Data Structures for Limited Oblivious Execution of Programs while Preserving Locality of Reference

Avinash V Varadarajan
Computer Science Division
UC Berkeley
avinash@cs.berkeley.edu

Ramarathnam Venkatesan
Microsoft Research
Redmond, WA
Bangalore, India
venkie@microsoft.com

C Pandu Rangan
Department of Computer Science
Indian Institute of Technology Madras
rangan@cse.iitm.ac.in

ABSTRACT

We introduce a data structure for program execution under a limited oblivious execution model. For fully oblivious execution along the lines of Goldreich and Ostrovsky [2], one transforms a given program into a one that has totally random looking execution, based on some cryptographic assumptions and the existence of secure hardware. Totally random memory access patterns do not respect the locality of reference in programs to which the programs generally owe their efficiency. We propose a model that limits the obliviousness so as to enable efficient execution of the program; here the adversary marks a variable and tries to produce a list of candidate locations where it may be stored in after T-steps of execution. We propose a randomized algorithm based on splay trees, and prove a lower bound on such lists.

Categories and Subject Descriptors

E.2 [**Data**]: Data Structures; F.m [**Theory of Computation**]: Miscellaneous

General Terms

Algorithms, Security, Performance, Theory

Keywords

Oblivious Execution, Software Protection, Secure Data Structures, Oblivious Data Structures, Secure Hardware, DRM

1. INTRODUCTION

Software Protection is one of the most challenging problems, both to define and to solve in a manner that is practically attractive. In one of its forms, which is called the obfuscation problem, impossibility results in the polynomial time adversary model have been shown in [4]. The other form, called oblivious simulation problem [2], is studied in this paper; here purely software solutions are impossible in the sense that software (no matter how encrypted) is just a binary sequence which a pirate can copy bit by bit and run on his own machine. Hence to protect software we need some support from hardware that can hold secrets, decrypt and execute, encrypt, add authentication tags to code and data touched, and put it back in main memory. This is needed since one needs both actual code and data to be hidden. If one uses simple encryption and authentication of the code, the attacker can gain information about the code by observing its memory access patterns. Given such a hardware, thus many issues simplify and one is left with the problem of making the memory accesses of data appear random, where it becomes a data structure problem: efficiently managing data in memory and accessing it with a given set of performance requirements, to which we now add some security requirements as well.

Goldreich and Ostrovksy [2] gave the first simulation with polynomial factor overhead. In this model, given a program P one constructs a new program $P^{'}$ such that P and $P^{'}$ are essentially equivalent in terms computations and input and output, but P' has additional security properties. Under suitable security assumptions, they show that a polynomial time adversary can not distinguish between the memory access pattern of $P^{'}$ and the memory access pattern of a dummy program R which just accesses random locations in the memory. They prove that if the number of data blocks in the programs memory is M and the number of memory accesses is T, $P^{'}$ needs to take $\Omega(T \log M)$ steps, and their solution actually runs for $O(T(\log^3 T))$ steps . We face two problems that affect the performance significantly.

1. To meet the basic requirement of having to simulate a sequence of random memory accesses, P' has to move data around in the memory heavily which is costly.

2. Usually, a program P has a lot of its efficiency dependent on the locality of reference (spatial or temporal). This is a widely used and somewhat a vague concept, which means that data and code which are visited in recent past are likely to be visited again. For P the entropy in an access sequence (formally defined later), which is a lower bound on the cost per access, can be a small constant; however, random patterns of P' will have much higher non-constant entropy. Our formal parameter for quantifying the locality of reference is the entropy of an access sequence. For typical programs this quantity is small.

DRM'07, October 29, 2007, Alexandria, Virginia, USA.

In any such transformation, one is bound to have overheads. Our goal is to suggest a parameter that is sensitive to performance by preserving the locality of reference, and lower the overheads as much as possible. However finding such a parameter is a non-trivial task. A few words of caution and caveats are in order. We would need to have complex enough programs, where the number n of variables are large enough. Given that some programs may have too few variables, at times it may be necessary to increase the number by introducing some *shadow variables* (defined later). Ideally one would want a model where the adversary does not know any information other than its run time or program's efficiency i.e., the entropy $H(S_P)$ of its access sequence to the memory; but even this may be an overkill in many applications. From efficiency considerations, if a program P accesses memory using a sequence whose entropy is H, we want the access sequence of P' to have entropy close to H. Our model is a step in that direction.

Now we return to the question of a proper model and its motivation. Consider a simple data access sequence $a_1, ...a_m$. It may be helpful to view this as an input to some compression algorithm, as it clarifies the entropy considerations. Assume that this sequence is generated by a stochastic source with q symbols $\{\sigma_1, ...\sigma_q\}$ each with probability $p_i, 1 \leq i \leq q$ and that the sequence is identically distributed and independent. We think of storing the sequence in a linked list. At each time j we look for the symbol $a_j = \sigma_i$ in the linked list. If it is already present, we output its position in the list and we move it to the front. Otherwise, we add it to the front of the list and output the item. (This is the move to front compression algorithm [6]). Let us assume that σ_i reappears at time $j + \tau$. By standard calculations, expected value of *inter-arrival time* τ is $1/p_i$. (Kac's theorem extends this phenomenon to a vastly larger class, namely stationary ergodic sources). Clearly, it will take $\leq \log \tau + 1$ bits to encode the integer τ, and thus the expected length of the output sequence $\sum_i p_i \log \frac{1}{p_i}$, this is the entropy of the source, and is thus optimal. This example strongly suggests that the inter-arrival times will be an effective parameter characterizing the achievable efficiency here. There are two problems in this example:

1. **Stochastic vs. Arbitrary inputs:** The data (access) sequence is stochastic, which may be too simplistic even for standard data structure issues without any security properties imposed. We need to be able to analyze *arbitrary* access sequences in P. (Hereafter we refer to a memory access sequence by P as a *program access sequence* and denote by S_P).

2. **Tampering vs Non-Tampering Adversary:** The adversary must be able to adaptively insert new memory access requests during its access, after some steps of P' (and thus some information on the coin flips of the transformation $P \longmapsto P'$ and that of P' may become available); we need to consider this since some programs may protect only parts of a program or an attacker may have partially successfully attacked some portions, where we desire graceful degradation.

To address the first issue, we use splay trees, which offer a very efficient and indeed a practical alternative to the linked list above. They are self-adjusting and provably (nearly) optimal in terms of amortized cost for *arbitrary sequences*. We could potentially use other data structures, but one should expect some extra complications. From a practical view point the amortized overhead other structures may be higher. In addition, formal amortized analysis for arbitrary access sequences becomes an issue.

Taking a cue from this, our weakening of the model is based on *inter-arrival time (*and related quantities) but it applies to arbitrary sequences S_P. Our transformation to P' will add N new variables and will result in $S_{P'}$ that will contain S_P but many other random accesses. In S_P let a variable be accessed at the j^{th} step, and then next at time $\tau + j$, and which will happen in the same order in $S_{P'}$ also. Roughly, we want the adversary who is watching the execution of P' not to be able to predict the location of the variable correctly during steps $i < j + \tau$: in fact, we want a suitable lower bound $L = L(\tau, M, n)$, on the list of possible locations given a data that has been accessed by a program. Here M is the main memory size and n is the number of variables. Such a lower bound applies to each variable. In fact, one can amplify the uncertainty by splitting a variable into k pieces and storing them (at the expense of performance) ; for complex enough programs we suggest that this will suffice. For programs that need more uncertainty, one may boost N, and the randomization parameters prescribed below, which in the limit will approach that of [2], thus giving one a performance tradeoff.

2. RELATED WORK

The problem of oblivious simulation (with polynomial overhead) of a random-access machine(RAM) was first addressed in [2]. They show a method for on-line simulation of an arbitrary RAM input by a probabilistic oblivious RAM with a poly-logarithmic slowdown in the running time. This method requires a trusted hardware to house and execute the program. Furthermore, they provide a lower bound for the slowdown factor, which makes it prohibitive to use in practice. In a practical vein, this is studied in [9] and [10]. In [9], they study address-space randomization(introducing artificial diversity by randomizing the memory location of certain system components) as a technique to prevent buffer overflow attacks. In [10], they propose a hardware-assisted method to hide memory access patterns, and control graphs of programs. Our contribution is a novel model for "measuring" the resource requirements of an attack, which we believe is both general and permits practical solutions. We introduce new types of adversaries, with appropriate motivations. We use splay trees to construct a efficient solution in the above model for the adversaries that we define.

The paper is structured as follows. In Section 2 we formally describe our model. In Section 3 we present our algorithm for limited obliviousness, along with its formal analysis. In Section 4, we present results of simulations of our model, to demonstrate its practical value, and in Section 5, we conclude by describing some applications of our ideas to problems in DRM.

3. THE MODEL

It is clear that it is not possible to hide the access pattern of a program by purely software solutions. Hence we introduce an additional piece of hardware in a *Random Access Machine* RAM called the *Oblivious Machine* OM which func-

tions between the *CPU* denoted as CPU and the *Memory* denoted as MEM.

A program consists of variables identified by unique name; for our purposes, as in [2], the values in these variables will not play a significant role; this can be justified by using suitable cryptographic primitives, namely, encryption and authentication. When the program runs in the CPU, these variables are accessed in a specific order in order to copy their values to a register or in order to update their values. These variables are stored in the MEM. So, we can view the CPU sending a sequence of variable names to the OM. This sequence of variable names is called the **program access sequence** S_P of the program P. Every time the OM receives a variable name, it accesses the location in the MEM where the variable is stored. Hereafter, when we refer to a variable v_i we mean the variable with variable name v_i. We proceed to give more formal definitions.

DEFINITION 1 (RAM). *A random access machine* RAM *consists of a* CPU, MEM, OM *interacting with each other as described below.*

DEFINITION 2 (CPU). CPU *is the processor of the* RAM *that runs a program having the variables named* $v_1, v_2 \ldots v_n$, *that generates a sequence of* m *variable names* $\in \{v_1, v_2, \ldots v_n\}$ *called the* access sequence. *Note that the variable names form a total order, namely* $v_1 < v_2 < \ldots < v_n$.

DEFINITION 3 (MEMORY). MEM *is the memory of the* RAM *having cells of size* s *bits each. Each cell has a unique address. The contents of any cell can be directly accessed using its address. Each cell stores one unique variable, possibly along with other control information. The following two operations are possible on* MEM

1. **read**(Address a) - *returns the contents of the cell with address* a.
2. **write**(Address a, Cell-Value v) - *Copies the* s *bit value in* v *into the cell with address* a.

Before the CPU *starts generating the access sequence, the* MEM *is initialized with all the* n *variables* $v_1, v_2, \ldots v_n$, *optionally along with some shadow variables.*

DEFINITION 4 (OBLIVIOUS MACHINE). OM *is a "black box" processor with internal memory cells. Whenever the* OM *receives a variable name* v_i *in the access sequence, it must issue a* read(addr) *call to* MEM *where* addr *is the address of the cell in* MEM *containing* v_i, *possibly along with read calls to other cells.*

DEFINITION 5 (SHADOW VARIABLES). *Before the* CPU *starts generating the access sequence,* k *additional variables for each variable* v_i *named* $v_{i1}, \ldots v_{ik}$ *are also (optionally) added to* MEM. *These are called shadow variables. These variables are accessed internally by* OM *in between accessing the actual variables* $v_1, v_2, \ldots v_n$.

3.1 Adversary

An adversary can view all communication between OM and MEM and thus he has full knowledge about the memory locations to which the OM performs read/write operation. This model becomes more interesting in view of recent cache attacks [11] and other side channel attacks using which an attacker can get some information about the memory locations accessed by a program. The idea is that, a program that remains oblivious even after revealing its complete memory transcript, must be secure against such side channel attacks.

The adversary cannot observe any computation performed within the OM nor the contents of its internal memory cells. Also, the adversary cannot know the clear text of the contents of any cell in MEM, as they are encrypted with the decryption key stored in the internal memory cells of OM. We define different types of adversaries, based on how much he controls the *access sequence* S_P.

DEFINITION 6 ($Adv1$). *This adversary generates* S_P *and sends it to* CPU *before the execution begins.*

DEFINITION 7 ($Adv2_L$). *This adversary generates the access sequence* S_P *and sends it to* CPU *before execution begins. Further, during the execution he adaptively chooses many variables* $u_1, u_2, ..., u_L$ *at execution instants* $t_1, ...t_L$, *and asks the oblivious machine to reveal their locations in the memory.*

Note that this is a natural model reflecting that the adversary may have gained partial information during execution; alternately the structure of the original program variables may be such that once a few variables are successfully traced, rest of them may be easy to unravel. One model would be to allow the adversary to insert access sequences during the execution of P' (as if originally they were in P), but the transformation of P into P' inserts its own variables and accesses, and with out revealing the secret keys that were involved in specifying how S_P is embedded in S'_P, allowing such insertions either does not increase the power of the adversary significantly, or it would be hard to formalize. For example allowing the adversary to access the memory at some location of his choice does not increase his power.

We will later formulate a (T, r)-adversary. The adversary marks a variable and then tries to make a list of candidate locations where that variable might be after T moves. The above adaptive queries may help the adversary win the game; in the event he asks for the variable he marked to be revealed, we declare he loses. Thus without loss of generality, in that game, the adversary does not query a marked variable.

Let $S_P = [a_1, a_2 \ldots a_m]$ be an access sequence, where $a_i \in \{v_1, v_2 \ldots v_n\}$. Let $q(i)$ be the number of times variable v_i occurs in the access sequence A. The *entropy of the access sequence* is defined as

$$H(S_P) = \sum_{i=1}^{n} q(i) log(\frac{m}{q(i)})$$

We use splay trees from [1]. Now we briefly survey the properties of splay trees relevant for us. They are self-adjusting binary search trees with an operation called *splaying*, which moves any item that is accessed to the root of the tree, and re-arranges the tree so that it can be still searched as binary search tree. Certain randomized variations of it have also been studied in [7] [8]. In this paper we stick to the original deterministic version in [1]. It has many optimality properties, and we quote the next two theorems.

THEOREM 1 (STATIC OPTIMALITY). *For any item i, let* q_i *be the access frequency of item i, that is, the total number of times i is accessed. If every item is accessed at least once, then the total access time is* $H(S_P)$.

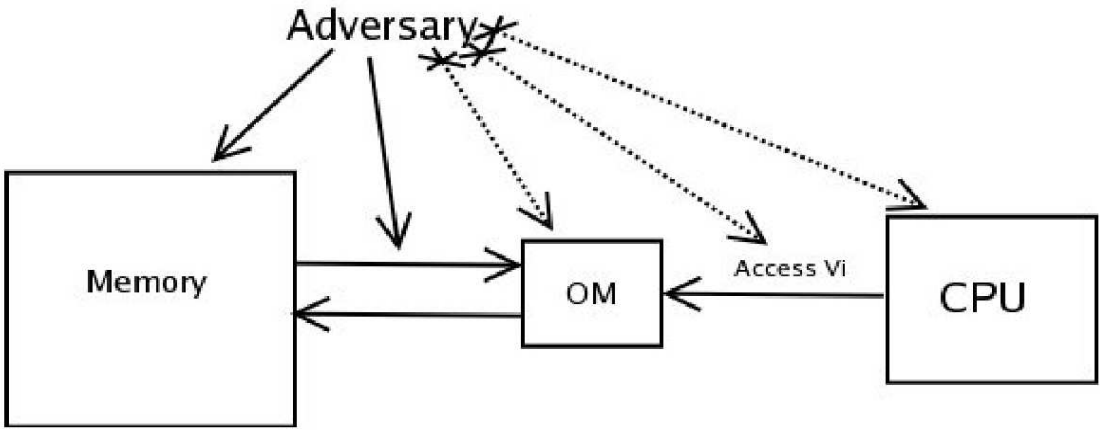

Figure 1: Block diagram of the model

By a standard theorem in information theory [12] this is optimal (within constant factor) since it achieves the access sequence entropy. What is important is that the analysis is for *any* access sequence, not merely those generated randomly & independently or some other stochastic manner. The access times and depth of the nodes during an access are closely related to the entropy of the sequence. In an access sequence, let $t(j)$ denote the total number of items accessed since the last access of the item being accessed in the j^{th} access. Then we have the following:

THEOREM 2 (WORKING SET). *The total access time is*

$$O(m + n \log n + \sum_{i=1}^{m} \log(t(i) + 1)$$

What this means is that the cost of accessing the item at i^{th} step is $\log(t(i) + 1)$, and the most recently accessed items (those in the working set) are the easiest to access. The term $(t(i) + 1)$ is can be viewed as an approximation to the inter arrival time. Now we define our notion of limited obliviousness.

DEFINITION 8. *((T, r)-oblivious variable). Just before the i^{th} variable in the access sequence is accessed, let the adversary label the variable stored in a particular memory cell as l [1], and let that memory cell be read during the i^{th} access. If after the $(i + T)^{th}$ access, if the adversary can output a list of r memory locations such that the current location of l is present in the list with very high probability , then we say that the adversary (T, r) breaks the variable. A variable is (T, r)-oblivious if an adversary cannot (T, r) break it. i.e., there is a lower bound $> r$ on the size of the list he has to output*

We do not insist that the variable to be uniformly distributed in those locations, as we do not assume any statistical properties of the access sequence. Its distribution may have an entropy proportional to that of access sequence. We can get an alternate definition for obliviousness if we replace r in the above definition by this entropy. Let $\mathbf{T} = (T_1, ..T_n)$, and $R = (r_1, r_2, ..., r_n)$. A program is called (T, R)-oblivious if each variable i is (T_i, r_i)-oblivious. We say a program is f-oblivious if each of its variable is i is (t, r)-oblivious where $t \leq T$, $r \geq f(t, n, M)$.

[1]Note that this need not be the actual name of the variable stored in that cell.

4. ALGORITHM SPLAY-RAND(K)

The variables $v_1, v_2 \ldots v_n$ are stored in MEM in the form of a **splay tree**. Each memory cell stores one variable, and pointers [2] (addresses) to the cells which are its left and right children in the tree. For analysis of security and performance, the values of the variables play no role. We therefore consider only the variable name to be stored in each cell. We can think of each **cell** in MEM as a **node** in the tree. The pointer to the **root** cell is stored in one of the internal memory cells of the OM. Therefore the size of each cell s in the memory is *sizeof(variable name) + 2(sizeof(pointer))*. When OM receives a variable $l \in \{v_1, v_2 \ldots v_n\}$ to be accessed, it does the following.

1. Starting from the root node, read all the nodes in the path from the root node to the node containing l, into the internal memory cells of OM.

2. Perform **splay** at the node containing l, by adjusting the child pointers in all the nodes in the path, and also the root pointer

3. Randomly permute the cell addresses of these nodes before rewriting them back, also adjust the root pointer, to point to the new address of the root.

For every k variables accessed by the CPU, randomly choose one of the variable $vr \in \{v_1, v_2 \ldots v_n\}$ and perform the same operations enumerated above.

4.1 Analysis

We now analyze the efficiency of the splay tree operations.

[2]This could be null.

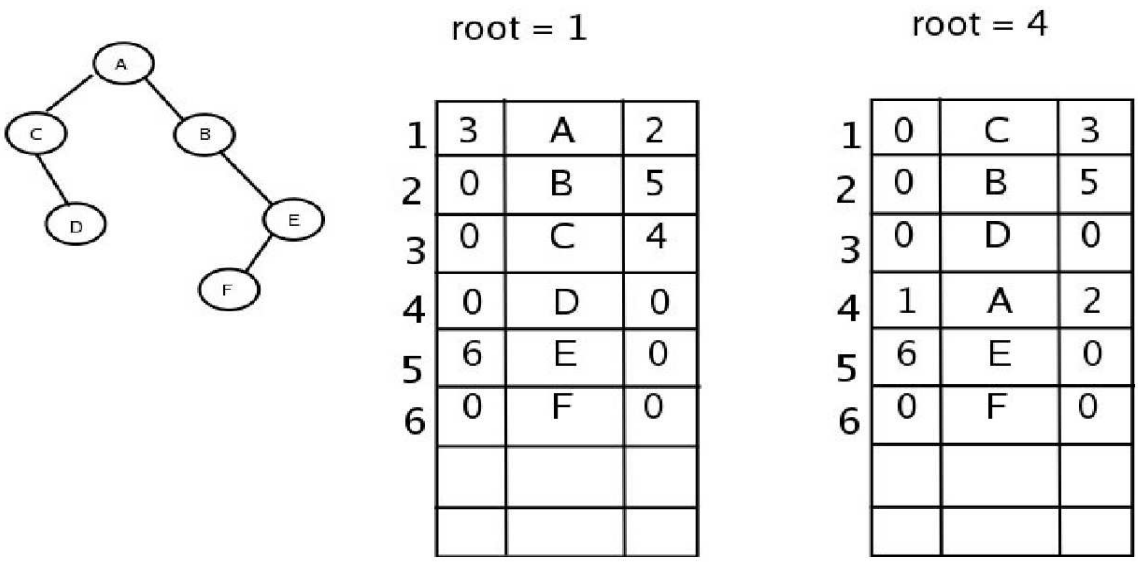

Figure 2: Explaining what permuting a path means in the memory

4.1.1 *Performance :*

Let the CPU y make T accesses. Let $m = T(1 + 1/k)$. Let q_i be the number of times the variable v_i occurs in the T accesses made by the *Adversary*. Let x_i be the number of times the v_i variable occurs in the T/k random accesses. Note that x_i are distributed according to binomial distribution each with bias $\frac{1}{n}$. From 1, the expected running time **Splay-Rand(k)** is

$$E(\sum_{i=1}^{n}(q_i + x_i)\log(\frac{m}{q_i + x_i}))$$

$$= \sum_{i=1}^{n} E((q_i + x_i)\log(\frac{m}{q_i + x_i}))\ )$$

$$\leq \sum_{i=1}^{n}(q_i + T/nk)\log(\frac{m}{q_i})$$

This can lead to an overhead of $O(\log n)$ in the worst case.

4.1.2 *Security :*

We analyze security for the adversary $Adv1$. In order to prove that a variable is (T, r)-oblivious, we need to get a lower bound on the number of cells which must appear in the list produced by the adversary for any variable v. We view the problem as following. In the splay tree, let the node containing v be marked "red" before it is accessed. Let every other node be "white". In view of the adversary, the red nodes are where the variable v could possibly be, and white nodes are where it cannot be. When v is accessed for the first time, all the nodes in the path are permuted, so every node in the path becomes red after the access. We analyze how the number of red nodes in the tree increase. We additionally note that the red nodes in the tree do not decrease.

Claim (By Induction): The root is always red.

After the first time v is accessed, we can inductively see that the root of the tree is always red. This is because, after every access, the accessed node becomes the new root, and it is red.

We first obtain a lower bound on the expected increase in the number of red nodes when a random node in the tree is splayed. Whenever a node is splayed, all the white nodes in the path from the root to that node become red.

Let S denote a splay tree of n nodes. Let $\gamma(S)$ be a color assignment to the splay tree S, which results in a colored splay tree $\tilde{S}$ with some of the nodes colored red, and the remaining nodes colored white. Let $\rho(\tilde{S})$ be the operation of splaying a random node in the tree $\tilde{S}$, and permuting the nodes in the access path as described in **Splay-Rand**. Note that after the operation ρ is performed, some of the white nodes in the tree $\tilde{S}$ become red nodes. Let $R(\tilde{S})$ be the number of red nodes in the colored splay tree $\tilde{S}$. We want to find lower bound on $\mathcal{E}(S, r, \gamma)$ which is defined as the expected value $E(R(\rho(\gamma(S)))|R(\gamma(S)) = r) - r$. Given a color assignment γ for a splay tree S, let $W_{\gamma,S}(v_i)$ be the number of white nodes in the path from node v_i to the root node in S. Clearly $\mathcal{E}(S, r, \gamma) = (1/n)\sum_{i=1}^{n} W_{\gamma,S}(v_i)$. Consider the following "no hole" condition for any color assignment γ on S

C1 : In the tree there exists no path p from the root to any node, with the nodes from the root being $v_{p1}, v_{p2}, \ldots v_{pk}$ where v_{pk} is colored red and $\exists i < k$ and v_{pi} is colored white.

The condition means that in S colored with γ, the red nodes form a connected subtree along with the root of S. If $R(\gamma(S)) = r$, then there will be $r + 1$ subtrees of white nodes "hanging" from this subtree as "external nodes".

LEMMA 1. *If a color assignment γ_1 for a tree S does not satisfy **C1**, then $\mathcal{E}(S, r, \gamma_1) \geq \mathcal{E}(S, r, \gamma^*)$. where γ^* satisfies **C1**.*

PROOF. If some coloring scheme γ_1 exists with such a path p, then by swapping the colors of v_{pk} and v_{pi}, we

get a new coloring scheme γ_2 where for every node v_i in S $W_{\gamma_1,S}(v) \leq W_{\gamma_2,S}$ and so $\mathcal{E}(S,r,\gamma_2) \leq \mathcal{E}(S,r,\gamma 2)$. By performing similar transformations we finally get a coloring scheme γ^* which satisfies **C1**. □

Consider the following coloring scheme with r red nodes for a tree S
A : Let $S(v_i)$ denote the number of white nodes in the subtree rooted at v_i. Color the node with maximum value of S (breaking ties arbitrarily) red, and repeat the same until r nodes are colored red.

LEMMA 2. *Let γ^* be a color assignment obtained from **A** . It will satisfy **C1**. Let γ_1 be any color assignment, then $\mathcal{E}(S,r,\gamma_1) \geq \mathcal{E}(S,r,\gamma^*)$*

PROOF. If γ_1 does not satisfy **C1**, then the proof follows from previous lemma. So, let us assume that γ_1 satisfies **C1** and is different from γ^*. There exists pairs of nodes u, v u is colored red in γ_1 and white in γ^* and vice versa for v. Since both γ_1 and γ^* satisfy **C1** u and v are roots of subtrees with every node (with the possible exception of the root node) white. From the description of γ^*, $W_{\gamma^*,S}(u) \geq W_{\gamma_1,S}(v)$, hence if we swap the colors of u, v in γ_1 we get a new scheme γ_2 with $\mathcal{E}(S,r,\gamma_2) \leq \mathcal{E}(S,r,\gamma_2)$, continuing we will get the scheme γ^*. □

Therefore, minimum value of $\mathcal{E}(S,r,\gamma)$ for a color assignment γ occurs when γ is obtained from **A**. Now we characterize the structure of a tree S of n nodes, which has a minimum value of $\mathcal{E}(S,r,\gamma)$ among all possible tree structures on n nodes, where γ is obtained from running **A** on S. We know that in S, after the color assignment, the r red nodes form a subtree containing the root of S. Therefore, the tree structure must have all the white nodes form $r+1$ balanced subtrees of the same number of nodes. This follows from the fact that $\mathcal{E}(S,r,\gamma) = (1/n)\sum_{i=1}^{n} W_{\gamma,S}(v_i)$ where $v_i's$ are the nodes in the tree. If we consider one particular subtree $S1$ with k nodes $v_{s1}, v_{s2} \ldots v_{sk}$, then the summation $(1/n)\sum_{i=1}^{k} W_{\gamma,S}(v_{si})$ is minimized if $S1$ is balanced. One such structure for S is when S is balanced i.e. a tree in which all the levels with the possible exception of the last level are completely filled with nodes. If we estimate the value of $\mathcal{E}(S,r,\gamma)$ where S is a balanced tree on n nodes and γ is obtained by running **A** on T, then it forms a lower bound for all trees on n nodes and all possible color assignments on them.

LEMMA 3. *$\mathcal{E}(S,r,\gamma)$ for a balanced tree on n nodes is $O(\log(n/r))$*

PROOF. We use the expression $\mathcal{E}(S,r,\gamma) = (1/n)\sum_{i=1}^{n} W_{\gamma,S}(v_i)$ to evaluate the required lower bound. To compute the lower bound, value of $W_{\gamma,S}(v_i)$ for all nodes v_i up to level $\lceil \log r \rceil$ to be at least 0, and $1, 2, \ldots$ for each increasing level, up to at least level $\lceil \log n \rceil$. Hence,

$$\begin{aligned}
\mathcal{E} &\geq 2^{(\lceil \log r \rceil + 1)} \times 1 + 2^{(\lceil \log r \rceil + 2)} \times 2 + \ldots + \\
&\quad 2^{(\lfloor \log(n+1) \rfloor - 1)}(\lfloor \log(n+1) \rfloor - 1 - \lceil \log r \rceil) \\
&= 2^{(\lceil \log r \rceil + 1)}(1 + 2 \times 2 + \ldots + \\
&\quad 2^{(\lfloor \log(n+1) \rfloor - 2 - \lceil \log r \rceil)} \times (\lfloor \log(n+1) \rfloor - 1 - \lceil \log r \rceil)) \\
&= 2^{(\lceil \log r \rceil + 1)}(1 + (\lfloor \log(n+1) \rfloor - 2 - \lceil \log r \rceil) \times \\
&\quad 2^{(\lfloor \log(n+1) \rfloor - 1 - \lceil \log r \rceil)}
\end{aligned}$$

Thus we have,

$$\begin{aligned}
\mathcal{E} &\geq \frac{2^{(\lceil \log r \rceil + 1)} + (\lfloor \log(n+1) \rfloor - 2 - \lceil \log r \rceil) 2^{\lfloor \log(n+1) \rfloor}}{n} \\
&= O(\log \frac{n}{r})
\end{aligned}$$

□

LEMMA 4. *In Algorithm **Splay-Rand(k)** any variable is $(n/2, n/2k - log(n/k))$ oblivious for the adversary $Adv1$.*

PROOF. Let Δ_i be the increase in the number of red nodes after a random splay. Let p_i be the probability that $\Delta_i > 0$. If the total number of red nodes $< \frac{n}{2}$, then $p_i > \frac{1}{2}$. If the length of the access sequence is m, then the number of random splays is $\ell = \frac{m}{k}$. We need to find the lower bound of $S = \sum_{i=1}^{\ell} \Delta_i$. But lower bound of S is greater than the lower bound of another random variable T, where $T = \sum_{i=1}^{\ell} x_i$, where x_i are independent bernoulli random variables with $p = \frac{1}{2}$. Therefore by Chernoff bound $prob(T < \frac{\ell}{2} - log\ell) < \frac{1}{\ell}$. Hence the result.

□

We can amplify the uncertainty of a particular variable v_i by introducing l shadow variables $v_{i1}, v_{i2} \ldots v_{il}$, whose access probability is similar to the variable v_i. This will increase the overhead by a factor of l. These shadow variables are particularly useful when there are only very few variables in the program.

To counter $Adv2_L$, the same algorithm is used. However, in this case, suppose $Adv2$ asks OM to reveal some L locations in the memory. If the variable v_i the adversary is tracking is one of the L locations, as the variable is declared to be broken by the adversary, and he restarts the game of tracking another variable. If the variable that the adversary is tracking does not lie in any of those j locations, then the number of red nodes for the variable v_i is just going to reduce by j.

5. EXPERIMENTAL RESULTS

We simulated the data structure for $n = 1023$, and compute the number of red nodes as a function of the number of accesses made, assuming that the variable being tracked is the first variable in the access sequence. The average number of red nodes(in 1000 trials) as a function of number of accesses is shown in Figure 5.These simulations give and idea of how effective the random accesses made every k steps in our algorithm are in causing uncertainty in the actual location of the variable being tracked. We observe, than in practice the number of red nodes after T random accesses is well above T, and so in our algorithm the number of red nodes after $n/2$ accesses is well above $n/2k$.

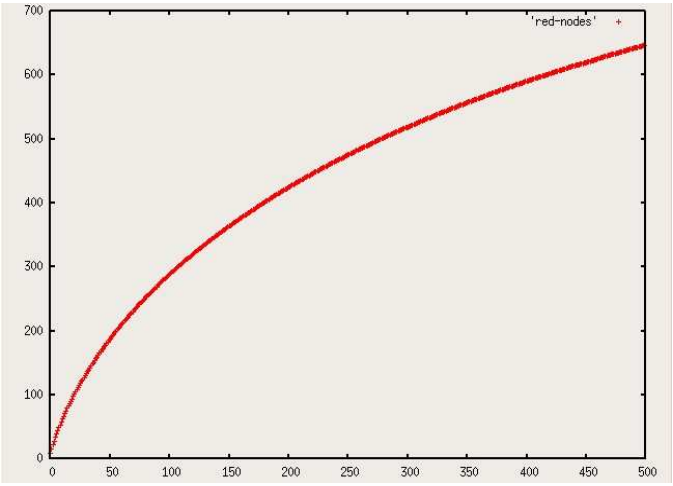

Figure 3: Simulation Results

6. APPLICATIONS

Our techniques can be use in DRM as well general system security issues.

First we consider the task of embedding an integer variable x in software so that it remains oblivious during execution in the sense defined earlier. We have relied on hardware assistance which in practice in most cases is unavailable. Hence we ask what can be done without the hardware support. Any solution can be used to hide secrets in software. In its full generality, against polynomial time adversaries, without secure hardware assistance this task is impossible. See [4]. Our observation here can be useful when there is a virtual machine implementation that simulates the hardware feature or some operating system support is available. In the former case, one may use a randomized virtual machine, that is individualized to each client and this may be appropriate in some applications (See [5]). In such a case our approach can be viewed as a modular development technique algorithmically and software-wise. Interestingly, even the simulator's data and code can be hosted on a limited oblivious data structure, not unlike bootstrapping a compiler.

Since our methods introduce uncertainty only in a limited sense, it is desirable to amplify by introducing many random variables x_i whose sum is x. Then we use each x_i as a variable in the oblivious data structure, where each of them is initially stored at random locations. After T steps of the algorithm each of the variables x_i will be potentially stored in one of many locations, thus x itself will be much harder to track. It is an open problem to estimate the entropy of x in such a case, but our empirical measurements indicate this increases as the run proceeds.

A first application would be whiteboxing. Here one has to embed the keys of a block cipher such as AES into its software. Our techniques can be used to blur the boundaries of the rounds and hide the secrets well. Another natural application if this can be used to defend against side channel attacks that use cache misses and timing. In addition, it can be reasonably expected that keeping certain critical data in an oblivious data structure will make it for exploitation by an external attack such as a virus. The last two items require further research. Use of these data structures with suitable message and control flow between those data structures can be helpful in estimating the strength of the protection schemes. See [3].

7. REFERENCES

[1] D.D. Sleator and R.E. Tarjan. Self-adjusting binary search trees. Journal of the ACM, 32 : 652(686, 1985).

[2] R. Ostrovsky and O. Goldreich. Software Protection and Simulation on Oblivious RAMs. Journal of the ACM, Vol. 43, No. 3, May 1996, pp.431 − 473.

[3] Nenad Dedic, Mariusz Jakubowski, Ramarathnam Venkatesan,A graph game model for software tamper protection, Lecture Notes in Computer Science, Proc of 9th Information Hiding, Brittany France, June 13-15, 2007

[4] B.Barak, O.Goldreich, R.Impagliazzo, S.Rudich, A.Sahai, S.Vadhan and K.Yang, On the (Im)possibility of Obfuscating Programs. CRYPTO 2001, LNCS 2139, p. 1 ff.

[5] Anckaert, B.; Jakubowski, M.; Venkatesan, R. Proteus: Virtualization for Diversified Tamper-Resistance. Proceedings of the Sixth ACM Workshop on Digital Rights Management. 2006. pp. 47-57 October 30 2006..

[6] J.L. Bentley, D.D. Sleator, R.E. Tarjan, and V.K. Wei. A locally adaptive data compression scheme. Communications of the ACM, 29(4):320–330, 1986.

[7] S. Albers, M. Karpinski Randomized splay trees: theoretical and experimental results. Information Processing Letters archive Volume 81 , Issue 4 (February 2002)

[8] M. Furer, Randomized Splay Trees. Tenth Annual ACM-SIAM Symposium on Discrete Algorithms, Baltimore, Maryland, 17-19 January 1999

[9] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, On the Effectiveness of Address-Space Randomization. 11'th ACM conference on Computer and Communications Security (CCS), pp. 298-307, 2004.

[10] X.Zhuang, T.Zhang, Hsien-Hsin S. Lee, S.Pande, Hardware Assisted Control Flow Obfuscation for Embedded Processors. CASES'04, September 22–25, 2004.

[11] D.A.Osvik, A.Shamir, E.Tromer, Cache attacks and countermeasures: the case of AES, proc. RSA Conference Cryptographers Track (CT-RSA) 2006, to appear

[12] Abramson, N. Information Theory and Coding. McGraw-Hill, New York, 1983.