A Parallel Algorithm for Extending Cryptographic Hash Functions (Extended Abstract)*

Palash Sarkar and Paul J. Schellenberg

Department of Combinatorics and Optimization, University of Waterloo, 200, University Avenue West, Waterloo, Ontario, Canada N2L 3G1, psarkar@cacr.math.uwaterloo.ca, pjschell@math.uwaterloo.ca

Abstract. We describe a parallel algorithm for extending a small domain hash function to a very large domain hash function. Our construction can handle messages of any practical length and preserves the security properties of the basic hash function. The construction can be viewed as a parallel version of the well known Merkle-Damgård construction, which is a sequential construction. Our parallel algorithm provides a significant reduction in the computation time of the message digest, which is a basic operation in digital signatures.

Keywords: cryptographic hash function, Merkle-Damgård construction, parallel algorithm, collision resistance.

1 Introduction

Hash functions are extensively used in cryptographic protocols. One of the main uses of hash functions is to generate a message digest from a message. This message digest is signed to get a digital signature. Due to the central importance of hash functions in cryptography, there has been a lot of work in this area. See [6] for a recent survey.

For a hash function to be used in cryptographic protocols, it must satisfy certain necessary conditions. In a recent paper [8], Stinson provides a comprehensive discussion of these conditions and also relations among them. The two most important properties that a cryptographic hash function must satisfy are the following: (a) finding a collision must be computationally infeasible and (b) finding a preimage of a given message digest must be computationally infeasible.

A hash function maps a set of longer messages into a set of shorter message digests. The range is finite, while the domain can possibly be (countably) infinite. Thus, theoretically, a hash function can map arbitrary length strings to finite length strings. However, hash functions with an infinite (or very large) domain can be difficult to construct directly. An alternative approach is to take a hash

* This research supported by a grant from the Mathematics of Information Technology and Complex Systems (MITACS) project.

function with a small finite domain and suitably extend it to tackle long strings. The extension must preserve the security properties (difficulty of finding collision and preimage) of the original hash function. An important construction for such extensions has been described by Merkle [3] and Damgård [2]. The construction is called the Merkle-Damgård (MD) construction.

The MD construction is a sequential construction. Suppose the basic hash function has domain $\{0,1\}^{512}$ and range $\{0,1\}^{128}$. Further, suppose that the message to be signed is long, say 1 Mbits (=2^{20} bits). If the MD construction is applied to the message, then the time taken to generate the digest would be proportional to $2^{20}/(512-128)$. For many applications this can cause an undesirable delay.

In this paper we build on the basic MD construction. We introduce a parallel version of this construction which preserves the security features of the basic hash function. The parallel version uses 2^t processors for some t and produces a significant speed up in the computation of the message digest.

Related Work: The concept of tree hashing has appeared before in the literature. Wegman and Carter [10] used tree hashing techniques to build universal hash functions. This was followed up by Naor and Yung [5] and Bellare and Rogaway [1] in the context of universal one way hash functions. Damgard [2] briefly outlines a tree hashing approach for extending collision resistant hash functions.

In this paper we concentrate exclusively on developing a parallel tree based algorithm for extending cryptographic hash functions. The main difference between our model and previous models is that we consider the number of available processors to be fixed while the length of the message can be arbitrarily long. Thus we consider a fixed processor tree and use it to hash arbitrarily long messages. Each processor simply computes the base hash function. The resulting increase in speed of computation of the message digest is almost linear in the number of processors. As an example, it may not be very expensive to use a tree of 32 or 64 processors to reduce the time required for message digest computation by a factor of 32 or 64 respectively. We believe that our algorithm has potential practical applications in digital signature computation.

Due to lack of space, proofs and detailed discussions cannot be presented in this paper. For these we refer the reader to [7].

2 Basics

2.1 Hash Functions

Our description of hash functions closely parallels that of Stinson [8]. An (n,m) hash function h is a function $h:\{0,1\}^n \rightarrow \{0,1\}^m$. *Throughout this paper we require that* $n \geq 2m$. Consider the following problem as defined in [8].

Problem :	Collision $Col(n,m)$
Instance :	An (n,m) hash function h.
Find :	$x, x' \in \{0,1\}^n$ such that $x \neq x'$ and $h(x) = h(x')$.

By an (ϵ, p) (randomized) algorithm for Collision we mean an algorithm which invokes the hash function h at most p times and solves Collision with probability of success at least ϵ.

The hash function h has a finite domain. We would like to extend it to an infinite domain. Our first step in doing this is the following. Given h and a positive integer L, we construct a hash function $h_L : \{0,1\}^L \rightarrow \{0,1\}^m$. The next step, in general, is to construct a hash function $h^\infty : \cup_{L\geq n}\{0,1\}^L \rightarrow \{0,1\}^m$. However, instead of doing this, we actually construct a hash function $h^* : \cup_{L=n}^{N}\{0,1\}^L \rightarrow \{0,1\}^m$, where $N = 2^{n-m} - 1$. Since we assume $n \geq 2m$, we have $n - m \geq m$. Practical message digests are at least 128 bits long meaning that $m = 128$. Hence our construction of h^* can handle any message with length $\leq 2^{128}$. This is sufficient for any conceivable application. (If we estimate that there are 32 billion computers, that is about 5 computers per man, woman and child, and each computer has 1024 gigabytes of disk storage, and each byte has eight bits, the number of bits that can be stored on all the these computer systems combined is a mere $2^5 \times 2^{30} \times 2^{10} \times 2^{30} \times 2^3 = 2^{78}$ bits. Our construction of h^* can be extended to construct h^∞ and will be provided in the full version of the paper.

We would like to relate the difficulty of finding collision for h_L, h^* to that of finding collision for h. Thus we consider the following two problems (see [8]).

Problem	: Fixed length collision $FLC(n,m,L)$
Instance	: An (n,m) hash function h and an integer $L \geq n$.
Find	: $x, x' \in \{0,1\}^L$ such that $x \neq x'$ and $h_L(x) = h_L(x')$.

Problem	: Variable length collision $VLC(n,m,L)$
Instance	: An (n,m) hash function h and an integer L with $n \leq L \leq 2^{n-m}$.
Find	: $x, x' \in \cup_{i=n}^{L}\{0,1\}^i$ such that $x \neq x'$ and $h^*(x) = h^*(x')$.

By an (ϵ, p, L) (randomized) algorithm $\mathcal{A}$ for Fixed length collision (resp. Variable length collision) we will mean an algorithm that requires at most p invocations of the function h and solves Fixed length collision (resp. Variable length collision) with probability of success at least ϵ. The algorithm $\mathcal{A}$ will be given an oracle for the function h and p is the number of times $\mathcal{A}$ queries the oracle for h in attempting to find a collision for h_L (resp. h^*).

Later we show Turing reductions from Collision to Fixed length collision and Variable length collision. Informally this means that given oracle access to an algorithm for solving $FLC(n,m,L)$ for h_L or $VLC(n,m,L)$ for h^* it is possible to construct an algorithm to solve $Col(n,m)$ for h. These will show that our constructions preserve the intractibility of finding collisions.

2.2 Processor Tree

Our construction is a parallel algorithm requiring more than one processors. *The number of processors is* 2^t. Let the processors be $P_0, \ldots, P_{2^t-1}$. For $i = 0, \ldots, 2^{t-1} - 1$, processor P_i is connected to processors P_{2i} and P_{2i+1} by arcs pointing towards it. The processors $P_{2^{t-1}}, \ldots, P_{2^t-1}$ are the *leaf processors* and the processors $P_0, \ldots, P_{2^{t-1}-1}$ are the *internal processors.* We call the resulting tree the processor tree of depth t. For $1 \leq i \leq t$, there are 2^{i-1} processors at level i. Further, processor P_0 is considered to be at level 0.

Each of the processors gets an input which is a binary string. The action of the processor is to apply the hash function h on the input if the length of the input is n; otherwise, it simply returns the input -

$$P_i(y) = \begin{cases} h(y) & \text{if } |y| = n; \\ y & \text{otherwise.} \end{cases} \tag{1}$$

For $0 \leq i \leq 2^t - 1$, we have two sets of buffers u_i and z_i. We will identify these buffers with the binary strings they contain. The buffers are used by the processors in the following way. There is a formatting processor P_F which reads the message x, breaks it into proper length substrings, and writes to the buffers u_i. For $0 \leq i \leq 2^{t-1} - 1$, the input buffers of P_i are z_{2i}, z_{2i+1} and u_i and the input to P_i is formed by concatenating the contents of these buffers. For $2^{t-1} \leq i \leq 2^t - 1$, the input buffer of P_i is u_i. The output buffer of P_i is z_i for $0 \leq i \leq 2^t - 1$.

Our parallel algorithm goes through several parallel rounds. The contents of the buffers u_i and z_i are updated in each round. To avoid read/write conflicts we will assume the following sequence of operations in each parallel round.

1. The formatting processor P_F writes into the buffers u_i, for $0 \leq i \leq 2^t - 1$.
2. Each processor P_i reads its respective input buffers.
3. Each processor P_i performs the computation in (1).
4. Each processor P_i writes into its output buffer z_i.

Steps (2) to (4) are performed by the processors $P_0, \ldots, P_{2^t-1}$ in parallel after Step (1) is completed by processor P_F.

2.3 Parameters and Notation

Here we introduce some notation and define certain parameters which are going to be used throughout the paper.
Number of processors: 2^t.
Start-up length: $2^t n$.
Flushing length: $(2^{t-1} + 2^{t-2} + \cdots + 2^1 + 2^0)(n - 2m) = (2^t - 1)(n - 2m)$.
Start-up + flushing length: $\delta(t) = 2^t n + (2^t - 1)(n - 2m) = 2^t(2n - 2m) - (n - 2m)$.
Steady-state length: $\lambda(t) = 2^{t-1}n + 2^{t-1}(n - 2m) = 2^{t-1}(2n - 2m)$.
Message: a binary string x of length L.
Parameters q, b and r:

1. If $L > \delta(t)$, then q and r are defined by the following equation: $L - \delta(t) = q\lambda(t) + r$, where r is the unique integer from the set $\{1, \ldots, \lambda(t)\}$. Define $b = \lceil \frac{r}{2n-2m} \rceil$.

2. If $L = \delta(t)$, then $q = b = r = 0$.

Note that $0 \leq b \leq 2^{t-1}$. *We will denote the empty string by* $<>$ *and the length of a binary string* x *by* $|x|$.

3 Fixed Length Input

In this section we describe the construction of the function h_L. The construction is naturally divided into two cases depending on whether $L \geq \delta(t)$ or $L < \delta(t)$. We first show that the case $L < \delta(t)$ reduces to the case $L \geq \delta(t')$ for some $t' < t$. Thus the case $L < \delta(t)$ is tackled using only a part of the processor tree.

3.1 Case $L < \delta(t)$

Let $t' < t$ be such that $\delta(t') \leq L < \delta(t'+1)$. We use the processor tree only upto level t' and use the parallel hashing algorithm of Section 3.2 with t replaced by t'. Thus we are not utilizing all the available processors. It can be shown that this results in a cost of at most one additional parallel round. We will present this proof in the full version of the paper.

3.2 Case $L \geq \delta(t)$

We first describe the parallel hashing algorithm. This algorithm uses several other algorithms as subroutines. We describe these later.

The parameters b and q defined in Section 2.3 will be used in the algorithms that we describe next. More specifically, the parameter q will be used in algorithm PHA and the parameter b will be used in algorithms FEG and FF. These parameters are assumed to be global parameters and are available to all the subroutines. It is quite simple to modify the subroutines such that the parameters are computed as and when required.

Parallel Hashing Algorithm (PHA)
Input: message x of length $L \geq \delta(t)$.
Output: message digest $h_L(x)$ of length m.

```
1. if $L > \delta(t)$, then
2.           $x := x||0^{b(2n-2m)-r}$
             (ensures that the length of the message becomes
             $\delta(t) + q\lambda(t) + b(2n-2m)$.)
3. endif.
4. Initialise buffers $z_i$ and $u_i$ to empty strings, $0 \leq i \leq 2^t - 1$.
5. Do FormatStartUp.
6. Do ParallelProcess.
7. for       $i = 1, 2, \ldots, q$ do
8.           Do FormatSteadyState.
9.           Do ParallelProcess.
```

```
10. endfor
11. Do FormatEndGame.
12. Do ParallelProcess.
13. for    $s = t-1, t-2, \ldots 2, 1$ do
14.             Do FormatFlushing($s$).
15.             Do ParallelProcess.
16. endfor
17. $z_0 = P_0(z_0||z_1||x)$.
18. return $z_0$.
19. end algorithm PHA
```

ParallelProcess (PP)
Action: Read buffers u_i and z_i, and update buffers z_i, $0 \le i \le 2^t - 1$.

```
1. for    $i = 0, \ldots, 2^t - 1$ do in parallel
2.            $z_i := P_i(z_{2i}||z_{2i+1}||u_i)$     if $0 \le i \le 2^{t-1} - 1$.
3.            $z_i := P_i(u_i)$                      if $2^{t-1} \le i \le 2^t - 1$.
4. endfor
5. end algorithm PP
```

Formatting Algorithms. There are four formatting subroutines which are invoked by PHA. Each of the formatting subroutines modifies the message x by removing prefixes which are written to the buffers u_i for $0 \le i \le 2^t - 1$. All the formatting subroutines are executed on the formatting processor P_F.

FormatStartUp (FSU)
Action: For $0 \le i \le 2^t - 1$, write a prefix of message x to buffer u_i and update the message x.

```
1. for    $i = 0, \ldots, 2^t - 1$ do
2.            Write $x = v||y$, where $|v| = n$.
3.            $u_i := v$.
4.            $x := y$.
5. endfor
6. end algorithm FSU
```

FormatSteadyState (FSS)
Action: For $0 \le i \le 2^t - 1$, write a prefix of message x to buffer u_i and update the message x.

```
1. for    $i = 0, \ldots, 2^{t-1} - 1$ do
2.            Write $x = v||y$, where $|v| = n - 2m$.
3.            $u_i := v$.
4.            $x := y$.
5. endfor
6. for    $i = 2^{t-1}, \ldots, 2^t - 1$ do
7.            Write $x = v||y$, where $|v| = n$.
```

```
8.              $u_i := v$.
9.              $x := y$.
10. endfor
11. end algorithm FSS
```

FormatEndGame (FEG)
Action: For $0 \leq i \leq 2^t - 1$, write a prefix of message x to buffer u_i and update the message x.

```
1.  for     $i = 0, 1, 2, \ldots, 2^{t-1} - 1$ do
2.              Write $x = v||y$ where $|v| = n - 2m$.
3.              $u_i := v$.
4.              $x := y$.
5.  endfor
6.  for     $i = 2^{t-1}, 2^{t-1} + 1, \ldots, 2^{t-1} + b - 1$ do
7.              Write $x = v||y$ where $|v| = n$.
8.              $u_i := v$.
9.              $x := y$.
10. endfor
11. for     $i = 2^{t-1} + b, 2^{t-1} + b + 1, \ldots, 2^t - 1$ do
12.             $u_i :=<>$.
13. endfor
14. end algorithm (FEG)
```

FormatFlushing(s) (FF(s))
Input: Integer s.
Action: For $0 \leq i \leq 2^t - 1$, write a prefix of message x to buffer u_i and update the message x.

```
1.  $k = \lfloor \frac{b + 2^{t-s-1} - 1}{2^{t-s}} \rfloor$.

2.  for     $i = 0, 1, 2, \ldots, 2^{s-1} + k - 1$ do
3.              Write $x = v||y$ where $|v| = n - 2m$.
4.              $u_i := v$.
4.              $x := y$.
5.  endfor
6.  for     $i = 2^{s-1} + k, 2^{s-1} + k + 1, \ldots, 2^t - 1$,
7.              Write $u_i :=<>$.
8.  endfor
9.  end algorithm FF
```

3.3 Correctness and Complexity

Here we state that algorithm PHA properly computes an m-bit message digest and state various properties of the algorithm. In Section 3.4 we will provide the security reduction of $Col(n, m)$ to $FLC(n, m, L)$. More detailed discussion and proofs can be found in [7]. Algorithm PHA executes the following sequence of parallel rounds.

1. Lines 5-6 of PHA execute one parallel round.
2. Lines 7-10 of PHA execute q parallel rounds.
3. Lines 11-12 of PHA execute one parallel round.
4. Lines 13-16 of PHA execute $t-1$ parallel rounds.
5. We consider Line 17 of PHA to be a special parallel round.

From this we get the following result.

Theorem 1. *Algorithm PHA executes $q+t+2$ parallel rounds.*

Each of the first $(q+t+1)$ parallel rounds consist of a formatting phase and a hashing phase. In the formatting phase, the formatting processor P_F runs a formatting subroutine and in the hashing phase the processors P_i $(0 \leq i \leq 2^t-1)$ are operated in parallel. Denote by $z_{i,j}$ the state of the buffer z_i at the end of round j, $0 \leq i \leq 2^t-1$, $1 \leq j \leq q+t+2$. Clearly, the state of the buffer z_i at the start of round j $(2 \leq j \leq q+t+2)$ is $z_{i,j-1}$. Further, let $u_{i,j}$ be the string written to buffer u_i in round j by the processor P_F. For $0 \leq i \leq 2^{t-1}-1$, the input to processor P_i in round j is $z_{2i,j-1}||z_{2i+1,j-1}||u_{i,j}$. For $2^{t-1} \leq i \leq 2^t-1$, the input to processor P_i in round j is the string $u_{i,j}$.

Theorem 2 (Correctness of PHA).

1. *Algorithm PHA terminates and provides an m-bit message digest.*
2. *Algorithm PHA provides each bit of the message x as part of the input to precisely one invocation of the hash function h.*

Proposition 1. *Let $\psi(L)$ be the number of invocations of h by PHA on a padded message of length $L = \delta(t) + q\lambda(t) + b(2n-2m)$. Then $\psi(L) = (q+2)2^t + 2b - 1$. Moreover, $\psi(L)$ is also the number of invocations of h made by the sequential MD algorithm.*

Remark: The time taken by the MD algorithm is proportional to the number of invocations of h whereas the time required by PHA is proportional to the number of parallel rounds. This is the basis for the speed-up obtained by PHA.

Proposition 2. *The maximum amount of padding added to any message is less than $2n-2m$.*

3.4 Security Reduction

In this section we provide a Turing reduction of $Col(n,m)$ to $FLC(n,m,L)$. This will show that if it is computationally difficult to find collisions for h, then it is also computationally difficult to find collisions for h_L. We provide only a sketch of the proof. The detailed proof can be found in [7].

Theorem 3. *Let h be an (n,m) hash function and for $L \geq n$ let h_L be the function defined by algorithm PHA. If there is an (ϵ,p,L) algorithm $\mathcal{A}$ to solve $FLC(n,m,L)$ for the hash function h_L, then there is an $(\epsilon, p+2\psi(L))$ algorithm $\mathcal{B}$ to solve $Col(n,m)$ for the hash function h.*

Sketch of proof: The idea of the proof is to show that if $\mathcal{A}$ can find x and x' such that $x \neq x'$ but $h_L(x) = h_L(x')$, then one can find w and w' such that $w \neq w'$ but $h(w) = h(w')$. The proof proceeds in the following manner. The output of PHA on input x is $h_L(x) = z_{0,q+t+2}$ and the output of PHA on input x' is $h_L(x') = z'_{0,q+t+2}$. Assume that $b > 0$ (the case $b = 0$ is similar). In this case the inputs to processor P_0 in round $q+t+2$ are $z_{0,q+t+1}||z_{1,q+t+1}||u_{0,q+t+2}$ and $z'_{0,q+t+1}||z'_{1,q+t+1}||u'_{0,q+t+2}$ corresponding to strings x and x' respectively. If $z_{0,q+t+1}||z_{1,q+t+1}||u_{0,q+t+2} \neq z'_{0,q+t+1}||z'_{1,q+t+1}||u'_{0,q+t+2}$, then we have a collision for h. Otherwise $z_{0,q+t+1} = z'_{0,q+t+1}$, $z_{1,q+t+1} = z'_{1,q+t+1}$ and $u_{0,q+t+2} = u'_{0,q+t+2}$. Note that $u_{0,q+t+2}$ and $u'_{0,q+t+2}$ are substrings of x and x' respectively. Thus not obtaining a collision for h at this stage implies a certain portion of x and x' are equal. At this point we use an reverse induction on round number to show that if there is no collision for h, then $x = x'$. Since by assumption we have $x \neq x'$, we must find a collision for h. $\square$

4 Variable Length Input

In the previous section we developed composition schemes which work for fixed input lengths. More precisely, given $h : \{0,1\}^n \rightarrow \{0,1\}^m$ and a positive integer L, we have shown how to construct $h_L : \{0,1\}^L \rightarrow \{0,1\}^m$. We now extend this to $h^* : \cup_{L=n}^{N}\{0,1\}^L \rightarrow \{0,1\}^m$, where $N = 2^{n-m} - 1$. For $0 \leq i \leq 2^s - 1$, let $bin_s(i)$ be the s-bit binary expansion of i. We treat $bin_s(i)$ as a binary string of length s. Then $h^*(x)$ is defined as follows.

$$h^*(x) = h\left((bin_{n-m}(|x|))||(h_{|x|}(x))\right). \tag{2}$$

In other words, we first apply $h_L(x)$ (where $|x| = L$) on x to obtain an m-bit message digest w. Let $v = bin_{n-m}(|x|)$. Then v is a bit string of length $n - m$. We apply h to the string $v||w$ to get the final message digest.

Remark: 1. We do not actually require the length of the message to be $< 2^{n-m}$. The construction can easily be modified to suit strings having length $< 2^c$ for some constant c. Since we are assuming $n \geq 2m$ and $m \geq 128$ for practical hash functions, choosing $c = n - m$ is convenient and sufficient for practical purposes.
2. The construction can be modified to tackle arbitrary length strings. For the usual Merkle-Damgård procedure this is described in [9]. We will provide the extension to arbitrary length strings for our construction in the full version of the paper.

Proposition 3. *Let τ_h and $\tau_h(L)$ be the time taken to compute h and h_L respectively. Then the time taken to compute h^* is $\tau_h(L) + \tau_h$ and the number of invocations of h is $1 + \psi(L)$.*

Theorem 4. *Let h be an (n,m) hash function and h^* be the function defined by Equation 2. If there is an (ϵ, p, L) algorithm $\mathcal{A}$ to solve $VLC(n,m,L)$ for the hash function h^*, then there is an $(\epsilon, p + 2 + 2\psi(L))$ algorithm $\mathcal{B}$ to solve $Col(n,m)$ for the hash function h.*

5 Concluding Remarks

We have considered only one property of hash functions - namely intractibility of finding collisions. There are other desirable properties that a hash function must satisfy. These are Zero Preimage and Preimage (see [8]). In [8], reductions between these properties have been studied. In our case, we are required to show that our constructions preserve the intractibility of these problems. In fact, these properties are indeed preserved and the proofs will be provided in the full version of the paper.

The second important point is that we have considered the processors to be organised as a binary tree. In fact, the same technique carries over to k-ary trees, with the condition that $n \geq km$. The computation can be made even faster by moving from binary to k-ary processor trees. However, the formatting processor will progressively become more complicated and will offset the advantage in speed up. Hence we have not explored this option further.

References

1. M. Bellare and P. Rogaway. Collision-resistant hashing: towards making UOWHFs practical. *Proceedings of CRYPTO 1997*, pp 470-484.
2. I. B. Damgård. A design principle for hash functions. *Lecture Notes in Computer Science*, 435 (1990), 416-427 (Advances in Cryptology - CRYPTO'89).
3. R. C. Merkle. One way hash functions and DES. *Lecture Notes in Computer Science*, 435 (1990), 428-226 (Advances in Cryptology - CRYPTO'89).
4. I. Mironov. Hash functions: from Merkle-Damgård to Shoup. *Lecture Notes in Computer Science*, 2045 (2001), 166-181 (Advances in Cryptology - EUROCRYPT'01).
5. M. Naor and M. Yung. Universal one-way hash functions and their cryptographic aplications. *Proceedings of the 21st Annual Symposium on Theory of Computing*, ACM, 1989, pp. 33-43.
6. B. Preneel. The state of cryptographic hash functions. *Lecture Notes in Computer Science*, 1561 (1999), 158-182 (Lectures on Data Security: Modern Cryptology in Theory and Practice).
7. P. Sarkar and P. J. Schellenberg. A parallel algorithm for extending cryptographic hash functions. CACR Technical Report, University of Waterloo, http://www.cacr.math.uwaterloo.ca
8. D. R. Stinson. Some observations on the theory of cryptographic hash functions. IACR preprint server, http://eprint.iacr.org/2001/020/.
9. D. R. Stinson. *Cryptography: Theory and Practice*, CRC Press, 1995.
10. M. N. Wegman and J. L. Carter. New Hash Functions and Their Use in Authentication and Set Equality. *Journal of Computer and System Sciences*, 22(3): 265-279 (1981)