Summary of *A Graph Theoretic Approach to Software Watermarking*

Danny Mandel

March 4, 2002

1 Motivation

This section starts with some basic watermarking concepts and goes through a couple of possible software watermarking schemes. The first scheme described is that of inserting false instructions into application binaries through the use of some special register. Of course, this algorithm is simple and could be easily cracked by a simple register analysis program which would identify these instruction as dead code and remove them. The authors then put forth the next complexity up from the previous idea, that of inserting the watermark into a function called at runtime that affects the live variables of the program. While this provides a better defense against automated attacks, upon careful inspection by a human this method is also easily discovered.

When viewing the watermark code and the original program code as control flow graphs, simple function calls are single edges which connect the two graphs. As such, they can easily be cut if one desired to separate the watermark from the program (i.e. attackers – like Bob). Though it may be difficult for humans to analyze complex control flow graphs, the authors point out that there are a number of graph algorithms for analyzing graphs and looking for portions of graphs that are *weakly connected*, which would highlight possible existence of small *cuts* (informally, a k-cut is a set of $k-1$ cuts across edges in a graph, breaking the graph into disjoint subgraphs). The authors term such an attack on the control flow graph a *graph based attack*. Given that these algorithms exist, the authors seek to create a way to *strongly connect* the watermark code to the rest of the program in order to make it difficult for automated tools to easily remove the watermark. Though they don't mention how or why they believe this to be the case, the authors believe any watermarking solution must contain tamperproofing as well.

1.1 Difficulties in designing a WM

In this section, the authors detail the reasons why it is difficult to watermark software. Though it is somewhat similar to cryptography, in that the goal is to prevent an adversary from illegally gaining access to or modifying something that we wish to keep secure, it is much more difficult to design one-way functions for software watermarks. In cryptography, the designers of the algorithm can create uniformly distributed instances of problems in which to embed the cyphertext. However, with software watermarking, the designers of the algorithm need to modify existing data (program binaries or sources), and must be careful not to modify them too greatly or suffer greatly decreased performance of programs. In addition, cracking cryptographic codes requires specific accuracy, while in many cases, software watermarking attacks need only be approximately successful.

To deal with these difficulties, the authors chose to approach the problem from a graph theoretic standpoint. The general strategy is to require the attacker to identify a specific cut from an exponential number of possibilities in order to remove the watermark. The connection to the watermark portion of the graph is achieved by picking a pseudorandom portion of a modified version of the original program's graph. This method is based on existing proven-difficult problems in graph theory. This technique will not be able to work on small executables without significantly increasing program size.

Another consideration when inserting a watermark into a program is hiding the code from programs that can recognize non-random distributions of instructions or data. Though such programs may not be able to pinpoint the exact location of a watermark, they may be able to rule out certain portions of the program as watermark locations, which makes an attacker's task easier.

2 Previous Work

In this section, the authors talk about previous papers, including Christian's. They also go over the difference between static and dynamic watermarks.

3 Goals and Assumptions

Here the authors go over their assumed notion of a program and functional equivalence.

3.1 Software Watermarking

A formal definition of what it means to watermark a program is found in this section. They also define the notion of a secure watermark - a watermark that can still be extracted after facing attacks from some adversary.

3.2 The adversary

The possible watermark attacks are discussed in this section of the paper. The terms *removing* and *jamming* adversaries are introduced in this section. A removing adversary is one who attempts to rewrite the entire program P with the aid of a decompiler or some other tool. A jamming adversary is one who attempts to distort the watermark so that it is difficult to recognize. In this paper, the authors worry about jamming adversaries, since it is impossible to prevent someone from rewriting a program. Additionally, the time and effort required to do so is generally so great that removing adversaries aren't considered a real threat.

4 Basic Principles

Here the authors give a formal explanation of what their algorithm will do. Given the flow graphs for the program, G, and the flow graph for the watermark, W, they produce a new graph, H, by embedding a number of edges between G and H that form a cut. Formally this is written as:

$G + W \rightarrow H$

They also introduce two other variables, G_H (the portion of H induced by G) and W_H (the portion of H induced by W). Two necessary conditions of a good watermark in their framework are:

- W_H must be *locally indistinguishable* from G_H.
- W_H must be *well connected* to G_H in H.

4.1 Hiding a cut

The authors formally define the problem of finding the cut between W and H as the ϵ-*separation* problem. It is stated as follows: 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 $G^{'}$ If this cut can be found, then the watermark can be found and removed. Thus, in order to have a safe watermark, the algorithm must hide the cut so that users cannot remove it. Though there are many algorithms for finding cuts of some size m, the correct cuts will still be difficult to locate if there are many cuts of roughly the same size in the new graph.

5 Embedding the waterark

The point of the embedding is to embed the cut so that the separation problem is likely to be hard on the newly constructed graph H.

5.1 Watermarking Algorithm

Assuming we have the following variables: Program P, watermarking code W, secret keys ω_1 and ω_2, integers m, n.

1) *Graph Step:* Compute flow graph G from P and flow graph W from W.

2) *Clustering Step:* Partition the graph G into n clusters using ω_1 as random seed. Call the partitioned graphs G_c and W_c respectively.

3) *Regularity Step:* Add edges between G_c and W_c using a *random walk*. Use the degrees of the vertices in the graphs as probabilities for choosing the next node to visit. These edges are implemented in the program by means of opaque construct (they use Christian's term without citing him – MS bastards).

5.2 Discussion

The clustering step (2) must be somewhat random, or at the very least, unpredictable. A predictable clustering alorithm would make identifying G and W easier.

A great deal of the discussion of the algorithm is dedicated to proving that their method truly simulates a random cut that is locally insdistinguishable from the rest of the program. It is important for their method to do so, since this is the true stealth of this algorithm. The reason this method is so stealthy is that there are an exponential number of possible ways of connecting G and W, so correctly guessing the connection between the two involves an exponential amount of work. It should be noted that one should assume the adversary knows m, the number of edges going between G and W, since they can try all possible values of m. It is also important that the number m not be either very small (close to 0) or very large (close to n^2), since algorithms that find very small or very large cuts will work.

5.3 Recovery of watermark

At this point, their algorithm breaks down a tad bit. For them to recover the watermark, there must be some part of the binary that effectively flags the location of the watermark. They don't specify how they propose to do this, other than it is encrypted. Ideally, one would publish the algorithm used for doing so in the way that most encryption or other watermarking algorithms are published.

Once they have used their secret black box scheme for identifying the nodes in W, they start choosing small subsets of from W, w and computing some "robust function" on the nodes in w. They don't really specify what the "robust function" is, though they say that it could be any of a number of different functions that rely on *vertex-invariants*. Example of such invariants are vertex degree or k-neighborhood size. It should be noted that looking at the individual adjacency matrix of W won't work since we assume that the adversary may have modified W in some manner. However, when we raise the adjacency matrix, A, to some power k, then it's expected that these small differences will become less apparent and the resultant matrix will be quite similar to the original matrix raised to the same power k.

6 Experiments

It should be noted that these experiments didn't involve actually trying to identify watermarks or any simulation of an adversarial model. The point of them was to show that their algorithm simulated the randomness that they had hoped while maintaining local indistinguishability. Their results showed that their algorithm was reasonable successful in choosing “random” locations to embed the edges. They also observed that their hypothesis of multiplying a slightly modified adjacency matrix to some power k was within 4% of the value of the original matrix value.

References

Christopher Jermaine. *Computing Program Modularizations Using the k-Cut Method.* http://www.cc.gatech.edu/people/home/jermaine/k-cut.pdf

Ramarathnam Venkatesan, Vijay Vazirani, and Saurabh Sinha. *A Graph Theoretic Approach to Software Watermarking*