A Parallelizable Design Principle for Cryptographic Hash Functions*

Palash Sarkar†
Cryptology Research Group
Applied Statistics Unit
Indian Statistical Institute
203, B.T. Road
Kolkata 700108
West Bengal, India
palash@isical.ac.in

Paul J. Schellenberg
Centre for Applied Cryptographic Research
Department of Combinatorics and Optimization
University of Waterloo
200 University Avenue West
Waterloo, Ontario
Canada N2L 3G1
pjschell@math.uwaterloo.ca

Contents

---

*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.

Abstract

We describe a parallel design principle for hash functions. Given a secure hash function $h : \{0,1\}^n \to \{0,1\}^m$ with $n \geq 2m$, 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 $\left\lfloor \frac{L}{2^t(n-m)} \right\rfloor + t$. 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 $2^{t'}$ $(0 \leq t' \leq t)$ processors.

Keywords : hash function, Merkle-Damgård construction, collision resistance, preimage resistance, parallel algorithm, binary tree.

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 [11] 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 [13], Stinson provides a comprehensive discussion of these properties and also relations among them. Depending on a particular application, a secure hash function must satisfy some or all of the following properties.

(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 $h(x) = z$. This property was first defined by Diffie and Hellman in their seminal paper on modern cryptology [7].

(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 $h(y) = z$. The notion of second preimage resistance was introduced by Merkle in [8].

(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 $h(x) = h(y)$. This property was first formally defined by Damgård in [6].

It is clear that if it is possible to find a second preimage, then it is possible to find collisions. Hence it is usually sufficient to study collision resistance. However, as pointed out in [13], 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 [4]). However, from a practical point of view such hash functions are unacceptably slow. Hence practical hash functions are constructed 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 Damgård [5]. An important point made in [5] 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 [8] and Damgård [5]. The construction is called the Merkle-Damgård (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 $n \geq 2m$ and a binary tree of 2^t processors, it is possible 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 arbitrary length messages. Since we require $n \geq 2m$ and practical hash functions have $m \geq 128$, the function h^* is adequate for any conceivable application and the construction of h^∞ is of theoretical interest only. The number of parallel rounds required to compute the digest of a message of length L is $\left\lfloor \frac{L}{2^t(n-m)} \right\rfloor + t$.

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 $0 \leq t' \leq t$ 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 $t \geq 0$. 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. Damgård [5] 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 [10, 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.

Parallelism in the design of hash functions have also been considered from a different direction. In our approach we assume the existence of a base hash function with a small domain and present a method to obtain a hash function with a very large domain. In the literature, the base hash function is called the *compression function* or *round function* and the method to extend the domain is called the *composition method*. Practical hash functions such as SHA or RIPEMD, use a specific round function and the Merkle-Damgård method to extend the domain. In [3], a detailed study has been made of the possible parallelism available in the round function of SHA and other practical hash functions. Note that our work is complementary to this effort in the sense that we exploit the parallelism that is obtainable in the *composition scheme* as opposed to the parallelism in the *round function*.

In another relevant paper, Schnorr and Vaudenay [12], design a hash function based on the fast Fourier transform (FFT) network and multipermutations. They present two hash functions based on this approach. The first hash function uses a compression function based on the FFT algorithm and multipermutations while the composition scheme is sequential. In the second algorithm, the compression function and the composition schemes are merged together to obtain a fast hash algorithm. This approach differs from our work in the following way. In our work we describe a parallel composition scheme which can be used with *any* compression function (whose input length is at least twice as large as the length of its output). As examples our parallel composition schemes can be used with the round function of SHA as well as the FFT based compression function of [12]. On the other hand, the approach in [12] is to construct a specific hash function. Thus we describe a design principle whereas [12] describes a particular hash function.

Parallelism in the design of hash functions has also been considered in the context of incremental hashing [2]. The incremental hash function described in [2] is computed by combining a set of elements of a group using the group law. Since the group operation is associative, this combining operation can be parallelized. However, [2] does not develop this theme any further.

2 Basics

This section consists of three parts. The first part formally describes collision resistant hash functions and related problems. The second part describes the processor tree model used in the paper. In the third part, we define certain parameters which will be required in the rest of the paper. *Throughout the paper we will denote the empty string by* $\boldsymbol{\lambda}$ *and the length of a binary string* y *by* $|y|$. *Further, the concatenation of two strings* y_1 *and* y_2 *will be denoted by* $y_1||y_2$.

2.1 Hash Functions

An (n, m) hash function h is a function $h : \{0,1\}^n \to \{0,1\}^m$. *Throughout this paper we require that* $n \geq 2m$. We are basically interested in collision resistant hash functions. As mentioned in Section 1, this means that it should be computationally infeasible to obtain two different messages whose hash values are the same. Describing this formally requires the consideration of a family of hash functions. The following definition of collision resistant hash function is based on [11].

Definition 1 *Let $\{H_s\}_{s\in\mathcal{S}}$ be a family of functions indexed by the set $\mathcal{S}$ where each H_s is an $(l(n), n)$ function. Here $l(n)$ is a function such that $l(n) > n$. We say that $\{H_s\}_{s\in\mathcal{S}}$ is collision resistant if the following condition holds.*

Let $\mathcal{A}$ be a randomized Las Vegas algorithm that takes as input an $s \in \mathcal{S}$, runs for time at most t and either returns "?" or finds x, x' such that $x \neq x'$ and $H_s(x) = H_s(x')$. For each such algorithm $\mathcal{A}$ with t bounded above by a polynomial in size of s, we should have

$$\mathsf{Prob}_{s\in\mathcal{S}}[\mathcal{A}(s) \neq \text{"?"}] \leq \epsilon.$$

In practice, it is usual to define a *single* hash function like SHA or RIPEMD instead of a keyed *family* of hash functions. In such a situation, it is not possible to apply Definition 1 to these hash functions. In fact, it seems almost impossible to provide a meaningful definition of collision resistance for a single hash function.

In this paper, we are interested in obtaining a design principle for practical hash functions. We will *assume* the existence of some collision resistant (n, m) hash function h, called the *compression function*. The function h can only be applied to inputs of length n. We would like to remove this restriction and define a hash function H which can be applied to strings of extremely large lengths. We would like the extension of h to H to be "secure" in the following sense: *If h is collision resistant, then H is also collision resistant.* The last statement is formalized in terms of a Turing reduction between two suitably defined problems (see below). The advantage of this method is that we only prove a reduction and at no point are we required to use Definition 1. This approach has been previously used in the study of hash functions [13].

We now turn to the task of defining our approach to reducibilities between different problems related to the property of collision resistance. Consider the following problem as defined in [13].

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 $L \geq 1$, we construct an (L, m) hash function h_L. The next step, in general, is to construct a hash function $h^\infty : \cup_{L\geq 1}\{0,1\}^L \to \{0,1\}^m$. However, instead of doing this, we first construct a hash function $h^* : \cup_{L=1}^{N}\{0,1\}^L \to \{0,1\}^m$, 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 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 $L \geq 1$.
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 $1 \leq L < 2^{n-m}$.
Find	:	$x, x' \in \cup_{i=1}^{L}\{0,1\}^i$ 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 $L \geq 1$.
Find	:	$x, x' \in \cup_{i=1}^{L}\{0,1\}^i$ 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 processor. *The number of processors is 2^t, for some $t > 0$.* 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. In particular, the arcs coming into processor P_0 are from processor P_1 and processor P_0 itself. 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 $\mathcal{T}_t$ the processor tree of height t* (see Figure 1 for the processor tree with $t = 3$). 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. We introduce the following notation which will be useful later.

$$\mathcal{I}_t = \{i : 0 \leq i \leq 2^{t-1} - 1\};\ \mathcal{L}_t = \{i : 2^{t-1} \leq i \leq 2^t - 1\};\ \mathcal{P}_t = \{i : 0 \leq i \leq 2^t - 1\};$$

When t is clear from the context, we will usually write $\mathcal{T}$, $\mathcal{I}$, $\mathcal{L}$ and $\mathcal{P}$ instead of $\mathcal{T}_t$, $\mathcal{I}_t$, $\mathcal{L}_t$ and $\mathcal{P}_t$ respectively.

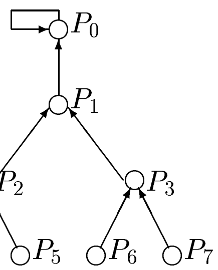

Figure 1: Processor Tree with $t = 3$.

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 $i \in \mathcal{P}$, 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 $i \in \mathcal{I}$, 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 $i \in \mathcal{L}$, the input buffer of P_i is u_i. The output buffer of P_i is z_i for $i \in \mathcal{P}$.

Algorithm PHA 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 $i \in \mathcal{P}$.
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 define certain parameters which are going to be used throughout the paper.

1. **Compression function:** An (n, m) function h with $n \geq 2m$.

2. **Height of the binary tree:** t.

3. **Message:** a binary string x of length $L \geq 1$.

4. **Start-up length (SUL):** $2^t n$.

5. **Maximum Flushing length (MFL):** $(2^{t-1}+2^{t-2}+\cdots+2^1+2^0)(n-2m) = (2^t-1)(n-2m)$.

6. **SUL + MFL:** $\delta(t) = 2^t n + (2^t - 1)(n - 2m) = 2^t(2n - 2m) - (n - 2m)$.

7. **Steady-state length (SSL):** $\lambda(t) = 2^{t-1}n + 2^{t-1}(n - 2m) = 2^{t-1}(2n - 2m)$.

8. **Parameters $\mathbf{q_t}$, $\mathbf{b_t}$ and $\mathbf{r_t}$:**

   Definition 2 *(a) 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 $b_t = \left\lceil \frac{r_t}{2n-2m} \right\rceil$.*

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

   Note that $0 \leq b_t \leq 2^{t-1}$.

9. **Number of Parallel Rounds R_t:** We define $R_t = q_t+t+2$. Later we will show in Theorem 6 that algorithm PHA executes R_t parallel rounds. We will usually write R instead of R_t.

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 height 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 $i \in \mathcal{I}$, the buffer u_i stores either an $(n-2m)$-bit string or the empty string and for $i \in \mathcal{L}$, 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 a certain number of 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 prefix from it. This prefix is divided into substrings and copied to the buffers u_i.

Intially 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 each processor P_i, $i \in \mathcal{I}$, 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. Each processor P_i, $i \in \mathcal{L}$, gets an n-bit input which is hashed to obtain the new value of the buffer z_i. The Steady-State lasts for q rounds (see Definition 2 above). 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, 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 gets 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 starting from level $(t-1)$ and ending at level 1. In the s^{th} stage of the flushing operation, all processors at levels greater than s get empty strings as inputs. Some of the processors at level s get an $(n-2m)$-bit string as input. The rest of the processors at level s get the empty string as input. All processors at levels $\leq s-1$ get an $(n-2m)$-bit string as input. This stage is called the Flusing stage.

At the end of the Flushing stage, the following two situations can occur. Either x is the empty string or it is an $(n-2m)$-bit string. If x is empty, then z_0 is returned as output. On the other hand, if x is an $(n-2m)$-bit string, then z_0 and z_1 are both m-bit strings. In this case, processor P_0 applies the function h 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 $L \geq \delta(t)$.
(2) t is the height of the processor tree.
Output: message digest $h_L(x)$ of length m.
Define: $q = q_t$, $r = r_t$ and $b = b_t$.

if $L > \delta(t)$, then
    $x := x||0^{b(2n-2m)-r}$
    (ensures that the length of the message becomes $\delta(t) + q\lambda(t) + b(2n - 2m)$.)
endif.
For $i \in \mathcal{P}$, initialise buffers z_i and u_i to empty strings.
Do FormatStartUp.
Do ParallelProcess.
for $i = 1, 2, \ldots, q$ do
    Do FormatSteadyState.
    Do ParallelProcess.
endfor
Do FormatEndGame.
Do ParallelProcess.
for $s = t-1, t-2, \ldots 2, 1$ do
    Do FormatFlushing(s).
    Do ParallelProcess.
endfor
if $x \neq \boldsymbol{\lambda}$ then $z_0 := P_0(z_0||z_1||x)$.
return z_0.
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, $i \in \mathcal{P}$.

for $i \in \mathcal{P}$ do in parallel
    If $i \in \mathcal{I}$, then $z_i := P_i(z_{2i}||z_{2i+1}||u_i)$.
    If $i \in \mathcal{L}$, then $z_i := P_i(u_i)$.
endfor
end algorithm PP

3.1 Formatting Subroutines

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 $i \in \mathcal{P}$. 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 $i \in \mathcal{P}$, write a prefix of message x to buffer u_i and update the message x.

for $i \in \mathcal{P}$ do
    Write $x = v||y$, where $|v| = n$.

$u_i := v$.
$x := y$.
endfor
end algorithm FSU

FormatSteadyState (FSS)
Action: For $i \in \mathcal{P}$, write a prefix of message x to buffer u_i and update the message x.

for $i \in \mathcal{I}$ do
    Write $x = v||y$, where $|v| = n - 2m$.
    $u_i := v$.
    $x := y$.
endfor
for $i \in \mathcal{L}$ do
    Write $x = v||y$, where $|v| = n$.
    $u_i := v$.
    $x := y$.
endfor
end algorithm FSS

FormatEndGame (FEG)
Action: For $i \in \mathcal{P}$, write a prefix of message x to buffer u_i and update the message x.

for $i \in \mathcal{I}$ do
    Write $x = v||y$ where $|v| = n - 2m$.
    $u_i := v$.
    $x := y$.
endfor
for $i = 2^{t-1}, 2^{t-1}+1, \ldots, 2^{t-1}+b-1$ do
    Write $x = v||y$ where $|v| = n$.
    $u_i := v$.
    $x := y$.
endfor
for $i = 2^{t-1}+b, 2^{t-1}+b+1, \ldots, 2^t - 1$ do
    $u_i := \boldsymbol{\lambda}$.
endfor
end algorithm (FEG)

FormatFlushing(s) (FF(s))
Input: Integer s.
Action: For $i \in \mathcal{P}$, write a prefix of message x to buffer u_i and update the message x.

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

for $i = 0, 1, 2, \ldots, 2^{s-1}+k_s-1$ do
    Write $x = v||y$ where $|v| = n - 2m$.
    $u_i := v$.
    $x := y$.
endfor

6. $\underline{\text{for}}$ $i = 2^{s-1} + k_s, 2^{s-1} + k_s + 1, \ldots, 2^t - 1,$
7. $\quad u_i := \boldsymbol{\lambda}.$
8. <u>endfor</u>
9. **end algorithm FF**

An example of the working of algorithm PHA is shown in Figure 2.

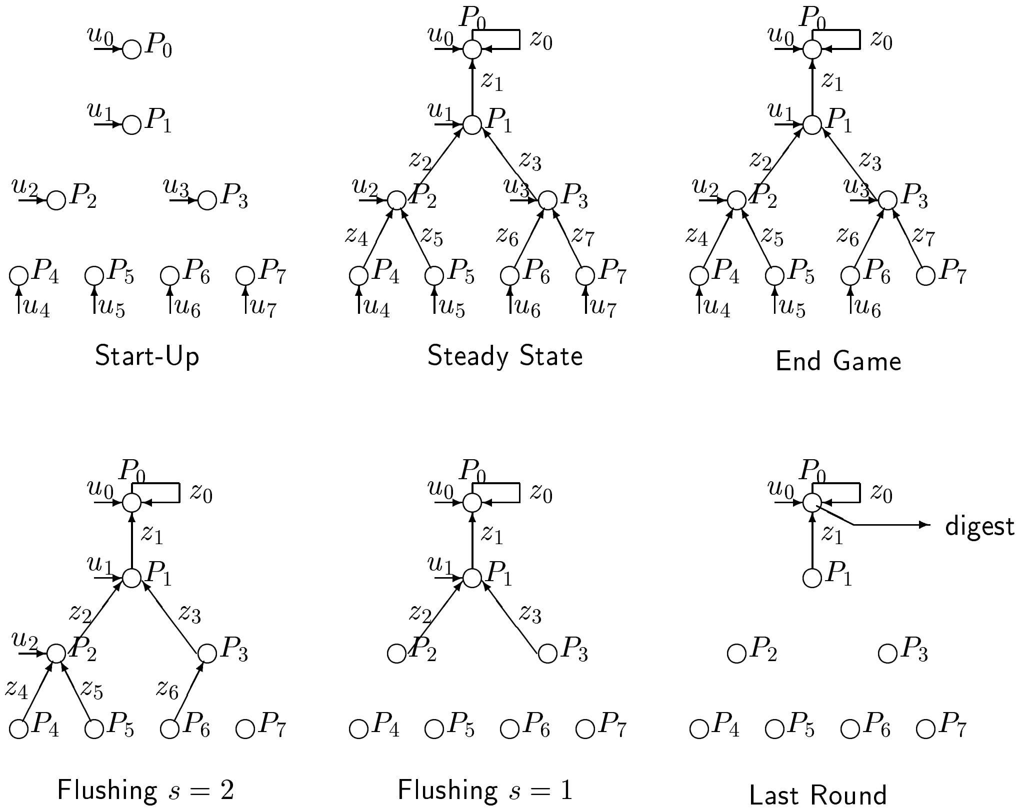

Figure 2: Algorithm PHA with $t = 3$, $q = 1$ and $b = 3$.

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 $u_i = \boldsymbol{\lambda}$ in all the rounds and for all $i \in \mathcal{I}$.

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 (s)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 height 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 $y = \text{PHA}(x, t)$ be produced by the processor tree $\mathcal{T}$. We describe an algorithm $\text{SimPar}(x, t, t')$ which also produces y using the processor tree $\mathcal{T}'$.

SimPar(x, t, t')

Input:

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

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

1. Change Lines 6,9 and 12 to "Do $\mathrm{SPP}(t,t')$".

2. Change Line 15 to "Do $\mathrm{SPP}(s,t')$".

end algorithm SimPar

The subroutine SPP() performs 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 upto height s. We define the subroutine SPP() to do these two tasks.

Algorithm SPP(s,t')

```
if s < t', then s = t'.
for j = 0 to 2^(s-t') - 1 do
        i = j2^(t')
        for ℓ = 0 to 2^(t') - 1 do in parallel
                use processor P'_ℓ to execute the task of processor P_(i+ℓ).
        endfor
endfor
end Algorithm SPP.
```

Proposition 3 *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 $s > t'$.*

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

4 Parallel Hash Function Definitions

The compression function is an (n,m) function h, with $n \geq 2m$. 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 $\cup_{i=1}^{n}\{0,1\}^i$ to $\{0,1\}^m$. The description of h_L and h^* is described below.

4.1 Definition of h_L

Let $L \geq 1$ be a positive integer and assume that a binary tree of 2^T processors is available. Then the (L,m) function h_L is defined as follows.

$$h_L(x) = \begin{cases} \mathrm{PHA}(x,T) & \text{if } L \geq \delta(T); \\ \mathrm{PHA}(x,t) & \text{if } 0 < t < T \text{ and } \delta(t) \leq L < \delta(t+1); \\ \mathrm{PHA}(x||0^{3n-2m-L},1) & \text{if } \delta(0) < L < \delta(1) = 3n-2m; \\ h(x) & \text{if } 1 \leq L \leq n = \delta(0). \end{cases} \quad (2)$$

When $t < T$, we use only 2^t of the 2^T processors available. Is is possible to significantly reduce the number of rounds by using more processors? The following lemma answers this question.

Lemma 4 *Any algorithm which provides input to a processor at level T in tree $\mathcal{T}$ requires at least $T+1$ rounds to compute the message digest; the computation of $h_L(x)$ described above requires at most*

$$\begin{cases} T+1 \text{ rounds when } t < T-1 \text{ and} \\ T+2 \text{ rounds when } t = T-1. \end{cases}$$

Proof. If we provide input to any processor of $\mathcal{T}$ at level T, then it requires at least $T+1$ rounds for the effects of this input to reach processor P_0 at level zero. Thus, at least $T+1$ rounds are required to compute the message digest.

From the definitions of $\delta(t)$ and $\lambda(t)$, we see that $\delta(t+1) = \delta(t) + 2\lambda(t)$. Therefore, if $\delta(t) \leq L < \delta(t+1)$, then $L = \delta(t) + q\lambda(t) + b(2n-2m)$ where $0 \leq q \leq 1$ and $0 \leq b \leq 2^{t-1}$. If $|x| = L$, then PHA(x,t) requires at most $t+3$ rounds to compute a message digest (see Subsection 5.1 below).

If $t < T-1$, then $t+3 < T+2$; if $t = T-1$, then $t+3 = T+2$. This establishes the result. ■

4.2 Definition of h^*

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

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

In other words, we first compute $h_L(x)$ (where $|x| = L$) 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 accomodate 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. 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 height of the binary tree used to generate the message digest. Suppose a user generates the digest using a binary tree of height t. Then any other user who needs to regenerate the digest has to have access to a binary tree of height t or should be able to simulate the binary tree of height 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 either one of the following two ways.

(1) The height T of the processor tree is fixed and is part of the hash function specification. Then any user who needs to generate $y =$PHA(x,T) and has access to a processor tree of height t, with

$t < T$ uses SimPar(x, T, t) to generate y. If $t \geq T$, then the user can run PHA(x, T) by not using processors at level greater than T.

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

Depending on the situation at hand either 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$(x, t, 0)$ to compute y =PHA(x, t). Thus User 2 can directly use his processor tree of 2^t processors to invoke PHA(x, t) 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

In this section we prove several properties of algorithm PHA.

5.1 Amount of Padding

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 independent of the number of processors and the length of the message.

Proposition 5 *The maximum amount of padding added to any message is less than* $2n - 2m$.

Proof. The only place where padding 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 : The maximum amount of padding required by PHA is $2(n - m) - 1$ and that required by the MD algorithm is $n - m - 1$.

5.2 Number of Parallel Rounds

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 6 *Algorithm PHA(x,t) executes $R = q+t+2 \le \left\lfloor \frac{L}{2^t(n-m)} \right\rfloor + t$ parallel rounds. Consequently, Algorithm SimPar(x,t,t') executes $(q+3)2^{t-t'} + t' - 1$ parallel rounds.*

Proof. Clearly the number of parallel rounds is $q+t+2$. From Definition 2, we have $q = \left\lfloor \frac{L-\delta(t)}{\lambda(t)} \right\rfloor$ if $\lambda(t) \not| L-\delta(t)$; and $q = \left\lfloor \frac{L-\delta(t)}{\lambda(t)} \right\rfloor - 1$ if $\lambda(t) | L - \delta(t)$. Hence,

$$\begin{aligned} q &\le \left\lfloor \frac{L-\delta(t)}{\lambda(t)} \right\rfloor \\ &= \left\lfloor \frac{L - 2^t(2n-2m) - (n-2m)}{2^{t-1}(2n-2m)} \right\rfloor \\ &\le \left\lfloor \frac{L-2^t(2n-2m)}{2^{t-1}(2n-2m)} \right\rfloor \\ &= \left\lfloor \frac{L}{2^t(n-m)} \right\rfloor - 2. \end{aligned}$$

Therefore, $q+t+2 \le \left\lfloor \frac{L}{2^t(n-m)} \right\rfloor + t$. ∎

5.3 Invocations of the Compression Function

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, r_t and b_t depend on the length L of the message. We write $q_t(L)$, $r_t(L)$ and $b_t(L)$ to denote the dependence of the parameters q_t and b_t on length L. Note that due to the padding done in line 2 of algorithm PHA we have $\psi(L) = \psi(L + b_t(L)(2n-2m) - r_t(L))$. We now have the following result.

Proposition 7 $\psi(L) = (q_t(L)+2)2^t + 2b_t(L) - 1$.

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 10 below, we have $\sum_{s=1}^{t-1} k_s = b-1$. Adding the above number of invocations we get the final result. ∎

We compare the number of invocations of h by PHA to that made by the MD algorithm. We do this for 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 [9]. 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 8 *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.*

5.4 Speed-Up over MD Algorithm

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 equal to $q_t(L)+t+2$. Further, both PHA and the MD algorithm must format the message. Hence if we ignore the time required to format the message, then the speed-up factor SF of PHA over MD is computed as follows.

$$\mathsf{SF} = \frac{\psi(L)}{R} = \frac{(q+2)2^t + 2b - 1}{R} = 2^t\left(\frac{q+2}{R}\right) + \frac{2b-1}{R} \approx 2^t\left(\frac{1}{1+\frac{t}{q+2}}\right).$$

The parameter $q = q_t$ is defined in equation (4). Using the values of $\delta(t)$ and $\lambda(t)$ we observe that $q+2 \approx \frac{L}{\lambda(t)}$. Hence $\mathsf{SF}\approx 2^t\left(\frac{1}{1+\frac{t\lambda(t)}{L}}\right)$.

The parameter t is the height of the binary tree and is fixed for a particular implementation. Hence $t\lambda(t)$ is a constant for a particular implementation of the algorithm. Thus SF$\rightarrow 2^t$ as $L \rightarrow \infty$. In other words, for long messages, the speed-up factor is roughly equal to the number of processors used.

5.5 Correctness of the Formatting Subroutines

The formatting subroutines of algorithm PHA divide the message into substrings and provide these as input to the compression function h. There are two things which require to be proved.

1. The formatting subroutines ensure that each bit of the message is provided as input to exactly one invocation of the compression function h.
2. The final output of algorithm PHA is an m-bit string.

The rest of the section is devoted to proving these two properties.

Each of the first $(R-1)$ parallel rounds in PHA(x,t) consists 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 $(i \in \mathcal{P})$ are operated in parallel. Denote by $z_{i,j}$ the state of the buffer z_i at the end of round j, where $i \in \mathcal{P}$ and $1 \le j \le R$. Clearly, the state of the buffer z_i at the start of round j $(2 \le j \le R)$ 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 $i \in \mathcal{I}$, the input to processor P_i in round j is $z_{2i,j-1}||z_{2i+1,j-1}||u_{i,j}$. For $i \in \mathcal{L}$, the input to processor P_i in round j is the string $u_{i,j}$.

The following lemma and corollary are required to prove Proposition 11.

Lemma 9 *For any nonnegative integer b, $\sum_{i\ge 1}\left\lfloor\frac{b+2^{i-1}}{2^i}\right\rfloor = b$.*

Proof. We prove this result by induction on b. Clearly the result holds for $b=0$.

Induction Hypothesis: For b a positive integer, assume that $\sum_{i\ge 1}\left\lfloor\frac{b+2^{i-1}-1}{2^i}\right\rfloor = b-1$.

It can be shown that

$$\left\lfloor\frac{m}{n}\right\rfloor = \begin{cases}\left\lfloor\frac{m-1}{n}\right\rfloor + 1 & \text{when } n|m,\\[2ex] \left\lfloor\frac{m-1}{n}\right\rfloor & \text{otherwise.}\end{cases}$$

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

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

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

Corollary 10 *For t a given positive integer and b an integer in the range $0 \leq b \leq 2^{t-1}$, let $k_s = \lfloor \frac{b+2^{t-s-1}-1}{2^{t-s}} \rfloor$ as defined in algorithm FF(s). Then $\sum_{s=1}^{t-1} k_s = \sum_{s\geq 1} k_s = b-1$.*

Proposition 11 *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 $0 \leq b \leq 2^{t-1}$. The formatting subroutines ensure that each bit of the message x is provided as input to some processor P_i exactly once; furthermore, the substring x presented to processor P_0 in step 17 of PHA is the empty string when $|x| = \delta(t)$ and is an $(n-2m)$-bit string when $|x| > \delta(t)$. The total time required by the formatting subroutines to format the message x over all the R rounds is proportional to*

(a) $|x| + (t-1)2^t - 2b + 2$ *steps when* $|x| > \delta(t)$ *or*

(b) $|x| + (t-1)2^t + 1$ *steps when* $|x| = \delta(t)$.

Proof. Each formatting algorithm defines $u_i = \boldsymbol{\lambda}$ or else defines u_i to be a prefix of x; namely,

$x = v||y$
$u_i = v$
$x = y$

In step 17, the substring x itself is presented to processor P_0. Hence, each bit of the message x is presented as input to some processor P_i exactly once. We now determine the length of the substring x presented to processor P_0 in step 17, i.e., in round R.

First assume that $L > \delta(t)$ and hence $b > 0$. 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, $i \in \mathcal{I}$, and an n-bit prefix to processor P_i, $i \in \mathcal{L}$. 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, $i \in \mathcal{I}$, with an $(n-2m)$-bit prefix of x, each leaf processor P_i, $2^{t-1} \leq i \leq 2^{t-1}+b-1$, 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, $0 \leq i < 2^{s-1}+k_s-1$, 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 $u_i = \boldsymbol{\lambda}$. This accounts for $(2^{s-1}+k_s)(n-2m)$ bits of x. The total number of bits presented to all the processors in the first $R-1$ rounds is

$$\begin{aligned}
& 2^t n + q\lambda(t) + bn + 2^{t-1}(n-2m) + \sum_{s=t-1}^{1}(2^{s-1}+k_s)(n-2m) \\
= \quad & 2^t n + q\lambda(t) + bn + \sum_{s=t}^{1} 2^{s-1}(n-2m) + \sum_{s=t-1}^{1} k_s(n-2m) \\
= \quad & 2^t n + q\lambda(t) + bn + (2^t-1)(n-2m) + (b-1)(n-2m) \quad \left(\text{since } \sum_{s=t-1}^{1} k_s = b-1\right) \\
= \quad & \delta(t) + q\lambda(t) + b(2n-2m) - (n-2m).
\end{aligned}$$

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 $s = t-1, t-2, \ldots, 2, 1$. 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 $u_i = \boldsymbol{\lambda}$ for $2^{t-1} + b \le i < 2^t$, and, for $1 \le s < t$, FF(s) defines $u_i = \boldsymbol{\lambda}$ for $2^{s-1} + k_s \le i < 2^t$. The number of assignments of the form $u_i = \boldsymbol{\lambda}$ is

$$\begin{aligned} & 2^{t-1} - b + \sum_{s=t-1}^{1} (2^t - 2^{s-1} - k_s) = 2^{t-1} - b + (t-1)2^t - \sum_{s=t-1}^{1} 2^{s-1} - \sum_{s=t-1}^{1} k_s \\ = \; & 2^{t-1} + (t-1)2^t - (2^{t-1} - 1) - 2b + 1 = (t-1)2^t - 2b + 2. \end{aligned}$$

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

Each step of the formatting algorithms consist of moving the leading bit of string x to some buffer u_i, or else assigning $u_i = \boldsymbol{\lambda}$. Therefore, the formatting algorithms require

(a) $\delta(t) + q\lambda(t) + b(2n-2m) + (t-1)2^t - 2b + 2$ 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 13.

Lemma 12 *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) $l_s = k_s + 1$ *if and only if* $2l_s = l_{s+1} + 1$.

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, $2\lfloor x + \frac{1}{2} \rfloor \le \lfloor 2x + 1 \rfloor \le 2\lfloor x + 1 \rfloor$. 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

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

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 $q+2+l$, 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.

At any round r, $1 \le r < R$, the input to processor P_i, $i \in \mathcal{L}$, is $u_{i,r}$, and the input to processor P_i, $i \in \mathcal{I}$, is $z_{2i,r-1}||z_{2i+1,r-1}||u_{i,r}$. In Theorem 13 we show that every bit of the string x is acted upon by hash function h. Furthermore, if $z_{i,r} \neq \boldsymbol{\lambda}$ then either $z_{i,r}$ is acted upon by h in round $r+1$ or else $z_{i,r}$ is passed on as the output of processor $P_{i/2}$ in round $r+1$; that is $z_{i/2,r+1} = z_{i,r}$. We establish these facts by showing that

$$\begin{aligned}&i \in \mathcal{I} \text{ and } |u_{i,r}| = n - 2m \text{ imply } |z_{2i,r-1}| = |z_{2i+1,r-1}| = m, \text{ and}\\ &i \in \mathcal{I} \text{ and } u_{i,r} = \boldsymbol{\lambda} \text{ imply } z_{2i+1,r-1} = \boldsymbol{\lambda}.\end{aligned}$$

Theorem 13 (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 $y = z_{0,R-1}||z_{1,R-1}||u_{0,R}$. Then, the output of algorithm PHA is, by definition,

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

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 $i \in \mathcal{P}$. Hence $|z_{i,1}| = m$ for $i \in \mathcal{P}$. Further, it is easy to verify that for $2 \le j \le q+1$, we have $|z_{i,j}| = m$ for $i \in \mathcal{P}$ and

$$|u_{i,j}| = \begin{cases} n - 2m & \text{if } i \in \mathcal{I}; \\ n & \text{if } i \in \mathcal{L}. \end{cases}$$

For $q+2 \le j \le R-1$, let $s = R - j$. Then $t \ge s \ge 1$ corresponding to $q+2 \le j \le R-1$. Define $l_s = \left\lfloor \frac{b+2^{t-s}-1}{2^{t-s}} \right\rfloor$. We now use 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 $i \in \mathcal{P}$. In round $q+2$, processor P_F executes FEG, and hence,

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

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 $q+2 \leq j-1 \leq q+t = R-2$, and let $s+1 = R-(j-1)$. Assume that in round $j-1$,

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

Now consider round j. Then $s = R - j$.

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

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

$$2i + 1 \leq 2^s + 2k_s - 1 \leq 2^s + l_{s+1} - 1 \quad \text{by Lemma 12.}$$

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

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

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

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: $2^{s-1} + l_s \leq i < 2^t$.

Since $2^{s-1} + k_s \leq 2^{s-1} + l_s \leq i$, $|u_{i,j}| = 0$. In addition, $2i \geq 2^s + 2l_s \geq 2^s + l_{s+1}$. 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 $z_{i,j} = \boldsymbol{\lambda}$.

Thus we have shown that

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

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

From the above argument, we see that, for $1 \leq j \leq R-1$, $|u_{i,j}| = n - 2m$ 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 $|z_{0,R-1}| = |z_{1,R-1}| = m$. From Proposition 11, we know that the substring x presented to processor P_0 in step 17 of PHA is of length $n - 2m$. Therefore, $z_{0,R} = h(z_{0,R-1}||z_{1,R-1}||x)$, an m-bit string, as required.

When message x has length $L = \delta(t)$, then $b = 0$. From the above result, we see that $|z_{0,R-1}| = m$ and $|z_{1,R-1}| = 0$. From Proposition 11, we know that the substring x presented to processor P_0 in step 17 of PHA is of length 0. Therefore, $z_{0,R} = z_{0,R-1}$, an m-bit string, as required. ∎

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

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 14 *Let $t \geq 0$, h be an (n,m) hash function and for $L \geq 1$ let h_L be the function defined by equation (2). 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 two 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.

For message x and round number j, define $\mathsf{ZList}(x,j)$ and $\mathsf{UList}(x,j)$ to be the following two lists: $\mathsf{ZList}(x,j) = \langle z_{0,j}, \ldots, z_{2^t-1,j} \rangle$; $\mathsf{UList}(x,j) = \langle u_{0,j}, \ldots, u_{2^t-1,j} \rangle$.

Note that the message x is equal to the concatenation of the strings in the lists $\mathsf{UList}(x,1), \ldots, \mathsf{UList}(x,R)$. Next we prove the following claim by backward induction on round number $j \leq R$.

Claim : If $h_L(x) = h_L(x')$ and there is no collision for the function h in rounds $j, \ldots, R$, then $\mathsf{UList}(x,j) = \mathsf{UList}(x',j)$ and $\mathsf{ZList}(x,j-1) = \mathsf{ZList}(x',j-1)$.

Proof of Claim : The base case is $j = R$. Note that $h_L(x) = h_L(x')$ implies $z_{0,R} = z'_{0,R}$. There are two cases to consider for round R according as $b = 0$ or $b > 0$. If $b = 0$, then the function h has not been invoked in round R and it is easy to see that $\mathsf{ZList}(x,R-1) = \mathsf{ZList}(x',R-1)$ and $\mathsf{UList}(x,R) = \mathsf{UList}(x',R)$. If $b > 0$, then the function h has been invoked in round R and either we have a collision for h in round R or $\mathsf{ZList}(x,R-1) = \mathsf{ZList}(x',R-1)$ and $\mathsf{UList}(x,R) = \mathsf{UList}(x',R)$.

Now suppose $j < R$. By the induction hypothesis for $j+1$ we know that $\mathsf{UList}(x,j+1) = \mathsf{UList}(x',j+1)$ and $\mathsf{ZList}(x,j) = \mathsf{ZList}(x',j)$. The condition $\mathsf{ZList}(x,j) = \mathsf{ZList}(x',j)$ states that for messages x and x' the outputs of all the processors are equal at the end of round j. The action of any processor in round j is to either copy its input to output or invoke the hash function h on its input. The inputs to the processors are the elements of the lists $\mathsf{ZList}(x,j-1)$ and $\mathsf{UList}(x,j)$ for message x (respectively, $\mathsf{ZList}(x',j-1)$ and $\mathsf{UList}(x',j)$ for message x). Thus if there is no collision for h in round j, we must have $\mathsf{UList}(x,j) = \mathsf{UList}(x',j)$ and $\mathsf{ZList}(x,j-1) = \mathsf{ZList}(x',j-1)$. This completes the inductive step and the proof of the claim.

From this claim it follows that if $h_L(x) = h_L(x')$ and there is no collision for the function h in any of the rounds, then $x = x'$. Since algorithm $\mathcal{A}$ succeeds with probability at least ϵ, we conclude that there is a collision for the function h also with probability at least ϵ. The number of invocations of h made by algorithm $\mathcal{B}$ is equal to the number of invocations of h made by algorithm $\mathcal{A}$ plus twice the number of invocations of h made by algorithm PHA(x,t). Hence the number of invocations made by algorithm $\mathcal{B}$ is equal to $p + 2\psi(L)$. ∎

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 15 *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 14 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 $h^\infty : \cup_{L \geq 1}\{0,1\}^L \to \{0,1\}^m$. Define $\delta_1(t) = \delta(t) - 1$ and $\lambda_1(t) = \lambda(t) - 1$. As in Definition 2, for $L \geq \delta_1(t)$, we define the parameters q, r and b as follows.

Definition 16 *1. If $L > \delta_1(t)$, then q and r are defined by the following equation:*

$$L - \delta_1(t) = q\lambda_1(t) + r, \tag{4}$$

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 $|v| = n - 1$.
$u_0 = v||0$, $x = y$.
<u>for</u> $i = 1, 2, \ldots, 2^t - 1$ <u>do</u>

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> $i = 1, 2, \ldots, 2^t - 1$ <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 (L, m) function computed by modified PHA be g_L. 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 $\delta_1(t_1) \leq L < \delta_1(t_1+1)$. Then perform Step 2 with t replaced by t_1.
2. If $L \geq \delta_1(t)$, 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 $t, \mu(t) \leq n-m$. Then we could let $bin(b)$ and $tbin(L)$ be $(n-m)$-bit strings. In this situtation, 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 17 *Let h be an (n,m) hash function and for $L \geq n$ let g_L be the function defined by the modified PHA algorithm. 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 18 *Let h be an (n,m) hash function and for $L \geq n$ 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 17 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

$$h_{m+\mu(t)}(w_1||tbin(L)) = w_2 = w_2' = h_{m+\mu(t)}(w_1'||tbin(L')).$$

We can argue as in Theorem 14 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 $L, L' \geq \delta_1(t)$. In this case we have $tbin(L) = tbin(L')$. If $w_1 \neq w'_1$, 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 14) to obtain $z_{i,q+1} = z'_{i,q'+1}$ for all $i \in \mathcal{P}$. Continuing the backward induction we obtain $z_{i,q-q'+1} = z'_{i,1}$ for all $i \in \mathcal{P}$. 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 19 *Algorithm modified PHA pads any message by at least $q+1$ bits where q is as defined in Defintion 16.*

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 16 we have

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

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-Damgård construction also uses an amount of padding which is linear in the length of the message (see [14]). 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. Depending on the application, there are other necessary properties that a hash function must satisfy. These are Preimage and Second Preimage. It is easy to see that the ability to find a second preimage implies the ability to find collisions. Hence if a function is collision resistant, it is automatically second preimage resistant. Thus we do not consider the property of second preimage resistance. However, we note that in general it is difficult to obtain a composition scheme that preserves the property of second preimage resistance.

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 $w = z_{0,R-1}||z_{1,R-1}||u_R$ if $b > 0$ or $w = z_{0,R-2}||z_{1,R-2}||u_{R-1}$ 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 $n \geq km$. 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 hash functions. We believe that our design principle will provide the basic structure for designing future practical hash functions.

Acknowledgement : We wish to thank Professor Bart Preneel for helpful comments on an earlier draft of the paper. We would also like to thank an anonymous referee for extensive comments on an earlier version of the paper. These comments have helped in significantly improving the presentation and technical quality of the paper.

References

[1] M. Bellare and P. Rogaway. Collision-resistant hashing: towards making UOWHFs practical. *Lecture Notes in Computer Science*, Proceedings of CRYPTO 1997, pp 470-484.

[2] M. Bellare and D. Micciancio. A New Paradigm for Collision-Free Hashing: Incrementality at Reduced Cost. *Lecture Notes in Computer Science*, (Advances in Cryptology - EUROCRYPT 1997), pages 163-192.

[3] A. Bosselaers, R. Govaerts and J. Vandewalle, SHA: A Design for Parallel Architectures? *Lecture Notes in Computer Science*, (Advances in Cryptology - Eurocrypt'97), pages 348-362.

[4] 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).

[5] I. B. Damgård. A design principle for hash functions. *Lecture Notes in Computer Science*, 435 (1990), 416-427 (Advances in Cryptology - CRYPTO'89).

[6] I. B. Damgård. Collision Free Hash Functions and Public Key Signature Schemes. *Lecture Notes in Computer Science*, (Advances in Cryptology EUROCRYPT 1987), pages 203-216.

[7] W. Diffie and Martin E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, volume IT-22, number 6, pages 644–654, year 1976.

[8] R. C. Merkle. One way hash functions and DES. *Lecture Notes in Computer Science*, 435 (1990), 428-226 (Advances in Cryptology - CRYPTO'89).

[9] I. Mironov. Hash functions: from Merkle-Damgård to Shoup. *Lecture Notes in Computer Science*, 2045 (2001), 166-181 (Advances in Cryptology - EUROCRYPT'01).

[10] 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.

[11] 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).

[12] C. Schnorr and S. Vaudenay. Parallel FFT-Hashing. *Lecture Notes in Computer Science*, Fast Software Encryption, LNCS 809, pages 149-156, 1994.

[13] D. R. Stinson. Some observations on the theory of cryptographic hash functions. IACR preprint server, http://eprint.iacr.org/2001/020/.

[14] D. R. Stinson. *Cryptography: Theory and Practice*, CRC Press, 1995.