# Towards a Theory of Software Protection

(Extended Abstract)

*Oded Goldreich*
Computer Science Department
Technion, Haifa 32000, Israel

## ABSTRACT

Software protection is one of the most important issues concerning computer practice. The problem is to sell programs that can be executed by the buyer, yet cannot be duplicated and/or distributed by him to other users. There exist many heuristics and ad-hoc methods for protection, but the problem as a whole did not receive the theoretical treatment it deserves.

In this paper, we make the first steps towards a theoretic treatment of software protection: First, we distill and formulate the key problem of *learning about a program from its execution*. Second, we present an *efficient* way of executing programs (i.e. a interpreter) such that it is infeasible to learn anything about the program by monitoring its executions. A scheme that protects against duplication follows.

How can one efficiently execute programs without allowing an adversary, monitoring the execution, to learn anything about the program ? Current cryptographic techniques can be applied to keep the contents of the memory unknow throughout the execution, but are *not applicable* to the problem of hiding the access pattern. Hiding the access pattern *efficiently* is the essence of our solution. We show how to implement (on-line and in an "oblivious manner") $t$ fetch instructions to a memory of size $m$ by making less than $t \cdot m^{\varepsilon}$ actual accesses, for every fixed $\varepsilon>0$.

## 1. INTRODUCTION

Software protection is one of the most important issues concerning computer practice. The problem is to sell programs that can be executed by the buyer, yet cannot be duplicated and/or distributed by him to other users. A lot of engineering effort is put into trying to provide "software protection", but this effort seems to lack theoretical foundations. In particular, there is no crisp definition of what the problems are and what should be considered as a satisfactory solution. In this paper, we make the first steps towards a theoretic

Work done while author was in the Laboratory for Computer Science, MIT.
Partially supported by a Weizmann Postdoctoral Fellowship, an IBM Postdoctoral Fellowship, and NSF Grant DCR-8509905.

A.M. Odlyzko (Ed.): Advances in Cryptology - CRYPTO '86, LNCS 263, pp. 426-439, 1987.

treatment of software protection, by distilling a key problem and solving it efficiently.

Before going any further, we distinguish between two intuitive notions: the problem of *protection against duplication* and the problem of *protection against distribution*. Loosely speaking, the first problem consists of ensuring that there is no efficient method for creating executable copies of the software; while the second problem consists of ensuring that, in case duplication succeeds, the illegal duplicator should be unable to prove in court that he has designed the program. In this paper we concentrate on the first problem, which clearly implies a solution to the second one.

We claim that *protection against duplication* must use some hardware measures: mere software (which is not physically protected) can always be duplicated. On the other extreme, the trivial solution is to rely only on hardware. That is, to sell physically-protected special-purpose computers for each task. This "solution" has to be rejected as infeasible and too expensive. We conclude that a real solution to protecting software from duplication should combine feasible software and hardware measures.

It has been suggested [Be, K] to protect software against duplication by selling a *physically shielded* CPU together with an *encrypted* program. The CPU will contain the corresponding decryption key, and will be installed in a computer system. The CPU will execute the program using the memory, I/O devices and other components of the computer. As customary, the CPU itself will contain only a small amount of storage space. We stress that only the CPU will be physically shielded and that all other components of the computer, including the memory in which the encrypted program and data are stored, will not be shielded.

The above setting is on the right track. It only uses a small amount of physical protection (shielding), and its implementation is feasible in current technology. However, the above setting does not constitute a full solution since it was not specified exactly how the CPU is to execute the program using the memory. A naive specification states that the computer operates as an ordinary Random Access Machine, except for the extra encryption and decryption performed by the CPU. This naive specification is not good enough, since certain properties of the program as its loop structure will not be kept secret from an observer. It is true that *straightforward duplication* of the program is not possible since one part of the program (i.e. the key) is in the shielded CPU which is unduplicatable. But protection against duplication should mean more than foiling straightforward attempts. In particular it should mean that the user is unable to learn enough about the program so that he can latter reconstruct it by himself. We thus view the above setting (i.e. a small shielded CPU and an encrypted program) as the start point for the study of software protection, rather than as a satisfactory solution. In fact, we will use this setting as the framework for our investigations, which are concerned with the following key question:

*What can a user learn about the program he bought ?*

### 1.1 What Can Be Learnt by Executing a Program

We recall that the program consists of an encrypted code and a shielded CPU capable of "executing" the code (on an external memory device which may be monitored by the user). The user can run the program on inputs of its choice and watch the sequence of memory accesses during such executions. Furthermore, he

can even interfere in the execution by changing the contents of the memory locations. In any case, the pattern of memory accesses certainly carries knowledge about the program. In many cases, one can easily infer from the access pattern essential properties of the program such as its loop structure. In some cases, this may suffice in order to reconstruct the program.

Our goal is to make it infeasible for the adversary to improve his ability of reconstracting the program by experimenting with it. If it is initially "easy" to reconstract the program then we require nothing, but in case this task is initially "hard" then experimenting with the program should not help. We meet our goal by requiring that the adversary can not learn anything about the encrypted program, except for its input/output relation and its running time. Certainly, if an adversary can learn nothing (except I/O relation and running-time) from his experiments then he can not improve his ability of reconstructing the program. Thus, the notion of a CPU which defeats experiments (i.e. prevents learning about a program from its executions) is the key to preventing software duplication. Intuitively, a CPU defeats experiments if it is infeasible to distinguish the sequences of memory accesses of any two programs run by it. The technical difficulty in the definition is the need to decouples the specified behaviour of the programs (i.e. input/output relation and running time) from the sequences of memory accesses made during their executions.

**Definition (sketch):** We say that a *CPU defeats experiments* if no probabilistic polynomial-time adversary can, on input an encrypted program, distinguish the two cases:

1) The adversary is *experimenting with the genuine CPU*, which is trying to execute the encrypted program through the external memory.

2) The adversary is *experimenting with a fake CPU*. The interactions of the fake CPU with the memory are almost identical to those that the genuine CPU would have had with the memory when executing a dummy program (e.g. *while* TRUE *do skip;*). The execution of the dummy program is timed-out by the number of steps of the real program. When timed-out, the fake CPU writes to the memory the same output that the genuine CPU would have written on the "real" program (and the same input).

**Constructing an efficient CPU which defeats experiments**

The problem of constructing a CPU which defeats experiments is not an easy one. Essentially there are two issues: The first issue is to hide from the adversary the values stored and retreived from memory, and to prevent the adversary's attempts to change these values and/or to launch an attack on the encryption function. This is done using traditional cryptographic techniques (e.g. probabilistic encryption [GM] and message authentication [GGM]) in an inovative manner. The second issue is to hide (from the adversary) the sequence of instructions and variables accessed during the execution (hereafter referred as *hiding the access pattern*).

Hiding the memory access pattern is a completely new problem and traditional cryptographic techniques are not applicable to it. A trivial but unacceptably wasteful solution consists of scanning through the entire memory each time a variable needs to be accessed. In this paper, we provide an efficient solution to the problem of hiding the access pattern. This solution is the basis of our construction of an efficient CPU which defeats experiments.

**Main Theorem:** Let $m$ denote the size of the external memory, and assume that one-way permutations exist. Then there exist a way to execute programs (through the memory) without leaking any knowledge about them, such that $t$ instructions of the original program require only $t \cdot m^{\varepsilon}$ memory accesses, $\varepsilon > 0$.

(The actual expression is $t \cdot 2^{\sqrt{2\log_2 m \cdot \log_2 \log_2 m}}$.)

### 1.2 The Hidden Access Game

The Main Theorem is proved by reducing the problem of executing programs without leaking knowledge about them, to a "hidden access game". The reduction uncouples the traditional cryptographic issues of encryption and authentication from the new issue of hiding an access sequence. The access game consists of a main player (called the *magician*), $m$ marked balls, and $2m$ boxes each capable of storing a single ball. Initially the $m$ balls are placed in the first $m$ boxes, such that ball $i$ is in the $i$th box. The magician can hold only a single ball in his hands at any time. There are two additional players called the *instructor* and the *adversary*. The game proceeds in rounds as follows. In each round, the instructor secretly specifies to the magician a ball (say ball $i$), and the magician "answers" by conducting a sequence of actions such that at the sequence's end the magician holds ball $i$ in his hands. The magician's actions consists of inserting his hand into a box for a moment, during which he either drops a ball or takes a ball or does nothing. The adversary can only sees into which box the magician has inserted his hand, but cannot see whether the magician dropped a ball, took a ball or did nothing. (It goes without saying that the adversary cannot see through the box.) The instructor is not collaborating with either magician or adversary. Can the magician follow the game without allowing the adversary to learn anything about the instruction sequence? More precisely, we require that the sequence of visible actions yields no information about the sequence of instructions.

There is a wasteful solution corresponding to the simple solution of the software protection problem: on every instruction the magician inserts his hand to all boxes in a predetermined order. Our proof of the above Theorem offers a better solution: in order to follow $t$ instructions the magician needs to make only $t \cdot 2^{\sqrt{2\log_2 m \cdot \log_2 \log_2 m}}$ actions (hand insertions).

**Remark:** The access game studied in this paper can be viewed as the Random Access Machine analogue of the *oblivious Turing Machine* problem studied by Pippenger and Fischer [PF]. The difference is that their solution heavily relies on the fact in their setting the instruction pattern is local (i.e. after asking for ball $i$, the instructor can only ask for either ball $i-1$ or ball $i+1$).

### ORGANIZATION

In Section 2 we establish a formal framework and present a definition of the phrase "a CPU executes programs without leaking knowledge about them". In Section 3 we sketch a reduction of the the problem of implementing such executions to the problem of implementing a magician in the above access game. *The reader who is merely interested in the access game is encourage to skip these sections and proceed directly to Sections 4 and 5.* Section 4 consists of the first non-trivial solution to the access game: a solution involving an

overhead factor of $\sqrt{m}$. In Section 5, a recursive solution involving an overhead of $2^{\sqrt{2\log_2 m \cdot \log_2\log_2 m}}$ - is presented. In Section 6, we present a $\Omega(\log m)$ lower bound on the overhead in a solution to the access game. We conclude with some remarks and open problems.

## 2. OUR DEFINITION OF SOFTWARE PROTECTION

Loosely speaking, our definition of protected software is that the adversary having the CPU and the encrypted program can "learn" nothing "substantial" about the program except for its input/output relation and running time. In order to present a formal definition we need first to define the interaction between the CPU, memory, adversary and to parameterize the encryption. We next turn to define transformations on programs (compilers) and define "learning substantially" as the ability to distinguish the original programs by monitoring the executions of their compiled mappings. Compilers which map programs in a manner that defeats any attempt to learn something substantial are then defined as protecting software. The reader may note that in this section we present the transformations on programs as compilers while in the introduction they were presented as interpreters. This difference is clearly not essential.

### 2.1. Interactive Machines, CPU, Memory, Programs, and Encryption

We start by defining the memory and the CPU as two interacting machines. The definition matches the standard notion of a RAM (e.g. [AHU]) in case the memory and CPU are interacting with each other. The only detail worth emphasis is that the CPU can only use space linear in its input parameter.

**Definition 1** (Probabilistic Interactive Machines - sketch): A *probabilistic interactive machine (PIM)* consists of a read-only input tape, a write-only output tape, a *work tape* and a finite control. In addition to the above the PIM may receive and send messages through a special communication channel.

**Definition 2** (Linear PIM): A *linear PIM* is a PIN that on input $x$ accesses only the first $O(|x|)$ cells of its work tape.

**Definition 3** (Memory): The *memory* is a (linear) PIM operating as hereby specified. On input a string $y$ partitioned (by special marks) into $m$ blocks, the memory copies the input to its work tape, and from this point on considers the $i$th block of $y$ as its $i$-th cell. Subsequently, the memory is message driven. When reading a new message of the form $(\sigma,i,z)$ the memory acts as follows. If $\sigma=S$ and $1 \le i \le m$ then the memory *sends* a message consisting of the current contents of its $i$-th cell. If $\sigma=P$ and $1 \le i \le m$ then the memory *puts* $z$ as the new contents of its $i$-th cell (if $z$ is too long -- it is truncated). If $\sigma=T$ the the memory outputs the contents of its work tape, and stops. In case none of the above holds, the memory remains idle.

**Remark:** For the sake of simplicity, we have assumes at this point that the programs conduct all their computation in the space occupied initially by the input. In practice, the actual input will be padded by blanks to allocate sufficient work space for the execution. The padded input will then serve as input to the program. An alternative approach, in which the memory size grows during the execution to meet workspace needs, will

be explored in the full version of this paper.

**Definition 4** (CPU - sketch): The *CPU* is a **linear** PIM which operates as hereafter specified. The input to the CPU is ignored, and its only purpose is to trigger the execution of the CPU, and to specify the permitted "length" of the CPU's work tape. The CPU starts its execution by sending a (fetch) message of the form $(S,1,\cdot)$. Subsequently it operates in "rounds". In each round it reads a new arriving message (into its work tape), applies a polynomial-time computation to its work tape (an "elementary operation in the terminology of the RAM model [AHU]), and concludes by sending a message (consisting of the contents of a portion of its work tape). (After sending a message of the form $(T,\cdot,\cdot)$ -- the CPU halts.)

**Definition 5** (programs, data, and computations): The input to the memory $(y)$ is partitioned (by a special symbol) into two parts called the *program* (denoted here as $\pi$) and the *data* (denoted $x$). The output of the memory (on input $y=(\pi,x)$), after interacting with the CPU, is denoted $\pi(x)$ and called *the result of $\pi$'s computation on input $x$*.

**Definition 6** (Probabilistic Encryption and its Security [GM] - sketch): A *probabilistic encryption scheme* is a triplet of probabilistic polynomial-time algorithms denoted $G,E,D$. On input $n$ (in unary) algorithm $G$ outputs a (legal) key $K$ of length $n$. On input a key $K$ and a message $M$, algorithm $E$ randomly selects an encryption denoted $E_K(M)$, such that $D_K(E_K(M))=M$. Loosely speaking, we say that the encryption scheme is *secure* if on input $n$ (in unary), and the messages $M_1$ and $M_2$, their probabilistic encryptions $E_K(M_1)$ and $E_K(M_2)$ (where $K=G(n)$) are polynomially-indistinguishable (even when given access to a black box implementing $E_K$).

**Remark**: We do not assume here that the encryption scheme is public-key.

### 2.2. Cryptographic CPU, Specification Oracle, and Compilers

**Definition 7** (Cryptographic CPU - sketch): The *Cryptograpic CPU* (*CCPU*) operates essentially as a CPU except for the following details:

1) The input is considered as a cryptographic key $K$ of length $n$ (and is not ignored).

2) The CCPU can effect (as an "elemetary operation") $E_K$ and $E_K^{-1}$ on any string of length $n$.

**Remark:** The time and space complexities of effecting $E_K$ and $E_K^{-1}$ are ignored in the above definition. In considering an implementation of a CCPU, the time complexity of effecting $E_K$ enters as a multiplicative factor, while the space complexity enters as an additive term. Both complexities depend only on the length of $K$, and thus are independent of the length of the data (to the program run by the CCPU). In the factoring-based implementation, the time complexity is $O(n^3)$ while the space complexity is $O(n)$.

**Definition 8** (A specification oracle): A *specification oracle for a program* $\pi$, is an oracle that on query $x$ returns $(\pi(x),t_\pi(x),s_\pi(x))$, where $\pi(x)$ is the output of $\pi$ on input $x$, $t_\pi(x)$ is the running-time of $\pi$ on input $x$, and $s_\pi(x)$ is the storage-requirement of $\pi$ on input $x$.

**Remark**: For the sake of simplicity, we assume in the rest of this extended abstract that both $t_\pi(x)$ and $s_\pi(x)$ depend only on the length of $x$. Furthermore, we will assume that these functions are easily computable.

Thus, the only interesting thing in the oracle's answer is $\pi(x)$.

Our objective is to claim that no adversary can learn anything about $\pi$, when given input $E_K(\pi)$ and interacting with the CCPU. This is false when $\pi$ is executed in the straightforward manner. What we do is map $\pi$ into a "functionally equivalent" program $\pi'$ and execute $\pi'$. We will require that no adversary given the encryption of $\pi'$ (and interacting with the CCPU) can learn anything about $\pi$.

**Definition 9** (Compiler): A compiler $C$ is a probabilistic polynomial time algorithm that on input an integer $n$ (in unary) and a program $\pi$ outputs an $n$-bit cryptograpic key $K = G(n)$ and an encrypted program $E_K(\pi')$, such that for every $x$, $\pi(x) = \pi'(x)$. We denote $\pi'$ by $C(\pi)$.

### 2.3. Software Protection and its Cost

Now we are ready to state our definition of software protection. Loosely speaking, a compiler is said to protect software if whatever can be efficiently computed on input an (encrypted) compiled program (when interacting with a CCPU (having the corresponding key)) -- can be efficiently computed given access only to the specification oracle for the program.

**Notation** (sketch): By $1^n$ we mean the unary representation of $n$. When writing $D_1(x) \equiv_P D_2(f(x))$, we mean that the probability distributions generated by the algorithms $D_1$ and $D_2$ on "random" $x$'s are polynomially indistinghishable [GM, Y]. Let $A$ and $B$ be interacting machines, then $A_{B(y)}(x)$ denote the probability distribution output by $A$ on input $x$, when $A$ is interacting with $B$ which gets $y$ as a private input (i.e. $A$ does not get $y$). Let $M$ be an oracle-machine, and $\pi$ be a program, then $M^{\pi}(x)$ denotes the output distribution of $M$ on input $x$ and access to a specification oracle for $\pi$.

**Definition I** (Software Protection -- sketch): Let $P$ denote the CCPU. The compiler $C$ *protects software* if for every probabilistic polynomial-time interacting machine $A$, there exists a probabilistic polynomial-time oracle-machine $M$, such that for all programs $\pi$ the following holds

$$A_{P(K)}(E_K(C(\pi))) \equiv_P M^{\pi}(1^{|K|}),$$

where $K$ is choosen randomly among all cryptographic keys of length $|K|$.

**Definition II** (cost of software protection): Let $\pi$ be a program, and $t_{\pi}(x)$ be as in Definition 8. Let $C$ be a compiler. Let $f_C^{\pi}$ be a function from integers to reals, such that $f_C^{\pi}(m)$ is the maximum, taken over all $m$-bit strings $x$, of $t_{C(\pi)}(x)/t_{\pi}(x)$. Let $f_C(m)$ be a function such that for every $\pi$ and for sufficiently large $m$, $f_C^{\pi}(m) \leq f_C(m)$. Then the *overhead created by the compiler C* is at most $f_C(m)$.

## 3. REDUCTION TO AN ACCESS GAME

The access game, described in the introduction, can be formulized as a randomized procedure that on input a sequence $\alpha$ of elements out of $\{1,2,...,m\}$ outputs two sequences, a *visable* sequence $\beta$ and a *secret* sequence $\gamma$. The sequence $\beta$ contains elements out of $\{1,2,...,2m\}$, while the sequence $\gamma$ consists of elements out of $\{T,D,N\}$. (The reader may think of $\alpha$ as being the instruction sequence, of the procedure as being the

magician, of the visable sequence as the sequence of cells into which the magician has inserted his hand, and of the secret sequence as of what he did when inserting his hand. $T$ stands for "take a ball", $D$ for "drop", and $N$ for "do nothing".) The visable sequence $\beta$ gives no information about the sequence $\alpha$ (i.e. the conditional probability that the input is $\alpha$ given that the visable output is $\beta$ equals the a-priori probability that the input is $\alpha$). The execution of $\beta$ with $\gamma$ gives a sequence $\delta$ which contains $\alpha$. ($\delta$ is the sequence of balls held in the magician's hand, when he inserts his hands to cells $\beta$ and acts in them according to $\gamma$.) Furthermore, the procedure should satisfy the above conditions when working on-line: every new element of $\alpha$ should cause the procedure to output new portions of $\beta$ and $\gamma$ such that they "contain" the element.

In addition, we require that the randomized procedure is efficient in the following sense:

1) The next output symbol is computed in time polynomial in $m$;

2) The space used is logarithmic in $m$, provided that the procedure has access to a random oracle;

3) The procedure can compute at each moment, the number of times each ball was taken out of a cell.

**Proposition** (The Reduction): Suppose that there exist one-way permutations, and there is a procedure satisfying the above conditions such that for every input of length $t$ it outputs sequences of length $t \cdot f(m)$. Then there exists a compiler that protects software with overhead at most $f(m)$.

**The proof** employes the following "traditional" cryptograpic techniques - probabilistic encryption [GM], pseudorandom function [GGM], and (provably secure) message authentication [GGM].

1) Probabilistic encryption is used in order to make it infeasible to tell anything about the contents of a memory location. The existence of one-way permutations implies the existence of probabilistic encryption schemes [GM, Y]. More efficient schemes exist under the intractability of factoring [ACGS, BG].

2) The pseudorandom functions replace the random oracle used by the procedure. It is crucial that they can be implemented using "small" space. Pseudorandom functions exist if one-way permutations exist [BM, Y, GGM].

3) Message authentication is used in order to prevent the adversary loancing a chosen ciphertext attack on the encryption scheme. Another use of authentication is to prevent the adversary from switching the contents of memory location, or to replace the contents by a previous contents of the same location. (It is thus crucial that the procedure satisfies the additional condition (3).) Message authentication is implemented using pseudorandom functions [GGM].

## 4. THE "SQUARE ROOT" SOLUTION

We will describe the solution, using the intuitive "magician" formalism of the introduction. Recall that there are $m$ balls marked 1 through $m$, which initially reside in the first $m$ cells such that ball $i$ is in the $i$-th cell. Altogether there are $2m$ cells, and suppose that $m \geq 2\sqrt{m}$.

The solution described below allows the magician to follow a sequence of $t$ instructions, by commiting at most $t \cdot \sqrt{m}$ actions. We desribe a solution in which the magician is allowed to hold up to 2 balls at any point in time.

Following is an outline of the magician's procedure:

0) Initially, for $1 \le i \le m$, the $i$ th cell contains ball number $i$. All other cells are empty.

*while* TRUE *do*;

1) Randomly permute the contents of the first $m+\sqrt{m}$ cells. That is, select a permutation $\pi$ over the integers 1 through $m+\sqrt{m}$ and relocate the contents of cell $i$ in cell $\pi(i)$.

2) Execute $\sqrt{m}$ instructions as follows. During the execution of these instructions, maintain the balls (accessed by these instructions) in cells number $m+\sqrt{m}+1$ through $m+2\sqrt{m}$. The instruction "get ball $i$" is executed as follows. First scan through the special $\sqrt{m}$ cells and check whether ball $i$ is in one of these cells. If the $i$ th ball is not found there then we retreive it from cell $\pi(i)$; else we access the next empty cell (i.e. one of the cells $m+1$ through $m+\sqrt{m}$ which was not accesssed before).

3) Return balls to their initial locations.

Before getting to the implementation details of the above steps, we provide some hints to as why no information about the instruction sequence is revealed by the sequence of viasble actions. Step (1) is syntactically independent of the instruction sequence. The accesses executed in step (2) are of two types: scanning through all cells from the $m+\sqrt{m}+1$-th to the $m+2\sqrt{m}$ -th, and accessing a new random cell between 1 and $m+\sqrt{m}$. No information about the instruction sequence is leak by this! The access pattern of Step (3) is identical to a combination of the second type of accesses made in Step (2), and the accesses of step (1).

**4.1 How to randomly permute the contents of the memory**

We first show how to implement a random permutation by using a random oracle and sorting, and next show how to implement sorting using a random oracle.

**Choosing and "storing" a random permutation**

We show how to choose and store a random permutation over $\{1,2,...,t\}$, using $O(\log t)$ storage and a random oracle. The idea is to use the oracle in order to tag the elements with random distinct (with high probability) integers. The permutation is obtained by sorting the elements by their tags [1]. (It suffice to have the tags being drawn at random from the set $\{1,2,...,t^{\log t}\}$.) Let $f:\{1,2,...,t\} \to \{1,2,...,t^{\log t}\}$ be a random function trivially constructed by the random oracle. Then $\pi(i)=k$ if and only if $f(i)$ is the $k$-th smallest element in $\{f(j): 1 \le j \le t\}$.

---

1) **Remark:** Luby and Rackoff [LR] showed that three iterations of the DES can be used to construct a pseudorandom permutation out of three random functions. However, this pseudorandom permutation is not good enough for our purposes since it can be distinguished from a random permutation with probability $\Theta(q^2/t)$, where $q$ is the number of permutation evaluations.

**Arranging the balls by the choosen permutation**

Now we face the problem of sorting the $t$ elements (by their tags) in a manner which leaks no information about the permutation. The crucial condition is that the magician which executes the sorting can store only a fix number of balls (say 2) at a time. The idea is to "implement" Batcher's Sorting Network [Bat], which allows to sort $t$ elements by $t\lceil \log_2 t \rceil^2$ comparisons. Each comparison is "implemented" by accessing both the corresponding cells, retreiving their contents, and then putting the contents back in the desired order. The sequence of accesses generated for this purpose is fixed and independent of the permutation to be implemented. Note that the magician can easily compute at each point which comparison he needs to implement next. This is due to the simple structure of Batcher's network, which is uniform with respect to logarithmic space [(2)].

**Computing the permutation in succeeding steps (2) and (3)**

The way the permutation $\pi$ is defined does not allow an immediate method of computing $\pi(i)$ on input $i$. This computation will be required in the subsequent executions of steps (2) and (3). We will "compute" $\pi(i)$ by conducting a binary search on the $f(\cdot)$'s, using the fact that (after step (1)) the $f(\cdot)$'s are "stored", in sorted order, in the cells. Note that $\pi(i)$ is computed in order to access the $\pi(i)$-th cell, and therefore the accesses done in the binary search do not add any information (since they are determined by $\pi(i)$).

**4.2 How to simulate a single access**

Now it is straightforward to give the details of Step (2). Throughout step (2), *count* maintains the number of single accesses simulated in the current run. *count* is initialy 0 and is increamented until it reaches $\sqrt{m}$. On instruction "take ball $i$" the magician proceeds as follows:

2a Scans through locations $m+\sqrt{m}+1$ to $m+2\sqrt{m}$. If the ball $i$ is in either of these cells then fetch it, and sets $j$ such that ball $i$ was taken from the $m+\sqrt{m}+j$-th cell. If neither of these cells contains ball $i$ then set $j=count$.

2b If $j \neq count$ then the magician accesses the $\pi(m+count)$-th cell (which is empty!); else the magician accesses the $\pi(i)$-th cell, and retrieve its contents (i.e. ball $i$).

2c Scans through locations $m+\sqrt{m}+1$ to $m+2\sqrt{m}$ again, and put ball $i$ in the $m+\sqrt{m}+j$-th cell. Increments *count* by 1.

**4.3 How to rearrange the balls**

Rearranging the balls is done in two substeps: first we undo the effect of the execution of Step (2) and next the effect of Step (1). Following is a description of the first substep. The second substep can be incorporated in the next execution of step (1).

For $j=1$ to $\sqrt{m}$ the magician proceeds as follows:

Accesses the $m+\sqrt{m}+j$-th cell. If it contains a ball, say ball $i$, then the magician accesses the $\pi(i)$-th cell and puts ball $i$ there. If the $m+\sqrt{m}+j$-th cell is empty then the magician accesses its $\pi(m+j)$-th cell (but puts nothing there).

---

2) The simplicity of Batcher sorting network is the main reason we prefare it upon the asymptotically superior Ajtai-Komlos-Szemeredi sorting network [AKS].

### 4.4 Analysis

The reader may easily verify that the sequence of accesses of the magician indeed yields no information about the sequence of instructions. It is left to calculate the overhead of the simulation (i.e. the ratio of accesses over instructions). The permutation applied after every $\sqrt{m}$ instructions causes an overhead of $O(m \cdot \log^2 m)$, which amounts to an amortized overhead of $O(\sqrt{m} \cdot \log^2 m)$ actions per instruction. In addition, each of the instructions causes $O(\sqrt{m})$ actions to be taken in step (2). Other actions taken in step (3) are negligible in mumber. The total overhead thus amounts to $O(\sqrt{m} \cdot \log^2 m)$ actions per instruction [3]. We get

**Theorem 1**: There exist a magician procedure with $O(\sqrt{m} \cdot \log_2 m)$ overhead.
Furthermore, this procedure is efficient in the sense of Section 3.

## 5. THE RECURSIVE SOLUTION

The recursive solution presented in this section is based on a generalization of the solution presented in Section 4. One can view the solution of Section 4 as consisting of two parts: the random shuffling and reshuffling of the cells contents every $\sqrt{m}$ original accesses (steps (1) and (3)), and the simulation of the instructions through their randomized locations (step (2)). Substeps (2a) and (2c) actually simulates a "powerful" magician which can hold up to $\sqrt{m}$ balls in its hands at any time: The magician looks whether he holds already the required ball. If the answer is negative then the magician fetches the ball, else he reaches for a "new" empty cell. Holding up to $\sqrt{m}$ balls was simulated in the obvious manner by scanning through extra $\sqrt{m}$ cells.

When trying to generalize the solution, we want to decrease the amortize cost of the random shuffling. Thus we will consider a more powerful magician capable of holding up to $f(m) > \sqrt{m}$ balls (say $f(m) = m^{3/4}$). The amortized cost of steps (1) and (3) is thus $m \cdot (\log_2 m)^2 / f(m)$. The key question is: *how are we going to simulate the magician with the $f(m)$-size hand ?*

We will think of the magician's hand as a heap containing up to $f(m)$ elements. We need to support $f(m)$ find operations and up to $f(m)$ element insertions to this heap. This translates to $f(m) \cdot \log_2 f(m)$ access operations to the data structure. We can view the situation as a new simulation, this time on $O(f(m))$ cells and for $O(f(m) \cdot \log f(m))$ instructions. In other words, we need to issue $\log_2 f(m)$ new instructions into the $f(m)$-size hand per each instruction into the original cells. We thus get.

$$overhead(m) = \frac{m \cdot (\log_2 m)^2}{f(m)} + O(\log f(m)) \cdot overhead(f(m))$$

Solving the recurence, we get $overhead(m) = O(2^{\sqrt{2 \log_2 m \cdot \log_2 \log_2 m}})$. Thus

3) Actually, the above choice of parameters is not optimal. Repermuting the balls after every $O(\sqrt{m} \cdot \log m)$ instructions, yields an overhead of $O(\sqrt{m} \cdot \log m)$ actions per instruction.

**Theorem 2**: There exist a magician procedure with $O(2^{\sqrt{2\log_2 m \cdot \log_2 \log_2 m}})$ overhead.
Furthermore, this procedure is efficient in the sense of Section 3.

## 6. A LOWER BOUND

A simple combinatorial argument shows that any oblivious simulation of arbitrary RAMs should have an average $\Omega(log\ m)$ overhead.

**Theorem 3**: Every successful magician procedure must make at least $m + (t-1)\cdot\log_3 m$ actions in order to implement $t$ instructions.

The proof uses very little of the structure of the problem, and therefore we do not believe that the lower bound obtained is tight.

## 7. CONCLUSIONS AND OPEN PROBLEMS

We have reduced software protection to a "hidden access game". The reduction was curried out on the instruction level. However, an identical reduction can be carried out on any level of programming modularity; e.g. cash memory accesses, paging mechanisms etc.

The hidden access game has also other applications, as allowing many users to run secretly private programs on a public computer, foiling flow analysis in distributed communication networks etc.

Formulating the problem of preventing software duplication has lead us to consider the question of what can be learnt about a program by watching its executions. It seems that formulating the problem of preventing software distribution (i.e. fingerprinting software) will lead to different questions. It will be very interesting to try and come up with a theoretical framework and definitions for "fingerprinted software".

A more technical problem is to provide better solutions to the hidden access game, and in turn to present compilers which protect software at lower cost. We believe that this should be possible. On the other hand, proving better lower bounds on the overhead of good magicians will be also interesting. At this point the gap between the known upper and lower bounds, on the overhead, is quite large: $O(2^{\sqrt{2\log_2 m \cdot \log_2 \log_2 m}})$ versus $\Omega(\log m)$.

## ACKNOWLEDGEMENTS

It is my pleasure to thank friends and colleagues for their contributions to this work and its presentation. In particular I wish to thank Baruch Awerbuch, Benny Chor, Shimon Even, Shafi Goldwasser, Silvio Micali, Ron Rivest, and Yacov Yacobi. Special thanks to Hugo Krawczyk for carefully reading an earlier version of the manuscript, pointing out some errors, and suggesting several improvements.

## REFERENCES


[AHU] Aho, A.V., J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publ. Co., 1974.

[AKS] Ajtai, M., J. Komlos, and E. Szemeredi, "An $O(n \cdot \log n)$ Sorting Network", *Proc. 15th STOC*, 1983, pp. 1-9.

[ACGS] Alexi, W., B. Chor, O. Goldreich, and C.P. Schnorr, "RSA and Rabin Functions: Certain Parts Are As Hard As The Whole", to appear in *SIAM Jour. on Computing*. Extended Abstract in *Proc. 25th FOCS*, 1984.

[Bat] Batcher, K., "Sorting Networks and their Applications", *AFIPS Spring Joint Computer Conference*, 32, 1968, pp. 307-314.

[Be] Best, R., "Microprocessor for Executing Encrypted Programs", US Patent 4,168,396. Issued September 1979.

[Blu] Blum, M., unpublished manuscript, 1983.

[BG] Blum, M., and S. Goldwasser, "An Efficient Probabilistic Public-Key Encryption Scheme which Hides All Partial Information", *Advances in Cryptology: Proceedings of CRYPTO 84*, Springer Verlag, Lecture Notes in Computer Science (196), 1985, pp. 289-299.

[BM] Blum, M., and Micali, S., "How to Generate Cryptographically Strong Sequences of Pseudo-Random Bits", *SIAM Jour. on Computing*, Vol. 13, 1984, pp. 850-864.

[GGM] Goldreich, O., S. Goldwasser, and S. Micali, "How to Construct Random Functions", *Proc. of 25th Symp. on Foundation of Computer Science*, 1984, pp. 464-479. To appear in *Jour. of ACM*.

[GM] Goldwasser S., and S. Micali, "Probabilistic Encryption", *Jour. of Computer and System Science*, Vol. 28, No. 2, 1984, pp. 270-299.

[K] Kent, S.T., "Protecting Externally Supplied Software in Small Computers", Ph.D. Thesis, MIT/LCS/TR-255, 1980.

[LR] Luby, M., and C. Rackoff, "Pseudo-random Permutation Generators and Cryptographic Composition", *Proc. of 18th STOC*, 1986, pp. 356-363.

[PF] Pippenger, N., and M.J. Fischer, "Relation Among Complexity Measures", *Jour. of ACM*, Vol. 26, No. 2, 1979, pp. 361-381.

[Y] Yao, A.C., "Theory and Applications of Trapdoor Functions", *Proc. of the 23rd IEEE Symp. on Foundation of Computer Science*, 1982, pp. 80-91.


## APPENDIX

**Definition** (polynomial indistinguishability [GM, Y]): Let $\{\Pi_1^n\}$ and $\{\Pi_2^n\}$ be two probability ensables; that is for every integer $n$ and $i$, $\Pi_i^n$ is a probability distribution on strings of length $\leq poly(n)$. The ensembles $\{\Pi_1^n\}$ and $\{\Pi_2^n\}$ are *polynomial indistinguishable* if the following holds:

> For every probabilistic polynomial-time algorithm $A$, every constant $c$, and sufficiently large $n$, the probability that $A$ outputs 1 on $n$ and a string selected according to $\Pi_1^n$ equals up to $n^{-c}$ the probability that $A$ outputs 1 on a string selected in $\Pi_2^n$.

**Definition 5** (Probabilistic Encryption and its Security [GM]): A *probabilistic encryption scheme* is a triplet of probabilistic polynomial-time algorithms denoted $G, E, D$ such that:

1) On input $n$ (in unary representation) algorithm $G$ outputs a key $K$ of length $n$.

2) On input a key $K$ and a message $M$, algorithm $E$ outputs an encryption denoted $E_K(M)$.

3) On input a key $K$ and an encrypted message $E_K(M)$, algorithm $D$ always outputs $M$. In case $D$ is given an illegal key-encryption pair it may behave arbitrarily.

The following security definition implies that both $G$ and $E$ can map an input to many (more than polynomially many) possible outputs. Let $M_1$ and $M_2$ be two messages and $A$ be a probabilistic polynomial-time machine which operates as follows. Algorithm $A$ receives as input two encryptions $E_K(M_1)$ and $E_K(M_2)$ in arbitrary order. In addition, algorithm $A$ is given access to a black box implementing $E_K$ (i.e. when sending $q$ to the black box, $A$ receives back as an answer a random encryption $E_K(q)$ ). Let $\Pi^n_{A,1}$ denote the probability that $A$ outputs 1 when $K$ is a key randomly chosen by $G$ on input $n$, and the encryption of $M_1$ was placed to the left of the encryption of $M_2$ on the input tape. Similarly, $\Pi^n_{A,2}$ denotes the probability of output 1 when $M_2$'s encryption was placed to the left of $M_1$'s. An encryption scheme is *secure* if for every two messages $M_1$ and $M_2$ and every probabilistic polynomial-time algorithm $A$, $\{\Pi^n_{A,1}\}$ and $\{\Pi^n_{A,2}\}$ are polynomially-indistinguishable.

**Remark**: The original definition of encryption security [GM] is for Public-Key Encryptions and thus access to a black box implementing $E_K$ is redundent. Above we gave a more general definition that suits both Private-Key and Public-Key encryptions.