838	9836.4	158	162.7	2	0.9
4	9682	9675.5	306	320.1	10	4.4
5	9480	9465.0	491	521.8	25	12.9
6	9238	9208.4	706	761.5	49	29.4

The table provides evidence in support of the validity of our assumption. More precise methods were used to to test the hypothesis with a specific high *level of significance*. χ^2?

7.2 Robust properties

Let A be the adjacency matrix of W_c and let d_W be its diameter. The hypothesis (see Section 5.1 and Section 6) is that the addition or deletion of a few edges in W_c do not cause a big change in $A^{\frac{d_W}{2}}$. We test this hypthesis on the n-node graph G_c, with the underlying assumption being that the same results will hold for W_c also. In our experiments, we chose $n = 5000$, and hence A is a 5000×5000 matrix. The diameter d was found to be 8, which is consistent with the known probabilistic bounds on the diamter of a random graph. ([Bollobas]) We then computed $A^{d/2}$, made some random additions and deletions to W_c and recomputed $A^{d/2}$. It was found that typically about 0.001% of the entries in this matrix had changed values, which implies that the matrix is robust to changes in the graph.

8 Conclusions

9 Acknowledgements

We thank Mariusz Jacubowski and Jayram Thatacher for their invaluable help in this project.

References

[Feller] William Feller An Introduction to Probability Theory and Its Applications

[Bollobas] Bollobas Random Graphs

[Collberg et al] Christian Collberg and Clark Thomborson Software Watermarking: Models and DynamicEmbeddings

[Anderson et al] Ross J. Anderson and Fabien A.P. Petitcolas On the limits of Steganography

[Frieze et al] Alan Frieze and Ravi Kannan The Regularity lemma and Approximation schemes for dense problems

[Diestel] Reinhard Diestel Graph Theory

[Guattery et al] Stephen Guattery and Gary L. Miller On the Performance of Spectral Graph Partitioning Methods

[Spielman et al] Daniel A. Spielman and Shang-Hua Teng Spectral Partitioning works: Planar graphs and finite element meshes.

[Sourcer] http://www.v-com.com/products/sourcer.html

[Ursoft] http://www.softseek.com/Programming/Assembly_Language/Review_9874_index.h tml

[Davidson] Robert L. Davidson and Nathan Myhrvold Method and system for generating and auditing a signature for a computer program. US Patent 5559884, September 1996. Assignee: Microsoft Corporation

[BCS] Council for IBM Corporation Software watermarks. Talk to BCS Technology of Software Protection Special Interest Interest Group. Reported in [AP].

[Shamir et al]

[Lipton-Tarjan]

[Metis]

[Sander et al] Tomas Sander, Christian F. Tschudin. On Software Protection Via Function Hiding, IHW'98 - Proc. of the International Information hiding Workshop, April. 1998.