A Graph Theoretic Approach to Software Watermarking

Ramarathnam Venkatesan[1], Vijay Vazirani[2], and Saurabh Sinha[3]

[1] Microsoft Research
venkie@microsoft.com
[2] Georgia Tech
vazirani@cs.gatech.edu
[3] University of Washington
saurabh@cs.washington.edu

Abstract. We present a graph theoretic approach for watermarking software in a robust fashion. While watermarking 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 control/data flow graphs and uses 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 graph are unlikely to disable detection by a probabilistic algorithm that has a secret. This is done by using some relatively robust graph properties and error correcting codes.
Under some natural assumptions about the code added to embed the WM, locating the WM by an attacker is related to some graph approximation problems. Since little theoretical foundation exists for hardness of typical instances of graph approximation problems, we present heuristics to generate such hard instances and, in a limited case, present a heuristic analysis of how hard it is to separate the WM in an information theoretic model. We describe some related experimental work. The approach and methods described here also suitable for solving the problem of *software tamper resistance*.

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.

I. S. Moskowitz (Ed.): IH 2001, LNCS 2137, pp. 157–168, 2001.

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 from automated attacks. 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 $W(x)$ affect the program variables, a data flow analysis program would detect the redundancy of $W(x)$ and remove it.

Scheme 2: Now 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:

$$\begin{aligned} & y := W(x) \\ & t := Encrypt(z, y)/*useyasakey*/ \\ & z := Decrypt(t, y) \\ & I(z) \end{aligned}$$

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 many other ways of linking 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 that there is a small cut. 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 wesoftware akly 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.

An attractive feature of the solution is that it provides a tool for Software Tamper Resistance. The goal here 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 detail here; briefly our method describes the construction of the graph that how different code fragments (each of which correspond to a node v_i of the graph) cross check the other fragments (corresponding to the nodes v_j that are adjacent to a given v_i). For robustness, 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 the specific criteria and actual generation of the inserted code 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 performance, and we note that the usual profiling and optimizing approaches work with the graph of the watermarked program as well.

1.1 Difficulties in Designing a 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 one-way functions on these domains and there are basic difficulties in accomplishing this in a rigorous or plausible way. For instance, to design and reason about one-way functions, one must specify the distribution of instances, which in cryptography, one can and usually takes to be uniform. But 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, for which little theory exists to reason about typical instances. Therefore, the flexibility for generating hard instances for the watermark removal problem is rather 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 pseudo-randomly extract a random-looking graph from the orignal flow graphs, using a k-partition algorithm. In spririt, this is similar to Szemeredi's regularity lemma, which embeds random looking structures [Diestel] in an arbitrary but large enough dense graph. The implicit constants for the lemma are truly astronomical and it is known that no improvement is possible, although weaker versions of regularity are often sufficient [FK]. Ours may be thought of as a poor man's version of these. Empirically, it appears plausible that such an extraction can be done on relatively large programs.

It is unlikely that we can 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 "**graph hashing**" function that uses a secret key and returns the same value on a graph even if it is subject to small alterations. Formally, this problem can use natural metrics such as edit distance bewtween graphs through addition and deletion of nodes and edges.

Local Indistinguishability The added code (or data) W 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, such as that developed by Adi Shamir and Nico van Somerin. 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.

1.2 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 the 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 to a program without changing its functionality, and can be used successfully for reverse engineering. For example, the Machine-SUIF CFG library [Holl] provides a Control Flow Graph interface to programs, where nodes are lists of instructions. Similarly, OPTIMIX [A] is a tool that allows tranformations and optimizations in a program through a graph rewrite specification. Examples of poweful disassemblers are [So] and *Ursoft*'s W32Dasm. A very powerful tool called *Vulcan* [Vulcan] serves as a disassembler and provides a flexible interface to the static flow graph. To restate what has been described above, any watermarking scheme of the future has to be robust to automated or semi-manual attacks that extensively use 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 taxonomy appears in [CT]. *Static* schemes embed watermarks in the *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]. But these 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, for example using data structures with some invariant properties to encode a WM. (One needs to ensure that the WM is insensitive to small distortions to these strcutures.) See [CT] for a brief descriptions, some weaknesses and possible attacks. Also see [AP].

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 secret key ω, and outputs a program P' (i.e., $E(P, W, \omega) = P'$) 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 K, i.e., $e(P', K) = W$. 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 $\exists P, W, \omega$ such that $P' = E(P, W, \omega)$ and $K = f(P, W, P', \omega)$
- $e(A(P'), K) = NULL$ otherwise.

In other words, the extractor must detect the presence or absence of watermarks in face of possible adversarial modifications, and furthermore, extract W if present.

3.2 The Adversary

Now we consider relevant adversarial models. An adversary modifies P' to produce a functionally equivalent program. Based on the extent to which the modification is done, we have *removing* adversaries and *jamming* adversaries. A *removing* adversary A_r is such that $A_r(P') = P''$, where P'' in an information theoretic sense has no information about W. 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, and our goal is to ensure that this the only possible model of an effective adversary. (By "undoing" a watermark, we mean rendering it impossible to detect by any efficient extractor.) The *jamming* adversary A_j modifies the program P' so that it is *difficult* for the extractor e to extract the watermark, even with the secret key K. This is a more practical model of adversary - it has more limited capabilities than the *removing* adversary. It cannot remove the watermark, but renders it hard to detect. We focus on

security against this type of adversary in the rest of the paper. By a *probabilistic* adversary we shall mean one that could succeed in undoing the watermark with a high probability. The adversary may also be *approximate*, i.e., it could undo a significant portion of the watermark.

4 Basic Principles

As noted earlier, we refer to programs by their 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 abstract the process of watermarking as $G+W \rightarrow H$. It *merges* G and W by adding edges. These edges form a *cut* that is discussed in much detail below. Addition of edges corresponds to automated ways of inserting code, data and control flow into the programs. Let G_H be the subgraph of H induced by the nodes of G. Similarly, let W_H be the subgraph induced by the nodes of W. We claim two necessary conditions for a good watermarking scheme in this framework:

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

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

4.1 Hiding a Cut

In the above framework, we define the ϵ-*separation* problem as: *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 given. Intuitively, the separation problem is to find the "right" cut, within a small margin of error. This "right" cut reveals W approximately, therefore $E(P, W, \omega)$ must *hide* the cut so that it is hard to recover. Our heuristic may be viewed as a steganographic hiding of a cut of size m in H such that it is hard to find, even approximately, from an information theoretic standpoint. We emphasize that this is only a pre-requisite for a secure watermark, since an easily detected cut exposes the watermark to a graph-based attack.

Literature is rich with separation algorithms that find cuts meeting various criteria: See [LT] for nearly equal partitioning of planar graphs with a low cut ratio, [GSV99] for approximating optimal separator cuts in planar graphs, [ST96, GM] for spectral methods and [LR88, LLR95] for multi-commodity flow based methods. Heuristic algorithms such as Metis [KK] are effective at partitioning in practice and this makes the task of hiding the cut non-trivial. We hide a cut of size m in such a way that there are many cuts of size m' for each integer $m' \in [m-\Delta, m+\Delta]$, for some suitably chosen $\Delta > 0$. Thus even if there is an algorithm to find cuts of size $m' \in [m-\Delta, m+\Delta]$, the "right" cut is hidden in an information theoretic sense.

It is now clear what we mean by the requirement that W_H must be *well connected* to G_H in H. (A sparse cut is easy to detect.)

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 those assumptions till Section 6.

5.1 Watermarking Algorithm

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

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, compute flow graph 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 (approximately). 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, also of order n.
3. *Regularity step*: Here we add edges to and between G_c and W_c using a *random walk*: Assume we are at a node $v \in G_c$. Let d_{gg} and d_{gw} be the current 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. If u is the node chosen to visit, an edge is added from v to u. We work similarly if $v \in W_c$ to begin with. We repeat this till m edges have been added *across* G_c and W_c. Let H be the resultant graph. Output the program P' corresponding to H. (An edge in H may be implemented in P' as an "opaque" call or control flow from one block to another.)

Recovery of the watermark must not only find the true cut, it must also deal with distortions made by the adversary, and we discuss this step in Section 5.3

5.2 Discussion

The clustering step (2) must have a way to find different clusterings for different values of ω_1, so that the adversary does not have any knowledge about the clustering used.

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 $G_{n,p}$, [B] with n nodes and edge probability p. (See Section 6.) We now

examine the effect of the regularity step (3). The basic goal of this step is to add m edges across G_c and W_c, sampling without replacement. The random walk heuristic roughly achieves this, and the motivation for it is to yield a better merged graph in terms of local indistinguishability for real world input graphs that are not truly random. (When this is not a concern, one may add edges randomly). The analysis is simplified by assuming henceforth that each of the n^2 possible edges is present in the cut independently with probability m/n^2. It is more realistic to assume that the graphs are as generated by the random walk procedure or sparse, but we shall not do this here. Also, we shall drop the subscript 'c' from G_c and W_c, so G and W are now the subgraphs of H, each of size n, with m edges straddling across.

Call an equi-partition $\{G', W'\}$ of H an *m-partition* if the cut size $|E(G', W')| = m$. Call an equi-partition $\{G', W'\}$ *good* (for the adversary) if $|W' \cap W|/|W| \geq 1 - \epsilon$. We now claim that 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 appropriately chosen value of m. This suggests that the m-cut between G and W is hidden in an information theoretic sense. The claim can be written as follows:

Let G and W be two random graphs following the $G_{n,p}$ model, each of size n. Let edges be added at random between them such that each edge is added with probability p. Let H be the resulting graph. Let $m = n^2p$ and let X be the random variable counting the number of good m-partitions of H, and let Y be the random variable for total number of m-partitions. Then, $\frac{E(X)}{E(Y)}$ is exponentially small in n.

We now outline our analysis. A partition will always mean an equi-partition. Let X' be the number of good partitions. Note that X' is a fixed constant. Clearly, $X < X'$, and therefore, $\frac{E(X)}{E(Y)} < \frac{X'}{E(Y)}$ (since $E(Y) > 0$). Consider an arbitrary equi-partition of the nodes of H into G' and W'. Let $|G' - G| = x$. Therefore, $|W' - W| = x$. X' is the number of such partitions with $x \leq \epsilon n$. Clearly, $X' = \sum_{x=0}^{\epsilon n} \binom{n}{x}^2 < (\epsilon n+1)\binom{n}{\epsilon n}^2$ (for $\epsilon < \frac{1}{2}$) $< 2\epsilon n 2^{2nH(\epsilon)} = \epsilon n 2^{2nH(\epsilon)+1}$, where $H(\lambda) = -\lambda log\lambda - (1-\lambda)log(1-\lambda)$ is the binary entropy function and the last inequality uses,

$$\frac{2^{nH(\lambda)}}{\sqrt{8\pi n\lambda(1-\lambda)}} \leq \binom{n}{\lambda n} \leq \frac{2^{nH(\lambda)}}{\sqrt{2\pi n\lambda(1-\lambda)}} \tag{1}$$

Let $I(A, B, m)$ be an indicator variable indicating that (A,B) is an m-partition. Then, $Y = \sum_{S \subset G} \sum_{S' \subset W, |S'|=|S|} I(G - S \cup S', W - S' \cup S, m) = \sum Y_i$ (here, we denote the i^{th} term in the summation by Y_i). By linearity of expectation, we have $E(Y) = \sum E(Y_i) = \sum Pr(Y_i = 1) = \sum_{x=0}^{n} \binom{n}{x}^2 p_x(m)$, where $p_x(m)$ is the probability that $(G - S \cup S', W - S' \cup S)$ is an m-partition, for $|S| = |S'| = x$. Note that this probability depends only on x, and not on the particular choice of (S, S').

Let Z_x be the random variable that counts the number of edges in an equi-partition (A, B) with $|G \cap B| = |W \cap A| = x$. Since both G and W follow

the $G_{n,p}$ random graph model, and since each of the n^2 possible edges across them is present with probability p, we have H being a graph with $2n$ nodes and each edge present with probability p, independently of others. So, for any equi-partition (A, B), the cut size has a binomial distribution. Therefore, Z_x follows the binomial distribution. Letting $b(N, p, k)$ denote the probability that a binomial variable with parameters N, p assumes a value k, we have $p_x(m) = Pr[Z_x = m] = b(n^2, p, m) = b(n^2, p, n^2p) = \Omega((n^2p(1-p))^{-1/2})$. Since p is a constant, we get

$$p_x(m) = \Omega(\frac{1}{n})$$

Therefore, $E(Y) = \sum_{x=0}^{n} \binom{n}{x}^2 p_x(m) = \Omega(\frac{1}{n}\sum_{x=0}^{n}\binom{n}{x}^2) = \Omega(\frac{\binom{2n}{n}}{n}) = \Omega(\frac{2^{2n}}{n^{1.5}})$ (using Inequality 1). And we finally have the desired result,

$$\frac{X'}{E(Y)} = O(\frac{\epsilon n 2^{2nH(\epsilon)+1}}{2^{2n}/n^{1.5}})$$

Note that we need to assume that the adversary knows m, since he can try each possible value. Also, we cannot expect the above claim to be true for all values of m. For example, if m is in the neighborhood of 0 or n^2, then good empirical attacks that find very small or very large cuts will do the job. But for values of m close to n^2p the analysis above indicates that the information theoretic hiding will succeed.

5.3 Recovery of Watermark

To recover the watermark, the extractor first needs to identify (most of) the nodes of W. To this end, 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. Recall that each node in the program flow graph is a sequence of instructions, which allows room to embed the flagging information. By applying a majority logic over a node and its neighborhoods, we can increase the resistance to tampering.

The extractor, having detected the nodes of W, then samples several small subsets w from W, using the secret key as the random seed. The sampling is done with probability proportional to the number of edges in the subset, so that relatively dense subsets are obtained. Then, a robust function is computed on each w, producing the watermark. Since the adversary cannot distinguish the nodes of W from G to any significant extent, we may assume that the distortive changes made by it to the program are at random places, and very few in number. Thus we need graph properties that are resistant to minor changes, and one could use all of them simultaneously. Construction of such "Graph hash" functions is an interesting problem by itself and our future research will address this problem in more detail. For now we present some elementary methods. If A is the adjacency matrix of w, $k < d/2$ where d is its diameter, one expects

then A^k to be robust to a small percentage of changes in the graph and it is observed to be so empirically. Using the adjacency matrix requires knowing a robust ordering (labelling) of the vertices, and we may use *vertex invariants* to solve the problem. A vertex invariant is a property of a vertex that does not change under an automorphism. The *degree* is such a property. k-neighborhood size is another. The idea is to compute labels of vertices using such invariants, and then computing the adjacency matrix powers. This is combined with a suitable error correcting code that filters out the small changes in a string extracted from adjacency matrix powers. Also we may use functions based on cuts and path lengths. While detecting the watermark, one may wish to use some local strategy that does not need using the whole graph. This is indeed possible when there are few changes. Finally, 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.

6 Experiments

We begin with simple experimental results that examine the randomness assumptions made about the graphs extracted in the clustering step. Let n = number of nodes in G_c and m = number of edges. We test the hypothesis that G_c is from the probability space $G_{n,p}$, with $p = \frac{m}{\binom{n}{2}}$, for several G_c. Fix some c. We pick at random many hyperedges of size c and count the number of them that are present and compare with the expected values. Additionally one takes many cuts and verifies that their values are as expected. 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 $Pr[\text{number of edges in a hyperedge is } c] = \binom{K}{c} p^c (1-p)^{K-c} = p_c$ (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	Observed	Expected	Observed	Expected	Observed	Expected
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

We can similary apply other randomness tests. These as well as actual empirical runs of various separation algorithms to locate a hidden cut can serve as a check that we are not introducing simple weaknesses.

Now we briefly address robust functions on graphs. A be the adjacency matrix of W_c and let d_W be its diameter. We test the small powers of adjacency matrix yield unique signatures of the graph, but are affected only in a few places when the graph is changed in small number of places. We used $A = 5000 \times 5000$ matrix, diameter $d = 8$ and $A^{d/2}$ was found to have about 4% of its entries changed.

7 Acknowledgements

We thank Mariusz Jakubowski(MS Research) and Jayram Thatacher (IBM Research) for their invaluable help early in this project.

References

[AP] Ross J. Anderson and Fabien A. P. Petitcolas. On the limits of Steganography. *IEEE J-SAC*, 16(4), May 1998. 160, 167

[A] U. Assmann. *OPTIMIX optimizer generator*. http://i44www.info.uni-karlsruhe.de/ assmann/optimix.html 160

[BCS] Council for IBM Corporation *Software birthmarks*. Talk to BCS Technology of Software Protection Special Interest Interest Group. Reported in [AP]. 160

[B] Bella Bollobas. 1985. *Random Graphs*. Academic Press. 163

[CT] C. Collberg and C. Thomborson. Software Watermarking: Models and Dynamic Embeddings. *Principles of Programming Languages 1999, POPL'99*. 160

[Davidson] R. L. Davidson and N. Myhrvold *Method and system for generating and auditing a signature for a computer program*. US Patent 5559884, September 1996. Assignee: Microsoft Corporation 160

[Diestel] Reinhard Diestel. 2000. *Graph Theory* Springer-Verlag, second edition. 159

[Feller] William Feller. 1993. *An Introduction To Probability Theory And Its Applications*, volume 1. Wiley Easter Limited, third edition.

[FK] A. Frieze and R. Kannan. The Regularity lemma and approximation schemes for dense problems. *37th Annual Symposium on Foundations of Computer Science*, 2-11, October 1996. IEEE. 159

[GM] S. Guattery and G. L. Miller. On the Performance of Spectral Graph Partitioning Methods. *Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, 233-242, ACM-SIAM, 1995. 162

[GSV99] N. Garg, H. Saran and V. V. Vazirani. Finding Separator Cuts in Planar Graphs within Twice the Optimal. *SIAM J. Computing*, vol 29, No. 1, 159-179 (1999). 162

[Holl] G. Holloway. *The Data Flow Analysis Library of Machine SUIF.* http://www.eecs.harvard.edu/hube/software/v130/dfa.html 160

[JK] Norman L. Johnson and Samuel Kotz. Discrete Distributions. *Wiley Series in Probability and Statistics*, 1999.

[KK] G. Karypis and V. Kumar. Multilevel k-way Hypergraph Partitioning. *DAC 1999*, 343-348. 162

[LLR95] N. Linial, E. London, and Y. Rabinovich. The geometry of graphs and some of its algorithmic applications. *Combinatorica*, 15:215–245, 1995. 162

[LR88] T. Leighton and S. Rao. An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. In *Proc. 29th Ann. IEEE Symp. on Foundations of Comput. Sci.*, pages 422–431, 1988. 162

[LT] R. J. Lipton and R. E. Tarjan. A Separator Theorem for Planar Graphs. *SIAM J. Appl. Math.*, 36 (1979), 177-189. 162

[So] V Communications. *Sourcer: Advanced Commenting Disassembler.* http://www.v-com.com/products/sourcer.html. 160

[ST96] D. A. Spielman and S. Teng. Spectral Partitioning works: Planar graphs and finite element meshes. Technical Report CSD-96-989, U. C. Berkley, February 1996. extended abstract in Proc. 37. IEEE Conf. Foundations of Comp. Sci., 1996. 162

[Vulcan] Amitabha Srivastava *Vulcan Tech Report* Technical Report Vol TR99, No 76, Microsoft Research Technical Reports, 1999. 160