PARSHA-256 - A New Parallelizable Hash Function and a Multithreaded Implementation ⋆

Pinakpani Pal and Palash Sarkar

Cryptology Research Group
ECSU and ASU
Indian Statistical Institute
203, B.T. Road, Kolkata
India 700108
{pinak,palash}@isical.ac.in

Abstract. In this paper, we design a new hash function PARSHA-256. PARSHA-256 uses the compression function of SHA-256 along with the Sarkar-Schellenberg composition principle. As a consequence, PARSHA-256 is collision resistant if the compression function of SHA-256 is collision resistant. On the other hand, PARSHA-256 can be implemented using a binary tree of processors, resulting in a significant speed-up over SHA-256. We also show that PARSHA-256 can be efficiently implemented through concurrent programming on a single processor machine using a multithreaded approach. Experimental results on P4 running Linux show that for long messages the multithreaded implementation is faster than SHA-256.
Keywords : hash function, SHA-256, parallel algorithm, binary tree.

1 Introduction

A collision resistant hash function is a basic primitive of modern cryptography. One important application of such functions is in "hash-then-sign" digital signature protocols. In such a protocol a long message is hashed to produce a short message digest, which is then signed. Since hash functions are typically invoked on long messages, it is very important for the digest computation algorithm to be very fast.

Design of hash functions has two goals – collision resistance and speed. For the first goal, it is virtually impossible to describe a hash function and *prove* it to be collision resistant. Thus we have to assume some function to be collision resistant. It seems more natural to make this assumption when the input is a short string rather than a long string. On the other hand, the input to a practical hash function can be arbitrarily long. Thus one has to look for a method of extending the domain of a hash function in a secure manner, i.e., the hash function on the larger domain is collision resistant if the hash function on the smaller domain

⋆ This work has been supported partially by the ReX program, a joint activity of USENIX Association and Stichting NLnet.

is collision resistant. In the literature, the fixed domain hash function which is assumed to be collision resistant is called the *compression function* and the method to extend the domain is called the *composition principle*.

A widely used composition principle is the Merkle-Damgård (MD) principle introduced in [3, 7]. To the best of our knowledge most known practical algorithms like MD5, SHA family, RIPEMD-160 [4], etc. are built using the MD composition principle. These functions vary in the design of the compression function. In fact, most of the work on practical hash function design has concentrated on the design of the compression function. This is due to the fact that the known attacks on practical hash functions are actually based on attacks on the compression function. See [9] for a survey and history of hash functions. As a result of the intense research on the design of compression function, today there are a number of compression functions which are widely believed to be collision resistant. Some examples are the compression functions of RIPEMD-160, SHA-256, etc.

As mentioned before, the other aspect of practical hash functions is the speed of the algorithm to compute the digest. One way to improve the speed is to use parallelism. Parallelism in the design of hash funcions has been studied earlier. The compression function of RIPEMD-160 has a built in parallel path [4]. In [2] the parallelism present in the compression function of the SHA family is studied. In a recent work, [8] studies the efficiency of parallel implementation of some dedicated hash functions. In [6], Knudsen and Preneel describe a parallelizable construction of secure hash function based on error-correcting codes. Another relevant paper is a hash function based on the FFT principle [11]. Also [1] describe an incremental hash function, which is parallelizable. However, most of the work seems to be concentrated on exploiting parallelism in the compression function. The other way to achieve parallelism is to incorporate it in the composition principle. One such work based on binary trees is by Damgård [3]. However, the algorithm in [3] is not practical since the size of the binary tree grows with the length of the message. One recent paper which describes a practical parallel composition principle is the work by Sarkar and Schellenberg [10].

In this paper we design a collision resistant hash function based on the Sarkar-Schellenberg (SS) composition principle. To actually design a hash function, it is not enough to have a secure composition principle; we must have a "good" compression function. As mentioned before, research in the design of compression functions have given us a number of such "good" functions. Thus one way to design a new hash function is to take the SS composition principle and an already known "good" compression function and combine them to obtain the new hash function. This new function will inherit the collision resistance from the compression function and the parallelism from the composition principle. Note that the parallelism in the composition principle is in addition to any parallelism which may be present in the compression function. Thus the studies carried out in [8, 2] on the parallel implementation of the standard hash functions are also relevant to the current work. In this paper we use this idea to design a new hash function – PARSHA-256 – which uses the SS composition principle along with the compression function of SHA-256.

PARSHA-256 can be implemented in both sequential and parallel manner. A fully parallel implementation of PARSHA-256 will provide a significant speed-up over SHA-256. However, for widespread software use, full parallel implementation might not always be possible. We still want our hash function to be used – without significantly sacrificing efficiency. One approach is to simulate the binary tree of processors with a binary tree of lesser height. Details of this simulation algorithm can be found in [10]. Another approach is to use concurrent programming using threads to simulate the parallelism.

We provide a multithreaded implementation of PARSHA-256. The SS composition principle is based on a binary tree of processors. In each round some or all of the processors work in parallel and invoke the compression function. The entire algorithm goes through several such parallel rounds. Our strategy is to simulate the processors using threads. The simulation is round by round, i.e., for each parallel round a number of threads (corresponding to the number of processors for that round) are started. All the threads execute the compression function in a concurrent manner. Also the inputs to the threads are different. The simulation of a round ends when all the threads have completed their tasks. This is repeated for all the parallel rounds.

Experimental results on P4 running Linux show that for long messages the above strategy of concurrent execution leads to a speed-up over SHA-256. This speed-up varies with the length of the message and the size of the binary tree. Thus we obtain a new hash function which is collision resistant if the compression function of SHA-256 is collision resistant; is significantly faster than SHA-256 if implemented in a full parallel manner, and on certain single processor platforms for long messages is still faster than SHA-256 if implemented as a concurrent program using threads.

2 Compression Function and Processor Tree

We describe our choice of the compression function and the processor tree used for the composition principle.

2.1 Choice of Compression Function

Let $h()$ be the compression function for SHA-256. The input to $h()$ consists of the following two quantities: (1) A : Sixteen 32-bit words and (2) B : Eight 32-bit words. In the intermediate stages, A is obtained from the message and B is the intermediate hash value. The output of $h()$ consists of eight 32-bit words. Thus the input to $h()$ is 768 bits and the output of $h()$ is 256 bits. *In the rest of the paper we will use $n = 768$ and $m = 256$.* In our algorithm, the inputs to $h()$ will be formed differently. *However, we do not change the definition of $h()$ and hence the assumption that $h()$ is collision resistant remains unchanged.*

2.2 Processor Tree

We will use a binary tree of processors. For $t > 0$, we define the processor tree $\mathcal{T}_t$ of height t in the following manner: There are 2^t processors, numbered $P_0, \ldots, P_{2^t-1}$. For $0 \leq i \leq 2^{t-1} - 1$, the children of processor P_i are P_{2i} and P_{2i+1}. The arcs point towards parents, i.e., the arc set of $\mathcal{T}_t$ is $A_t = \{(P_{2i}, P_i), (P_{2i+1}, P_i) : 0 \leq i \leq 2^{t-1} - 1\}$. Thus the arcs coming into P_0 are from P_1 and P_0 itself. We define $\mathcal{I} = \{0, \ldots, 2^{t-1} - 1\}$, $\mathcal{L} = \{2^{t-1}, \ldots, 2^t - 1\}$ and $\mathcal{P} = \{0, \ldots, 2^t - 1\}$. Figure 1 shows $\mathcal{T}_3$.

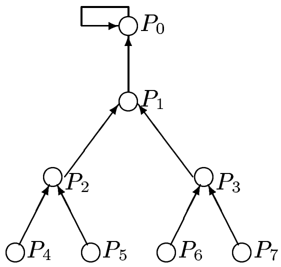

Fig. 1. Processor Tree with $t = 3$.

The input to the processors are binary strings and the behaviour of any processor P_i is described as follows:

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

Thus P_i invokes the hash function $h()$ on the string y if the length of y is n; otherwise it simply returns the string y. We note that in the digest computation algorithm the length of y will always be n, m or 0.

3 A Special Case

In this section, we describe a special case with a suitable message length and a processor tree with $t = 3$ and without the use of initialization vector. The purpose of this description is to highlight the basic idea behind the design. In Section 4, we provide the complete specification of PARSHA-256. The special case described here is intended to help the reader to better appreciate the different parameters of the general specification.

Let x be the message to be hashed with length $L = 2^t(p + 2)(n - m) - (n - 2m)$ for some integer $p \geq 0$. Consider the processor tree $\mathcal{T}_3$ having processors $P_0, \ldots, P_7$. During the hash function computation, the message x will be broken up into disjoint substrings of lengths n or $n - 2m$. These substrings will be provided as input to the processors in the different rounds. Let us denote by $u_0, \ldots, u_7$ the substrings of x which are provided to the processors $P_0, \ldots, P_7$ in a particular round. The computation will be done in $(p+4)$ parallel rounds. In each round some or all of the processors work in parallel and apply the compression function to its input to obtain its output. Let us denote by $z_0, \ldots, z_7$ respectively the outputs of the processors $P_0, \ldots, P_7$ in a particular round. The description of the rounds is as follows.

1. In round 1, each processor P_j, with $0 \leq j \leq 7$, gets as input an n-bit substring u_j of the message x and produces an m-bit output z_j.
2. In rounds 2 to $(p+1)$ the computation proceeds as follows.
   (a) Processors $P_0, \ldots, P_3$ each get an $(n-2m)$-bit substring of the message x. These substrings are $u_0, \ldots, u_3$. Processor P_j $(0 \leq j \leq 3)$ concatenates the m-bit strings z_{2j} and z_{2j+1} of the previous round to u_j to form an n-bit input. For example, P_0 concatenates z_0, z_1 to u_0; P_1 concatenates z_2, z_3 to u_1 and so on. Note that all the intermediate hash values $z_0, \ldots, z_7$ of the previous rounds are used up.
   (b) Processors $P_4, \ldots, P_7$ each get an n-bit substring of the message as input, i.e., the strings $u_4, \ldots, u_7$ are all n-bit strings.
   (c) Each of the processors invoke the compression function on their n-bit inputs to produce an m-bit output.
3. In round $(p+2)$, processors $P_0, \ldots, P_3$ each get an $(n-2m)$-bit string, i.e., the strings $u_0, \ldots, u_3$ are each $(n-2m)$-bit strings. None of the processors $P_4, \ldots, P_7$ get any input. Each processor P_j $(0 \leq j \leq 3)$ then forms an n-bit string as described in item 2a above. These strings are hashed to obtain m-bit outputs $z_0, \ldots, z_3$.
4. In round $(p+3)$, processors P_0 and P_1 each get an $(n-2m)$-bit string. (The other processors do not get any input.) These processors then form n-input using the strings $z_0, \ldots, z_3$ as before. The n-bit strings are hashed to produce two m-bit strings z_0 and z_1.
5. In round $(p+4)$ only processor P_0 gets an $(n-2m)$-bit string. The m-bits outputs z_0, z_1 of the round $(p+3)$ are concatenated to this $(n-2m)$-bit string to form an n-bit input. This input is hashed to obtain the final message digest.

Figure 2 shows the working of the algorithm. Note that the total number of bits that is hashed is equal to

$$2^t(n) + p(2^{t-1}(n-2m) + 2^{t-1}(n)) + (n-2m)(2^{t-1} + \cdots + 1).$$

A routine simplification shows that this is equal to the length L of the message x. Hence the entire message is hashed to produce the m-bit message digest. Now we consider the modifications required to handle the general situation.

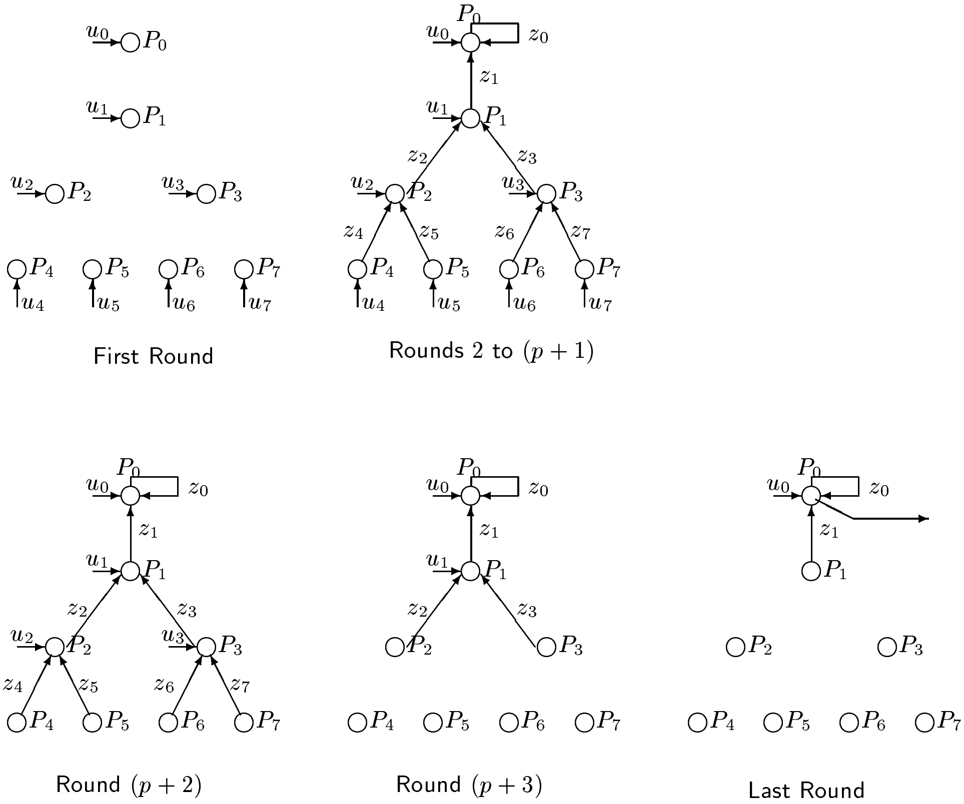

Fig. 2. Example for the special case.

3.1 Arbitrary Lengths

The message length that we have chosen is of a particular form. In general we have to tackle arbitrary length messages. This requires that the original message be padded with 0's to obtain the length in a desirable form.

3.2 Processor Tree

The special case described above is for $t = 3$. Depending upon the availability of resources, one might wish to use a larger tree. We have provided the specification of PARSHA-256 using the tree height as a parameter. Suppose the height of the available processor tree is T. However, the length of the message might not be large enough to utilize the entire processor tree. In this case, one has to utilize a subtree of height $t \leq T$, which we call the effective height of the processor tree.

3.3 Initialization Vector

The description of the special case does not use an initialization vector (IV). As a result there are invocations of the compression function where the input is formed entirely from the message bits. This implies that any collision for the compression function immediately provides a collision for the hash function. To avoid this situation, one can use an initialization vector as part of the input to the invocations of the compression function. This ensures that to find a collision for the hash function, one has to find a collision for the compression function where a portion of the input is fixed.

Using an IV is relatively simple in the Merkle-Damgård composition scheme. The IV has to be used only for the first invocation of the compression function. For the tree based algorithm, the IV has to be used at several points. It has to be used for all invocations of the compression function in the first round and all invocations of the compression function by leaf level processors in the subsequent rounds. The disadvantage of using an IV is the fact that the number of invocations of the compression functions increases. Further, this value increases as the length of the IV increases. To allow more flexibility to the user we provide for three different possible lengths for the IV. The effect of the length of IV on the number of parallel rounds and the number of invocations of the compression function is discussed in Section 5.

4 PARSHA-256 Specification

In this section we provide the detailed technical specification of the new hash function PARSHA-256. This includes the padding, formatting of the message and the digest computation algorithm. The choice of the compression function and the SS composition principle is also a part of these specifications.

4.1 Parameters and Notation

1. $n = 768$ and $m = 256$.
2. Compression function $h : \{0,1\}^n \to \{0,1\}^m$.
3. Message x having length $|x| = L$ bits.
4. Height of available processor tree is T.
5. Effective height of processor tree is t.
6. Initialization vector IV having length $|\mathsf{IV}| = l \in \{0, 128, 256\}$.
7. Functions $\delta(i)$ and $\lambda(i)$: $\delta(i) = 2^i(2n - 2m - l) - (n - 2m)$; $\lambda(i) = 2^{i-1}(2n - 2m - l)$.
8. q, r and b are defined from L and t as follows:
   If $L > \delta(t)$, then write $L - \delta(t) = q\lambda(t) + r$,
   where r is the unique integer from the set $\{1, \ldots, \lambda(t)\}$.
   If $L = \delta(t)$, then $q = r = 0$.
9. $b = \left\lceil \frac{r}{2n-2m-l} \right\rceil$.
10. Number of parallel rounds : $R = q + t + 2$.
11. The empty string will be denoted by **NULL**.

The initialization vector IV of length l is specified as followed. The specification of SHA-256 describes a 256-bit initialization vector **IV**. If $l = 256$, then $\mathsf{IV} = \mathbf{IV}$; if $l = 128$, then IV is the first 128 bits of **IV**; and if $l = 0$, then $\mathsf{IV} = \mathbf{NULL}$.

4.2 Formatting the Message

The message x undergoes two kinds of padding. In the first kind of padding, called end-padding, zeros are appended to the end of x to get the length of the padded message in a certain form. This padding is defined in Step 5 of PARSHA-256 in Section 4.4. The other kind of padding is what we call IV-padding. If $l > 0$, then IV-padding is done to ensure that no invocation of $h()$ gets only message bits as input.

We now describe the formatting of the message into substrings. Let the end-padded message be written as $U_1||U_2||\ldots||U_R$, where for $1 \leq i \leq R-1$, $U_i = u_{i,0}||\ldots||u_{i,2^t-1}$ and $u_{i,j}, U_R$ are strings of length $0, n-2m$ or $n-l$ as defined in Equation (2).

$$\left.\begin{array}{rl} |u_{i,j}| = & \begin{cases} n-l & \text{if } (i=1) \text{ or } (2 \leq i \leq q+1 \text{ and } j \in \mathcal{L}); \\ n-l & \text{if } i = q+2 \text{ and } 2^{t-1} \leq j \leq 2^{t-1}+b-1; \\ 0 & \text{if } i = q+2 \text{ and } 2^{t-1}+b \leq j \leq 2^t; \\ 0 & \text{if } q+2 < i < R \text{ and } i \in \mathcal{L}; \\ n-2m & \text{if } 2 \leq i \leq q+2 \text{ and } j \in \mathcal{I}; \\ n-2m & \text{if } q+2 < i < R \text{ and } 0 \leq j \leq K_i - 1; \\ 0 & \text{if } q+2 < i < R \text{ and } K_i \leq j \leq 2^{t-1}-1. \end{cases} \\ |U_R| = & \begin{cases} n-2m & \text{if } b > 0; \\ 0 & \text{otherwise.} \end{cases} \end{array}\right\} \quad (2)$$

Here

$$K_i = 2^{s-1} + k_s \quad (3)$$

where $s = R - i$ and $k_s = \left\lfloor \frac{2^{t-s-1}+b-1}{2^{t-s}} \right\rfloor$. For $1 \leq i < R$ and $0 \leq j \leq 2^t - 1$, the input to processor P_j in round i is a string $v_{i,j}$ and the output is $z_{i,j}$. These strings are defined as follows.

$$\left.\begin{array}{rll} z_{i,j} &= P_j(v_{i,j}); & \\ v_{i,j} &= u_{i,j}||\mathsf{IV} & \text{if } i = 1 \text{ or } j \in \mathcal{L}; \\ &= z_{i-1,2j}||z_{i-1,2j+1}||u_{i,j} & \text{if } 1 < i < R \text{ and } j \in \mathcal{I}. \end{array}\right\} \quad (4)$$

For $l = 0$, the correctness of the above formatting algorithm can be found in [10]. The same proof also holds for the case $l > 0$ and hence we do not repeat it here.

4.3 Computation of Digest

The digest computation algorithm is described as follows.
ComputeDigest(x, t)
Inputs : message x and effective tree height t.
Output : m-bit message digest.

1. for $1 \leq i \leq R-1$
2.   for $j \in \mathcal{P}$ do in parallel

3. $\quad\quad z_{i,j} = P_j(v_{i,j})$;
4. $\quad$ enddo;
5. enddo;
6. if $b > 0$ then $w = P_0(z_{R-1,0}||z_{R-1,1}||U_R)$; else $w = z_{R-1,0}$;
7. $z = h(w||\mathsf{bin}_{n-m}(L))$;
8. return z.

The function $\mathsf{bin}_k(i)$ is defined in the following manner: For $0 \leq i \leq 2^k - 1$, $\mathsf{bin}_k(i)$ denotes the k-bit binary representation of i.

Remark 1. In rounds 1 to $(q+1)$ all the processors invoke the compression function on their n-bit inputs. However, in rounds $(q+2)$ to $(q+t+1)$ only some of the processors actually invoke the compression function. The compression function is invoked only if the input to the processor is an n-bit string. Otherwise the processor simply outputs its input (see equation (1)). This behaviour of the processors is controlled by the formatting of the message. The precise details are as follows: Let i be the round number and $s = R - i$. In round $i = q+2$, processors $P_0, \ldots, P_{2^{t-1}+b-1}$ invoke the compression function; in round $q+2 < i < q+t+2$, processors $P_0, \ldots, P_{K_i - 1}$ invoke the compression function, processor P_j (if any) where $2^{s-1} + k_s - 1 < j < 2^{s-1} + l_s - 1$, $l_s = \lfloor((b + 2^{t-s} - 1)/2^{(t-s)})\rfloor$ simply outputs its m-bit input. All other processors in these rounds are inactive.

4.4 Digest Generation and Verification

We are now in a position to define the digest of a message x. Suppose that we have at our disposal a processor tree of height T. Then the digest z of x is defined in the following manner.
$\mathsf{PARSHA\text{-}256}(x, T)$
Inputs : message x and height T of available binary tree.

1. if $L \leq \delta(0) = n - l$, then return $h(h(x||0^{n-l-L}||\mathsf{IV})||\mathsf{bin}_{n-m}(L))$;
2. if $\delta(0) < L < \delta(1)$, then $x = x||0^{\delta(1)-L}$; $L = \delta(1)$;
3. Determine t as follows :
$$\begin{aligned} t &= T \quad \text{if } L \geq \delta(T); \\ &= i \quad \text{if } \delta(i) \leq L < \delta(i+1),\ 1 \leq i < T; \end{aligned}$$
4. Determine q, r and b from L and t; (see Section 4.1)
5. $x = x||0^{b(2n-2m-l)-r}$;
6. $z = \mathsf{ComputeDigest}(x, t)$;
7. output (t, z).

Clearly the digest z depends upon the height of the tree t. Hence along with z, the quantity t is also provided as output. Note that the height t of the tree used to produce the digest may be less than the height T of the tree that is available. The reason for this is that the message length L may not be long enough to utilize the entire tree. Thus t is the effective height of the tree used to compute the digest. *During verification, Step 3 of* PARSHA*-256 is not executed, since the effective height of the tree is already known.*

This raises the following question: What happens if the verifier does not have access to a tree of height t? In [10], it is shown that any digest produced using a tree of height t can also be produced using a tree of height t' with $0 \leq t' < t$. The same algorithm will also work in the present case and hence we do not repeat it here. Moreover, in this paper we provide a multithreaded implementation of algorithm ComputeDigest() where the processors are implemented using threads. This also shows that access to a physical processor tree is not necessary for digest computation.

5 Theoretical Analysis

In this section we perform a theoretical analysis of collision resistance and speed-up of PARSHA-256. The speed-up is with respect to SHA-256, which is built using the Merkle-Damgård composition principle.

5.1 Collision Resistance

We first note that the composition scheme used in the design of PARSHA-256 is the parallel Sarkar-Schellenberg scheme described in [10]. Hence we have the following result.

Theorem 1 (Sarkar-Schellenberg [10]). *If the compression function $h()$ of SHA-256 is collision resistant then so is* PARSHA-256.

If no initialization vector is used, i.e., if $l = 0$, then the ability to obtain a collision for $h()$ immediately implies the ability to obtain a collision for PARSHA-256. Hence we can state the following result.

Theorem 2. *If $l = 0$, then $h()$ is collision resistant if and only if* PARSHA-256 *is collision resistant.*

What happens if $l > 0$? In this situation the initialization vector IV is non-trivial. The intuitive idea is to increase the collision resistance of the hash function beyond that of the compression function. If there is no IV, then a collision for the compression function immediately leads to a collision for the hash function. However, if an IV is used, then the adversary has to find a collision for the compression function under the condition that a certain portion of the input is fixed. Intuitively, this could be a more difficult task for the adversary. On the other hand, Dobbertin [5] has shown that for MD4 the use of IV does not lead to any additional protection. Still the use of IV is quite common in hash function specification and hence we also include it in the specification of PARSHA-256.

5.2 Speed-Up over SHA-256

Let the end-padded length L of the message x be such that $L = \gamma(n - m)$ for some positive integer γ. To hash a message x of length L, SHA-256 requires γ invocations of $h()$ and hence the time required to hash x is γT_h, where T_h is the time required by one invocation of $h()$. (We are ignoring the last invocation of $h()$ where the length of the message is hashed; this step is required by both SHA-256 and PARSHA-256.) We now compare this to the number of invocations of $h()$ and the number of parallel rounds required by PARSHA-256.

Recall from Section 4.1 that the number of parallel rounds required by PARSHA-256 is R. Thus the time required for parallel execution of PARSHA-256 is RT_h. The number of invocations of $h()$ by PARSHA-256 is same as the number of invocations of $h()$ by PHA in [10].

Proposition 1. *The number of invocations of* $h()$ *by* PARSHA-256 *on a message of length* L *is equal to* $(q+2)2^t + 2b - 2$.

The parameters q and b depend on L, t, l, n and m. We have the following result.

Proposition 2. $\frac{L}{\lambda(t)} - 1 < q + 2 < \frac{L}{\lambda(t)}$.

Note that $\lambda(t)$ depends on t, l, n and m. For convenience of comparison we assume $\gamma = 2^\rho$ for some positive integer ρ. Also $\lambda(t) = 2^{t-1}(2n - 2m - l)$. Hence we have

$$\frac{2^{\rho - t}}{1 - \frac{l}{2(n-m)}} - 1 < q + 2 < \frac{2^{\rho - t}}{1 - \frac{l}{2(n-m)}}. \tag{5}$$

We define two ratios to compare PARSHA-256 and SHA-256. The first ratio is the round factor RF which compares the number of parallel rounds required by PARSHA-256 to the number of rounds required by SHA-256. The lesser the value of RF the more is the speed-up attained by PARSHA-256 over SHA-256. The second ratio is the invocation factor IF which compares the number of invocations of the compression function $h()$ made by PARSHA-256 to SHA-256. These two ratios are defined as follows.

$$\left.\begin{aligned} \mathsf{RF} &= \tfrac{q+t+2}{2^\rho}; \\ \mathsf{IF} &= \tfrac{(q+2)2^t + 2b - 2}{2^\rho}. \end{aligned}\right\} \tag{6}$$

Note that both RF and IF depend on l. Table 1 provides the values of RF and IF under different values of l. For practical implementations the value of t will be small (typically between 3 and 8). The number of processors used is 2^t and hence ideally we should have the round factor RF to be 2^{-t}. For moderately long messages (around 1Mbyte) this is true for $l = 0$. For $l > 0$ the RF is more. However, for all l, the round factor RF decreases with increase in message length.

Table 1. Comparison of RF and IF for $L = 2^\rho(n - m)$

l	RF	IF
0	$\approx 2^{-t} + \frac{t}{2^\rho}$	≈ 1
128	$\approx 1.14 \times 2^{-t} + \frac{t}{2^\rho}$	≈ 1.14
256	$\approx 1.33 \times 2^{-t} + \frac{t}{2^\rho}$	≈ 1.33

Also for a fixed length L (i.e., a fixed ρ), the factor $\frac{2^{-t}}{\mathsf{RF}}$ decreases as t increases which implies that for a fixed length the efficiency of speed-up decreases with increasing t. However, the actual speed-up increases as t increases.

For $l = 0$, the number of invocations of $h()$ made by PARSHA-256 is equal to the number of invocations of $h()$ made by SHA-256. For $l > 0$, the number of invocations made by PARSHA-256 is more than that made by SHA-256. This is due to the use of IV. Thus a strictly sequential simulation of PARSHA-256 will require more time than SHA-256. In the next section, we show that for long messages a multithreaded implementation of PARSHA-256 on a single processor machine can lead to a speed-up over SHA-256.

6 Multithreaded Implementation

We implement the algorithm to compute PARSHA-256 using threads. The processors are implemented using threads and the simultaneous operation of the processors is simulated by concurrent execution of the respective threads. There are R parallel rounds in the algorithm.

Each round consists of two phases – a formatting phase and a hashing phase. In the formatting phase, the inputs to the processors are formed using the message and the outputs of the previous invocations of $h()$. Once this phase is completed, the hashing phase starts. In the hashing phase all the processors operate in parallel to produce the output. We use two buffer sets – the input and the output buffer sets. The input buffer set consists of 2^t strings of length n each. Similarly, the output buffer set consists of 2^t strings of length m each. Thus each processor has its own input buffer and output buffer. In the formatting phase, the input buffer sets are updated using the message and the output buffers. In the hashing phase, the input buffers are read and the output buffers are updated. During implementation we declare the buffers to be global variables. This avoids unnecessary overhead during thread creation.

The formatting phase prepares the inputs to all the processors. This phase is executed in a sequential manner. That is, first the input to processor P_0 is prepared, then the input to processor P_1 is prepared and so on for the required number of processors. After the formatting phase is complete, the hashing phase is started. The exact details of processor invocation are given as follows.

Rounds 1 to $q + 1$:
  $P_0, \ldots, P_{2^t-1}$ each invoke the compression function.
Round $q + 2$:
  $P_0, \ldots, P_{2^{t-1}+b-1}$ each invoke the compression function.
Round i with $q + 2 < i < R$:
  $P_0, \ldots P_{K_i-1}$ each invoke the compression function.
Round R :
  if $b > 0$, then P_0 invokes the compression function.

Here K_i is as defined in Equation (3). Note that in rounds $q + 2$ to R at most one processor may additionally output the m-bit input that it receives in the previous round. (See Remark 1 for further explanation.)

Table 2. Details of different test platforms

	Silicon Graphics O2	P4
Number of CPU	1	1
Processor	MIPS R12000A	Intel Pentium 4
Processor Speed	400MHz	1.40 GHz
Main Memory	512 MB	256 MB
OS	IRIX 6.5	RedHat Linux 8.0

Each processor is simulated using a thread. In the hashing phase of each round, the required threads are started. Each thread is given an integer j, which identifies the processor number and hence the input and the output buffers. Also each thread gets the address of the start location of the subroutine $h()$. The subroutine $h()$ is implemented in a thread safe manner, so that conflict free concurrent execution of the same code is possible. The management strategy for the input and output buffers described above ensure that there is no read/write conflict for the buffers even during concurrent execution. The hashing phase is completed only when all the started threads successfully terminate. This also ends one parallel round of the algorithm. Finally the algorithm ends when all the parallel rounds are completed.

There is another way in which concurrent execution can be further utilized. As described before there are two phases of each round – the reading/formatting phase and the hashing phase. It is possible to introduce concurrency in these two phases in the following manner. Suppose the system is in the hashing phase of a particular round. At this point it is possible to concurrently execute the reading/formatting phase of the next round. The advantage is that in the next round the hashing phase can be started immediately, since the reading/formatting phase of this round has been completed concurrently with the hashing phase of the previous round. In situations where memory access is slow, this method will provide speed improvements. On the other hand, to avoid read/write conflict, we have to use two sets of buffers, leading to a more complicated buffer management strategy. For our work, we have chosen not to implement this idea.

7 Experimental Results

First we note that if PARSHA-256 is simulated sequentially then the time taken is proportional to the number of invocations of $h()$. From Table 1, we know that for a *strict sequential execution* PARSHA-256 will be roughly as fast as SHA-256 when $l = 0$ and PARSHA-256 will be $\frac{1}{\mathsf{IF}}$ times slower than SHA-256 when $l > 0$. Also for full parallel implementation the speed-up of PARSHA-256 over SHA-256 is determined by the factor RF in Table 1.

In this section we compare the performance of the *multithreaded implementation* of PARSHA-256 with SHA-256. The experiments have been carried out on two platforms (see Table 2). The algorithms have been implemented in C and the same code was executed on both the platforms. The order of bytes in a long on the two platforms are different; this was the only factor taken into account while running the program.

Remark 2. The compression function $h()$ for SHA-256 is also the compression function of PARSHA-256. We implemented $h()$ as a subroutine and this subroutine was invoked by both SHA-256 and PARSHA-256. Thus the comparison of the two implementation is really a comparison of the two composition principles. Any improvement in the implementation of the compression function $h()$ will improve the speed of both SHA-256 and PARSHA-256 but the comparative performance ratio would roughly remain the same.

To provide a common platform for comparison, the same background machine load was maintained for the execution of both SHA-256 and PARSHA-256. For comparison purposes we have calculated the difference in clock() between the start and end of the program for both SHA-256 and PARSHA-256. Extensive experiments were carried out for comparison purpose and a summary of the main points is as follows.

- On P4 running Linux, the following was observed. For long messages of around 1 Mbyte or more, the multithreaded implementation of PARSHA-256 was faster by a factor of 2 to 3 for all values of l.
- On SG, the speed of both the algorithms was roughly same for $l = 0$ and 128. For $l = 256$, the speed of PARSHA-256 was roughly 0.85 times the speed of SHA-256.
- For short messages, the multithreaded implementation was slower. This is possibly due to higher thread management overhead.
- The gain in speed decreases as l increases. This is due to the increase in the number of invocations of the compression function as shown in Table 1.
- The gain in speed increases with increase in message length. However, the rate of increase is slow.

As an outcome of our experiments, we can conclude that on P4 running Linux and for long messages, the multithreaded implementation of PARSHA-256 is roughly 2 to 3 times faster than SHA-256.

8 Conclusion

In this paper, we have presented a new hash function PARSHA-256. The hash function is built using the SS composition principle and the compression function of SHA-256. Since the SS composition principle is parallelizable, our hash function is also parallelizable. A full parallel implementation of PARSHA-256 will show a significant speed-up over SHA-256. In this paper, we have described a concurrent implementation of PARSHA-256 on a single processor machine. Experimental results show that for long messages the concurrent implementation is still faster than SHA-256.

The basic idea explored in the paper is that it is possible to obtain secure and parallelizable hash functions by combining the SS composition principle with a "good" compression function. We have done this using the compression function of SHA-256. Using other "good" compression functions like RIPEMD-160 or other SHA variations will also yield new and fast parallel hash functions. We believe this task will be a good research/industrial project with many practical applications.
Acknowledgement : We would like to thank the reviewers of the paper for their detailed comments, which helped to considerably improve the description of the hash function.

References

1. 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.
2. 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.
3. I. B. Damgård. A design principle for hash functions. *Lecture Notes in Computer Science*, 435 (1990), 416-427 (Advances in Cryptology - CRYPTO'89).
4. H. Dobbertin, A. Bosselaers and B. Preneel. RIPEMD-160: A strengthened version of RIPEMD. *Cambridge Workshop on Cryptographic Algorithms*, 1996, LNCS, vol 1039, Springer-Verlag, Berlin 1996, pp 71-82.
5. H. Dobbertin. Cryptanalysis of MD4. *Journal of Cryptology*, 11(4): 253-271 (1998).
6. L. Knudsen and B. Preneel. Construction of Secure and Fast Hash Functions Using Nonbinary Error-Correcting Codes. *IEEE Transactions on Information Theory*, vol. 48, no. 9, September 2002, pp 2524–2539.
7. R. C. Merkle. One way hash functions and DES. *Lecture Notes in Computer Science*, 435 (1990), 428-226 (Advances in Cryptology - CRYPTO'89).
8. J. Nakajima, M. Matsui. Performance Analysis and Parallel Implementation of Dedicated Hash Functions. *Lecture Notes in Computer Science*, (Advances in Cryptology - EUROCRYPT 2002), pp 165-180.
9. 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).
10. P. Sarkar and P. J. Schellenberg. A Parallelizable Design Principle for Cryptographic Hash Functions. *IACR e-print server*, 2002/031, http://eprint.iacr.org.
11. C. Schnorr and S. Vaudenay. Parallel FFT-Hashing. *Lecture Notes in Computer Science*, Fast Software Encryption, LNCS 809, pages 149-156, 1994.

A Test Vector

Our implementation of PARSHA-256 is available at `http:`
`www.isical.ac.in/~ crg/software/parsha256.html.`

The test vector that we use is the string : $(abcdefgh)^{128}$. (Note that the corresponding files for little and big endian architectures are going to be different.)

Each of the characters represent a byte and the entire string is of length 1 Kbyte. We run PARSHA-256 for $t = 3$ and for $l = 0, 128$ and 256. Denote the resulting message digests by d_1, d_2 and d_3. Each d_i is a 256 bit value and we give the hex representations below.

d_1 is as follows.

```
4d4c2b13 3e516dc1 35065779 536fd4bf
74f98189 bc6b2a92 10803d38 77e3b656
```

d_2 is as follows.

```
e554c47b 1538c9db 5cbff219 2d620fd3
ae21d04a 5ae6fa50 150888cc da6cf783
```

d_3 is as follows.

```
459142c5 fcd6eff6 839d6740 177b54d5
2e8bc987 a7438438 a588441a 7113e8d3
```