A Parallelizable Design Principle for Cryptographic Hash Functions*

Palash Sarkar [†]	Paul J. Schellenberg
Cryptology Research Centre	Centre for Applied Cryptographic Research
Applied Statistics Unit	Department of Combinatorics and Optimization
Indian Statistical Institute	University of Waterloo
203, B.T. Road	200 University Avenue West
Kolkata 700108	Waterloo, Ontario
West Bengal, India	Canada N2L 3G1
e-mail: palash@isical.ac.in	pjschell@math.uwaterloo.ca

Contents

1	1 Introduction							
2	Basics							
	2.1	Hash Functions	4					
	2.2	Processor Tree	5					
	2.3	Parameters and Notation	5					
3	Para	allel Hashing Algorithm	6					
	3.1	Formatting Algorithms	7					
	3.2	Simulating Trees	9					
4	Para	allel Hash Function Definitions	10					
	4.1	Definition of h_L	10					
	4.2	Definition of h^*	11					
	4.3	Specifying Parallelism	11					
5	\mathbf{Cor}	rectness and Complexity of PHA	12					
6	Sec	urity Reductions for h_L and h^*	18					
	6.1	Collision Resistance of h_L	18					
	6.2	Collision Resistance of h^*	20					
7	Con	struction of h^∞	21					

^{*}An earlier abridged version of the paper appeared in the Proceedings of Indocrypt 2001, LNCS 2247, pages 40-49. † Part of the work was done while the author was visiting the Centre for Applied Cryptographic Research, University of Waterloo.

8 Preimage Resistance

9 Concluding Remarks

Abstract

We describe a parallel design principle for hash functions. Given a secure hash function $h: \{0,1\}^n \to \{0,1\}^m$ with $n \ge 2m$, and a binary tree of 2^t processors we show how to construct a secure hash function h^* which can hash messages of lengths less than 2^{n-m} and a secure hash function h^∞ which can hash messages of arbitrary length. The number of parallel rounds required to hash a message of length L is $\lfloor \frac{L}{2^t} \rfloor + t + 2$. Further, our algorithm is incrementally parallelizable in the following sense : given a digest produced using a binary tree of 2^t processors, we show that the same digest can also be produced using a binary tree of $2^{t'}$ $(0 \le t' \le t)$ processors.

Keywords : cryptographic hash function, Merkle-Dam $\overset{\circ}{g}$ ard construction, parallel algorithm, collision resistance, preimage resistance, second preimage resistance, zero preimage resistance.

1 Introduction

Hash functions are extensively used in cryptographic protocols. One of the main uses of hash functions is in digital signature protocols, where the message digest produced by the hash function is signed. Due to the central importance of hash functions in cryptography, there has been a lot of work in this area. See [7] for a survey.

For a hash function $h : \{0, 1\}^n \to \{0, 1\}^m$ to be used in cryptographic protocols, it must satisfy certain well known necessary properties. In a recent paper [8], Stinson provides a comprehensive discussion of these properties and also relations among them. The important properties that a cryptographic hash function must satisfy are the following.

- (a) **Preimage Resistance :** Finding a preimage of a given message digest must be computationally infeasible. In other words, given $z \in \{0, 1\}^m$ it should be computationally infeasible to find $x \in \{0, 1\}^n$ such that h(x) = z.
- (b) Second Preimage Resistance : Finding a second preimage of a digest given one preimage of the same digest must be computationally infeasible. In other words, given $x \in \{0, 1\}^n$ and $z \in \{0, 1\}^m$ such that h(x) = z, it should be computationally infeasible to find $y \in \{0, 1\}^n$ such that $x \neq y$ and h(y) = z.
- (c) Collision Resistance : Finding a collision must be computationally infeasible. In other words, it should be computationally infeasible to find $x, y \in \{0, 1\}^n$ such that $x \neq y$ but h(x) = h(y).

It is clear that if it is possible to find second preimage, then it is possible to find collisions. Hence it is usually sufficient to study collision resistance. However, as pointed out in [8], there is no satisfactory reduction from collision resistance to preimage resistance or vice versa. Hence the goal of a practical hash function should be to achieve both preimage and collision resistance.

It is possible to construct hash functions where one can prove that finding collisions is equivalent to solving certain known hard problems (see for example [2]). However, from a practical point of view such hash functions are unacceptably slow. Hence practical hash functions are constructed

 $\mathbf{24}$

from simple arithmetic/logical operations so that they are very fast. The trade-off is that for such hash functions it is not possible to relate the difficulty of finding collisions to known hard problems.

Research in design of hash functions have evolved certain principles for designing "secure" and practical hash functions. One of the important papers in this area is by Damgard [3]. An important point made in [3] is that it is easier to design a "secure" hash function with a short fixed domain than a hash function with a very large (or infinite) domain. However, for a hash function to be useful it must be possible to hash arbitrary long messages. Hence one must look for techniques that can extend the domain of a hash function while preserving the relevant security properties.

An important construction for securely extending the domain of a secure hash function has been described by Merkle [4] and Damgard [3]. The construction is called the Merkle-Damgard (MD) construction. The MD construction is a sequential construction and provides a basic guideline for designing practical hash functions.

In this paper we develop an alternative design principle for securely extending the domain of a secure hash function. Our design principle is based on a binary tree of processors and allows for parallelism in the computation of the hash function. We show that given a secure hash function $h : \{0,1\}^n \to \{0,1\}^m$ with $n \ge 2m$ and a binary tree of 2^t processors, it is possible to construct a secure hash function h^* which can hash messages on lengths less than 2^{n-m} and a secure hash function h^∞ which can hash arbitrary length messages. Since we require $n \ge 2m$ and practical hash functions have $m \ge 128$, the function h^* is adequate for any conceivable application and the construction of h^∞ is of theoretical interest only. The number of parallel rounds to compute the digest of a message of length L is $\lfloor \frac{L}{2^t} \rfloor + t + 2$.

Our design principle allows for incremental parallelism in the following sense. If a message digest can be produced using a binary tree of 2^t processors, then the same message digest can be produced using a binary tree of $2^{t'}$ processors for $0 \le t' \le t$ with a proportional loss in speed of computation. In the extreme case of t' = 0 this means that using a single processor it is possible to produce a digest which has been produced using a binary tree of 2^t processors for any $t \ge 0$. We stress that this is an extremely important point for practical application of our design principle. In a multi-user setting where different users have different resource capabilities, it is important that a digest produced by one user can be produced by any other user irrespective of the amount of resources available to him.

Related Work : The concept of tree hashing has appeared before in the literature. Damgard [3] showed that for a message of length n, it is possible to compute the digest in $O(\log n)$ steps using O(n) processors. Note that the number of processors is proportional to the length of the message. Hence the result yields an impractical algorithm. Tree hashing has also been considered in relation to universal one-way hash functions [6, 1]. However, these papers also assume a model where the number of processors grows with the length of the message.

Our model improves upon the previous work on tree hashing in the following two ways.

- 1. In our model the number of processors is fixed while the length of the message can be very long.
- 2. A digest which can be produced by a binary tree with a certain number of processors can also be produced by a binary tree with lesser number of processors and in the extreme case by a single processor.

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 $h : \{0,1\}^n \to \{0,1\}^m$. Throughout this paper we require that $n \ge 2m$. 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 $L \ge 1$, we construct a hash function $h_L : \{0,1\}^L \to \{0,1\}^m$. The next step, in general, is to construct a hash function $h^{\infty} : \bigcup_{L\ge 1} \{0,1\}^L \to \{0,1\}^m$. However, instead of doing this, we first construct a hash function $h^* : \bigcup_{L=1}^N \{0,1\}^L \to \{0,1\}^m$. However, instead of doing this, we first construct a hash function $h^* : \bigcup_{L=1}^N \{0,1\}^L \to \{0,1\}^m$, where $N = 2^{n-m} - 1$. Since we assume $n \ge 2m$, we have $n - m \ge m$. Practical message digests are at least 128 bits long meaning that m = 128. Hence our construction of h^* can handle any message with length less than 2^{128} . This is sufficient for any conceivable application. The construction of h^{∞} presents certain technical difficulties. We overcome these difficulties and describe the construction of h^{∞} in Section 7.

We would like to relate the difficulty of finding collisions for h_L, h^* and h^{∞} to that of finding a collision for h. Thus we consider the following problems.

Problem	:	Fixed length collision $FLC(n,m,L)$
Instance	:	An (n, m) hash function h and an integer $L \ge n$.
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 $n \le L \le 2^{n-m}$.
Find	:	$x, x' \in \bigcup_{i=n}^{L} \{0, 1\}^{i}$ such that $x \neq x'$ and $h^{*}(x) = h^{*}(x')$.

Problem	:	Arbitrary length collision $ALC(n,m,L)$
Instance	:	An (n, m) hash function h and an integer $L \ge n$.
Find	:	$x, x' \in \bigcup_{i=n}^{L} \{0, 1\}^i$ such that $x \neq x'$ and $h^{\infty}(x) = h^{\infty}(x')$.

By an (ϵ, p, L) (randomized) algorithm \mathcal{A} for Fixed length collision we will mean an algorithm that requires at most p invocations of the function h and solves Fixed 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 . Similar definitions are true for Variable length collision and Arbitrary length collision.

Later we show Turing reductions from Collision to Fixed length collision, Variable length collision and Arbitrary 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^* or ALC(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 $P_{2t-1}, \ldots, P_{2^t-1}$ 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 -

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

For $0 \le i \le 2^t - 1$, 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 $0 \le i \le 2^{t-1} - 1$, 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 $2^{t-1} \le i \le 2^t - 1$, the input buffer of P_i is u_i . The output buffer of P_i is z_i for $0 \le i \le 2^t - 1$.

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 $0 \le i \le 2^t 1$.
- 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. In the construction of h_L we will not always use the processor tree upto depth T. We will denote by t the depth of the processor tree used. When the processor tree is used upto depth t, the number of processors used is 2^t . Next we describe several parameters with respect to t - the useful depth of the processor tree.

Start-up length: $2^t n$.

Flushing length: $(2^{t-1} + 2^{t-2} + \dots + 2^1 + 2^0)(n - 2m) = (2^t - 1)(n - 2m).$ 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 $L \ge n$.

Parameters q_t , b_t and r_t :

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

2. If
$$L = \delta(t)$$
, then $q_t = b_t = r_t = 0$.

Note that $0 \le b_t \le 2^{t-1}$. We will denote the empty string by $\langle \rangle$ and the length of a binary string y by |y|.

3 Parallel Hashing Algorithm

We first describe a parallel hashing algorithm which is the basic building block used for the construction of hash functions. The main algorithm uses other algorithms as subroutines which are described later. Before presenting the actual algorithm we present the basic idea behind the algorithm.

Let x be a message of length L and \mathcal{T} be the binary tree of processors of depth t as described in Section 2.2. There are also two sets of 2^t buffers z_0, \ldots, z_{2^t-1} and u_0, \ldots, u_{2^t-1} . Each of the buffers z_i can store m-bit strings. For $0 \leq i \leq 2^{t-1} - 1$, the buffer u_i stores either an (n-2m)-bit string or the empty string and for $2^{t-1} \leq i \leq 2^t - 1$, the buffer u_i stores either an n-bit string or the empty string. Each buffer z_i stores the output of processor P_i . The buffers u_i are obtained as prefixes from the message x.

The algorithm consists of some parallel rounds where in each parallel round all the 2^t processors operate in parallel. Further, in each of the parallel rounds the message x is shortened by removing a prefic from it. This prefix is divided into substrings and copied to the buffers u_i .

Initially all the buffers z_i are empty. Thus the first step of the algorithm is to initialise the z_i 's which is done in the following manner. Each processor P_i is given an *n*-bit string u_i as input. Processor P_i hashes u_i to produce the digest z_i . This step is called Start-Up.

The algorithm then enters the Steady-State. In the Steady-State, processors $P_0, \ldots, P_{2^{t-1}-1}$ gets an (n-2m)-bit input u_i . Also P_i reads the buffers z_{2i} and z_{2i+1} . Processor P_i then forms an input of length n by concatenating z_{2i}, z_{2i+1} and u_i . This n-bit string is hashed to obtain the new value of the buffer z_i . The processors $P_{2^{t-1}}, \ldots, P_{2^{t}-1}$ each get an n-bit input which is hashed to obtain the new values of the buffers $z_{2^{t-1}}, \ldots, z_{2^{t}-1}$. The Steady-State lasts for a certain number of rounds which we determine later. It is clear that after a certain stage it will not be possible to provide inputs to all the processors.

After the Steady-State ends we have a single round called the End-Game. This round starts the mopping up operation. In this round only some of the leaf level processors get *n*-bit strings as input while all other processors get the empty string as input. In this round each of the internal processors still get an (n - 2m)-bit input.

After the End-Game, there are (t-1) rounds which flush the processor tree. The flushing proceeds in a bottom-up fashion. In the *i*th stage of the flushing operation all processors at level $\geq s - i + 1$ get empty strings as inputs. Some of the processors at level s - i get an (n - 2m)-bit string as input. The rest of the processors at level s - i get the empty string as input. All processors at levels $\leq s - i - 1$ get an (n - 2m)-bit string as input. This stage is called the Flusing stage.

At the end of the Flushing stage, z_0 and z_1 are *m*-bit strings while all other buffers are empty strings. Further, the remaining part of x is an (n - 2m)-bit string. Processor P_0 applies the hash function to the *n*-bit string $z_0 ||z_1||x$ to obtain the final message digest.

We now present the formal description of the algorithm.

Parallel Hashing Algorithm (PHA(x,t))

Inputs:

- (1) message x of length $L \ge \delta(t)$.
- (2) $t (\leq T)$ is the depth up to which the processor tree must be used.

Output: message digest $h_L(x)$ of length m.

Define: $q = q_t$, $r = r_t$ and $b = b_t$.

1. <u>if</u> $L > \delta(t)$, <u>then</u> 2. $x := x ||0^{b(2n-2m)-r}|$

(ensures that the length of the message becomes $\delta(t) + q\lambda(t) + b(2n - 2m)$.)

- $3. \underline{\text{endif}}.$
- 4. Initialise buffers z_i and u_i to empty strings, $0 \le i \le 2^t 1$.
- 5. <u>Do</u> FormatStartUp.
- 6. <u>Do</u> ParallelProcess.
- 7. $\underline{\text{for}}$ $i = 1, 2, \dots, q \underline{\text{do}}$
- 8. $\underline{\text{Do}}$ FormatSteadyState.
- 9. <u>Do</u> ParallelProcess.
- 10. \underline{endfor}
- 11. <u>Do</u> FormatEndGame.
- 12. <u>Do</u> ParallelProcess.
- 13. <u>for</u> $s = t 1, t 2, \dots 2, 1$ <u>do</u>
- 14. <u>Do</u> FormatFlushing(s).
- 15. <u>Do</u> ParallelProcess.
- 16. \underline{endfor}

```
17. z_0 = P_0(z_0||z_1||x).
```

- 18. return z_0 .
- 19. end algorithm PHA

We now describe the different subroutines used by PHA. We assume that the message x is globally manipulated by the different formatting algorithms and the input t of PHA is available to all the subroutines. Further, we assume that the parameter b is available to the subroutines FEG and FF.

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

1. <u>for</u> $i = 0, ..., 2^{t} - 1$ <u>do in parallel</u> 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. <u>endfor</u> 5. **end algorithm PP**

3.1 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 $0 \le i \le 2^t - 1$. The message x is available as either an array or a file. We assume that the message is read sequentially bit by bit. The formatting algorithms copy a prefix of the message into a buffer and suitably advance the file (or array) pointer. 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. <u>for</u> $i = 0, ..., 2^t - 1$ <u>do</u> 2. Write x = v ||y|, where |v| = n. 3. $u_i := v$. 4. x := y. 5. <u>endfor</u> 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.

```
i = 0, \dots, 2^{t-1} - 1 do
1.
     for
               Write x = v || y, where |v| = n - 2m.
2.
3.
               u_i := v.
4.
               x := y.
    \underline{endfor}
5.
               i = 2^{t-1}, \dots, 2^t - 1 do
6.
     for
               Write x = v || y, where |v| = n.
7.
8.
               u_i := v.
9.
               x := y.
10. endfor
11. end algorithm FSS
```

FormatEndGame (FEG)

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

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

FormatFlushing(s) (FF(s))

Input: Integer s.

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

Remark: 1. The assignments x := y is an assignment of the relevant file or array pointer and can be done in constant time.

2. If n = 2m, then $u_i = <>$ for $0 \le i \le 2^{t-1} - 1$. This significantly simplifies the formatting algorithms. Thus if one is allowed to choose the parameters n and m, then it is best to choose n to be equal to 2m.

3.2 Simulating Trees

One potential problem in the use of PHA to generate a message digest is the fact that the verifier might not have access to a binary tree of processors or he might have access to a binary tree of a lesser height. In such a situation, it will not be possible to verify the message digest. We show how this problem can be solved by allowing a smaller tree of processors to simulate a larger tree of processors. A more detailed discussion of this issue is given in Section 4.3.

Let t, t' be two non-negative integers with t > t'. Let \mathcal{T} (resp. \mathcal{T}') be a tree of depth t (resp. t') consisting of 2^t (resp. $2^{t'}$) processors $P_0, \ldots, P_{2^{t}-1}$ (resp. $P'_0, \ldots, P'_{2^{t'}-1}$) connected in the manner described in Section 2.2. Let y = PHA(t, x) be produced by the processor tree \mathcal{T} . We describe an algorithm SimPar(t, t', x) which also produces y using the processor tree \mathcal{T}' .

$\mathbf{SimPar}(t, t', x)$

Input:

- (1) message x of length $L \ge \delta(t)$.
- (2) t is the depth of the original processor tree.
- (3) t' is the depth of the available processor tree.

Output: message digest $h_L(x) = PHA(t, x)$ of length m. The algorithm is identical to PHA(t, x) with the following changes.

- 1. Change Lines 6,9 and 12 to "Do SPP(t, t')".
- 2. Change Line 15 to "Do SPP(s, t')".

end algorithm SimPar

The subroutine SPP() perform the task of simulating the processor tree \mathcal{T} using the tree \mathcal{T}' . For the first q + 2 rounds the entire tree \mathcal{T} needs to be simulated. However, for the next t - 1 rounds we need to simulate \mathcal{T} only up to depth s. We define the subroutine SPP() to do these two tasks.

Algorithm SPP(s, t')

 $\underbrace{ \text{for } i = 0 \text{ to } 2^{t'-1} \text{ do in parallel} }_{ \text{use processor } P'_i \text{ to execute the task of processor } P_i.$ 1. 2.3. <u>endfor</u> $\underline{\text{if}} s \leq t' \text{ then stop.}$ 4. for $j_1 = 0$ to s - t' - 1 do 5. $\underline{\text{for } j_2 = 0 \text{ to } 2^{j_1} - 1 \underline{\text{do}}}$ 6. $i_1 = 2^{t'+j_1} + j_2(2^{t'})$ 7. for i = 0 to $2^{t'} - 1$ do in parallel 8. use processor P'_i to execute the task of processor P_{i_1+i} . 9. 10. <u>endfor</u> endfor 11. 12.endfor

13. end Algorithm SSP1.

Proposition 2 The number of parallel rounds required by SPP(s, t') is equal to one if $s \leq t'$ and is equal to $2^{s-t'}$ if s > t'.

Proof. If $s \leq t'$, then the result is obvious. If s > t', then the number of rounds required is $1 + (1 + 2 + 2^2 + \ldots + 2^{s-t'-1}) = 2^{s-t'}$.

Remark: If there is only one processor (i.e., \mathcal{T}' consists only of P'_0), then the number of rounds required by SPP(s, 0) is 2^s .

4 Parallel Hash Function Definitions

The base hash function is $h : \{0,1\}^n \to \{0,1\}^m$, with $n \ge 2m$. If x is a binary string with |x| < n, then we apply the hash function h to the string $x||0^{n-|x|}$ to get the message digest. Thus effectively h is a map from $\bigcup_{i=1}^n \{0,1\}^i$ to $\{0,1\}^m$. The description of h_L and h^* is described below.

4.1 Definition of h_L

Let $L \ge 1$ be a positive integer and assume that a binary tree of 2^T processors is available. Then h_L is defined as follows.

$$h_{|x|}(x) = \begin{cases} PHA(T,x) & \text{if } |x| \ge \delta(T); \\ PHA(t,x) & \text{if } 0 < t < T \text{ and } \delta(t) \le |x| < \delta(t+1); \\ PHA(0,x) & \text{if } \delta(0) < |x| < \delta(1) \\ h(x) & \text{if } 1 \le |x| \le n = \delta(0). \end{cases}$$
(2)

When t < T we are not utilizing all the available processors. We now show that not utilizing all the processors leads to at most one extra round when t = T - 1. We first note the following result. Recall the definition of q_t from Definition 1.

Lemma 3 If $\delta(t) \leq L < \delta(t+1)$, then $0 \leq q_t \leq 1$.

Proof. We use the following two facts, which are easy to verify.

1. $\delta(t+1) = 2\delta(t) + n - 2m$.

2. $2\lambda(t) = \delta(t) + n - 2m$.

Since $\delta(t) \leq L < \delta(t+1)$, we have $0 \leq \left\lfloor \frac{L-\delta(t)}{\lambda(t)} \right\rfloor < \left\lfloor \frac{\delta(t+1)-\delta(t)}{\lambda(t)} \right\rfloor$. Using the above two facts we get $0 \leq \left\lfloor \frac{L-\delta(t)}{\lambda(t)} \right\rfloor < 2.$

From the definition of q_t this shows that $0 \leq q_t \leq 1$.

The processor tree has T levels plus an additional level containing only P_0 . Thus if any processor at the leaf level is used, then at least T + 1 rounds will be required to obtain a message digest. On the other hand using the processor tree upto level t requires $t + q_t + 2$ rounds (see Theorem 5 below). Thus not utilizing all the processors require extra parallel rounds only if $t + q_t + 2 > T + 1$. By choice of t and Lemma 3, we have $q_t = 1$. Thus extra parallel round is required when t > T - 2. Since $t \leq T - 1$, we get that t = T - 1. Further, in this case only one extra round will be required.

4.2 Definition of h^*

Given $h: \bigcup_{i=1}^{n} \{0,1\}^i \to \{0,1\}^m$ and a positive integer $L \ge 1$, Equation (2) defines the function $h_L: \{0,1\}^L \to \{0,1\}^m$. We now extend this to $h^*: \bigcup_{L=1}^{N} \{0,1\}^L \to \{0,1\}^m$, where $N = 2^{n-m} - 1$. For $0 \le i \le 2^s - 1$, 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.

$$h^*(x) = h\left((bin_{n-m}(|x|))||(h_{|x|}(x))\right).$$
(3)

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 $n \ge 2m$ and $m \ge 128$ for practical hash functions, choosing c = n - m is convenient and sufficient for practical purposes.

2. In Section 7, we present the construction for arbitrary length strings.

4.3 Specifying Parallelism

We consider the following problem. Suppose a set of users agree to choose $h^*()$ as a hash function standard. The message digest produced on a message clearly depends on the depth of the binary tree used to generate the message digest. Suppose a user generates the digest using a binary tree of depth t. Then any other user who needs to regenerate the digest has to have access to a binary of depth t or should be able to simulate the binary tree of depth t. It is quite possible that the user has access to only one processor. In this case also the user should be able to generate the message digest. This can be ensured in any one of the following two ways.

(1) The depth T of the processor tree is fixed and is part of the hash function specification. Then any user who needs to generate y = PHA(T, x) and has access to a processor tree of depth t, with t < T uses SimPar(T, t, x) to generate y. If $t \ge T$, then the user can run PHA(T, x) by not using processors at level greater than T.

(2) The depth of the processor tree is not part of the hash function specification. In this case the actual depth of the processor tree is output with the message digest, i.e. the output on input x is (t, PHA(t, x)). Any other user who wishes to regenerate the digest and has access to a tree of depth t' runs SimPar(t, t', x) if t < t' or runs PHA(t, x) if $t \ge t'$.

Depending on the situation in hand any one of the above two strategies may be adopted. We would like to highlight another aspect of Strategy 2. Suppose User 1 has only a single processor and wishes to compute the digest on a message x. User 1 also knows that the digest will be recomputed by User 2 who has access to a processor tree of 2^t (t > 0) processors. User 1 then invokes SimPar(t, 0, x) to compute y = PHA(t, x). Thus User 2 can directly use his processor tree of 2^t processors to invoke PHA(t, x) and recompute y. In this manner the total time required to compute both the digests is minimized.

Fundamentally our design principle follows the simple basic rule : Users with more resources can speed up computation of the digest, without affecting the efficiency of users with lesser resources to compute the same digest.

5 Correctness and Complexity of PHA

Here we consider the correctness and complexity of computing h_L . In Section 6.1 we will provide the security reduction of Col(n, m) to FLC(n, m, L). By correctness of h_L we mean that every bit of the message is hashed and algorithm PHA outputs an *m*-bit message digest.

The following result shows that the maximum amount of padding added to a message depends only on the parameters n and m. In particular, the maximum amount of padding is indepedent of the number of processors and the length of the message.

Proposition 4 The maximum amount of padding added to any message is less than 2n - 2m.

Proof. The only place where padding (if any) is done is at line 2 of algorithm PHA. The amount of padding is b(2n-2m)-r. Since $b = \left\lceil \frac{r}{2n-2m} \right\rceil < \frac{r}{2n-2m} + 1$, we have b(2n-2m)-r < 2n-2m.

Remark : Using a naive padding scheme will result in the padding length being proportional to $2^t(2n - 2m)$. This will result in many zeros being appended to the message which is clearly an undesirable feature. Further, it is not difficult to verify that the use of a naive padding scheme does not reduce the number of parallel rounds required and neither does it make the parallel hashing algorithm any simpler. Thus we discourage the use of a naive padding scheme.

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 5 Algorithm PHA(t, x) executes q + t + 2 parallel rounds. Consequently, Algorithm SimPar(t, t', x) executes $(q + 3)2^{t-t'} + t$ parallel rounds.

Each of the first (q+t+1) parallel rounds in PHA(t, x) 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 $(0 \le i \le 2^t - 1)$ are operated in parallel. Denote by $z_{i,j}$ the state of the buffer z_i at the end of round j, $0 \le i \le 2^t - 1$, $1 \le j \le q + t + 2$. Clearly, the state of the buffer z_i at the start of round j $(2 \le j \le q + t + 2)$ 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 $0 \le i \le 2^{t-1} - 1$, the input to processor P_i in round j is $z_{2i,j-1}||z_{2i+1,j-1}||u_{i,j}$. For $2^{t-1} \le i \le 2^t - 1$, the input to processor P_i in round jis the string $u_{i,j}$.

The following lemma and corollary are required to prove Proposition 8.

Lemma 6 For any nonnegative integer b, $\sum_{i\geq 1} \left\lfloor \frac{b+2^{i-1}}{2^i} \right\rfloor = b$.

Proof. We prove this result by mathematical induction on *b*. Clearly the result holds for b = 0. Induction Hypothesis: For *b* a nonnegative integer, assume that $\sum_{i\geq 1} \left\lfloor \frac{b+2^{i-1}-1}{2^i} \right\rfloor = b-1$.

It can be shown that

$$\left\lfloor \frac{m}{n} \right\rfloor = \begin{cases} \left\lfloor \frac{m-1}{n} \right\rfloor + 1 & \text{when } n \mid m, \\ \\ \left\lfloor \frac{m-1}{n} \right\rfloor & \text{otherwise.} \end{cases}$$

In addition, $2^{i}|(b+2^{i-1})$ if and only if $b = 2^{i-1}c$ where c is an odd integer. Combining these facts with the induction hypothesis, we get that

$$\sum_{i \ge 1} \left\lfloor \frac{b+2^{i-1}}{2^i} \right\rfloor = 1 + \sum_{i \ge 1} \left\lfloor \frac{b+2^{i-1}-1}{2^i} \right\rfloor = b.$$

Thus, by mathematical induction, we conclude that the result holds for all nonnegative integers b.

Corollary 7 For t a given positive integer and b an integer in the range $0 \le b \le 2^{t-1}$, let $k_s = \lfloor \frac{b+2^{t-s-1}-1}{2^{t-s}} \rfloor$ as defined in algorithm PHA. Then $\sum_{s=1}^{t-1} k_s = \sum_{s\ge 1} k_s = b-1$.

Proposition 8 Let x be a message of length $L = \delta(t) + q\lambda(t) + b(2n-2m)$, where q is a nonnegative integer and b is an integer in the range $0 \le b \le 2^{t-1}$. The formatting algorithms present every bit of message x to exactly one of the processors P_i ; furthermore, the substring x presented to processor P_0 in step 17 of PHA is the empty string $\langle \rangle$ when $|x| = \delta(t)$ and is an (n-2m)-bit string when $|x| > \delta(t)$. The formatting algorithms require

- (a) $|x| (n-2m) + (t-1)2^t 2b + 2$ steps when $|x| > \delta(t)$ or
- (b) $|x| + (t-1)2^t + 1$ steps when $|x| = \delta(t)$.

Proof.

Each formatting algorithm defines $u_i = \langle \rangle$ or else defines u_i to be a prefix of x; namely,

 $\begin{aligned} x &= v || y \\ u_i &= v \end{aligned}$

x = yIn step 17, the substring x itself is presented to processor P_0 . Hence, every bit of message x is presented to exactly one processor P_i , as claimed. We now determine the length of the substring x presented to processor P_0 in step 17.

Formatting algorithm FSU provides a prefix of length n to each processor P_i . This accounts for $2^t n$ bits of x. Algorithm FSS provides an (n-2m)-bit prefix to processor P_i , $0 \le i < 2^{t-1}$, and an n-bit prefix to processor P_i , $2^{t-1} \le i < 2^t$. This accounts for $2^{t-1}(2n-2m) = \lambda(t)$ bits of x. Since FSS is invoked q times, this accounts for $q\lambda(t)$ bits of x. Formatting algorithm FEG provides each internal processor P_i , $0 \le i < 2^{t-1}$, with an (n-2m)-bit prefix of x, each leaf processor P_i , $2^{t-1} \le i \le 2^{t-1} + b - 1$, with an n-bit prefix of x, and all the other leaf processors with an empty string. This accounts for $2^{t-1}(n-2m) + bn$ bits of x. For $s = t - 1, t - 2, \ldots, 2, 1$, formatting algorithm FF(s) presents each processor P_i , $0 \le i < 2^{s-1} + k_s - 1$, where $k_s = \lfloor \frac{b+2^{t-s-1}-1}{2^{t-s}} \rfloor$, with an (n-2m)-bit prefix of x and all the other processors P_i with $u_i = <>$. This accounts to $(2^{s-1} + k_s)(n-2m)$ bits of x. The total number of bits presented to the processors P_i , $0 \le i < 2^t$, is

$$2^{t}n + q\lambda(t) + bn + 2^{t-1}(n - 2m) + \sum_{s=t-1}^{1} (2^{s-1} + k_{s})(n - 2m)$$

= $2^{t}n + q\lambda(t) + bn + \sum_{s=t}^{1} 2^{s-1}(n - 2m) + \sum_{s=t-1}^{1} k_{s}(n - 2m)$
= $2^{t}n + q\lambda(t) + bn + (2^{t} - 1)(n - 2m) + (b - 1)(n - 2m)$ (since $\sum_{s=t-1}^{1} k_{s} = b - 1$)
= $\delta(t) + q\lambda(t) + b(2n - 2m) - (n - 2m)$.

Hence, the substring x presented to processor P_0 in step 17 of PHA is of length (n-2m) as claimed.

In the special case when x is of length $L = \delta(t)$, b = q = 0. This in turn implies that $k_s = 0$ for s = t - 1, t - 2, ..., 2, 1. Hence, the total number of bits presented to the processors P_i is just $\delta(t)$, and the substring x presented to processor P_0 in step 17 of PHA is the empty string.

Formatting algorithm FEG defines $u_i = <>$ for $2^{t-1} + b \leq i < 2^t$, and, for $1 \leq s < t$, FF(s) defines $u_i = <>$ for $2^{s-1} + k_s \leq i < 2^t$. The number of assignments of the form $u_i = <>$ is

$$2^{t-1} - b + \sum_{s=t-1}^{1} (2^t - 2^{s-1} - k_s) = 2^{t-1} - b + (t-1)2^t - \sum_{s=t-1}^{1} 2^{s-1} - \sum_{s=t-1}^{1} k_s$$

= $2^{t-1} + (t-1)2^t - (2^{t-1} - 1) - 2b + 1 = (t-1)2^t - 2b + 2.$

In the special case when x has length $L = \delta(t)$, there are $(t-1)2^t + 1$ assignments of the form $u_i = <>$.

Each step of the formatting algorithms consist of moving the leading bit of string x to some buffer u_i , or else assigning $u_i = <>$. Therefore, the formatting algorithms require

(a)
$$\delta(t) + q\lambda(t) + b(2n - 2m) - (n - 2m) + (t - 1)2^t - 2b + 2$$
 steps when $L > \delta(t)$ or

(b) $\delta(t) + (t-1)2^t + 1$ steps when $L = \delta(t)$. This establishes the result.

=

We require the following lemma in the proof of Theorem 10.

Lemma 9 For any integers b and t, $b \ge 0$ and $t \ge 1$, define $k_s = \lfloor \frac{b+2^{t-s-1}-1}{2^{t-s}} \rfloor$ for $1 \le s < t$ and $l_s = \lfloor \frac{b+2^{t-s}-1}{2^{t-s}} \rfloor$ for $1 \le s \le t$. Then

(a)
$$k_s \le l_s \le k_s + 1$$
,
(b) $2k_s \le l_{s+1} \le 2l_s$, and
(c) $l_s = k_s + 1 \Rightarrow 2l_s = l_{s+1} + 1$

Proof.

Clearly,

$$k_s = \left\lfloor \frac{b-1}{2^{t-s}} + \frac{1}{2} \right\rfloor \le \left\lfloor \frac{b-1}{2^{t-s}} + 1 \right\rfloor = l_s \le \left\lfloor \frac{b-1}{2^{t-s}} + \frac{3}{2} \right\rfloor = k_s + 1.$$

For any nonnegative real number x, $2\lfloor x + \frac{1}{2} \rfloor \leq \lfloor 2x + 1 \rfloor \leq 2\lfloor x + 1 \rfloor$. Setting $x = (b-1)/2^{t-s}$, we get

$$2k_s \le l_{s+1} \le 2l_s$$

Now let $x = \frac{b-1}{2^{t-s}} = I + f$ where I is an integer and $0 \le f < 1$. Then

$$l_s = \lfloor x+1 \rfloor = \lfloor I+f+1 \rfloor = I+1.$$

If $l_s = k_s + 1$, then

$$I + 1 = l_s = k_s + 1 = \lfloor x + 1/2 \rfloor + 1 = \lfloor I + f + 1/2 \rfloor + 1 = I + 1 + \lfloor f + 1/2 \rfloor.$$

Hence $\lfloor f + 1/2 \rfloor = 0$ which means $0 \le f < 1/2$. Then

$$l_{s+1} = \lfloor 2x + 1 \rfloor = \lfloor 2I + 2f + 1 \rfloor = 2I + 1 = 2l_s - 1.$$

Observe that in round q+2, the formatting algorithm FEG defines $u_{i,q+2} = <>$ iff $2^{t-1}+b \leq i \leq 2^t$. Furthermore, in round j, q+2 < j < q+t+2, formatting algorithm FF(s) defines $u_{i,j} = <>$ iff $2^{s-1}+k_s \leq i < 2^t$ where s = q+t+2-j. In Theorem 10, we show that for $q+2 \leq j \leq q+t+2$, we have $z_{i,j} = <>$ iff $2^{s-1}+l-s \leq i < 2^t$ where s = q+t+2-j and $l_s = \left\lfloor \frac{b+2^{t-s}-1}{2^{t-s}} \right\rfloor$. The correctness of algorithm PHA depends on showing that, for q+2 < j < q+t+2, we have $u_{i,j} = <>$ iff either $2^{t-1} \leq i < 2^t$ or $z_{2i+1,j-1} = <>$. This means that

$$z_{i,j} = \begin{cases} z_{2i,j-1} & \text{whenever } u_{i,j} = <> = z_{2i+1,j-1}, \\ h(z_{2i,j-1}||z_{2i+1,j-1}||u_{i,j}) & \text{whenever } u_{ij} \neq <> \neq z_{2i+1,j-1}, \\ <> & \text{whenever } 2^{t-1} \leq i < 2^t. \end{cases}$$

In round q + 2, the formatting subroutine FEG is invoked. This subroutine defines the strings $u_{2^{t-1},q+2}, \ldots, u_{2^{t-1}+b-1,q+2}$ to be non empty and $u_{2^{t-1}+b}, \ldots, u_{2^t-1,q+2}$ to be empty strings. As a result in rounds q + 2 + l $(1 \le l \le t - 1)$ only some of the buffers $u_{i,q+2+l}$ are non empty. If $u_{i,q+2+l}$ is defined then processor P_i will get an *n*-bit input and invoke the hash function on this input. Thus in this case $z_{i,q+2+l}$ will be an *m*-bit string. Further, it may happen that $z_{2i,q+l+1}$ is an *m*-bit string but $z_{2i+1,q+l+1}$ is not an *m*-bit string. In this case, $u_{i,q+2+l}$ must be empty and processor P_i gets the *m*-bit string $z_{2i,q+l+1}$ as input which it copies to output, i.e., $z_{i,q+2+l} = z_{2i,q+l+1}$. Thus there are two things to consider.

1. The maximum value of *i* such that $u_{i,q+2+l}$ is non empty.

2. The maximum value of *i* such that $z_{i,q+2+l}$ is non empty.

Let s = t - l. Then the maximum value of *i* such that (1) happens is $2^{s-1} + k_s - 1$ and the maximum value of *i* such that (2) happens is $2^{s-1} + l_s - 1$. These two facts are crucial to the correctness of PHA and are proved as part of Theorem 10 below.

Remark : We would like to point out the connection of the values k_s and l_s respectively to the inorder successor and predecessor of the processor P_i . In round q + 2 + l = q + 2 + t - s, processor P_i outputs an *m*-bit output if and only if the inorder predecessor (which is at the leaf level) of P_i received an *n*-bit input in round q + 2. Further, in round q + 2 + l = q + 2 + t - s, processor P_i invokes the hash function (equivalently $u_{i,q+2+l}$ is defined) if the inorder successor (again at the leaf level) of P_i received an *n*-bit input in round q + 2. These considerations also provide the expressions for k_s and l_s .

Theorem 10 (Correctness of PHA) Given any message x with $|x| \ge \delta(t)$, algorithm PHA(x, t) applies hash function h to every bit of x and produces an m-bit message digest.

Proof. Let $y = z_{0,q+t+1} || z_{1,q+t+1} || u_{0,q+t+2}$. Then, the output of algorithm PHA is, by definition,

$$z_{0,q+t+2} = \begin{cases} h(y) & \text{if } |y| = n, \\ y & \text{otherwise.} \end{cases}$$

Therefore, we must show that if $|y| \neq n$, then |y| = m.

In round 1, processor P_F writes *n*-bit strings to each of the buffers u_i , i.e., $|u_{i,1}| = n$ for $0 \le i \le 2^t - 1$. Hence $|z_{i,1}| = m$ for $0 \le i \le 2^t - 1$. Further, it is easy to verify that for $2 \le j \le q+1$, we have $|z_{i,j}| = m$ for $0 \le i \le 2^t - 1$ and

$$u_{i,j}| = \begin{cases} n - 2m & \text{if } 0 \le i \le 2^{t-1} - 1; \\ n & \text{if } 2^{t-1} \le i \le 2^t - 1. \end{cases}$$

For $q+2 \leq j \leq q+t+1$, let s = q+t+2-j. Then $t \geq s \geq 1$ corresponding to $q+2 \leq j \leq q+t+1$. Define $l_s = \left\lfloor \frac{b+2^{t-s}-1}{2^{t-s}} \right\rfloor$. We now use mathematical induction to show that for these values of j and s,

$$|z_{i,j}| = \begin{cases} m & \text{for } 0 \le i \le 2^{s-1} + l_s - 1, \\ 0 & \text{for } 2^{s-1} + l_s \le i < 2^t. \end{cases}$$

Basis Case. For j = q + 2, s = t and $l_s = b$; furthermore, $|z_{i,q+1}| = m$ for $0 \le i < 2^t$. In round q + 2, processor P_F executes FEG, and hence,

$$|u_{i,q+2}| = \begin{cases} n - 2m & \text{for } 0 \le i \le 2^{t-1} - 1, \\ n & \text{for } 2^{t-1} \le i \le 2^{t-1} + b - 1, \\ 0 & \text{for } 2^{t-1} + b \le i < 2^t. \end{cases}$$

Therefore,

$$|z_{i,q+2}| = \begin{cases} m & \text{for } 0 \le i \le 2^{t-1} + b - 1, \\ 0 & \text{for } 2^{t-1} + b \le i < 2^t. \end{cases}$$

Induction Hypothesis: Let j - 1 be any integer in the range $q + 2 \le j - 1 \le q + t$, and let s + 1 = q + t + 2 - (j - 1). Assume that in round j - 1,

$$|z_{i,j-1}| = \begin{cases} m & \text{for } 0 \le i \le 2^s + l_{s+1} - 1, \\ 0 & \text{for } 2^s + l_{s+1} \le i < 2^t. \end{cases}$$

Now consider round j. Then s = q + t + 2 - j.

Case 1: $0 \le i \le 2^{s-1} + k_s - 1$.

Then algorithm FF(s) defines $u_{i,j}$ to be a nonempty (n-2m)-bit string. Furthermore,

$$2i + 1 \le 2^s + 2k_s - 1 \le 2^s + l_{s+1} - 1$$
 by Lemma 9.

By our Induction Hypothesis, $|z_{2i,j-1}| = |z_{2i+1,j-1}| = m$. Hence, $|z_{2i,j-1}||z_{2i+1,j-1}||u_{i,j}| = n$. This implies $|z_{i,j}| = m$.

Case 2: $2^{s-1} + k_s \le i \le 2^{s-1} + l_s - 1$.

This case is vacuous whenever $l_s = k_s$. When $l_s = k_s + 1$, then $2^{s-1} + k_s = i = 2^{s-1} + l_s - 1$ and $|u_{i,j}| = 0$ from the definition of algorithm FF(s). Then

$$2i = 2^s + 2l_s - 2 = 2^s + l_{s+1} - 1$$
 (since $2l_s = l_{s+1} + 1$ when $l_s = k_s + 1$).

Therefore, $|z_{2i,j-1}| = m$ by our Induction Hypothesis. Since $2i + 1 = 2^s + l_{s+1}$, our Induction Hypothesis implies $|z_{2i+1,j-1}| = 0$. Therefore, $|z_{2i,j-1}||z_{2i+1,j-1}||u_{i,j}| = m$ and $z_{i,j} = z_{2i,j-1}$, a nonempty *m*-bit string.

Case 3: $2^{s-1} + l_s \le i < 2^t$. Since $2^{s-1} + k_s \le 2^{s-1} + l_s \le i$, $|u_{i,j}| = 0$. In addition, $2i \ge 2^s + 2l_s \ge 2^s + l_{s+1}$. Therefore, $|z_{2i,j-1}| = |z_{2i+1,j-1}| = 0$. Hence, $|z_{2i,j-1}| |z_{2i+1,j-1}| |u_{i,j}| = 0$ and $z_{i,j} = <>$.

Thus we have shown that

$$|z_{i,j}| = \begin{cases} m & \text{for } 0 \le i \le 2^{s-1} + l_s - 1, \\ 0 & \text{for } 2^{s-1} + l_s \le i < 2^t. \end{cases}$$

By mathematical induction, this holds for all j in the range $q+2 \le j \le q+t+1$ and s = q+t+2-j.

From the above argument, we see that, for $1 \leq j \leq q + t + 1$, $|u_{i,j}| = n - 2m$ if and only if $|z_{2i,j-1}| = |z_{2i+1,j-1}| = m$. In this case, $z_{i,j} = h(z_{2i,j-1}||z_{2i+1,j-1}||u_{i,j})$. As well, it is immediate that whenever a formatting algorithm defines $|u_{i,j}| = n$, then $z_{i,j} = h(u_{i,j})$. Thus the hash function h processes each of the prefixes $u_{i,j}$.

When message x has length $L > \delta(t)$, then b > 0. From the above result, we see that $|z_{0,q+t+1}| = |z_{1,q+t+1}| = m$. From Proposition 8, we know that the substring x presented to processor P_0 in step 17 of PHA is of length n - 2m. Therefore, $z_{0,q+t+2} = h(z_{0,q+t+1}||z_{1,q+t+1}||x)$, an *m*-bit string, as required.

When message x has length $L = \delta(t)$, then b = 0. From the above result, we see that $|z_{0,q+t+1}| = m$ and $|z_{1,q+t+1}| = 0$. From Proposition 8, we know that the substring x presented to processor P_0 in step 17 of PHA is of length 0. Therefore, $z_{0,q+t+2} = z_{0,q+t+1}$, an m-bit string, as required.

We now turn to computing the number of invocations of h made by PHA(x, t). Let $\psi(L)$ be the number of invocations of h made by PHA(x, t) on a message of length L. The parameters q_t and b_t depend on the length L of the message. We write $q_t(L)$ and $b_t(L)$ to denote the dependence of the parameters q_t and b_t on length L. We now have the following result.

Proposition 11 $\psi(L) = (q_t(L) + 2)2^t + 2b_t(L) - 1.$

Proof. We first note that $q = q_t = q_t(L)$ and $b = b_t = b_t(L)$. In each of the first $q_t(L) + 1$ rounds h is invoked 2^t times. In round $q_t(L) + 2$, the number of invocations of h is $2^{t-1} + b_t(L)$. In rounds $q_t(L) + 3$ to $q_t(L) + t + 1$, the total number of invocations of h is $\sum_{s=1}^{t-1} (2^{s-1} + k_s)$. Lastly, in round $q_t(L) + t + 2$, there is one invocation of h. Using Corollary 7, we have $\sum_{s=1}^{t-1} k_s = b - 1$. Adding the above number of invocations we get the final result.

We now compare the number of invocations of h by PHA to that made by the MD algorithm. The maximum amount of padding required by PHA is 2(n - m) - 1 and that required by the MD algorithm is n - m - 1. We compare the number of invocations of h on message lengths which do not require padding by PHA. It turns out that these message lengths also do not require padding by the MD algorithm.

Let the length of the message be $L = \delta(t) + q_t(L)\lambda(t) + b_t(L)(2n - 2m)$. Then PHA makes $\psi(L) = (q_t(L) + 2)2^t + 2b_t(L) - 1$ invocations of h.

Here we use the description of the MD algorithm given in [5]. For the MD algorithm the first invocation uses n bits and each of the subsequent invocations uses n - m bits. Hence the total number of invocations of h is

$$1 + \frac{L-n}{n-m} = 1 + \frac{2^{t}(2n-2m) + q2^{t-1}(2n-2m) + b(2n-2m) - (n-2m) - n}{n-m} = \psi(L).$$

Thus we get the following result.

Theorem 12 The number $\psi(L)$ of invocations of h made by PHA(x, t) on a message x of length $L = \delta(t) + q_t(L)\lambda(t) + b_t(L)(2n - 2m)$ is equal to the number of invocations of h made by the MD algorithm on a message of the same length L.

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 which is $q_t(L) + t + 2$. Further, both PHA and the MD algorithm must format the message. Hence if we ignore the time required to format the message, then PHA is faster by a factor of

$$\frac{\psi(L)}{q+t+2}.$$

For moderately large q, the increase in speed is almost linear in the number of processors.

6 Security Reductions for h_L and h^*

In this section we show that finding collisions for h_L and h^* is difficult provided finding collisions for h is difficult.

6.1 Collision Resistance of h_L

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 .

Theorem 13 Let h be an (n,m) hash function and for $L \ge n$ 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.

Proof. The algorithm \mathcal{B} does the following. It first runs \mathcal{A} to obtain 2 strings x and x' such that $x \neq x'$, |x| = |x'| = L, and with probability at least ϵ , $h_L(x) = h_L(x')$. Then \mathcal{B} runs PHA on both x and x' and stores all the intermediate states of the buffers z_i and u_i . Let z_{ij} and z'_{ij} be the states

of buffer z_i at the end of round j corresponding to the messages x and x' respectively. Similarly, let u_{ij} and u'_{ij} be the strings written to buffer u_i in round j corresponding to the messages x and x' respectively.

We now proceed by reverse induction on j to show that if $x \neq x'$ and $h_L(x) = h_L(x')$, then we can find a collision for hash function h by applying algorithm PHA to x and x'. To allow us to start the proof with round q + t + 1, we must extend our definition of u_{ij} and u'_{ij} to round j = q + t + 2. Define $u_{0,q+t+2}$ and $u'_{0,q+t+2}$ to be the substrings x and x', respectively, provided to processor P_0 in Step 17 of algorithm PHA. Define $u_{i,q+t+2} = \langle \rangle = u'_{i,q+t+2}$ for $0 < i < 2^t$.

Induction Basis Step :
$$j = q + t + 2$$
.

We show that, either there is a collision for h in round q + t + 2, or else

$$z_{i,q+t+1} = z'_{i,q+t+1}$$
 and $u_{i,q+t+2} = u'_{i,q+t+2}$ for $0 \le i < 2^t$.

Case 1: b = **0.** In this case, $u_{0,q+t+2} = <> = u'_{0,q+t+2}$ by Proposition 7. Hence, $u_{i,q+t+2} = u'_{i,q+t+2}$ for all $i, 0 \le i < 2^t$. By Theorem 9, $z_{i,q+t+1} = <> = z'_{i,q+t+1}$ for all $i, 0 < i < 2^t$. Furthermore, $z_{0,q+t+1} = z_{0,q+t+2} = h_L(x) = h_L(x') = z'_{0,q+t+2} = z'_{0,q+t+1}$. This completes the proof of Case 1.

Case 2: b > **0.** In this case, $u_{0,q+t+2} = x \neq <> \neq x' = u'_{0,q+t+2}$ by Proposition 7. By Theorem 9, $z_{i,q+t+1} = <> = z'_{i,q+t+1}$ for all *i*, $1 < i < 2^t$ and $z_{0,q+t+1}$, $z_{1,q+t+1}$, $z'_{0,q+t+1}$, $z'_{1,q+t+1}$ are all nonempty *m*-bit strings. Hence,

$$\begin{split} h(z_{0,q+t+1}||z_{1,q+t+1}||u_{0,q+t+2}) \\ &= z_{0,q+t+2} = h_L(x) = h_L(x') = z'_{0,q+t+2} \\ &= h(z'_{0,q+t+1}||z'_{1,q+t+1}||u'_{0,q+t+2}). \end{split}$$

Now, 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 hash function h; otherwise, $z_{0,q+t+1} = z'_{0,q+t+1}$, $z_{1,q+t+1} = z'_{1,q+t+1}$ and $u_{0,q+t+2} = u'_{0,q+t+2}$. This completes the proof of Case 2, and also the proof of the Induction Basis Step.

Induction Hypothesis: For $q + t + 1 \ge j \ge 2$, assume

$$z_{i,j} = z'_{i,j}$$
 and $u_{i,j+1} = u'_{i,j+1}$ for $0 \le i < 2^t$.

In the proof of Theorem 9 we show that, for $0 \le i < 2^{t-1}$,

$$z_{i,j} = \begin{cases} h(z_{2i,j-1}||z_{2i+1,j-1}||u_{i,j}) & \text{whenever} \quad z_{2i+1,j-1} \neq <> \neq u_{i,j}, \\ z_{2i,j-1} & \text{whenever} \quad z_{2i+1,j-1} = <> = u_{i,j}, \end{cases}$$

and, for $2^{t-1} \le i < 2^t$,

$$z_{i,j} = \begin{cases} h(u_{i,j}) & \text{whenever } u_{i,j} \neq <>, \\ <> & \text{whenever } u_{i,j} = <>. \end{cases}$$

The corresponding statements hold for the $z'_{i,j}$'s and the $u'_{i,j}$'s. Because messages x and x' are of the same length L, $z_{i,j} = <>$ if and only if $z'_{i,j} = <>$ and $u_{i,j} = <>$ if and only if $u'_{i,j} = <>$. We now consider each of these 4 cases individually.

For $0 \leq i < 2^{t-1}$ and $z_{2i+1,j-1} \neq <> \neq u_{i,j}$, we have that

$$\begin{aligned} h(z_{2i,j-1}||z_{2i+1,j-1}||u_{i,j}) &= z_{i,j} \\ &= z'_{i,j} \quad \text{(by the Induction Hypothesis)} \\ &= h(z'_{2i,j-1}||z'_{2i+1,j-1}||u'_{i,j}). \end{aligned}$$

If $(z_{2i,j-1}||z_{2i+1,j-1}||u_{i,j}) \neq (z'_{2i,j-1}||z'_{2i+1,j-1}||u'_{i,j})$, then we have a collision for hash function h; otherwise,

$$z_{2i,j-1} = z'_{2i,j-1}, \ z_{2i+1,j-1} = z'_{2i+1,j-1}, \ \text{ and } \ u_{i,j} = u'_{i,j}$$

For $0 \le i < 2^{t-1}$ and $z_{2i+1,j-1} = <> = u_{i,j}$, we have that

$$egin{array}{rcl} z_{2i,j-1}&=&z_{i,j}\ &=&z_{i,j}'\ &=&z_{i,j-1}'\ &=&z_{2i,j-1}'. \end{array}$$

For $2^{t-1} \leq i < 2^t$ and $u_{i,j} \neq <>$, we have that

$$h(u_{i,j}) = z_{i,j}$$

= $z'_{i,j}$ (by the Induction Hypothesis)
= $h(u'_{i,j})$.

If $u_{i,j} \neq u'_{i,j}$, then we have a collision for hash function h; otherwise, $u_{i,j} = u'_{i,j}$.

For $2^{t-1} \leq i < 2^t$ and $u_{i,j} = <>$, we have that $u_{i,j} = <>= u'_{i,j}$.

Combining these 4 cases, we get that, either there is a collision for hash function h in round j, or else

$$z_{i,j-1} = z'_{i,j-1} \ \ ext{and} \ \ u_{i,j} = u'_{i,j} \ \ ext{for} \ \ 0 \leq i < 2^t.$$

By mathematical induction, it follows that, either there is a collision for hash function h in some round $j = q + t + 2, q + t + 1, \ldots, 3, 2$, or else

$$z_{i,j-1} = z'_{i,j-1}$$
 and $u_{i,j} = u'_{i,j}$ for $0 \le i < 2^t$ and $2 \le j \le q + t + 2$.

In round 1, $u_{i,1}$ is a nonempty *n*-bit string for $0 \le i < 2^t$. If there is no collision for *h* in rounds $q + t + 2, q + t + 1, \ldots, 3, 2$, then

$$h(u_{i,1}) = z_{i,1} = z'_{i,1} = h(u'_{i,1}).$$

If $u_{i,1} \neq u'_{i,1}$ for some *i*, then there is a collision for hash function *h* in round 1; otherwise, $u_{i,1} = u'_{i,1}$ for $0 \leq i < 2^t$.

Thus we obtain that if there is no collision in any invocation of h by PHA on messages x and x', then $u_{i,j} = u'_{i,j}$ for $0 \le i < 2^t$ and $1 \le j \le q + t + 2$. The padded messages x and x' are equal to the concatenations of the $u_{i,j}$'s and the $u'_{i,j}$'s, respectively, for $0 \le i < 2^t$ and $1 \le j \le q + t + 2$. Hence, if there is no collision in any invocation of h by PHA on the messages x and x', then x = x'. But algorithm \mathcal{A} ensures that with probability at least ϵ , we have messages x and x' such that $|x| = |x'| = L, x \ne x'$, and $h_L(x) = h_L(x')$. Hence, with probability at least ϵ , we obtain a collision for h.

The number of invocations of h by algorithm \mathcal{B} is equal to the number of invocations of h by \mathcal{A} plus the number of invocations of h by PHA on x and x'. Thus the total number of invocations of h by algorithm \mathcal{B} is $p + 2\psi(L)$.

6.2 Collision Resistance of h^*

The security of h^* is easily derived from the security of h_L . The details are given below.

Theorem 14 Let h be an (n,m) hash function and h^* be the function defined by Equation 3. 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.

Proof. The algorithm \mathcal{B} does the following. It first runs \mathcal{A} to obtain two messages x and x'. Then with probability at least ϵ , we have $h^*(x) = h^*(x')$ and $x \neq x'$. Algorithm \mathcal{B} then runs h^* on both x and x' to obtain $h^*(x) = y$ and $h^*(x') = y'$ storing all the intermediate values that are generated. Let $w = h_{|x|}(x)$, $w' = h_{|x'|}(x')$, $v = bin_{n-m}(|x|)$ and $v' = bin_{n-m}(|x'|)$. There are two cases.

Case 1: $|x| \neq |x'|$. In this case $v \neq v'$ and hence $v||w \neq v'||w'$. However, h(v||w) = y = y' = h(v'||w') with probability at least ϵ . Thus in this case we can find a collision for h with probability at least ϵ .

Case 2: |x| = |x'| = L. In this case v = v'. If $w \neq w'$, then we have a collision for h. If w = w' then we have a collision for h_L . We can now argue as in the proof of Theorem 13 that with probability at least ϵ we obtain a collision for h.

The computation of h^* requires $1 + \psi(L)$ invocations of the hash function h. This shows that the number of invocations of h made by \mathcal{B} is at most $p + 2 + 2\psi(L)$.

7 Construction of h^{∞}

In this section we describe the construction and the security reduction for the function h^{∞} : $\bigcup_{L \ge n} \{0, 1\}^L \to \{0, 1\}^m$. Define $\delta_1(t) = \delta(t) - 1$ and $\lambda_1(t) = \lambda(t) - 1$. As in Definition 1, for $L \ge \delta_1(t)$, we define the parameters q, r and b as follows.

Definition 15 1. If $L > \delta_1(t)$, then q and r are defined by the following equation: $L - \delta_1(t) = q\lambda_1(t) + r$, where r is the unique integer from the set $\{1, \ldots, \lambda_1(t)\}$. Define $b = \lceil \frac{r}{2n-2m} \rceil$.

2. If $L = \delta_1(t)$, then q = b = r = 0.

Algorithm PHA computes the function h_L . We first define a modification of PHA. More specifically, we define the modifications required in the formatting subroutines. We will call the resulting algorithm the modified PHA algorithm.

Modification to FSU: Replace Step 1 of FSU by the following sequence of operations:

Write x = v || y where |v| = n - 1. $u_0 = v || 0, x = y$.

for
$$i = 1, 2, \dots, 2^t - 1$$
 do

Modification to FSS: Replace Step 1 of FSS by the following sequence of operations:

Write x = v || y where |v| = n - 2m - 1.

$$u_0 = v || 1, \ x = y.$$

<u>for</u> $i = 1, 2, \dots, 2^t - 1$ <u>do</u>

Informally, during start up we are providing P_0 with an input whose last bit is 0 and during steady state we are providing P_0 with an input whose last bit is 1.

Let the function computed by modified PHA be $g_L : \{0,1\}^L \to \{0,1\}^m$. We now describe the construction of the function h^{∞} .

The parameter b is at most 2^{t-1} and can be represented in binary by a t-bit string. Note that the length of the binary representation of b depends only on t and is independent of the message length L. We denote the t-bit binary representation of b by bin(b). Let $\mu(t) = \lceil \log(\delta_1(t) + 1) \rceil$. Let tbin(L) be a binary string of length $\mu(t)$, such that tbin(L) is the $\mu(t)$ -bit binary representation of L if $L < \delta_1(t)$, else tbin(L) is the $\mu(t)$ -bit binary representation of $\delta_1(t)$.

The output of the function h^{∞} is defined by the following algorithm.

Algorithm ArbLength

input : message x of length L. **output** : m-bit message digest $h^{\infty}(x)$.

- 1. If $L < \delta_1(t)$, then find the unique t_1 such that $\delta_1(t_1) \le L < \delta_1(t_1 + 1)$. Then perform Step 2 with t replaced by t_1 .
- 2. If $L \ge \delta_1(t)$, then apply modified PHA to x to obtain an m-bit message digest $w = g_L(x)$.
- 3. Let $w_1 = h_{m+t}(w || bin(b))$.
- 4. Let $w_2 = h_{m+\mu(t)}(w_1 || tbin(L)).$
- 5. output w_2 .

Remark: It is reasonable to assume that both $t, tbin(L) \leq n - m$. Then we could let bin(b) and tbin(L) be (n - m)-bit strings. In this situation, Steps 3 and 4 above can be replaced by $w_1 = h(w||bin(b))$ and $w_2 = h(w_1||tbin(L))$ respectively.

We now turn to the security reduction for h^{∞} . First we note the fact that the security of g_L is preserved in a manner similar to that of h_L .

Theorem 16 Let h be an (n, m) hash function and for $L \ge n$ let g_L be the function defined by the modified algorithm PHA. If there is an (ϵ, p, L) algorithm \mathcal{A} to solve FLC(n, m, L) for the hash function g_L , then there is an $(\epsilon, p + 2\psi_1(L))$ algorithm \mathcal{B} to solve Col(n, m) for the hash function h, where $\psi_1(L)$ is the number of invocations of h made by g_L .

Theorem 17 Let h be an (n,m) hash function and for $L \ge n$ let h^{∞} be the function defined by algorithm ArbLength. If there is an (ϵ, p, L) algorithm \mathcal{A} to solve ALC(n, m, L) for the hash function h^{∞} , then there is an $(\epsilon, p + 2\psi_2(L))$ algorithm \mathcal{B} to solve Col(n,m) for the hash function h, where $\psi_2(L) = \psi_1(L) + \psi(t+m) + \psi(\mu(t)+m)$ is the number of invocations of h made by h^{∞} .

Proof. Algorithm \mathcal{B} runs algorithm \mathcal{A} to obtain two strings x and x' such that with probability at least ϵ we have $h^{\infty}(x) = h^{\infty}(x')$ and $x \neq x'$. Let L = |x| and L' = |x'|. Further, we will denote the parameters for the message x by unprimed symbols and the parameters for the message x' by primed symbols. First assume that L = L'. Then tbin(L) = tbin(L') and bin(b) = bin(b'). We can now use Theorem 13 to obtain a collision for h with probability at least ϵ . Thus for the rest of the proof we will assume $L \neq L'$. There are two cases to consider.

Case 1: At least one of L or L' is less that $\delta_1(t)$. In this case $tbin(L) \neq tbin(L')$. We have

$$egin{array}{rcl} h_{m+\mu(t)}(w_1||tbin(L))&=&w_2\ &=&w_2'\ &=&h_{m+\mu(t)}(w_1'||tbin(L')) \end{array}$$

We can argue as in Theorem 13 that either we obtain a collision for h or $w_1||tbin(L) = w'_1||tbin(L')$ which in turn implies tbin(L) = tbin(L'). Since we know $tbin(L) \neq tbin(L')$, it follows that we must obtain a collision for h. **Case 2**: Both $L, L' \ge \delta_1(t)$. In this case we have tbin(L) = tbin(L'). If $w_1 \ne w'_1$, then the inputs to $h_{\mu(t)+m}$ in Step 4 of ArbLength are different for x and x'. This will again provide a collision for h. So suppose $w_1 = w'_1$. There are two subcases to consider.

Subcase 2a : $b \neq b'$: In this case $bin(b) \neq bin(b')$. We have

$$h_{m+t}(w||tbin(L)) = w_1 = w'_1 = h_{m+t}(w'||tbin(L')).$$

Again the inputs to h_{m+t} are different and hence we have a collision for h_{m+t} . As before, this will necessarily provide a collision for h.

Subcase 2b: b = b': In this case bin(b) = bin(b'). If $w \neq w'$, then this will provide a collision for h. So assume that w = w'.

So we are in the situation where $g_L(x) = w = w' = g_{L'}(x')$, b = b' and $L \neq L'$. We have the (padded) message lengths in the following forms: $L = \delta_1(t) + q\lambda_1(t) + b(2n - 2m)$ and $L' = \delta_1(t) + q'\lambda_1(t) + b'(2n - 2m)$. Since b = b' and $L \neq L'$ we have $q \neq q'$. Assume without loss of generality q' < q.

The last t + 1 rounds of both PHA and modified PHA are the same. Suppose that none of the invocations of h in the last t + 1 rounds of modified PHA provides a collision for h. Now using the fact that b = b' we can use a backward induction on the round number (as in the proof of Theorem 13) to obtain $z_{i,q+1} = z'_{i,q'+1}$ for all $0 \le i \le 2^t - 1$. Continuing the backward induction we obtain $z_{i,q-q'+1} = z'_{i,1}$ for all $0 \le i \le 2^t - 1$. We now look at the output of processor P_0 . Let p = q - q'. We have

$$\begin{array}{rcl} z_{0,p+1} & = & P_0(z_{0,p}||z_{1,p}||u_{0,p+1}), \\ z_{0,1}' & = & P_0(u_{0,1}'). \end{array}$$

The string $u_{0,p+1}$ is obtained from FSS and the string $u'_{0,1}$ is obtained from FSU. By the modifications made to these algorithms to get modified PHA, we know that $u_{0,p+1} = v||1$ and $u'_{0,1} = v'||0$ for some strings v and v' of lengths n-1 and n-2m-1 respectively. Hence $z_{0,p}||z_{1,p}||u_{0,p+1} \neq u'_{0,1}$. But $z_{0,p+1} = z'_{0,1}$ and so we obtain

$$P_0(z_{0,p}||z_{1,p}||u_{0,p+1}) = h(z_{0,p}||z_{1,p}||u_{0,p+1}) = z_{0,p+1} = z'_{0,1} = h(u'_{0,1}) = P_0(u'_{0,1}).$$

This is a collision for h.

We next consider the amount of padding required by algorithm ArbLength. This is determined by the padding introduced by algorithm modified PHA.

Theorem 18 Algorithm modified PHA pads any message by at least q+1 bits where q is as defined in Definition 15.

Proof. The modification to FSU introduces one bit of padding and the modification to FSS introduces one bit of padding per round. Since FSS is executed q times a total of q bits of padding is introduced by FSS.

From Definition 15 we have

$$\left\lfloor \frac{L - \delta_1(t)}{\lambda_1(t)} \right\rfloor \le q + 1 \le 1 + \left\lfloor \frac{L - \delta_1(t)}{\lambda_1(t)} \right\rfloor$$

Since t, n, m are constants for a particular implementation of modified PHA, the amount of padding is linear in the length of the message. We note that the Merkle-Damgard construction also uses an

amount of padding which is linear in the length of the message (see [9]). Moreover, the constant of proportionality is lesser for our construction. However, it is undesirable to have a padding scheme which grows with the length of the message. The amount of padding required in the construction of h^* is at most 2(n-m)-1 and hence is independent of the message length. Further, the function h^* can take as input any message of practical length. Thus algorithm ArbLength and the function h^{∞} are mainly of theoretical interest.

8 Preimage Resistance

We have formally considered only one property of hash functions - namely intractibility of finding collisions. There are other necessary properties that a hash function must satisfy. These are Preimage and Second Preimage (see [8]). We are required to show that our constructions preserve the intractibility of these problems. In fact, these properties are indeed preserved and the proofs are easy. We informally describe the reduction for Preimage.

Informally the preimage problem for a hash function h is the following. The adversary is given a message digest y and has to obtain a message x such that h(x) = y. Suppose that there is a (probabilistic) algorithm \mathcal{A} to solve the preimage problem for any of our extensions h_L, h^* or h^{∞} . For the sake of concreteness we only consider h_L , the others being similar. We argue that \mathcal{A} can be used to obtain an algorithm \mathcal{B} which will solve the preimage for h with the same probability of success. Given y, algorithm \mathcal{B} will first run \mathcal{A} to obtain a preimage x for h_L . Then \mathcal{B} runs PHA and outputs $w = z_{0,q+t+1} ||z_{1,q+t+1}||u_{q+t+2}$ if b > 0 or $w = z_{0,q+t} ||z_{1,q+t}||u_{q+t+1}$ if b = 0. It is now easy to see that w is a preimage for h (with the probability of success being at least that of \mathcal{A}).

9 Concluding Remarks

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 $n \ge km$. More speed up can be achieved 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.

To summarize our contribution, in this paper, we have presented an incrementally parallelizable design principle for cryptographic hash functions. We believe that our design principle will provide the basic structure for designing future practical hash functions. In a future communication, we will describe parallel modifications of MD5, RIPEMD-160 and SHA-2 hash functions. Our plan is to keep the "core" operations of these hash functions intact but build the iterative part based on the design principle developed in this paper.

Acknowledgement : We wish to thank Professor Bart Preneel for helpful comments on an earlier draft of the paper.

References

 M. Bellare and P. Rogaway. Collision-resistant hashing: towards making UOWHFs practical. Proceedings of CRYPTO 1997, pp 470-484.

- [2] D. Chaum, E. van Heijst and B. Pfitzmann. Cryptographically strong undeniable signatures, unconditionally secure for the signer. *Lecture Notes in Computer Science*, 576 (1992), 470-484, (Advances in Cryptology - CRYPTO'91).
- [3] I. B. Damgard. A design principle for hash functions. Lecture Notes in Computer Science, 435 (1990), 416-427 (Advances in Cryptology CRYPTO'89).
- [4] R. C. Merkle. One way hash functions and DES. Lecture Notes in Computer Science, 435 (1990), 428-226 (Advances in Cryptology CRYPTO'89).
- [5] I. Mironov. Hash functions: from Merkle-Damgard to Shoup. Lecture Notes in Computer Science, 2045 (2001), 166-181 (Advances in Cryptology - EUROCRYPT'01).
- [6] 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.
- [7] 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).
- [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)