Hash Function *Luffa*

Supporting Document

Christophe De Cannière
ESAT-COSIC, Katholieke Universiteit Leuven

Hisayoshi Sato, Dai Watanabe
Systems Development Laboratory, Hitachi, Ltd.

15 September 2009

Contents

1 Introduction

This document provides the currently known results on the security and the performances of a family of hash function *Luffa*. We refer to [28] for the specification and the notations used throughout this document.

1.1 Updates of This Document

The algorithm of *Luffa* in Round 2 is modified from the algorithm submitted to Round 1. We modified two functions; `SubCrumb` and the finalization. Please refer to [29] for the modifications and their reasons. The Round 1 algorithm and the Round 2 algorithm of *Luffa* are denoted by *Luffa* v1 and *Luffa* v2 respectively.

This document mainly deals with the evaluation results of *Luffa* v2 and the version is omitted if it does not matter. In fact, most of the security evaluation results are common between two algorithms because the modifications are minor. The significant update in the supporting document is that the higher order differential attack on *Luffa* v2 is studied in Section 3.4. The higher order differential attack on *Luffa* v1 is studied in the separate document [30]. All implementation results are updated due to the modifications of the algorithm. In addition, we mention the free-start attacks in Section 5.

1.2 Organization of This Document

The rest of this document is organized as follows: Firstly the design rationale of *Luffa* is presented in Section 2 to clarify that there is no backdoor in our design. Secondly the security issues of *Luffa* are discussed. The security of some randomness properties, especially the differential characteristics, of the non-linear permutation is discussed in Section 3. The security of the chaining is discussed in Section 4 by assuming that the underlying permutations are ideal function. We mention the free-start attacks in Section 5. After that the implementation issues are discussed in Section 6. Also the performances on some platforms are given in Section 6.

2 Design Rationale

Luffa is a family of cryptographic hash functions suitable for multipurpose. Besides NIST's requirements, we would like to make the design as simple as possible.

2.1 Chaining

The chaining method of *Luffa* hash function is a variant of a sponge function [7, 8] whose security is based only on the randomness of the underlying permutation. Although the simple construction is very attractive, a sponge function costs more than a traditional block cipher based hash construction from the viewpoint of implementations. In the case of PGV construction, the block length n_b of the underlying block cipher is equal to the hash length n_h (A key scheduling function should be taken into account in practice). On the other hand, a sponge function requires a permutation of $n_b \geq 2 \cdot n_h + m$ bits length, where m is the message block length. Generally speaking, the calculation cost of a function increases if the input (and the output) length get larger. In addition, constructing proper large permutations for each hash length is far from a scalable design.

These undesirable properties of a naive sponge lead us to its variant such that the underlying function is a family of sub-permutations with shorter input/output. The resulting chaining method *Luffa* employs the message injection function and one blank round.

The message injection functions of *Luffa* are linear functions. These functions are designed to avoid the serious attack described in Section 4.4 caused by the simple property of keeping the independency of input/output of each sub-permutations. More precisely, the message injection functions are designed to have the maximum branch number, thus the input of each round is sufficiently mixed and the independency is kept to the minimum. Consequently, it is conceivable that these components make the construction *Luffa* sufficiently intractable to find a collision. Moreover we believe that these components also provide sufficient resistance against the preimage/second preimage attacks.

The finalization of *Luffa* consists of one blank round and an output function. One blank round is appended in order to be resistant against unknown future attacks.

The block length of the sub-permutations is chosen to be suitable for 256, 384 and 512 bits hash length. It is possible to choose the shorter block length, however all sub-permutations must be different each other so that too small block length is not practical. In addition, a bit slice permutation has more flexibility in the design if the block length is large. Therefore 256 bits is chosen as a block length of a sub-permutation.

2.2 Non-Linear Permutation

The sub-function of *Luffa* adopts a bit slice substitution permutation network (SPN). The reasons to choose an SPN more than a look up table (LUT) based function are as follows:

- If the CPU design is evolved, the throughput increases.
- A cipher consisting of logical operations is believed to be secure against cash timing attacks.
- It seems easier to implement compared to LUTs.

The first is the main reason to adopt a bit slice permutation. For example, four `SubCrumb` can be executed at once with the SSE instructions of Intel Core2 processors and the throughput is much faster than the code with 32-bit instructions only.

2.2.1 Sbox in `SubCrumb`

Serpent [4, 9] and Noekeon [12] are typical examples of a bit slice block cipher and they adopt Sboxes of 4 bits input/output.

In the proposal of Serpent, the designers explained the eight Sboxes are almost randomly generated and ones satisfying good differential/linear properties are chosen. In addition, those representation as Boolean functions should have the highest degree, namely three. The efficiency of the implementations is not well considered. Osvik proposed an efficient method to find a good sequence for a given Sbox on Intel 586 processors [24]. In his experimental results, S_2 of Serpent is the fastest, the number of instructions is 16 and it can be executed in 8 cycles. Note that his method is not exhaustive, so that resultant sequence might not be optimal. In addition, how to choose the optimal Sbox in software implementation is not clarified in his approach.

On the other hand, the Sbox of Noekeon is defined as a sequence of instructions by nature. Noekeon is intended to have symmetric property, i.e., the encryption and the decryption can be done by the same function. The reason to design the Sbox in this manner might be that the Sbox and the inverse must be equal. Unfortunately, the given set of instructions of Noekeon's Sbox is not suitable for software implementations.

Our approach to design the Sbox is similar to Noekeon. In other words, it is defined by the sequence of instructions to have a desirable property such that the implementation on Intel Core2 Duo processors is optimal. By simple thought experiment, we believe that at least five cycles are needed in order to achieve optimal differential/linear probability. In fact, we found a few Sboxes satisfying those properties. The smallest Sbox consists of 9 instructions and it is executable in 5 cycles.

The Sbox chosen for *Luffa* consists of 16 instructions and it is executable in 6 cycles[1]. It seems to have some good properties as follows:

- The maximum differential probability is 1/4.
- The maximum linear probability is 1/4.
- It has no fixed point.
- The degree of the Boolean representation is 3 for all output bits.

2.2.2 `MixWord`

The linear diffusion layer of the sub-permutation of *Luffa* consists of XORings and rotations in the same way as Serpent and Noekeon. Those Sboxes do not mix bits at a different bit position in a word, but mix bits at a same bit position in different words, so that the linear diffusions are intended to mix the bits at different bit positions.

Different from those 128-bit block ciphers, the linear diffusion of *Luffa* is required to mix the outputs of different `SubCrumb`. In order to achieve these

[1] The Sbox of *Luffa* v2 is different from that of *Luffa* v1. Please refer to [29] for the reason to change the Sbox

requirements, a traditional Feistel ladder with rotations is chosen, where the inputs are x_k and x_{k+4}. The rotations are removed from the rundles of the ladder to execute an XORing and a rotation in parallel in software implementations. The number of iterations is chosen to be four because we believe that the weight of the non-linear mixing should be nearly equal to that of the linear diffusion.

Now we are going to explain how to choose the rotations σ_1, σ_2, σ_3 and σ_4. Firstly, the linear code given by the iteration of `MixWord` is considered. Let A be the representation matrix of `MixWord` and $G_n = I||A||\cdots||A^n$ be the generator matrix of a code. We searched for the lowest weight code word generated from low weight inputs. Most of the candidates seems to have the same diffusion property for $n \leq 4$, and the chosen parameter shows the best property.

Next, the polynomial representation of `MixWord` is considered. An XORing and a rotation can be represented as the operations on a polynomial ring GF(2)$[t]$. Let the two input words be $a(t)$ and $b(t)$, then the output is given by

$$\begin{aligned} a'(t) &= (1+t^{\sigma_1}+t^{\sigma_2}+t^{\sigma_3}+t^{\sigma_1+\sigma_3})a(t)+(1+t^{\sigma_2}+t^{\sigma_3})b(t), \\ b'(t) &= t^{\sigma_4}(1+t^{\sigma_1}+t^{\sigma_2})a(t)+t^{\sigma_4}(1+t^{\sigma_2})b(t). \end{aligned}$$

It is clear from the equations that the branch number of `MixWord` is upper-bounded by six (In practice, it is five).

The notable input is $(0, b)$ and the corresponding output becomes $(a', 0)$ if $b(x)(1 + x^{\sigma_2}) = 0$, i.e. b should have a zero divisor. Such a differential path with small number of active Sboxes is not desirable from the security viewpoint. The lowest Hamming weight of such inputs b is given by $32/p$, where $\sigma_2 = 2^p(2\tau+1)$. In addition, the Hamming weight of the corresponding output $(a', 0)$ is also $32/p$. This fact indicates that the factor of 2 in σ_2 should be small.

On the other hand, we restricted the parameters σ_1, σ_2 and σ_3 to be even in order to make it feasible to search for the best (truncated) differential path for 4 steps. Then a step of `MixWord` does not mix odd bits and even bits. In other words, the transformation can be separated into two independent functions. Note that the above choice might reduce the diffusion property in general. σ_4 should be odd in order to mix odd bits and even bits at the next step.

Throughout the above process, a set of parameters $\sigma_1 = 2, \sigma_2 = 14, \sigma_3 = 10, \sigma_4 = 1$ is chosen for *Luffa*.

2.2.3 Constants

The step constants are used for several reasons as follows:

- The sub-permutation Q_j MUST be different each other.
- The step function at each step SHOULD be different each other to prevent slide attacks and to remove fixed points.
- There SHOULD not be a kind of symmetry in the input/output.

From the viewpoint of implementations, it is better to generate the constants by a simple circuit with a fixed starting variable. Therefore we designed a small constant generator with a linear update function for *Luffa*.

2.2.4 Number of Steps

In order to achieve good throughput comparable to SHA-256, the number of steps is fixed to 8.

On the other hand, we found the differential path for 8 steps with probability 2^{-224}, and proved that there is no differential path with probability more than 2^{-124}. We do not think it is necessary for the sub-permutation Q_j to achieve full security (namely, the maximum differential probability should be not more than 2^{-256}, etc.). The collision attack based on a differential path is discussed in Section 3.2.

2.2.5 Tweaks

A tweak is applied in order to differentiate each permutation. And we also expect that the tweak will break a nature of the message injection function defined over a direct product ring. Especially, it will become hard to choose same input differences for the different permutations. The operations are chosen not to increase the implementation cost in terms of both the size and the performance.

Table 1: The differential profile of the Sbox S

	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
1	4	0	2	0	2	2	2	2	0	0	0	0	0	0	2
2	0	4	0	0	0	0	0	0	0	0	0	4	4	0	4
3	0	0	2	0	0	0	2	0	0	4	2	2	2	2	0
4	0	0	0	4	0	0	4	4	0	0	4	0	0	0	0
5	0	0	2	0	0	4	2	0	0	0	2	2	2	2	0
6	0	0	0	2	2	2	2	2	2	2	2	0	0	0	0
7	4	0	2	2	0	0	0	0	2	2	2	0	0	0	2
8	0	4	0	0	2	2	0	0	2	2	0	0	4	0	0
9	0	0	2	2	4	0	0	2	0	0	0	2	2	2	0
a	0	4	0	2	0	0	2	2	0	0	2	0	0	4	0
b	4	0	2	0	0	0	2	2	2	2	0	0	0	0	2
c	0	4	0	0	2	2	0	0	2	2	0	0	0	4	0
d	4	0	2	2	2	2	0	0	0	0	2	0	0	0	2
e	0	0	0	0	2	2	0	0	2	2	0	4	0	0	4
f	0	0	2	2	0	0	0	2	4	0	0	2	2	2	0

3 Security Analysis of Permutation

This section shows some known properties of the non-linear permutation Q_j.

3.1 Basic Properties

3.1.1 Sbox S (*Luffa*v2)

Table 1 shows the differential probabilities corresponding to input and output differences. The maximum differential probability of the Sbox S is 2^{-2}.

Table 2 shows the biases of the linear approximation defined by corresponding input and output masks. The maximum linear probability of the Sbox S is 2^{-2}.

Let x_0, x_1, x_2, x_3 and y_0, y_1, y_2, y_3 be the 4-bit input and output of the Sbox. Then the algebraic normal form of the Sbox is given by

$$\begin{aligned}
y_0 &= 1 + x_0 + x_1 + x_1x_2 + x_0x_3 + x_1x_3 + x_0x_1x_3 + x_0x_2x_3, \\
y_1 &= x_0 + x_0x_1 + x_1x_2 + x_3 + x_0x_3 + x_1x_3 + x_0x_1x_3 + x_0x_2x_3, \\
y_2 &= 1 + x_1 + x_0x_2 + x_1x_2 + x_0x_1x_2 + x_3 + x_1x_3 + x_0x_1x_3 + x_2x_3, \\
y_3 &= 1 + x_1 + x_2 + x_0x_2 + x_1x_2 + x_0x_1x_2 + x_0x_3 + x_1x_3 + x_0x_1x_3 + x_2x_3.
\end{aligned}$$

Note that the number of monomials which appear in the polynomial expression is smaller than that of a randomly generated Sbox. The polynomial expression of the Sbox adopted to *Luffa* v2 has more monomials than the Sbox for *Luffa* v1, but it should be noted that $y_0 + y_1$, $y_2 + y_3$ are still very sparse. We have not found any practical attack on *Luffa* v2 using this property. The higher order differential attack on *Luffa* v2 is studied in Section 3.4.

3.1.2 Differential Propagation

It is easy to see that the branch number of `MixWord` is 5. We confirmed that the minimum number of active Sboxes of 4 steps is 31. Therefore the maximum differential characteristic probability (MDCP) of Q_j is upper bounded by 2^{-124}. The MDCP of 4 steps is estimated by the exhaustive 4-bit-wise truncated differential path search, but the direct path search for more than 4 steps is computationally infeasible.

We also searched for the good differential path for 8 steps to get close to the real bound. We consider only the case that the output differences of Sboxes at different positions in the same step are equal. Under this assumption, the search for a good differential path of the permutation Q_j is equivalent to search for a low weight code word of the linear code defined by

Table 2: The linear profile of the Sbox S

	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
1	4	0	0	0	4	0	0	0	4	0	0	0	4	0	0
2	2	2	0	0	2	2	4	4	2	2	0	0	2	2	0
3	2	2	0	4	2	2	0	0	2	2	4	0	2	2	0
4	2	2	0	2	4	0	2	2	0	0	2	4	2	2	0
5	2	2	0	2	0	0	2	2	4	0	2	4	2	2	0
6	0	4	0	2	2	2	2	2	2	2	2	0	0	0	4
7	0	0	0	2	2	2	2	2	2	2	2	0	0	4	4
8	0	0	4	2	2	2	2	2	2	2	2	0	0	4	0
9	0	4	4	2	2	2	2	2	2	2	2	0	0	0	0
a	2	2	4	2	0	4	2	2	0	0	2	0	2	2	0
b	2	2	4	2	0	0	2	2	0	4	2	0	2	2	0
c	2	2	0	0	2	2	0	0	2	2	0	4	2	2	4
d	2	2	0	0	2	2	0	0	2	2	0	4	2	2	4
e	0	0	0	4	0	4	0	4	0	4	0	0	0	0	0
f	4	0	0	0	0	0	4	0	0	0	4	0	4	0	0

the iteration of `MixWord` (See Section 2.2.2 in detail for the definition of this code). We applied Leon's probabilistic algorithm [23] to find a low weight code word and Table 3 shows the best differential path so for.

Table 3: The best known (truncated) differential path

0	00000000000100000000111101010100 00000100000101000100010100010100	16
1	00000111000010000011011000000000 00101101101010001110110000100010	22
2	00000000000000000000000001010000 00000000000100010001000100010100	8
3	00000000000000000000000000000000 10001000100000000010000000000000	4
4	10000000000000000000000000000010 00000001000000000000010001000001	6
5	100000000000000011100001000001101 00000000010000111001000000010011	16
6	00011110101010000000000000100000 00000101010000010100010001000001	16
7	00010101010000101000011110011110 01100000101000100000010000110110	24

The most left column in the table means the step so that r-th line (counting from 0) means the 4-bit-wise truncated differences which is input to the $r+1$-th step function. The most right column means the number of active Sboxes at each step. The above path has 112 active Sboxes in total so that the differential characteristic probability of the path is 2^{-224}. It should be remarked that 8 step functions cannot be considered a perfect random permutation.

Table 4: The best known differential probabilities

Number of steps	Diff. probability
4	$\leq 2^{-62}$
5	2^{-100}
6	2^{-144}
7	2^{-176}
8	2^{-224}

The best known differential characteristic probabilities up to 8 steps are summarized in Table 4. Note that the best known differential paths for more than 4 steps is a part of the 8 steps differential path. The line 2-6, line 1-6, 0-6 in Table 3 correspond to the probability for 5, 6, 7 steps respectively in Table 4.

3.2 Collision Attack Based on A Differential Path

Here we discuss a differential based collision attack consisting of two message blocks. In other word, how to find a message pair $(M^{(i)}, M^{(i+1)})$ and $(M^{(i)} \oplus \Delta^{(i)}, M^{(i+1)} \oplus \Delta^{(i+1)})$ such that $MI(\texttt{Round}(H^{(i-1)}, M^{(i)}), M^{(i+1)}) = MI(\texttt{Round}(H^{(i-1)}, M^{(i)} \oplus \Delta^{(i)}), M^{(i+1)} \oplus \Delta^{(i+1)})$ and the cost are discussed.

3.2.1 General Discussion

Let p_j be the maximum differential characteristic probability of Q_j and $p = \max_j p_j$. If some input bits are chosen adequately, the differential probability with the condition tends to be higher than p. This technique is well-known as a message modification in general. In order to simplify the discussion, we assume that a bit constraint of an input improves the differential probability double, i.e., the differential characteristic probability under m bits constraints is assumed to be $2^m \cdot p$. Note that this is hard to happen in practice because some of the constraints are often conflictive with the others.

The attack with messages of two rounds requires all the output differences of Q_j are not zero. Algorithm 1 describes the procedure of the attack.

Algorithm 1 Differential based collision attack

```
  **Step 1** Choose a good internal state $H^{(i-1)}$ (which can satisfy constraints
  as many as possible) by moving $M^{(i-1)}$ randomly.
  **Step 2** Choose a part of the message block $M^{(i)}$ to satisfy constraints.
  **Step 3** Move the rest of the message block $M^{(i)}$ and check if the output
  differences are equal. If not, go back to Step 1.
```

Let H_{pre}, M_{pre} and M_{online} be the numbers of messages to be used in Step 1, 2 and 3 respectively. Let H_{online} be the number of iterations of the whole procedure. The differential probability under the constraints is roughly given by $p^w \cdot H_{pre} \cdot M_{pre}$ and the total number of trials is given by $H_{online} \cdot M_{online}$. Then an inner collision will be found if the following inequality is satisfied:

$$p^w \cdot (H_{pre} \cdot M_{pre})(H_{online} \cdot M_{online}) \geq 1.$$

The calculation complexity of the attack is given by $H_{online}(H_{pre} + M_{online})$ and it should be smaller than $2^{\frac{w-1}{4}n_b}$ for the attack being faster than a birthday attack to find a collision of outputs. Therefore $H_{pre} \cdot H_{online} < H_{online}(H_{pre} + M_{online}) \leq 2^{\frac{w-1}{4}n_b}$ is the necessary condition. In addition, $M_{pre} \cdot M_{online}$ is upper-bounded by 2^{n_b}. Therefore the lower bound of the maximum differential characteristic probability of Q_j (for a successful attack) is given by

$$p \geq 2^{-\frac{n_b}{4}(1+\frac{3}{w})}.$$

For $w = 3, 4, 5$, $p \geq 2^{-128}$, 2^{-112}, $2^{-102.4}$ are the necessary conditions respectively.

Finding an inner collision should be also considered because an inner collision can be used to find a second preimage and preimage as well as to find a collision for sponge variants. The discussion for the inner collision can be done in the same manner. In this case, the collision attack is successful if the number of operations is less than $2^{n_b(w-1)/2}$. Therefore $p \geq 2^{-170.7}$, 2^{-160}, $2^{-153.6}$ are the necessary conditions for $w = 3, 4, 5$ respectively.

The currently known best differential characteristic probability of Q_j is 2^{-224} and the tweaks will make it difficult to find a tuple of good differential paths, therefore we believe that *Luffa* is secure against this attack.

3.2.2 How to Find A Collision for 5 Steps without Tweaks

As an example to find an inner collision in practice, we describe a naive attack for 5 steps using the line 1-5 in Table 3. In order to simplify the discussion, we ignore the influences of the multiplications in the message injection function and the tweaks. In addition, we consider only *Luffa*-256, i.e., $w = 3$.

Before starting the attack, the bit-wise differences should be chosen and we can easily find an iterative sequence of differences (`0x1` $\rightarrow$ `0x1`) and (`0x2` $\rightarrow$ `0x2`) from Table 1. A 4-bit data $a_{j,3,l}||a_{j,2,l}||a_{j,1,l}||a_{j,0,l}$ (or $a_{j,7,l}||a_{j,6,l}||a_{j,5,l}||a_{j,4,l}$) are called a *crumble* in the following. Let us consider the crumbles corresponding to ones (we call them active crumbles) in the difference at the line 1. If all the input bits to these Sboxes are chosen to be adequate, then the input differences `0x1` are mapped to `0x1` with probability 1. Each input to a crumble should hold two bits of constraints in order to satisfy the above condition. In Step 1, the attacker tries to find a state $H^{(i-1)}$ whose active crumbles satisfy constraints. The number of constraints here is $22 \cdot 2 \cdot 3 = 132$. In Step 2, the attacker chooses a message $M^{(i)}$ such that the above mentioned 4-bit crumbles are all zero. The number of constraints in this step is $22 \cdot 4 = 88$. Now the attacker has $256 - 88 = 168$ bits freedom in the message $M^{(i)}$. On the other hand, there are still $(8+4+6+16) \cdot 3 = 102$ active Sboxes, so that 2^{204} trials are needed to find an input pair which follows the differential path. Therefore $2^{204-168} = 2^{36}$ iterations of whole steps are needed. As a result, the total complexity of the attack is estimated at $2^{36}(2^{132} + 2^{168}) \approx 2^{204}$.

We expect that more detailed analysis such as the message modification at the next step certainly reduces the calculation complexity of the attack on 5 steps without tweaks, and may allow to attack 5 steps with tweaks or 6 steps. But the rough estimate in Section 3.2.1 tells that this kind of approach never reach to more than 6 steps. A multiple differential path search also helps to improve the attack, but we think the attack on 7 steps is hard to expect.

3.3 Birthday Problem on The Unbalanced Function

The standard birthday problem assumes that the underlying set is uniformly distributed. Bellare and Kohno discussed the birthday problem for unbalanced distributions and proved that the collision happens more often if the underlying distribution is not uniform [6]. The "non-randomness" of the permutation Q_j may tempt to apply their result to *Luffa*. Though the distribution of the outputs of the differential of Q_j is not uniform, to find a collision is equivalent to find a preimage of zero in the differential of Q_j. Therefore we believe that the application of the result on the unbalanced birthday problem is not possible.

3.4 Higher Order Differential Distinguisher

NIST requires the SHA-3 Candidates to support the random number generation defined in SP 800-90 [14]. Therefore the deterministic random bit generator (DRBG) which adopts SHA-3 Candidates is necessary to indistinguishable from a random function.

On the other hand, a hash function does not have an additional input other than a message so that it is not a pseudorandom function. The random distribution of the output is not a necessary condition for a collision resistance. However, non-random distribution of the output of the hash function is considered an undesirable property for applications mentioned above.

In this section, we study the randomness property of *Luffa* and the underlying permutation Q_j. We use the terminology *distinguisher* for functions which detect a kind of non-randomness according to [2]. Note that a distinguisher is usually a terminology for a function which distinguishes two random variables.

3.4.1 Higher Order Difference

An application of a higher order difference to cryptanalysis is suggested by Lai [22] and firstly applied to a block cipher by Knudsen [20]. Here we briefly introduce the definition of higher order difference and its basic property.

Let $Y = E(X; K)$ be a function where $X \in \mathrm{GF}(2)^n$, $Y \in \mathrm{GF}(2)^m$ and $K \in \mathrm{GF}(2)^s$. Let $\{A_1, \ldots, A_i\}$ be a set of linearly independent vectors in $\mathrm{GF}(2)^n$ and $V^{(i)}$ be the sub-space spanned by these vectors. The i-th order difference is defined by

$$\Delta_{V^{(i)}} E(X; K) = \sum_{A \in V^{(i)}} E(X + A; K).$$

In the following, $\Delta^{(i)}$ denotes $\Delta_{V^{(i)}}$ if the choice of $V^{(i)}$ does not matter in the discussion. The basic fact of the higher order difference is that $\Delta^{(D+1)} E(X; K) = 0$ if the algebraic degree of E with respect to X is D. Therefore the higher order difference is uses as the tool to evaluate the algebraic property, especially the degree of the target function.

3.4.2 Higher Order Differential Attack on *Luffa*v1

Here the higher order differential distinguishing attack on step-reduced variant of *Luffa* v1 is introduced. Please refer to [30] for the detailed attack.

First of all, we found that the algebraic degree of the underlying non-linear permutation Q_j grows slower than an ideal case. In addition, we found that there are very efficient distinguishers for the step-reduced variant of *Luffa* v1. According to our theoretical estimate, we can construct distinguishers for step-reduced variants of *Luffa* v1 up to 7 out of 8 steps by using a block message. The expected degrees of the distinguishers are summarized in Table 5.

Table 5: The summary of the algebraic degrees of distinguishers for *Luffa* v1

Number of steps	1	2	3	4	5	6	7	8
Degree	–	2	5	13	33	84	214	545

Table 5 indicates that the algebraic degree of the distinguisher for 7 steps is expected 214 and the distinguisher requires 2^{216} inputs. This distinguisher can be extended to the step-reduced variant of the hash function by the careful choice of the messages without an extra cost because the hash value for a block message is just the XORing of the outputs of Q_j in *Luffa* v1.

3.4.3 Higher Order Differential Property of Q_j of *Luffa*v2

Now we consider the resistance of *Luffa* v2 against the higher order differential attack. Table 6 shows that the theoretical estimates of algebraic degrees of distinguishers for Q_j. In the estimation we ignore the influence of the shuffling of the order of the input words to `SubCrumb` because it make the estimate very difficult. In other words, we estimated the weakened variant of `SubCrumb` in which the order of the input words are the same as that of *Luffa* v1, namely,

```
SubCrumb(a[0],a[1],a[2],a[3]);
SubCrumb(a[4],a[5],a[6],a[7]);
```

Table 6: Algebraic degrees of distinguishers for weakened *Luffa* v2

Number of steps	1	2	3	4	5	6	7	8
Degree	–	2	5	13	35	94	252	675

Next we estimate the success probability of the distinguishing attack by the computational experiments. We move only bits in `a[0]` so that a bit per an Sbox is varying. In other words, we applied t-th order difference by moving the most least t bits of the variable $x_0^{(0)}$. The distinguisher can skip the effect by the SubCrumb in the first step by this choice of varying bits.

As mentioned in Section 3.1.1, $y_0 + y_1$, $y_2 + y_3$ are of low degrees so that it is expected that $x_k^{(r)} + x_{k+1}^{(r)}$ for $k = 0, 2, 4, 6$ are useful distinguishers.

Table 7 summarizes the experimental results for up to 5 steps. The numerical values in the table shows the ratio such that one of the equations $x_0 = x_1$, $x_2 = x_3$, $x_4 = x_5$, $x_6 = x_7$ holds. In other words, the values means the ratio of the distinguishing attack being successful. We calculated each higher order difference for 100 times by generating the initial states randomly. Besides, the values in the parentheses shows the ratio for the weakened variant of SubCrumb mentioned above.

3.4.4 Higher Order Differential Attack on *Luffa*v2

Table 6 indicates that the distinguishing attack on 7 step functions is still possible if only the Sbox is replaced. However, *Luffa* v2 always applies a blank round in the finalization so that it has sufficient security margin. In addition, Table 7 indicates that the shuffling of the order of input words to SubCrumb make the attack more difficult. As a consequence, we believe that the higher order differential attack does not threaten the security of *Luffa* v2.

4 Security Analysis of Chaining

In this section, the underlying permutations are assumed to be random and the security of the chaining of *Luffa* is discussed. We found a generic attack to find an inner collision which queries to the permutation Q_j only $2^{\frac{w-1}{w+1}n_b}$ times. Therefore we cannot claim that *Luffa* has sufficient security in terms of the current stream of security proof concerning only the number of queries to a random function. On the other hand, all of the attacks we considered require not less than $2^{\frac{w-1}{2}n_b}$ calculations. We believe that none of the attacks threatens the practical security of *Luffa*.

At the beginning of this section, the basic properties of the message injection functions are given.

After that, three generic attacks to find an inner collision are presented. Finding an inner collision of a sponge function yields a second preimage and

Table 7: Experimental results

	Number of steps				
Order	1	2	3	4	5
1	1.00 (1.00)	.05 (.10)	.00 (.00)	.00 (.00)	.00 (.00)
2	1.00 (1.00)	1.00 (1.00)	.00 (.02)	.00 (.00)	.00 (.00)
3	1.00 (1.00)	1.00 (1.00)	.05 (.28)	.00 (.00)	.00 (.00)
4	1.00 (1.00)	1.00 (1.00)	.23 (.56)	.00 (.00)	.00 (.00)
5	1.00 (1.00)	1.00 (1.00)	.76 (.96)	.00 (.00)	.00 (.00)
6	1.00 (1.00)	1.00 (1.00)	1.00 (1.00)	.00 (.00)	.00 (.00)
7	1.00 (1.00)	1.00 (1.00)	1.00 (1.00)	.00 (.01)	.00 (.00)
8	1.00 (1.00)	1.00 (1.00)	1.00 (1.00)	.00 (.01)	.00 (.00)
9	1.00 (1.00)	1.00 (1.00)	1.00 (1.00)	.00 (.04)	.00 (.00)
10	1.00 (1.00)	1.00 (1.00)	1.00 (1.00)	.00 (.46)	.00 (.00)
11	1.00 (1.00)	1.00 (1.00)	1.00 (1.00)	.00 (.78)	.00 (.00)
12	1.00 (1.00)	1.00 (1.00)	1.00 (1.00)	.00 (.91)	.00 (.00)
13	1.00 (1.00)	1.00 (1.00)	1.00 (1.00)	.01 (1.00)	.00 (.00)
14	1.00 (1.00)	1.00 (1.00)	1.00 (1.00)	.01 (1.00)	.00 (.00)
15	1.00 (1.00)	1.00 (1.00)	1.00 (1.00)	.09 (1.00)	.00 (.00)
16	1.00 (1.00)	1.00 (1.00)	1.00 (1.00)	.18 (1.00)	.00 (.00)
17	1.00 (1.00)	1.00 (1.00)	1.00 (1.00)	.34 (1.00)	.00 (.00)
18	1.00 (1.00)	1.00 (1.00)	1.00 (1.00)	.53 (1.00)	.00 (.00)
19	1.00 (1.00)	1.00 (1.00)	1.00 (1.00)	.73 (1.00)	.00 (.00)
20	1.00 (1.00)	1.00 (1.00)	1.00 (1.00)	.86 (1.00)	.00 (.00)
21	1.00 (1.00)	1.00 (1.00)	1.00 (1.00)	.97 (1.00)	.00 (.00)
22	1.00 (1.00)	1.00 (1.00)	1.00 (1.00)	.99 (1.00)	.00 (.00)
23	1.00 (1.00)	1.00 (1.00)	1.00 (1.00)	1.00 (1.00)	.00 (.00)
24	1.00 (1.00)	1.00 (1.00)	1.00 (1.00)	1.00 (1.00)	.00 (.00)
25	1.00 (1.00)	1.00 (1.00)	1.00 (1.00)	1.00 (1.00)	.00 (.00)
26	1.00 (1.00)	1.00 (1.00)	1.00 (1.00)	1.00 (1.00)	.00 (.00)
27	1.00 (1.00)	1.00 (1.00)	1.00 (1.00)	1.00 (1.00)	.00 (.01)
28	1.00 (1.00)	1.00 (1.00)	1.00 (1.00)	1.00 (1.00)	.00 (.00)
29	1.00 (1.00)	1.00 (1.00)	1.00 (1.00)	1.00 (1.00)	.00 (.03)
30	1.00 (1.00)	1.00 (1.00)	1.00 (1.00)	1.00 (1.00)	.00 (.05)
31	1.00 (1.00)	1.00 (1.00)	1.00 (1.00)	1.00 (1.00)	.00 (.07)
32	1.00 (1.00)	1.00 (1.00)	1.00 (1.00)	1.00 (1.00)	.00 (.16)

preimage as well as a collision, same applies to *Luffa*. Therefore, *Luffa* is not secure if there is an attack to find an inner collision whose calculation complexity is less than $2^{\frac{w-1}{2}n_b}$.

All attacks presented in this section concern an inner collision, but the conditions assumed for the message injection functions MI are different. The first attack does not require any property to the message injection function more than a surjectivity. The second attack is adjusted to *Luffa*. The third attack is applicable only to very simple message injection function, and does not threaten the security of *Luffa* itself. However, this attack illuminate the necessary condition for the message injection function so that we also describe the attack. The expected number of queries and the computational complexities required for the attacks are summarized in Table 8.

Table 8: The complexity of generic attacks

MI	Num. of queries (exponent of 2^a)	Calc. complexities (exponent of 2^a)	Section
Any	$\frac{w-1}{w}n_b$	$\frac{w-1}{2}n_b$	4.2
Luffa	$\frac{w-1}{w+1}n_b$	$\geq \frac{w-1}{2}n_b$	4.3
XORings	$\frac{n_b}{2}$	$\frac{n_b}{2}$	4.4

In the last of this section, the security of the finalization process is discussed.

4.1 Basic Properties of The Message Injection Functions

The message injection functions of *Luffa* are defined over a direct product ring $\mathrm{GF}(2^8)^{32}$. They have branch numbers 4, 5, 6 for $w = 3, 4, 5$ respectively. On the other hand, they have a kind of "non-diffusing" property as follows. Let a be any 32-bit data and n be a non-negative integer less than 32. A 32-bit data concatenating $32 - n$ bits consecutive zeros and the least significant n bits of a is denoted by $0||a[n]$. For a 256-bit data $X = (x_0, \ldots, x_7)$, $0||X[n]$ is defined by $(0||x_0[n], \ldots, 0||x_7[n])$. Then the message injection function MI satisfies $LH_j(0||H^{(i-1)}[n]) \oplus LM_j(0||M^{(i)}[n]) = 0||X[n]$ for all j and n, where X is an n_b bit data. This property is used in the attack described in Section 4.3.

4.2 First Naive Attack

The first attack does not require any special property to the message injection function. The attack requires about $2^{\frac{w-1}{w}n_b}$ queries to each permutation Q_j, and also requires to estimate $2^{\frac{w-1}{2}n_b}$ intermediate states. Nevertheless the number of required queries is small, the latter calculations seem dominant so that we believe that this attack does not threaten the practical security of *Luffa*.

4.2.1 Long Message Attack to Find An Inner Collision

Let us denote the transformation by the message injection function MI by

$$X_j = LH_j(H^{(i-1)}) \oplus LM_j(M^{(i)}), \quad 0 \le j < w.$$

If LM_j are surjective, then there are subspaces V_j over GF(2) of dimension $\lceil \frac{w-1}{w} n_b \rceil$ such that for any state $H^{(i-1)}$ there is a message $M^{(i)}$ such that $X_j \in V_j$ for all j.

At the pre-computation phase, the attacker queries elements of V_j to Q_j, then the queries and the corresponding answers are stored. After that, he chooses an adequate message block $M^{(1)}$ such that $LH_j(H^{(0)}) \oplus LM_j(M^{(1)}) \in V_j$ for $0 \le j < w$, then he accesses to the storage in order to get the next state $H^{(1)}$. Iterations of this process generates amount of intermediate states without an extra query to Q_j. An inner collision will be found if the number of intermediate states becomes more than $(2^{\lceil \frac{w-1}{w} n_b \rceil})^{\frac{w}{2}} \approx 2^{\frac{w-1}{2}n_b}$ because all the outputs of the message injection function MI are included in $\prod_{j=0}^{w-1} V_j$. In terms of the number of queries, the attack requires $2^{\lceil \frac{w-1}{w} n_b \rceil}$ queries to each permutation Q_j.

4.2.2 How to Reduce The Message Length

The attack in Section 4.2.1 requires a message of about $2^{\frac{w-1}{2}n_b}$ block length. However, an attack applicable to shorter messages is more attractive in practice and the message length is upper bounded by 2^{64} bits for *Luffa*-224 and

-256, and is upper bounded by 2^{128} bits for *Luffa*-384 and -512 respectively. Here we show how to reduce the message length drastically with a negligibly small additional cost.

Assume that the attacker allows V_0 to be 1 dimension larger. Then it is possible to find two messages such that $LH_j(H^{(i-1)}) \oplus HM_j(M^{(i)}) \in V_j$ for $0 \le j < w$. This approach allows the attacker to construct a binary tree of intermediate states instead of a long sequence. If the depth of the tree becomes larger than $\frac{w-1}{2}n_b$, i.e., the length of any chain in the tree is not less than $\frac{w-1}{2}n_b$, then it includes $2^{\frac{w-1}{2}n_b}$ states.

Note that this idea is applicable without an additional cost in the case of *Luffa*-256 and *Luffa*-512, because $n_b = 256$ is not divisible by $w = 3$ nor 5.

4.2.3 Complexity of The Naive Attack

The presented attack requires a huge memory of size $w \cdot 2^{\frac{w-1}{w}n_b}$ and accesses to the memory $w \cdot 2^{\frac{w-1}{2}n_b}$ times. If a memory access is much faster than a query to the permutation Q_j, the attack can violate the security claimed for *Luffa*. However, a memory access is very slow in general and a faster memory access such as a cache memory is very expensive. In addition, the required time to reach an objective memory address increases as the size of the memory gets larger. On the other hand, only 500 instructions are necessary to calculate the output of the permutation Q_j. We believe an access to such a huge memory is not less costly than a direct calculation of the permutation.

4.3 Meet-In-The-Middle Attack on Luffa

Next, a meet-in-the-middle (MIM) attack using a kind of non-diffusion property of the message injection function of *Luffa* is presented. The attack requires about $2^{\frac{w-1}{w+1}n_b}$ queries to each permutation Q_j, and also requires to apply the MIM $(2^{\frac{w-1}{w+1}n_b})^{\frac{w-1}{2}}$ times. The total calculation complexity of the second attack is considered not less than $2^{\frac{w-1}{2}n_b}$.

4.3.1 Attack Description

Here we use the notations defined in Section 4.1. In addition, the concatenation of any $32 - n$ bits and the least significant n bits of a is denoted by

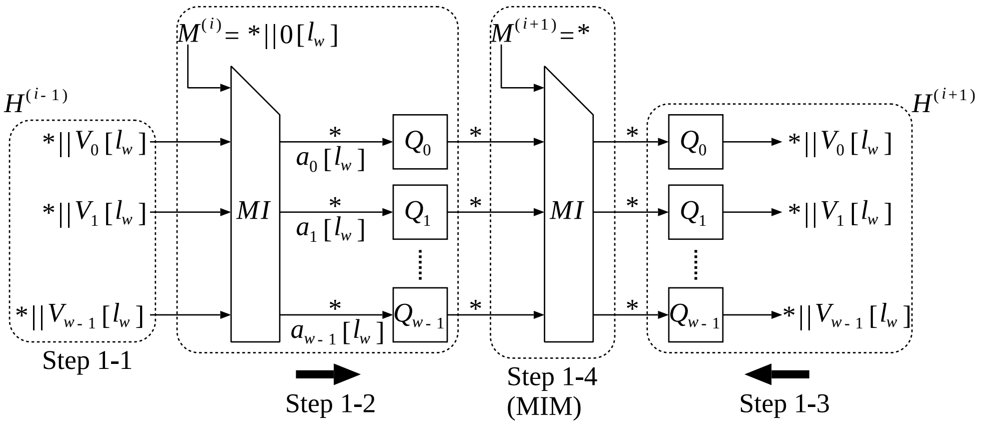

Figure 1: The first step of the meet-in-the-middle attack

$*||a[n]$ and $*||X[n]$ is also defined in the similar manner.

Let l_w be an integer $n_b - \lceil \frac{w-1}{w+1} n_b \rceil$. Then the procedure of the attack is given in Algorithm 2 and Figure 1 shows the step 1 of the attack. It describes a long message attack, but a modification for the shorter message is possible in the similar manner to the first attack.

4.3.2 Complexity of The MIM Attack

Let q be the maximum number of queries to Q_j in Step 1-2 and 1-3. A inner collision is found if $q \cdot q^w \cdot 2^{n_b} \geq (2^{n_b})^{\frac{w}{2}}$, so that $q \geq 2^{\frac{w-1}{w+1} n_b}$ is necessary.

Step 2 is almost same as the long message attack presented in Section 4.2. Only the difference is to use the states generated in Step 2-3. The states are of the form $(V_0, *||X_1[l_w], \ldots, *||X_{w-1}[l_w])$, where $X_j[l_w]$ are constants, so that $(2^{\lceil \frac{w-1}{w+1} n_b \rceil})^{\frac{w-1}{2}}$ states are necessary to find an inner collision and it is still smaller than $2^{\frac{w-1}{2} n_b}$.

On the other hand, the total complexity of the attack becomes larger because the calculation complexity of Step 1 is not negligible any more. Let $\mathcal{H}$ be the set consisting of inputs to MI in Step 1, $\mathcal{X}_j$ be the set of the j-th output blocks. The calculation complexity to find a collision such that $LH_j(H^{(i-1)}) \oplus LM_j(M^{(i)}) = X_j$ for $0 \leq j < w$ is not less than the maximum size of the sets $\mathcal{H}$ and $\mathcal{X}_j$ because $\mathcal{H}$, $\mathcal{X}_j$ are random sets. Therefore the

Algorithm 2 Meet-in-the-middle attack for *Luffa*

```
  **Step 1: MIM to find a connection**
  **1-1 (Upper side):** Fix a state $H^{(i-1)} = (*||V_0[l_w], \dots, *||V_{w-1}[l_w])$.
  **1-2 (Upper side):** Move $M^{(i)} = *||0[l_w]$ and query $M^{(i)}$ to each $Q_j$ to generate inputs of $MI$ at $i+1$-th round.
  **1-3 (Bottom):** For $0 \le j < w$, move $*||V_j[l_w]$ and query to $Q_j^{-1}$ to generate outputs of $MI$ at $i+1$-th round.
  **1-4 (MIM):** Move $M^{(i+1)}$ and find an inner collision such that $Q_j(LH_j(H^{(i-1)}) \oplus LM_j(M^{(i)})) = Y_j$ for all $j$.
  **Step 2: Long message attack**
  **2-1:** Set $i = 1$.
  **2-2:** Set $H^{(0)} = (V_0, \dots, V_{w-1})$.
  **repeat**
    **2-3:** Apply the MIM in Step 1 at round $i$. Let $H^{(i+1)} = (Y_0, \dots, Y_{w-1})$ be the resultant state.
    **2-3:** Choose $\tilde{M}^{(i+2)} = *||0[l_w]$ such that $LH_0(H^{(i+1)}) \oplus LM_0(\tilde{M}^{(i+2)}) = V_0$. Note that this state $S^{(i+1)} = MI(H^{(i+1)}, \tilde{M}^{(i+2)})$ is a "sprig" in the chain.
    **2-4:** $i = i + 2$.
  **until** An inner collision happens in $\{S^{(i+1)}\}_i$
```

complexity of Step 1 is at least $2^{\frac{w-1}{w+1}n_b}$.

The following algorithm shows a natural approach to find an inner collision in Step 1 and it requires $2^{\frac{2(w-1)}{w+1}n_b}$ calculations.

Algorithm 3 How to find a inner collision in Step 1-4

```
  **1-4-1:** Choose any $H^{(i-1)} \in \mathcal{H}$ and $X_0 \in \mathcal{X}_0$.
  **1-4-2:** Choose a message $M^{(i)} \in \mathcal{M}$ such that $X_0 = LH_0(H^{(i-1)}) \oplus LM_0(M^{(i)})$.
  **1-4-3:** Check if $LH_j(H^{(i-1)}) \oplus LM_j(M^{(i)}) \in \mathcal{X}_j$ for $1 \le j < w$. If yes, output $H^{(i-1)}, M^{(i)}$. Otherwise go back to Step 1-4-1.
```

The above discussion concludes that the total complexity of the second attack is not less than $2^{\frac{w-1}{w+1}n_b} \cdot (2^{\lceil \frac{w-1}{w+1}n_b \rceil})^{\frac{w-1}{2}} \approx 2^{\frac{w-1}{2}n_b}$. Therefore this attack does not threaten the practical security of *Luffa*.

4.4 An Application of The Multicollision Attack on A Weakened MI

In this section, an generic attack more effective than that described in Section 4.2 and 4.3 is presented. The attack is applicable only to the chaining with weak message injection function MI and it is not applicable to *Luffa*. But this attack indicates the necessity of a strong mixing function for MI. The message injection function MI of the variant is defined by

$$X_j = H_j^{(i-1)} \oplus M^{(i)}, \quad 0 \leq j < w.$$

A remarkable point of this variant is that each line is totally independent from others until the finalization is applied. The basic idea of the attack is finding a collision for each block independently.

4.4.1 Attack Description for $w = 3$

In order to simplify the explanation, the width w is fixed to three. The applications in the case of the larger width are considered later.

The first step of the attack is finding a partial collision of first two blocks, consisting of two round inputs $M^{(i)}, M^{(i+1)}$, where the second message block $M^{(i+1)}$ is chosen to cancel the difference of $M^{(i)}$. After that a birthday attack by randomly chosen $M^{(i)}$ is applied to the second block. The calculation complexity of this step is approximately given by that of the birthday attack at the second block, so that it is about $2^{\frac{n_b}{2}}$ queries to the permutation Q_j.

The next step is constructing a partial collision chain of length $\frac{n_b}{2}$, i.e., the chain consists of n_b rounds. Then the attacker gets $2^{\frac{n_b}{2}}$ states whose first 2 blocks are all equal so that there will be a collision at the third block. This is an application of the multi-collision technique proposed by Joux [19].

4.4.2 Extension to $w > 3$

The vulnerability is caused by the fact that the intermediate values of each block can be independently calculated. This property holds even for $w > 3$, so that recursive application of the multi-collision technique is possible. For example, the multi-collision chain consists of $(\frac{n_b}{2})^2$ partial collisions allows to find a collision for $w = 4$. Both the number of queries to the permutation Q_j

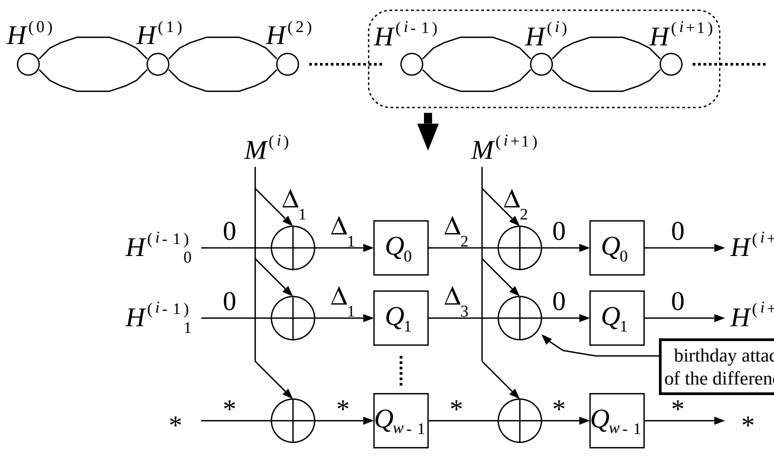

Figure 2: Partial collision chain for the weakened message injection

and the computational complexity are $n_b^w \cdot 2^{\frac{n_b}{2}}$ for any w. The complexity of the attack increases if w becomes large, but very slowly.

4.5 Collision, Second Preimage, And Preimage

Bertoni *et al.* proved that the best attack on a sponge function is to find an inner collision [8]. And to find a collision of outputs, a second preimage, and a preimage all belong to the inner collision. We believe that it is also the case for *Luffa* and we have not found any serious attack to find an inner collision even though *Luffa* has no security proof so far. Therefore we think *Luffa* has the sufficient collision resistance, second preimage resistance, and preimage resistance.

4.6 Security Analysis of Finalization

4.6.1 Saturation Property of *Luffa*v2

A saturation attack (or Square attack) was originally proposed by Daemen *et al.* as the dedicated attack on a block cipher SQUARE [11]. The basic idea of the attack is similar to truncated differential attack, but it uses another property preserved by a permutation mapping. Assume that the attacker takes all possible inputs and gets the outputs of the permutation. Then the sum of outputs becomes zero.

The message injection function MI and the consecutive permutations Q_j preserves this property. The output function OF also preserves the property. If there is no blank round and the attacker can take all values of the last message block, the sum of the outputs becomes zero. This is a kind of distinguisher and requires 2^{n_b} calculations. However, a blank round is always applied in *Luffa* v2 so that the saturation property does not hold after the blank round.

4.6.2 Slide Attack

A slide attack was originally proposed by Biryukov and Wagner [10] for block ciphers whose key scheduling is simple. Gorski *et al.* pointed out a slide attack is applicable under the keyed hashing context [16] and showed that it can break some sponge-like hash functions. Their attack is applicable if the round function is very thin and the finalization is not well considered. In the case of *Luffa*, the round function is much stronger than that of the broken sponge variants. In addition, the message padding and the fixed message inputs to a blank round are defined to avoid the sliding property. Therefore we believe that a kind of slide attacks does not threaten the security of *Luffa*.

5 On The (Semi-)Free-start Setting

In this section, (semi-)free-start setting on sponge variants is discussed. A free-start setting assumes that the attacker can choose the IVs which are usually fixed and a semi-free-start setting assumes that the attacker can choose only an IV in his attack. These settings relax the constraints for the attacker so that they are considered a useful step to study the security of a hash function. Especially, (semi-)free-start attacks on the underlying primitive are taken seriously if the target hash function is a compression function based design.

In this case, the security of the chaining usually depends on that of the underlying compression function. And the free-start setting is natural in terms of the security of a compression function because the constraints to control the chaining value are derived from the usage of the compression function.

In Round 1 of the SHA-3 competition, several (semi-)free start attack on the compression functions were proposed. In addition, some reports discussed the security of sponge based hash function under (semi-)free-start setting. However, a sponge function and its variants are permutation-based designs, so that they are not secure even if the underlying primitives are assumed ideal. For example, a collision and a preimage can be easily found with negligible calculation complexity for the ideal model of a sponge function.

From this natural property of the sponge function, we think that it is not important to discuss the security of a sponge function and its variants under (semi-)free-start setting. And it might be better to note that the IV of sponge variants should not be variable in their applications.

In the case of *Luffa*, Jia proposed a free-start (second) preimage attacks [18]. As claimed above, we consider his attacks do not threaten the practical security of *Luffa* under the normal setting.

And it might be better to note that *Luffa* is not secure under semi-free-start setting. In this case, the attacker can choose the inputs to the permutations Q_j independently so that the generic semi-free-start collision attack on *Luffa* is trivially reduced to search for a multi-collision of n_b-bit data. Therefore the computational complexity of the attack is upper-bounded by $2^{\frac{w-1}{w}n_b}$ and it is smaller than the birthday bound in the case of $w = 5$. Namely, the computational complexity in this case is $2^{(5-1)/5256} = 2^{204.8} < 2^{512/2}$.

In addition, the discussion in 3.2 assumes a normal setting, i.e., the IV is fixed. Therefore the calculation complexities of the differential based collision attacks and a kind of rebound attacks under semi-free-start setting could be smaller than what we expect. However, we believe these semi-free-start attacks does not threaten the security of *Luffa* under the normal setting, which are required by NIST.

6 Implementation Aspects

6.1 Performance Figures for Software Implementations

Here, we show the software performances of the *Luffa* v2 hash family. The hash family was implemented on several processors; the Intel Core2 [17], the Atmel AVR [1], Renesas H8 [25] and ARM ARM9 [5] processors. We have evaluated the speed and memory usage of the hash family on the processors.

As for speed performance, we have measured two types of figures; execution time to hash a one-block message and execution time to hash a very long message. For the former, "one-block" means that the padded message consists only of a 256-bit block. That is, the length of the original message is no more than 255 bits. For the latter, the length of the message is set to be so large that a contribution of the finalization function to a throughput speed is negligible. More precisely, the figure is the execution time of an invocation of the round function divided by 32.

6.1.1 8-bit Processors

Luffa has been implemented for the Atmel AVR processor and the Renesas H8 processor in assembly languages.

Atmel AVR Processor Our target processor model is ATmega8515. The processor has 8192 bytes of flash memory and 512 bytes of SRAM. We used Atmel's AVR Studio as a development environment and measured execution time on the AVR Simulator of ATmega8515 bundled with the AVR Studio.

The execution time and memory requirements of an assembly code are shown in Table 9. In the table, the second column lists the execution time to hash a one-block message. The third column lists the execution time to hash a very long message. The fourth and fifth columns are for the sizes of the implementation. There is no setup of the algorithm in *Luffa*, hence the setup time is zero.

Renesas H8 Processor Our target processor model is H8 38024F (H8/300L core). The processor has 32 kbytes of flash memory and 1 kbytes of SRAM.

Table 9: Execution time and memory requirements on AVR ATmega8515

Bit length of hash value	Execution time		Memory requirements	
	One-block msg. (cycles/message)	Very long msg. (cycles/byte)	Code size + constant data (bytes)	RAM (bytes)
224	46,243	732.1	688+120	134
256	46,243	732.1	688+120	134
384	98,285	1,055.4	774+160	166
512	133,673	1,427.0	840+200	198

Table 10: Execution time and memory requirements on Renesas H8/300L

Bit length of hash value	Execution time		Memory requirements	
	One-block msg. (cycles/message)	Very long msg. (cycles/byte)	Code size + constant data (bytes)	RAM (bytes)
224	101,990	1,624.8	856+120	144
256	101,990	1,624.8	856+160	144
384	215,618	2,296.8	976+160	176
512	284,930	3,028.8	1,112+200	208

We used Renesas's High-performance Embedded Workshop (HEW) as a development environment and measured execution time on the Simulator of H8/300L bundled with HEW.

The execution time and memory requirements of an assembly code are shown in Table 10. In the table, the second column lists the execution time to hash a one-block message. The third column lists the execution time to hash a very long message. The fourth and fifth columns are for the sizes of the implementation. There is no setup of the algorithm in *Luffa*, hence the setup time is zero.

6.1.2 32-bit Processors

Here, we will show some performance figures of *Luffa* for 32-bit processors. The hash family has been implemented for the Intel Core2 Duo processor in C and assembly languages and for the ARM ARM9 processor in C language.

Intel Core2 Duo Processor Our target processor model is the Intel Core2 Duo E6600 2.4GHz processor in 32-bit mode, which will be used by NIST. We have measured speed performances of three different types of codes, a C code obeying the ANSI C grammar, a C code using Visual C++ SSE intrinsics and an assembly code. Both of the C codes obey the NIST API.

Table 11: 32-bit platforms used for measurement

Programming language	Processor	Memory	OS	Compiler or assembler
C	Core2 Duo E6600 (2.4GHz)	2GBytes	Windows Vista 32-bit Edition	Visual Studio 2005 C++
Assembly	Core2 Duo E6600 (2.4GHz)	2GBytes	Ubuntu Linux 8.04 32-bit distribution	gnu as

In accordance with the test environment of NIST, we have used the Microsoft's Visual Studio 2005 C++ compiler and Windows Vista Ultimate 32-bit operating system for the measurement of the C codes. The assembly code was measured on a 32-bit Linux distribution. These environments are shown on Table 11.

We measured two types of figures, the execution time to hash a very long message and the execution time to hash a one-block message. The figures are shown at Table 12 and 13. In Table 12, a throughput speed is also listed. Note that the results of the C codes include overheads coming from the NIST API, but that of the assembly code does not. Also note that there is no setup of the algorithm in *Luffa*, hence the setup time is zero.

Since the fast implementation of *Luffa* does not use look-up tables, only small size of memory is required to implement *Luffa*. Therefore, a relation of time-memory trade-off in 32-bit implementation is relatively weak.

ARM ARM9 Processor Our target processor model is ARM ARM-926EJ-S. We used ARM RealView Development Suite as a development environment and measured execution time on the Simulator of ARM926EJ-S. The model ARM926EJ-S has an instruction cache and a data cache. The size of each cache can be from 4 kbytes to 128 kbytes. In our measurement, the simulator is set to have 4 kbytes for each.

Table 12: Throughput speed and execution time to hash a very long message on Core2 Duo in 32-bit mode

Bit length of hash value	ANSI C		C using SSE intrinsics		Assembly	
	Exec. time (cycles/byte)	Speed (Mbps)	Exec. time (cycles/byte)	Speed (Mbps)	Exec. time (cycles/byte)	Speed (Mbps)
224	31.1	617.2	19.8	970.7	13.8	1,391.3
256	31.2	614.6	19.8	970.7	13.8	1,391.3
384	46.7	410.8	22.3	862.2	15.5	1,238.7
512	64.9	295.9	36.0	533.3	26.8	716.4

Table 13: Execution time to hash a one-block message on Core2 Duo in 32-bit mode

Bit length of hash value	ANSI C	C using SSE intrinsics	Assembly
	Execution time (cycles/message)	Execution time (cycles/message)	Execution time (cycles/message)
224	2,162	1,509	886
256	2,157	1,513	886
384	4,622	2,439	1,497
512	6,457	3,799	2,573

The execution time of an ANSI C code is shown in Table 14. This code is slightly different from the ANSI C code used on the Intel Core2 Duo processor. In the table, the second column lists the execution time to hash a one-block message. The third column lists the execution time to hash a very long message. Note that the result of the C code includes overheads coming from the NIST API. Also note that there is no setup of the algorithm in *Luffa*, hence the setup time is zero.

6.1.3 64-bit Processor

Here, we show some performance figures of *Luffa* for a 64-bit processor. The figures are for the Intel Core2 Duo processor, which will be used by NIST.

Intel Core2 Duo Processor Our target processor model is the Intel Core2 Duo E6600 2.4GHz Processor in 64-bit mode. We have measured speed performances of three different types of codes, an C code obeying the

Table 14: Execution time on ARM ARM926EJ-S in C language

Bit length of hash value	Execution time	
	One-block msg. (cycles/message)	Very long msg. (cycles/byte)
224	7,054	91.1
256	7,111	91.1
384	14,209	129.5
512	18,564	169.7

Table 15: 64-bit platforms used for measurement

Programming language	Processor	Memory	OS	Compiler or assembler
C	Core2 Duo E6600 (2.4GHz)	2GBytes	Windows Vista 64-bit Edition	Visual Studio 2005 C++
Assembly	Core2 Duo E6600 (2.4GHz)	2GBytes	Ubuntu Linux 8.04 64-bit distribution	gnu as

ANSI C grammar, a C code using Visual C++ SSE intrinsics and an assembly code. Both of the C codes obey the NIST API.

In accordance with the test environment of NIST, we have used the Microsoft's Visual Studio 2005 C++ compiler and Windows Vista Ultimate 64-bit operating system for the measurement of the C codes. The assembly code was measured on a 64-bit Linux distribution. These environments are listed on Table 15.

We measured two types of figures, the execution time to hash a very long message and the execution time to hash a one-block message. The figures are shown at Table 16 and 17. In Table 16, a throughput speed is also listed. Note that the results of the C codes include overheads coming from the NIST API, but that of the assembly code does not. Also note that there is no setup of the algorithm in *Luffa*, hence the setup time is zero.

Since the fast implementation of *Luffa* does not use look-up tables, only small size of memory is required to implement *Luffa*. Therefore, a relation of time-memory trade-off in 64-bit implementation is relatively weak.

Table 16: Throughput speed and execution time to hash a very long message on Core2 Duo in 64-bit mode

Bit length of hash value	ANSI C		C using SSE intrinsics		Assembly	
	Exec. time (cycles/byte)	Speed (Mbps)	Exec. time (cycles/byte)	Speed (Mbps)	Exec. time (cycles/byte)	Speed (Mbps)
224	26.0	738.2	16.3	1,175.8	13.3	1,443.6
256	26.2	731.7	16.3	1,175.8	13.3	1,443.6
384	40.2	478.2	18.5	1,036.2	15.0	1,280.0
512	55.6	345.4	31.7	604.9	23.8	806.7

Table 17: Execution time to hash a one-block message on Core2 Duo in 64-bit mode

Bit length of hash value	ANSI C	C using SSE intrinsics	Assembly
	Execution time (cycles/message)	Execution time (cycles/message)	Execution time (cycles/message)
224	1,815	1,225	854
256	1,830	1,236	854
384	3,990	1,973	1,446
512	5,600	3,265	2,289

6.2 Performance Figures for Hardware Implementations

Here, we show the software performances of the *Luffa* v2 hash family. A hardware performance evaluation of the *Luffa* hash family was done by synthesizing the proposed designs using UMC 0.13 μm CMOS standard cell library. The code was first written in GEZEL [15] and tested for the functionality using the test vectors provided by the software implementations. The GEZEL code was then compiled to VHDL and synthesized using Synopsys Design Compiler version Y-2006.06 [26]. Our goal was to show that the family of cryptographic hash functions *Luffa* can be implemented efficiently in hardware. We targeted both, the compact and the high-throughput implementations.

Table 18: Throughput optimized implementations of the *Luffa* hash family.

Design	Area [GE]	Frequency [MHz]	# of cycles per round	Throughput [Mbps]	
				OB	LM
Luffa-224/256	30,834	1,124	9	15,980.0	31,960.0
Luffa-384	50,068	813	9	11,563.0	23,126.0
Luffa-512	65,102	690	9	9,808.5	19,617.0

6.2.1 Throughput-Optimized Implementations

The synthesis results for the throughput optimized version of *Luffa* are given in Table 18. The high-throughput designs are synthesized regardless of the gate count and show that the *Luffa* hash algorithm achieves the throughput of more than 31 Gbps. The throughput for "one-block" message was calculated according to the following equation:

$$\text{Throughput}_{\text{OB}} = \frac{\text{Frequency}}{\#\text{ of Cycles} \times 2} \times 256 \text{ bit} \ ,$$

while the throughput for the very long message was calculated as:

$$\text{Throughput}_{\text{LM}} = \frac{\text{Frequency}}{\#\text{ of Cycles}} \times 256 \text{ bit} \ .$$

For the high-throughput implementations, the goal was to minimize the critical path. We used w permutation blocks in parallel and each of them contained 64 Sboxes and 4 `MixWord` blocks. The straightforward implementation resulted in the critical path of 1.21 ns (*Luffa*-224/256) and the cycle count of 8. The critical path was placed from the input of the message injection function to the output of the permutation block. As the message injection function is performed only once at the beginning of every round, we moved the state registers at the input of the permutation blocks. This resulted in the faster design, shortening the critical path to only 0.89 ns (*Luffa*-224/256). One more clock cycle had to be spent in order to perform the complete round, but the final throughput got increased for about 20 %.

To show the possible trade-offs regarding the throughput-optimized implementations we provide Figure 3. More details about the exact values are given in appendix (Table 21).

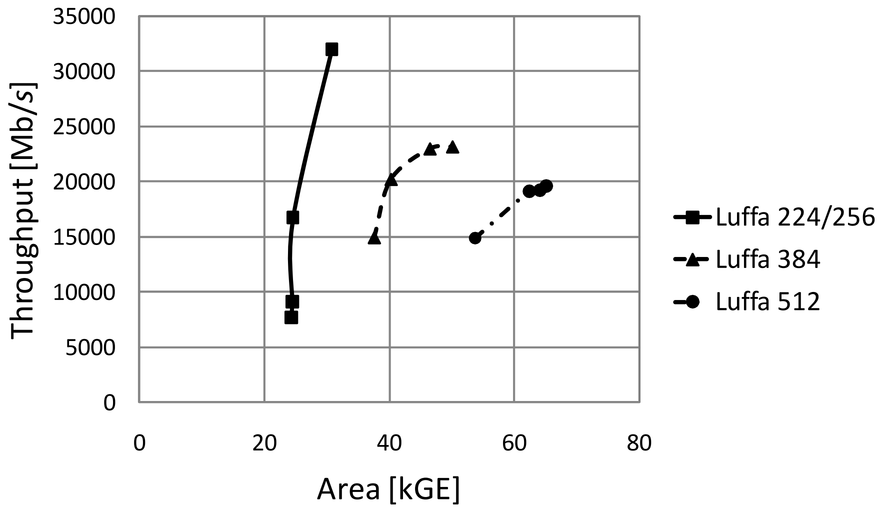

Figure 3: Throughput-optimized implementations.

Table 19: Area optimized implementations of the *Luffa* hash family.

Design	Area [GE]	Frequency [MHz]	# of cycles per round	Throughput [Mbps] OB	Throughput [Mbps] LM
Luffa-224/256	19,646	344	891	49.3	98.7
Luffa-384	29,466	344	1188	36.9	73.8
Luffa-512	39,803	344	1485	17.6	35.2

6.2.2 Area-Optimized Implementations

The compact implementations were made using only one non-linear permutation block. Inside the permutation we used a single Sbox and a single `MixWord` block. This approach resulted in a large number of cycles, while on the other hand it efficiently reduced the final gate count. We used w 256-bit registers to maintain the internal state and to keep the results for the blank round. Eight more 32-bit registers were used inside the permutation block. The results are given in Table 19.

As can be seen from Table 19, the most compact implementation is obtained for *Luffa*-224/256 algorithm and consumes approximately 19.6 kGE.

Note that our only goal for the compact implementation was to have a small circuit size, regardless of the achieved speed. Hence, we fixed the frequency to 344 MHz and synthesized our designs.

6.2.3 Pipelined Implementation

When hashing independent message blocks, one can benefit from using the pipelined architecture. Multiple non-linear permutation blocks need to be added ($8w$ blocks) as well as w pipelined 256-bit register for each round ($8w$ in total). This approach can effectively increase the throughput for more than 8 times at the cost of additional area overhead. As can be seen from Table 20, a throughput of 115.6 Gbps[2] is achieved for the *Luffa*-224/256 version at the cost of 156.6 kGE.

Table 20: Pipelined implementations of the *Luffa* hash family.

Design	Area [GE]	Frequency [MHz]	# of cycles per round	Throughput* [Mbps]	
				OB	LM
Luffa-224/256	156,613	508	9	57,799.0	115,598.0
Luffa-384	217,936	483	9	54,954.5	109,909.0
Luffa-512	272,413	478	9	54,385.5	108,771.0

* Throughput for independent message blocks.

6.2.4 Summary of Hardware Implementations

The hardware implementations of the *Luffa* family of hash functions have been evaluated in this report. We conclude that the design is very well suited for both compact and high-throughput implementations. The most compact size of 19,646 GE was achieved for the 224/256 version of *Luffa*. The same version achieves the highest throughput of 31.96 Gbps, while the pipelined design approaches the throughput of 115.6 Gbps. Due to the ample of parallelism provided by *Luffa* hash family, it is possible to make a plenty of trade-offs and choose the most appropriate design for the target application.

[2]This is true only for hashing independent message blocks in parallel.

Table 21: Hardware performance of the *Luffa* hash family.

Design	Area	Frequency	# of cycles	Throughput [Mbps]	
	[GE]	[MHz]	per round	OB	LM
Luffa-224/256	30,834	1,124	9	15,980.0	**31,960.0**
Luffa-224/256	24,586	588	9	8,366.0	16,732.0
Luffa-224/256	24,450	320	9	4,558.5	9,117.0
Luffa-224/256	24,303	270	9	3,844.0	7,688.0
Luffa-224/256	**19,646**	344	891	49.3	98.7
Luffa-384	50,068	813	9	11,563.0	**23,126.0**
Luffa-384	46,476	806	9	11,469.5	22,939.0
Luffa-384	40,279	709	9	10,086.5	20,173.0
Luffa-384	37,612	523	9	7,446.0	14,892.0
Luffa-384	**29,466**	344	1,188	36.9	73.8
Luffa-512	65,102	690	9	9,808.5	**19,617.0**
Luffa-512	64,155	676	9	9,609.5	19,219.0
Luffa-512	62,433	671	9	9,545.0	19,090.0
Luffa-512	53,734	523	9	4,772.5	14,892.0
Luffa-512	**39,803**	344	1,485	17.6	35.2

Table 21 shows the detailed implementation results of different versions of the *Luffa* hash family. The fastest and the most compact versions for each version of the *Luffa* hash family are given in bold.

6.3 Security against Side Channel Attacks

A side channel attack observes physical information leakage in addition to the input and the output of the cipher in order to recover the secret data. There is no secret for a naive hash function so that it is not necessary to consider the threat of side channel attacks. However, some applications such as the HMAC use a secret information. In such applications, side channel attacks also should be considered. Hereinafter, we discuss the abstract property of the sub-permutations of *Luffa* against side channel attacks both in software and hardware implementations.

Tsunoo *et al.* proposed a cache timing attack for a software implementation of DES [27]. The attack observes the delay of the operation caused by the difference between cache hits and cache misses. This attack can be widely applicable to ciphers in which Sboxes are implemented by reference tables and amount of papers have been published to improve the attack and the countermeasure. Besides, a bit slice permutation does not refer the cache and implements the Sboxes by a set of logical instructions. This feature of a bit slice permutation avoids any kind of attacks based on cache timing.

A differential power analysis (DPA) observes the power consumption of certain part of operations depending on the secret information [21]. It is hard to perfectly avoid the DPA, instead amount of techniques to increase the cost to apply the attack have been proposed. The most likely approach is obscuring the power consumption of each operation. We believe that the cost to obscure the operations of *Luffa* is not costly in comparison to MD-like hash functions.

Trademarks

- ARM and RealView are registered trademarks and ARM926EJ-S is the name of products of ARM Limited in the United States and/or other countries.
- Atmel, AVR, and AVR Studio are registered trademarks of Atmel Corporation in the United States and/or other countries.
- Intel is a registered trademark and Core is the name of products of Intel Corporation in the U.S. and other countries.
- Linux is a registered trademark of Linus Torvalds in the U.S. and other countries.
- Microsoft, Windows Vista, and Visual Studio are registered trademarks of Microsoft Corporation in the United States and/or other countries.
- Renesas and H8 are registered trademarks of Renesas Technology Corporation in the United States and/or other countries.
- Synopsys is a registered trademark and Design Vision is the name of products of Synopsys, Inc. in the United States and/or other countries.

- Ubuntu is a registered trademark of Canonical Ltd. in the United States and/or other countries.

References

[1] Atmel Corporation, "8-bit AVR Instruction Set," available at `http://www.atmel.com/dyn/resources/prod_documents/doc0856.pdf`.

[2] J.P. Aumasson, I. Dinur, W. Meier and A. Shamir "Cube Testers and Key Recovery Attacks On Reduced-Round MD6 and Trivium," *Fast Software Encryption, FSE 2009*, Lecture Notes in Computer Science, vol. 5665, Springer-Verlag, pp. 1–22, 2009.

[3] J.P. Aumasson and W. Meier, "Zero-sum distinguishers for reduced Keccak-f and for the core functions of Luffa and Hamsi," 2009. Available at `http://www.131002.net/data/papers/AM09.pdf`.

[4] R. Anderson, E. Biham and L. Knudsen, "Serpent: A Proposal for the Advanced Encryption Standard," available at `http://www.cl.cam.ac.uk/~rja14/serpent.html`.

[5] "Reference Manuals of the ARM architectures and processors," available at `http://infocenter.arm.com/help/index.jsp`.

[6] M. Bellare and T. Kohno, "Hash function balance and its impact on birthday attacks," *Advances in Cryptology - Eurocrypt'04*, Lecture Notes in Computer Science, Vol. 3027, Springer-Verlag, pp. 401–418, 2004.

[7] G. Bertoni, J. Daemen, M. Peeters and G. Van Assche, "Sponge Functions," Ecrypt Hash Workshop 2007.

[8] G. Bertoni, J. Daemen, M. Peeters and G. Van Assche, "On the Indifferentiability of the Sponge Construction," *Advances in Cryptology, Eurocrypt'08*, Lecture Notes in Computer Science, Vol. 4965, Springer-Verlag, pp. 181–197, 2008.

[9] E. Biham, R. Anderson and L. Knudsen, "Serpent: A New Block Cipher Proposal," *Fast Software Encryption, FSE'97*, Lecture Notes in Computer Science, Vol. 1372, Springer-Verlag, pp. 222–238, 1998.

[10] A. Biryukov and D. Wagner, "Slide Attacks," *Fast Software Encryption, FSE'99*, Lecture Notes in Computer Science, Vol. 1636, Springer-Verlag, pp. 245–259, 1999.

[11] J. Daemen, L. Knudsen, V. Rijmen, "The Block Cipher Square," *Fast Software Encryption, FSE'97*, Lecture Notes in Computer Science, Vol. 1267, Springer-Verlag, pp. 149–165, 1997.

[12] J. Daemen, M. Peeters, G. Van Assche and V. Rijmen, "Nessie Proposal: NOEKEON," available at `http://gro.noekeon.org/`.

[13] National Institute of Standards and Technology, "Secure Hash Standard," FIPS 180-2.

[14] National Institute of Standards and Technology, "Recommendation for Random Number Generation Using Deterministic Random Bit Generators (Revised)," NIST Special Publication 800-90, March 2007.

[15] GEZEL, `http://rijndael.ece.vt.edu/gezel2/index.php/Main_Page`.

[16] M. Gorski, S. Lucks and T. Peyrin, "Slide Attacks on Hash Functions," Cryptology ePrint Archive 2008/263, 2008.

[17] Intel Corporation, "Intel 64 and IA-32 Architectures Software Developer's Manual," available at `http://www.intel.com/products/processor/manuals/index.htm`.

[18] K. Jia, "Practical Pseudo-Cryptanalysis of Luffa," Cryptology ePrint Archive, Report 2009/224, 2009.

[19] A. Joux, "Multicollisions in iterated hash functions. Application to cascaded constructions," *Advances in Cryptology, CRYPTO'04*, Lecture Notes in Computer Science, Vol. 3152, Springer-Verlag, pp. 306–316, 2004.

[20] L. R. Knudsen, "Truncated and Higher Order Differentials," *Fast Software Encryption, FSE '94*, Lecture Note in Computer Science vol. 1008, pp. 196–211, Springer-Verlag, 1994.

[21] P. Kocher, J. Jaffe, and B. Jun, "Introduction to differential power analysis and related attacks," 1998. Available at `http://www.cryptography.com/dpa/technical/index.html`.

[22] X. Lai, "Higher order derivatives and differential cryptanalysis," *Proc. Symposium on Communication, Coding and Cryptography*, pp. 227–233, Kluwer Academic Publishers, 1994.

[23] J. S. Leon, "A Probabilistic Algorithm for Computing Minimum Weights of Large Error-Correcting Codes," *IEEE Trans. on Infomation Theory*, Vol. 34, No. 5, pp. 1354–1359, 1988.

[24] D. A. Osvik, "Speeding up Serpent," *The 3rd AES Conference, Proceedings*, pp. 317–329, 2000.

[25] Renesas Technology Corporation, "H8/300L Series Software Manual," available at `http://documentation.renesas.com/eng/products/mpumcu/rej09b0214_h8300l.pdf`.

[26] Synopsis, `http://www.synopsys.com/`.

[27] Y. Tsunoo, T. Saito, T. Suzaki, M. Shigeri, H. Miyauchi, "Cryptanalysis of DES Implemented on Computers with Cache," *Cryptographic Hardware and Embedded Systems, CHES'03*, Lecture Notes in Computer Science, Vol. 2779, Springer-Verlag, pp. 62–76, 2003.

[28] C. De Cannière, H. Sato, and D. Watanabe, "Hash Function *Luffa*, Specification, " NIST SHA3 Competition, 2008.

[29] C. De Cannière, H. Sato, and D. Watanabe, "The Reasons for The Change of *Luffa*, " to be supplied with the Second Round Package. Also available at `http://www.sdl.hitachi.co.jp/crypto/luffa/Reason4Mod.pdf`.

[30] D. Watanabe and Y. Hatano, "Higher Order Differential Attack on Reduced Round *Luffa*," to be supplied with the Second Round Package. Also available at `http://www.sdl.hitachi.co.jp/crypto/luffa/HigherOrderDifferentialAttackOnLuffa_v1.pdf`.