SHAMATA HASH FUNCTION ALGORITHM SPECIFICATIONS

October 30, 2008

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

SHAMATA is a register based hash function with a variable digest size. SHAMATA can hash messages of length up to $2^{64}-1$ bits. The chaining value is of length 2048 bits and is formed by the internal states of two registers. Message is incorporated into the registers as 128-bit data blocks. The registers are clocked twice after incorporation of each data block. SHAMATA also employs the MD-strengthening to pad the message to a multiple of 128 bits. Finally, the output is composed from the last internal state of one of the registers.

SHAMATA uses one of the AES primitive functions `MixColumns` to incorporate the message into the internal state and a modified version of the AES round function to mix the internal state.

2 Notation

$\oplus$	:	XOR operation
$\|\|$	:	Concatenation
$A[i]$	:	i-th most significant block (128 bits) of register A
$A(i)$	:	i-th most significant word (64 bits) of register A
$A\{i\}$	:	i-th most significant bit of register A
CV	:	Chaining Value

3 Specification

SHAMATA employs the MD-strengthening to pad the message to a multiple of 128 bits similar to the padding of SHA. A "1" bit and sufficient number of "0" bits are concatenated to the message so that the length of the padded message equals 64 modulo 128. Then the length of the original message represented as a 64-bit value is concatenated at the end of the padded message. The digest lengths are multiples of 32 bits starting from 224 bits up to 512 bits.

We always take the left-most bit as the MSB and the data given by the smallest index is the most important. Message blocks are shown as $M_1, M_2, ..., M_{NoMB}$ where the last block, M_{NoMB}, is the padding.

SHAMATA internal state consists of two registers:

- B **register:** main mixing register, consists of 4 blocks of size 128 bits each for a total of 512 bits.
- K **register:** second mixing register, consists of 12 blocks of size 128 bits each for a total of 1536 bits.

SHAMATA can be separated into 3 phases:

1. **Initialization:**

   Initial internal states of B and K registers are formed in this phase. The IV vector is formed by the digest size. Then IV is incorporated into the registers B and K and the registers are updated eight times. The resulting internal state is taken as the initial internal state.

2. **Message Loading:**

   This is the main part of SHAMATA, where message blocks are incorporated into the registers B and K. Then the B and K registers are updated using the modified AES round function, which we call ARF.

3. **Finalization:** The number of message blocks ($NoMB$) is taken as a 128-bit message block. For example, if $NoMB = 10$ then it is taken as 0x0000 0000 0000 0000 0000 0000 0000 000A as a 128-bit message block.

   The B and K registers are updated with $NoMB$ and then clocked using the modified AES round function, ARF. This procedure is repeated 32 steps. Then the B register is used to produce the message digest.

Hash Length	IV in hex
224	0x0000 0000 0000 0000 0000 0000 0000 00E0
256	0x0000 0000 0000 0000 0000 0000 0000 0100
384	0x0000 0000 0000 0000 0000 0000 0000 0180
512	0x0000 0000 0000 0000 0000 0000 0000 0200

Table 1: Examples of IV's which are the hash lengths written as 128-bit vectors.

Each phase of SHAMATA uses a function called *UpdateRegister*. The high level description of SHAMATA is given in Algorithm 1.

We define r as the number of iterations for the modified AES round function ARF during message loading. We determine that r is equal to one if the digest size is less than or equal to 256 bits and r is equal to two if the digest size is greater than 256 bits.

3.1 *UpdateRegister*(State, D, $blockno, r$)

This is the main function used throughout SHAMATA. As input it takes r to specify the clocking behavior and a 128-bit value D as the data block to be incorporated into the registers with its index *blockno* which indicates its order in the message. The high level description of *UpdateRegister* is given in Algorithm 2.

UpdateRegister function can be decomposed into 2 sub-functions:

3.1.1 *LoadDataBlock*(State, D, *blockno*)

The function *LoadDataBlock* loads an extended copy of the given data D and its index *blockno*, which is the order of D in the message, into the registers B and K.

Let $P = \texttt{MixColumns}(D)$ and $Q = \texttt{MixColumns}(D^T)$ where D^T is the transpose of the data D when D is considered as a 4×4 matrix as in AES. That is, i-th most important four-byte forms the i-th column for $i = 1, 2, 3, 4$. The `MixColumns` matrix is applied to the columns. Let $P' = P(1)||Q(0)$ and $Q' = Q(1)||P(0)$. Then

$$\begin{aligned} B[2] &= B[2] \oplus P \oplus blockno \\ B[3] &= B[3] \oplus Q \oplus blockno \\ K[3] &= K[3] \oplus P' \\ K[5] &= K[5] \oplus Q \\ K[7] &= K[7] \oplus P \\ K[11] &= K[11] \oplus Q' \end{aligned}$$

where *blockno* is a 64-bit vector but it is considered as a 128-bit vector while XOR'ing.

Example. The *blockno* is 10 while incorporating the 10-th message block (which is M_{10}). Consider *blockno* as a 128-bit vector to incorporate it into the register B. So, 10 will be written as 0x0000 0000 0000 0000 0000 0000 0000 000A.

3.1.2 *ClockRegister*(State, r)

The function *ClockRegister* is used to update the contents of B and K registers by clocking the registers twice using the modified AES round function. B and K registers are considered as shift registers with 128-bit cells and at each clocking their feedbacks are calculated as:

$$\begin{aligned} feedK &= ARF^r(B[2]) \oplus B[0] \\ feedB &= feedK \oplus K[9] \oplus K[0] \end{aligned}$$

Algorithm 1: High Level Description of SHAMATA.

Data: 128-bit blocks of the message with padding, h: digest size given as a 128 bit vector. h must be a multiple of 32, between 224 and 512.
Result: h bit message digest
begin
  Initialization Phase:
  if $h \leq 256$ **then** $r := 1$ **else** $r := 2$
  begin
    Initialize the registers: B:=`0x0`, K:=`0x0`
    $IV := h$: Consider h as a 128-bit vector IV (see Table 1)
    for $i \leftarrow 1$ **to** 8 **do**
      UpdateRegister(**State**, IV, i, r)
  end
  Message Loading Phase:
  begin
    for $i \leftarrow 1$ **to** $NoMB$*:Number of Message Blocks* **do**
      UpdateRegister(**State**, $M[i]$, i, r)
  end
  Finalization Phase:
  begin
    for $i \leftarrow 1$ **to** 32 **do**
      UpdateRegister(**State**, $NoMB$, i, r)
    ProduceOutput(**State**, h): Produce h bit output from the block register B
  end
end

Algorithm 2: High Level Description of *UpdateRegister*.

Data: 128 bit value D with its *blockno* and r
Result: Updated B and K
begin
  LoadDataBlock(**State**, D,*blockno*): Incorporate the data block with its index *blockno* into the registers B and K
  ClockRegister(**State**, r): Clock the B and K Registers twice. Run the modified AES round function r rounds for each clocking
end

Then B and K registers are shifted to the left by 128 bits and feedbacks are placed at their corresponding low 128 bits for each clocking, i.e.

$$\begin{aligned} B[i] &= B[i+1], \text{ for } i = 0, 1, 2 \\ B[3] &= feedB \\ K[i] &= K[i+1], \text{ for } i = 0, ..., 10 \\ K[11] &= feedK \end{aligned}$$

Clocking the registers B and K once is depicted in Figure 1. Clocking the registers twice is given as

$$\begin{aligned} feedK_1 &= ARF^r(B[2]) \oplus B[0] \\ feedB_1 &= feedK_1 \oplus K[9] \oplus K[0] \\ feedK_2 &= ARF^r(B[3]) \oplus B[1] \\ feedB_2 &= feedK_2 \oplus K[10] \oplus K[1] \\ B[i] &= B[i+2], \text{ for } i = 0, 1 \\ B[2] &= feedB_1 \\ B[3] &= feedB_2 \\ K[i] &= K[i+2], \text{ for } i = 0, ..., 9 \\ K[10] &= feedK_1 \\ K[11] &= feedK_2. \end{aligned}$$

ARF is the AES [21] Round Function without key addition. So, ARF consists of `SubByte, ShiftRows` and `MixColumns` operations of AES. The parameter r determines how many times the ARF function is iterated for each clocking. Hence, *ClockRegister* function calls the ARF function $2r$ times.

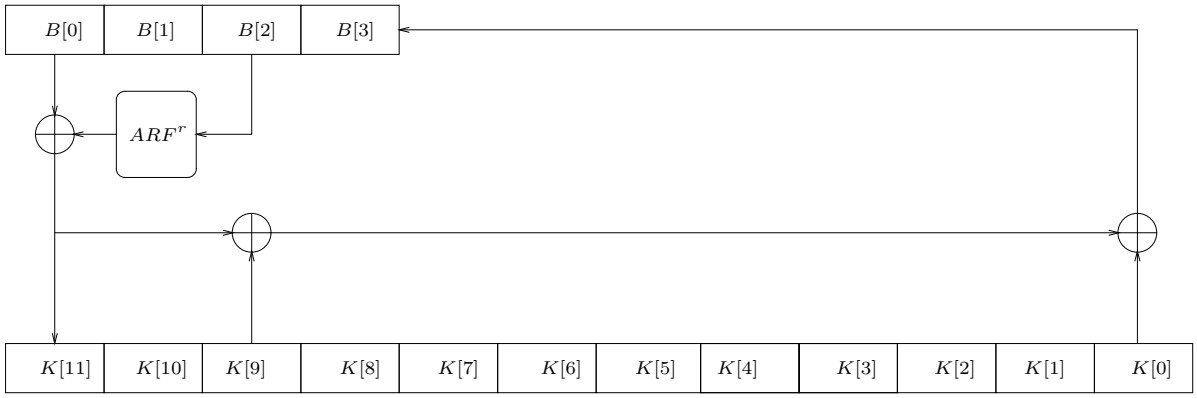

Figure 1: Clocking the Registers B and K once.

3.2 *ProduceOutput*(State, h)

ProduceOutput produces the digest using the register B. The output is the first h Least Significant Bits of the register B where the i-th LSB of the output is the i-th LSB of B:

$$\text{Output} = B\{512-h\}||B\{512-h+1\}||\cdots||B\{511\}.$$

4 Design Rationale

SHAMATA hash function is designed as a secure, fast, efficient, and flexible hash function. It can provide a satisfactory performance both on ultimate software platforms such as 64-bit processors and on hardware platforms such as FPGAs.

Recall that SHAMATA is a register based hash function. The registers B and K are loaded with the current message blocks and then updated. The registers B and K are operated like a block and a key of a Feistel type block cipher respectively. The register B can be considered as the block register and the register K can be considered as the subkey register of the block cipher (see Figure 2). Unlike classical block ciphers, the subkeys are updated by data, by the block register and by the key register.

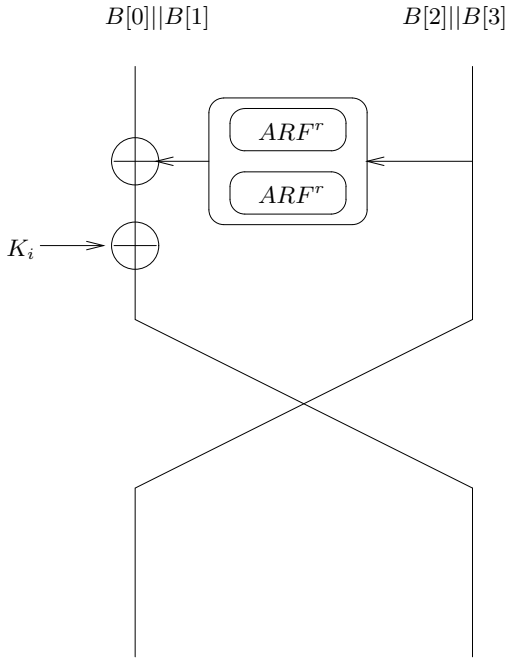

Figure 2: The High Level Structure of SHAMATA as a Feistel Network. K_i corresponds the data from the K register which is XOR'ed to produce two feedback values of the B register. K_i can be considered as the subkey of i-th step.

The main philosophy behind supplying security by SHAMATA is to maximize the number of active S-boxes in the AES round function ARF for any given difference in message blocks. It is well known that ARF has an excellent confusion-diffusion performance. Any message block is encoded by a linear code before it is incorporated into the registers B and K by `MixColumns` operation of AES. After encoding a message block M, we obtain the blocks (P, Q) and (P', Q'). The `MixColumns` operation ensures that at least 8 bytes of (P, Q) (and 8 bytes of (P', Q') also) are active for any given difference to the message block M.

Besides encoding message blocks, loading several copies of an encoded message block provides a fast diffusion.

In general, current data block is confused fully before loading next data in block cipher based hash functions. Unlike classical block ciphers in hash modes, the linear redundancies in blocks and high diffusion property of SHAMATA provides us of fast data loading securely. By this way, we obtain a fast hash function.

The index value *blockno* used in updating the registers B spoils the monotony and ensures that the feedback function is a time-depending function and changes at each data loading. This strengths against several attacks such as fixed-point attacks. Moreover, incorporating *blockno* destroys a structural property

of ARF and `MixColumns`: If all the sixteen bytes of an input of ARF (or `MixColumns`) are equal, then all the sixteen bytes of the output of ARF (or `MixColumns`) are also equal. All the operations in SHAMATA consist of these two functions except XOR. On the other hand, XOR preserves this property too. Assume that the internal state is zero at a time. Loading only the message blocks whose all the sixteen bytes are equal, would produce an internal state whose blocks would have this property also if *blockno* were not incorporated. This property is valid also for XOR differences of the message blocks whose all bytes are equal.

The output function takes values only from B so that preimage resistance is consolidated. Remark that the next state function of the nonlinear registers B and K are invertible. So the registers can also be clocked backward easily if their contents are completely known.

SHAMATA hash function is suitable in parallel implementation. Each of encoding and data loading can be implemented in parallel. Encoding of next data block can be done while applying *UpdateRegister* to the current data. Moreover, the input of the feedback function for the registers is formed from the third cell of the register B. It is straightforward that the feedback function can be run twice and the two feedback values of both B and K can be calculated in parallel.

4.1 Design Criteria for Building Blocks

- ARF: The AES round function ARF is used to compute both the feedbacks of K ($feedK_1, feedK_2$) and the feedbacks of B ($feedB_1, feedB_2$). This is the only nonlinear part of SHAMATA.

- Encoding: Data is encoded as $M \longrightarrow (P, Q)$ by `MixColumns` operation which is an $[32, 16, 8]$ block code over $GF(256)$. Hence, any difference in a data block results in at least 8 active bytes. In addition, small differences are diffused faster because of the concordant diffusion properties of ARF with `MixColumns`. For example, one byte difference in M causes 8-byte difference in (P, Q) and these differences cause 32-byte (full-byte) differences after ARF in two clocking since each difference in 8-byte differences comes to a different column after `ShiftRows` operation of ARF.

  Encoding restricts the control of attacker while trying to find collisions. Because, the attacker cannot impose any difference at a certain time. On the other hand, encoding is implemented by using tables of size 8x32 in optimized codes. So, we can obtain endian-free implementation of data loading.

- Incorporating Data: Data is incorporated into the registers B and K by XOR'ing. XOR operation is a simple operation both for software and for hardware. On the other hand, XOR is compatible with the building blocks of AES in terms of security. Any difference in (P, Q) remains unchanged after incorporating into the registers if there are no differences in the corresponding registers. For instance, one byte difference in M causes 8-byte difference in (P, Q) and this 8-byte difference is incorporated into $B[2]$ and $B[3]$ by XOR'ing.

- Registers: The register B is confused fast whereas the register K keeps history of differences for a long time (during 6 loading) by a large internal state. The fast confusion of the register B provides several iterations of ARF function. In fact, exactly n iterations of ARF is applied to the initial state of $B[2]$ and $B[3]$ before the $(n+1)$-th loading.

- Feedback Taps: There are two feedback taps except the most significant blocks of the registers B and K, namely $B[2]$ and $K[9]$. $B[2]$ is the input for the ARF function. Also, P is incorporated into $B[2]$. So, we obtain a fast confusion of data throughout the B register. On the other hand, since the feedback taps of B itself are all of even index number ($B[0]$ and $B[2]$), there is no diffusion between the blocks having even index number with the blocks having odd index number ($B[1]$ and $B[3]$) by means of B register. This can be seen easily from the Feistel approach depicted in Figure 2. The outputs of two ARF functions in Figure 2 are seen not to be diffused by a diffusion layer after two ARF functions. Whereas, $K[9]$ supplies diffusion among the blocks of B having even index number with the blocks of B having odd index number.

- IV: IV is determined by the digest size and is used to construct the initial states of the registers. The length of the initial states is 2048-bits and it is not efficient to use 2048-bit constants as initial

states for different digest sizes, particularly for restricted environments. So, we suggest an easy way to construct initial states.

4.2 Related Constructions

The design philosophy of SHAMATA hash function is based on some ideas and security criteria which have been discussed recently in the literature. For example we have adopted that the large internal state size improves security. Lucks has proved some statements in this direction [15]. It may be considered that the SHAMATA construction resembles the sponge construction [1]. Sponge constructions are also register based. Message blocks are incorporated into a register and then the register is updated. As one difference, we encode (and hence enlarge) the message before incorporating. Also, unlike sponge construction, the output is formed from the final internal state of the register at once. Moreover, we adopt that incorporating the index i into the registers enhances the security against certain attacks such as slide attacks or fixed point attacks. A similar idea was introduced in the framework HAIFA [3] by Biham and Dunkelman. However, the number of bits hashed so far is incorporated in HAIFA. Kelsey and Schneier also have proposed to add the message index number to the inputs of compression function [13]. We utilize the digest sizes so as to generate the initial states of the register like HAIFA construction. The only constant value is IV and it is determined by the digest size.

4.3 Tunable Parameters

SHAMATA has several tunable parameters. The specific values of the parameters are determined mainly by the digest size. The main parameter is r. The number of rounds of ARF, r, in each clock determines the confusion and the diffusion performance of the feedback function. Increasing r causes higher confusion/diffusion with a cost of more time. There is a straightforward tradeoff between security and efficiency for the choice of the parameter r.

The confusion of the current data block into the registers B and K is mostly given by the number of clocking the registers for each data loading which we determine as two. The number of clocking for each data loading can be increased. However, this causes extra overhead in time meanwhile the security is expected to be enhanced.

The initialization phase consists of 8 steps of IV loading. We need different initial internal states for different digest sizes. These internal states are not supposed to be so much confused and uncorrelated. So, we find 8 steps enough to obtain different initial internal states.

The finalization phase consists of 32 steps of $NoMB$ loading. The finalization phase is four times longer than the initialization phase. Because, the last data blocks must be well confused fairly. The latest block incorporated into the ARF function is $K[8]$ and it is incorporated into the ARF function after 5 steps. Hence, 10 iterations of ARF is applied to the value in $K[8]$ after 15 steps for $r = 1$. Assuming 10 rounds of ARF is secure, we expect that all the block of B are well confused by each message bit after 16 steps of the finalization phase. Adding another 16-step for security margin, we decide the finalization phase to consist of 32 steps.

4.4 Weakened Version

Several weakened version of SHAMATA can be designed by tuning the parameters such as r, number of clocks per each loading and the number of loadings in finalization phase.

Apart from changing the parameters, one smart way of weakening SHAMATA is cancel the encoding $M \longrightarrow (P, Q)$ by `MixColumns` operation and considering (P, Q) as a message block itself. This will increase the length of message block to 256-bit. Hence, the throughput is increased by a factor of two.

The initialization and the finalization steps may stay unchanged. It is an interesting question whether this weakened version satisfies all the security requirements.

4.5 Pros and Cons

SHAMATA has the following advantages:

- Not different algorithms for different digest sizes.
- Uses AES round function ARF: This gives advantages both in security and in implementation. All the results and developments for efficient implementation of AES both in the literature and in industry can be applied to SHAMATA. For example, a new set of Single Instruction Multiple Data (SIMD) instructions that will be introduced in the next generation of the Intel® processor family will enable much faster ARF implementation [26].

  In addition, AES and SHAMATA can be implemented in a crypto chip with a mutual implementation of ARF function, saving area. Moreover, the security of AES is well analyzed in the literature which gives information about the security of SHAMATA.
- Simple design. Easy to analyze.
- Flexible. Any flexibility property of AES can be used in order to achieve good performance in terms of area-time tradeoff.
- Highly parallelizable: AES is itself highly parallelizable. Moreover, two AES round function can be implemented in parallel. Encoding the data and incorporating it into the registers can be also done in parallel.
- No pitfalls of MD construction.
- Small memory for data since data block length is only 128-bit. In addition, almost no memory needed for constants.
- Promising high performance on next generation CPUs.
- Easily tweakable. Tunable parameters can provide different versions of SHAMATA easily.
- Word oriented even though it can supply endian-free implementation.

SHAMATA has the following disadvantages:

- Large internal state size. 2048 bits of registers makes implementation of SHAMATA in small areas like RFID chips difficult. We think that the internal state size must be twice as large as the digest size as security criterion like the security criterion on the internal state size of stream ciphers. One can design a hash function foiling the security pitfalls of MD construction by adopting this criterion.

  The digest size of SHAMATA can be arbitrarily large, up to 512 bits. In fact, implementation difficulties of hash functions having large digest size like 512 bits in a small area is a common problem. On the other hand, the technology of chips develops quite fast, resulting in significant increase in number of gates.
- Slow finalization. Finalization of SHAMATA consists of 32 steps which slows down SHAMATA while hashing small data. Anyway, finalization takes a few thousands of cycles which is, we think, tolerable for almost all the applications.

5 Security Analysis

We claim that SHAMATA satisfies full security against (second) preimage attacks and collision attacks. Moreover, SHAMATA hash function is secure against multicollision attacks such as the one given in [11].

5.1 Collision Resistance

We claim that the best attack to find a collision with SHAMATA is the brute force attack with workload $2^{n/2}$ where n is any digest size including 224, 256, 384 and 512. SHAMATA can provide the corresponding security level when n is replaced by a smaller number m, by choosing a fixed subset of output bits.

We can categorize the collision into four groups: For some messages of same block length, the Chaining Values (CV's) at the end of message loading (in the beginning of finalization) may be different but the digest values may be same. For some messages of same block length, the CV's at the end of message loading may be same. For some messages of different block length, the CV's at the end of message loading may be different but the digest values may be same. For some messages of different block length, the CV's at the end of message loading may be same (see Table 2).

$NoMB$'s vs CV's	Same	Different
Same	Same CV's with same finalization functions	Different CV's with same finalization functions
Different	Same CV's with different finalization functions	Different CV's with different finalization functions

Table 2: The four categories of hash collision. Any difference on the number of message blocks ($NoMB$'s) results in different finalization functions.

Same CV- Same $NoMB$: This is the most common attack scenario for finding collisions. In fact, the most famous attacks on SHA such as those in [2, 5, 19, 20] families are also of this group.

We should remark at this point that finding collision on chaining value is as hard as finding collision on hash value for MD constructions. This is one of the main weaknesses of MD constructions causing easy collisions (collisions on one block or collisions on two blocks if the compression function is weak), easy multicollisions (finding 2^k collisions costs only k times more than finding one collision [11]), and makes MD construction vulnerable to herding attack [13], long message second- preimage attacks [12], etc. All such attacks aim at finding collision on chaining value to obtain a collision on hash value.

One of the design principle adopted in SHAMATA is that finding collision on chaining value is much harder than finding collision on hash value. This is supplied by choosing a large CV length and by encoding the message block before incorporating it into the registers.

It is trivial that finding collision on CV just by imposing a difference on a single block is impossible. Even we can prove that it is also impossible to find collision on CV by imposing differences on 8 consecutive blocks. We prove this fact in the following statements.

Lemma 1 *If there exists a nonzero difference in $K[0]$ after loading the i-th message blocks, then it is impossible to find a collision on CV's after loading the $(i+1)$-th message blocks.*

Proof. Assume on the contrary that the statement is wrong. The next state function of the registers is bijective. So, it is impossible to destroy a nonzero difference in the clocking step. So, the only step in destroying a nonzero difference is the message loading. There must be differences in $B[2], B[3], K[11], K[7], K[5]$ and $K[3]$ (these are where data is incorporated) and there must be no difference in the other cells. On the other hand, the difference in $B[2]$ must be equal to the difference in $K[7]$ since the same data difference(ΔP) is incorporated both into $B[2]$ and into $K[7]$. If we clock the registers backward twice with these differences, we obtain that there is no difference in $K[0]$. This is a contradiction. QED

Theorem 1 *If the first feedback of the K register after a data loading ($feedK_1$) has a nonzero difference, then there exists no collision on CV's during forthcoming 7 data loadings.*

Proof. The difference that occurs in the first feedback (which is $ARF^r(B[2]) \oplus B[0])$) is loaded into $K[12-2i]$ before i-th loading for $i = 1, ..., 6$. Observe that $12-2i$ is an even number and there is no data loading on cells of K with even index. So, the difference will move throughout K register without being

cancelled by any data loading. In particular, the difference will come to $K[0]$ before loading 6-th forthcoming data blocks. It will live there after loading 6-th data blocks since there is no data incorporation into $K[0]$. This implies that there will be no collision on CV's after loading 7-th forthcoming data blocks by Lemma 1. QED

Corollary 1 *It is impossible to find a collision on CV's by imposing differences only on eighth consecutive blocks.*

Proof. The first feedbacks of K will be different just after the first different blocks are loaded since ARF^r is bijective and hence produces nonzero difference for nonzero input difference. Hence $ARF^r(B[2]) \oplus B[0]))$ produces nonzero difference since there is a difference on $B[2]$ but there is no difference on $B[0]$ after loading the first different blocks. On the, other hand, since the first feedbacks of K will be different, then there exists no collision on CV's during forthcoming 7 data loading by Theorem 1. Hence, 8 pairs of loadings which can be different results in different CV's. QED

Corollary 2 *Assume there is a collision on CV's after loading several pairs of message blocks. Then, the last seven pairs of $feedK_1$'s before the collision are all equal to each other.*

The proof of the Corollary 2 is immediate by Theorem 1. Hence, an attacker must try to obtain zero differences for the last seven $feedK_1$'s to obtain a collision.

Remark. We make use of just high level structure of SHAMATA to obtain the statements above. We do not use specific properties of ARF function or encoding the data before incorporating it into the registers.

The common technique to find a collision on CV for sponge construction is meet-in-the-middle attack type. Unlike finding collisions on SHA family, differences in several blocks are used to find a collision. Certain differences are imposed while clocking the registers in forward direction one one hand and certain differences are imposed while clocking the registers in backward direction on the other hand. Finally, the differences in forward direction meets the differences in backward direction, causing collision on CV at the beginning of backward direction. A collision attack by Peyrin to GRINDAHL ([18]) and two attacks to RadioGatún by Khovratovich and Biryukov ([14]) are recent examples of such attacks on sponge constructions.

It is certain that one must impose several difference on several blocks to find collision on CV for SHAMATA. The techniques used in [18] or in [14] can be tried for SHAMATA also. However, the problem seems more difficult since the internal state size of SHAMATA is larger, data is encoded before loading and three copies of encoded data is loaded. Moreover, the number of active bytes is increased dramatically by the ARF function and by encoding if the number of active bytes in the message block is very small (see Table 3). This property increases the complexity of finding collision using active bytes by the truncated differential approach.

Different CV**- Same** $NoMB$: In this case we have the same finalization function F for both messages. However, the inputs of F are different. Different inputs produces different and unrelated outputs even though the inputs are quite related. Because F is bijective and consists of 64 clocks applying 32 iterations of ARF. On the other hand, there may be collision on the internal states of the B register. However, the output is highly confused because of many iterations of ARF. So, it is expected that the probability of occurring collision on the internal states of the B register after the finalization gives no advantages to attacker.

Same CV**- Different** $NoMB$: Consider two messages of different block length. Assume that they have same CV's before the finalization. This case is quite difficult to occur because of MD strengthening at the end of message loading. In this situation there is a collision on CV's after the MD strengthening. If there would be no finalization step then this would result in collision on hash values. We expect that, even coming to this point is practically impossible.

Different $NoMB$'s cause different finalization functions. Let F_1 be the finalization function determined by $NoMB_1$ and F_2 be the finalization function determined by $NoMB_2$. Then there will be no relationship between the outputs $F_1(CV)$ and $F_2(CV)$ since 32 iterations of ARF function is applied. This problem is similar to the problem of finding two keys producing the same ciphertexts from a given plaintext where the block cipher is a 32-round Feistel network using ARF as round function (see Figure 2).

Different CV- Different $NoMB$: Consider two messages of different block length. Assume that they have different CV's before the finalization. This is the most difficult case to form a collision on the final internal states. Because both the finalization functions and their inputs are different. Let's note that attacker do not have any control during the finalization since there is no message loading in this phase. The registers are clocked 64 times since there are 32 loadings.

5.2 Preimage Resistance

We claim that the best attack to find a preimage with SHAMATA is the brute force attack with workload 2^n where n is any digest size including 224, 256, 384 and 512. SHAMATA can provide the corresponding security level when n is replaced by a smaller number m, by choosing a fixed subset of output bits.

The output is formed from certain bits of the register B. Assume that the final internal state of B is given to the attacker. Then, she cannot clock the registers without knowing the register K. Even, it is a difficult problem to recover the internal states just before the finalization step starts. This problem is similar to the problem of finding the plaintext of a given ciphertext which is decrypted by a 32-round Feistel cipher whose round function is ARF (see Figure 2), without knowing the key.

5.3 Second-Preimage Resistance

We claim that the best attack to find a second-preimage with SHAMATA is the brute force attack with workload 2^n even for extremely long messages bounded by 2^{64} bits, where n is any digest size including 224, 256, 384 and 512. SHAMATA can provide the corresponding security level when n is replaced by a smaller number m, by choosing a fixed subset of output bits.

The large internal state size of SHAMATA, the time dependent compression function and the finalization step foils second-preimage attacks, including the long message second-preimage attack by Kelsey and Schneier in [13]. See section 5.10.

One interesting property that SHAMATA attains is given in Corollary 2. According to the corollary, to have a collision on a chaining value CV, the previous 7 $feedK_1$ values must be equal for two messages where one of the messages is given to the attacker. In this case, the attacker has difficulty in satisfying even this condition. Because, one must choose the message blocks besides choosing the differences for corresponding blocks so as to satisfy 7 $feedK_1$ values to be equal. However, the attacker has no ability to choose message blocks for a given message.

5.4 Collision on Randomized Hash

SHAMATA can be used as randomized hash as proposed by Halevi and Krawczyk in [10] and also as a new proposal given in Section 8.6. We claim that the minimum complexity of finding a pair (M', r') for a given pair (M, r) such that $RMX - SHAMATA(r, M) = RMX - SHAMATA(r', M')$ where M is given by attacker (and of arbitrary length bounded by 2^{64} bits) and r is randomly chosen after M is provided, is 2^n where n is any digest size including 224, 256, 384 and 512. We assume that predicting r is almost impossible. The length of the message M does not determine the security level since the length of CV is much larger than the digest length.

This is a special type of second-preimage attack where the attacker has more control on the message. The attacker cannot choose the blocks that are incorporated into the register. However, she can choose any difference between any two blocks in a message (except the first block) incorporated into the registers. However, this does not give extra advantages to the attacker in comparison with the second-preimage attack since the differences in message blocks of a given message are not used to find a collision.

To find a collision on a CV, the previous 7 $feedK_1$ values must be pairwise equal for a message M' of same length as that of M by Corollary 2. Hence, one must choose both the message blocks and the differences for each incorporation to satisfy just this condition. However, the message blocks (with randomized value) of the message M cannot be chosen by the attacker.

5.5 Forgery Attacks on HMAC

We claim that any distinguishing attack on PRF-HMAC with SHAMATA requires at least $2^{n/2}$ queries with a time complexity of at least 2^n steps for any digest size n, including 224, 256, 384 and 512 bits.

Two iterations of SHAMATA supplies at least 64 iterations of ARF with an unknown key. The obtained key dependent function is expected to be PRF, since AES is expected to be PRF.

5.6 Differential Analysis

Differential analysis was first discovered by Biham and Shamir and applied to DES [4]. The attack exploits the statistical deficiencies in the output differences corresponding to certain input differences. For the hash functions, this corresponds to finding collision when the output difference is chosen to be zero. Therefore, differential analysis can be considered as the most substantial analysis for hash functions. Indeed, almost all the analyses including those in [2, 5, 19, 20] on SHA family have been based on differential analyses.

Our most important design principle may be considered as to maximize the number of active S-Boxes in ARF in order to provide statistically uniform distribution in output differences in a short time. This is achieved by data encoding and utilizing the proved fine confusion/diffusion properties of AES round function.

Any message block is encoded first and then the ARF function is applied to the encoded message. Any difference in one byte in a message block result in at least eight different bytes in the encoded block for (P, Q). These differences are confused in ARF. In general, the minimum number of active bytes are given in Table 3. For instance, 3-byte difference in M results in at least 14-byte difference in (P, Q). Assuming that this is the difference for the inputs of ARF, we obtain at least 16-byte difference after applying ARF function in two clocks.

M	(P, Q)	After ARF
1	8	32
2	11	24
3	14	16
4	17	8

Table 3: Minimum number of active bytes.

5.7 Fixed Point Analysis

One of the conspicuous pitfalls of the Merkle-Damgård construction is its vulnerability against second-preimage attacks if the underlying compression function has many fixed points and if it is easy to deduce the fixed points [8]. The large internal state of SHAMATA and applying a different feedback function at each data loading by incorporating the order of each data, renders second- preimage attacks based on fixed-points infeasible.

5.8 Multicollision Attacks

The multicollision attack of Joux (see [11]) is not feasible for SHAMATA because of the large internal state. Remark that multicollisions are formed by using collisions of chaining values in [11]. Chaining values are the internal states of the registers B and K in SHAMATA. Hence, it is much more difficult to find collision on chaining values than to find collision on hash values due to the large internal state size.

5.9 Herding Attacks

The herding attack by Kelsey and Kohno (see [12]) is also not feasible for SHAMATA because of large internal state. The collisions are all searched on chaining values in the attack. The large internal state size foils finding collisions on chaining values as easy as finding collisions on hash values.

5.10 Long Message Second-Preimage Attacks

The long message second-preimage attack by Kelsey and Schneier ([13]) is also not feasible for SHAMATA because of large internal state and time varying compression function. Recall that the order of a message block is also incorporated into the registers. Hence, any $(k, k + 2^k - 1)$ *expandable message* defined in [13] is unusable to find a second-preimage. Because the messages in the set of an *expandable message* are of different length and hence the collision on the chaining values for the *expandable message* makes no sense in case for SHAMATA since after the collision different compression functions are applied by incorporating different message orders.

5.11 Slide Attacks

A very recent application of slide attack on hash functions is introduced by M. Gorski, S. Lucks and T. Peyrin ([9]). The slide attack works on sponge like constructions. The self-similarity of the update function is exploited. The extended data which is incorporated into the register K is never zero. Also, there is a nonzero padding block at last. Even the index *blockno* spoils the self-similarity of updating the registers. In summary, it seems that the slide attacks does not work against SHAMATA.

5.12 Length-Extension Attacks

Length-extension attacks are applied to find hash value of an extended message M' from the hash value of M and its length where M is kept secret. The most important property that the attack exploits is that the hash value can be a chaining value at the same time. This is not the case in SHAMATA. There is a finalization step in SHAMATA and a hash value can not be a chaining value. Anyway, they are of different length.

Chaining values must be kept secret to third parties for some applications such as HMAC, especially if the last input block of the compression function is public. However, last chaining value is given as the hash value for large variety of constructions such as MD construction. Such a property renders the scheme vulnerable against length-extension attacks. The output function is different from the compression function in SHAMATA. So, the last chaining value is not given as output. In addition, the finalization function is different from the compression function, foiling the length-extension attack. Also, the index is reset in the finalization step. So, an attacker cannot use the finalization step as a part of message loading.

A special kind of length-extension attack is a kind of slide attack and is given in [9]. However, as explained in Section 5.12, this attack is not applicable to SHAMATA.

6 Hardware Performance

SHAMATA was implemented on a Xilinx Virtex-5 LX30 chip with the design language VHDL for 224-bit and 256-bit digest sizes as examples. Two parallel ARF functions were implemented in parallel with 32 parallel `SubBytes` operations as look-up tables. The throughput is around 9 Gbit per second with 95 MHz clock frequency. The total area is around 28K gates. This is a time optimized implementation of SHAMATA.

7 Software Performance Figures

7.1 8-bit Processors

An implementation of SHAMATA has been simulated on the Atmel AT89C51ED2 micro-controller in 33 MHz using Keil development environment with memory capacity of 1792 Bytes on-chip XRAM, 64K Flash on-chip and 2048 Bytes EEPROM. There is no operation system available.

Code lengths and execution times are given in Table 4. We implement AES S-box as a table. MixColumns operation is implemented as in the reference code (byte-wise implementation). The code is a C code. We expect that the code size and the speed can be enhanced significantly by implementing it in assembly.

Hash Length	224-bit	256-bit	384-bit	512-bit
MOIP	0.08 s	0.08 s	0.09 s	0.09 s
TOIP	0.00047 s	0.00047 s	0.00047 s	0.00047 s
Code Size with MOIP	4384 Byte	4384 Byte	4744 Byte	4744 Byte
Code Size with TOIP	5369 Byte	5361 Byte	5727 Byte	5722 Byte
Memory with MOIP	684 Byte	688 Byte	720 Byte	736 Byte
Memory with TOIP	684 Byte	688 Byte	720 Byte	736 Byte
Finalization	0.35 s	0.35 s	0.39 s	0.39 s
Update (16 Byte)	0.01 s	0.01 s	0.01 s	0.01 s
Update (1024 Byte)	0.66 s	0.66 s	0.74 s	0.74 s

Table 4: Performance of SHAMATA on 8-bit microprocessor. TOIP: Time Optimized Initialization Phase (Initialization phase is pre-computed). MOIP: Memory Optimized Initialization Phase (Initialization phase is performed). Update is message loading part.

7.2 Optimized ANSI C in 32 Bit Architecture

SHAMATA has been implemented on a 2.66 GHz Pentium Core2Duo, running Windows Vista Ultimate 32Bit with 2GB of RAM. MS Visual Studio 2005 is used as a compiler with optimization option O2. The implementation is serial and portal implementation using only one CPU. Parallelization properties of the CPU are not utilized. The code sizes are around 20-25 KBytes for any digest size.

The `Mixcolumns` operations together with `SubBytes` are implemented as four tables of size 8×32 as proposed by Daemen and Rijmen [6]. Moreover the `Mixcolumns` operations during encoding the data are also implemented as four tables of sizes each 8×32. All the 8 tables are together cover 8 KBytes of memory. This provides us of fast implementation since we get rid of multiplications in $GF(256)$. Indeed `Mixcolumns` as a byte-wise implementation with multiplications takes 376 cycles whereas it takes 16 cycles as table based implementation.

Initialization phase is done in two ways: Either the initial states of the registers B and K are pre-computed and saved in memory, or the initial states are computed on-line from the IV. The former method costs extra memory of 2048 bits with negligible time whereas the latter method costs time of roughly 2000 cycles for 224 and 256 bit hash lengths and 3000 cycles for 384-bit and 512-bit hash lengths with negligible extra memory for initialization.

Performance of the code is given in Table 5.

Hash Length	TOIP as cycle	MOIP as cycle	Hashing as Cycle/Byte	Finalization as cycle
224 & 256	136	1976	15	6120
384 & 512	136	2944	22	8704

Table 5: Performance of SHAMATA on 32-bit processor. TOIP: Time Optimized Initialization Phase (Initialization phase is pre-computed). MOIP: Memory Optimized Initialization Phase (Initialization phase is performed). The overhead due to padding is included in Finalization phase.

The tables allocate around 8200 Bytes in memory and the variables allocate around 1100 Bytes in memory for any digest sizes for TOIP implementation.

7.3 Optimized ANSI C in 64 Bit Architecture

SHAMATA has been implemented in a serial mode on a 2.66 GHz Pentium Core2Duo, running Windows Vista Ultimate 64 Bit with 2GB of RAM. MS Visual Studio 2005 is used as a compiler. Performance of the code is given in Table 6. The code sizes are around 20-25 KBytes for any digest size.

The `Mixcolumns` operations together with `SubBytes` are implemented as four tables of size 8×32. Moreover the `Mixcolumns` operations during encoding the data are also implemented as four tables of sizes each 8×32. All the 8 tables are together cover 8 KBytes of memory. This provides us of fast implementation since we get rid of multiplications in $GF(256)$. Indeed `Mixcolumns` as a byte-wise implementation with multiplications takes 376 cycles whereas it takes 16 cycles as table based implementation.

Initialization phase is done in two phase: Either the initial states of the registers B and K are pre-computed and saved in memory, or the initial states are computed on-line from the IV. The former method costs extra memory of 2048 bits with negligible time whereas the latter method costs time of roughly 750 cycles for 224 and 256 bit hash lengths and 1400 cycles for 384-bit and 512-bit hash lengths with negligible extra memory for initialization.

Hash Length	TOIP as cycle	MOIP as cycle	Hashing as Cycle/Byte	Finalization as cycle
224 & 256	104	752	8	3384
384 & 512	104	1368	11	5184

Table 6: Performance of SHAMATA on 64-bit processor. TOIP: Time Optimized Initialization Phase (Initialization phase is pre-computed). MOIP: Memory Optimized Initialization Phase (Initialization phase is performed). The overhead due to padding is included in Finalization phase.

The tables allocate around 8200 Bytes in memory and the variables allocate around 1100 Bytes in memory for any digest sizes for TOIP implementation.

8 Functionality

SHAMATA hash function can be used in any cryptographic applications currently specified in FIPS and NIST Special publications that require a NIST-approved hash algorithm.

8.1 Digital Signature Standard

SHAMATA hash function is suitable with FIPS 180-2 Secure Hash Algorithm Standard. Hence, it is also suitable for FIPS 186-2 Digital Signature Standard and can be used in digital signature applications.

8.2 HMAC

SHAMATA hash function can be used as the HMAC algorithm (FIPS 198 [23]) by using inner pad (ipad) and outer pad (opad) with a suitable block length. Recall that the block length of SHAMATA is 128-bit whereas the block lengths of SHA-256 and SHA-512 are 512-bit and 1024-bit respectively.

The key "K_0" in FIPS 198 is determined by the original key "K" and the block length "B" in [23]. We propose to use the block length of the corresponding SHA to determine "K_0" when using SHAMATA as the HMAC defined in FIPS 198. That is, if the output of the hash is less than or equal to 256-bit, then determine "K_0" by comparing the the length of the key "K" with 512 (take "B" as 512) and if the output of the hash is greater than 256-bit, then determine "K_0" by comparing the the length of the key "K" with 1024 (take "B" as 1024) in steps 1-2-3 of Table 1 in FIPS 198. Any other applications using HMAC in FIPS 198 such as SP 800-56A, SP 800-90 or FIPS 186-2, can use SHAMATA as HMAC as proposed here.

8.3 PRF- HMAC

SHAMATA hash function with HMAC algorithm in FIPS 198 can be used as a Pseudo Random Function (PRF). Recall that the proposed length of the key used with ipad and opad in hashing with SHAMATA as HMAC is slightly different than the original length described in FIPS 198. See section 8.2. The PRF can be generated as in RFC 4868 ([25]) by replacing SHA with SHAMATA.

8.4 Pair-Wise Key Establishment

SHAMATA hash function is suitable with FIPS 180-2 Secure Hash Algorithm Standard. Hence, it is also suitable for SP 800-56A Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography.

8.5 Deterministic Random Bit Generator

Since SHAMATA hash function is suitable with FIPS 180-2, it can also be used in Deterministic Random Bit Generators (DRBGs).

8.6 Randomized Hash

Randomized Hash is a kind of modes of operation for hash functions using a random salt value r, proposed by Halevi and Krawczyk in [10]. If H is a hash function, the corresponding randomized hash is given as

$$RMX - H = H_r(M) = H(r||M_1 \oplus r||M_2 \oplus r|| \cdots ||M_n \oplus r).$$

SHAMATA can be used as randomized hash with RMX construction where r is a multiple of block length. There is another version of randomized hash in [10]. However, we adopt this version even though we think that there is no difference in terms of practical security for SHAMATA.

New Proposal: This proposal is same as RMX-SHAMATA, except different IV and $NoMB$ are used. This proposal can be used when the salt, r, is of length 128 bits or 256 bits. If the length of r is 128 bit then $IV' = IV \oplus r$ and $NoMB' = NoMB \oplus r$. If the length of r is 256 bit then $IV' = IV \oplus r_0$ and $NoMB' = NoMB \oplus r_1$ where r_0 is the first 128-bit part and r_1 is the last 128-bit part of r. Then everything is same as in the case for RMX-SHAMATA, except using IV' instead of IV and $NoMB'$ instead of $NoMB$. We call this proposal as RMXX-SHAMATA.

9 Test Vectors

```
224 bit hash length:
Message = 52A608AB21CCDD8A4457A57EDE782176
MD = 976BE2195E6097092A0F8FA11C1EC930FFC205585B9EEC325872E98C


256 bit hash length
Message = 52A608AB21CCDD8A4457A57EDE782176
MD = 4A6A43A58A6240672714269A7FD6819C097F23E209EE326BC06B2C8577A4A3E7

384 bit hash length
Message = 52A608AB21CCDD8A4457A57EDE782176
MD =33BB1FF0CD78A9F3E78E87A613B4C495894028402AA367C3510679469FC8083CAD
757AC0BBA59E3E4550825E83F62FBD


512 bit hash length
Message = 52A608AB21CCDD8A4457A57EDE782176
MD =9ECB44C6EFBCFB8F6993DC0CAE2F10AD79A70168167B0318D32C03BB4298FEED8A
985873183CEA6B33F97FFEF88D2B042FD592D5359F0843761D3F906B93CA86
```

Acknowledgement

We are grateful to Önder Yetiş, Alparslan Babaoğlu and Murat Apohan for their invaluable supports. We thank to Ümit Göğüsgeren who implemented SHAMATA in FPGA, Halil Özçiçek and Mahmut Sağıroğlu who helped in implementation in 8-bit processor. Moreover, we would like to thank to Fatih Sulak and Ali Aydın Selçuk for their useful comments.

References

[1] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Sponge Functions. In Proc. of *Third NIST Cryptographic Hash Workshop,* 2007.

[2] E. Biham and R. Chen. Near-Collisions of SHA-0. In M.K. Franklin, editor *Advances in Cryptology, CRYPTO 2004*, LNCS 3152, pp.290-305, Springer, 2004.

[3] E. Biham and O. Dunkelman. A Framework for Iterative Hash Function: HAIFA. In Proc. of *Second NIST Cryptographic Hash Workshop,* 2006.

[4] E. Biham and A. Shamir. *Differential Cryptanalysis of Data Encryption Standard,* Springer, 1993.

[5] F. Chabaud and A. Joux. Differential Collisions in SHA-0. In H. Krawczyk, editor *Advances in Cryptology, CRYPTO 1998*, LNCS 1462, pp.56-71, Springer, 1998.

[6] J. Daemen and V. Rijmen. *Design of Rijndael. AES-The Advanced Encryption Standard*, Springer-Verlag, 2002.

[7] I. Damgård. A Design Principle for Hash Functions. In G. Brassard, editor *Advances in Cryptology, CRYPTO 1989*, LNCS 435, pp.416-427, Springer, 1989.

[8] R. D. Dean. *Formal Aspects of Mobile Code Security.* Ph.D Dissertation, Princeton University, 1999.

[9] M. Gorski, S. Lucks and T. Peyrin. Slide Attacks on a Class of Hash Functions. *To appear in ASIACRYPT 2008.*

[10] S. Halevi, H. Krawczyk. Strengthening Digital Signatures Via Randomized Hashing. *Advances in Cryptology, CRYPTO 2006*, LNCS 4117, pp.41-49, Springer, 2006.

[11] A. Joux. Multicollisions in Iterated Hash Functions. In M.K. Franklin, editor *Advances in Cryptology, CRYPTO 2004*, LNCS 3152, pp.306-316, Springer, 2004.

[12] J. Kelsey and T. Kohno. Herding Hash Functions and the Nostradamus Attack. In S. Vaudenay, editor *Advances in Cryptology, EUROCRYPT 2006*, LNCS 4004, pp.183-200, Springer, 2006.

[13] J. Kelsey and B. Schneier. Second-Preimages on n-bit Hash Functions for Much Less Than 2^n work. In R. Cramer, editor *Advances in Cryptology, EUROCRYPT 2005*, LNCS 3494, pp.474-490, Springer, 2005.

[14] D. Khovratovich and A. Biryukov. Two Attacks on RadioGatún, *To appear in Proc. of Indocrypt 2008.*

[15] S. Lucks. A failure-Friendly Design Principle for Hash Functions. In B. Roy, editor *Advances in Cryptology, ASIACRYPT 2005*, LNCS 3788, pp.474-494, Springer, 2005.

[16] A. J. Menezes, S. A. Vanstone and P. C. Van Oorschot. *Handbook of Applied Cryptography.* CRC Press, Inc., Boca Raton, FL, USA, 1996.

[17] R.C. Merkle. One Way Hash Functions and DES. In G. Brassard, editor *Advances in Cryptology, CRYPTO 1989*, LNCS 435, pp.428-446, Springer, 1989.

[18] T. Peyrin. Cryptanalysis of GRINDAHL. In K. Krosawa, editor *Advances in Cryptology, ASIACRYPT 2007*, LNCS 4833, pp.551-567, Springer, 2007.

[19] X. Wang, H. Yu and Y.L. Yin. Efficient Collision Search Attacks on SHA-0. In V. Shoup, editor *Advances in Cryptology, CRYPTO 2005*, LNCS 3621, pp.1-16, Springer, 2005.

[20] X. Wang, H. Yu and Y.L. Yin. Finding Collisions in the Full SHA-1. In V. Shoup, editor *Advances in Cryptology, CRYPTO 2005*, LNCS 3621, pp.17-36, Springer, 2005.

[21] The National Institute of Standards and Technology. FIPS 197: ADVANCED ENCRYPTION STANDARD(AES), 2001.

[22] The National Institute of Standards and Technology. FIPS 180-2 Secure Hash Algorithm Standard, 2002.

[23] The National Institute of Standards and Technology. FIPS 198 The Keyed-Hash Message Authentication Code (HMAC), 2002.

[24] `http://csrc.nist.gov/groups/ST/hash/sha-3/index.html`

[25] RFC 4868 `http://www.faqs.org/rfcs/rfc4868.html`.

[26] `http://software.intel.com/en-us/articles/advanced-encryption-standard-aes-instructions-set.`

Appendix : Modified AES Round Function ARF

The feedback function of the registers B and K makes use of the AES round function ARF without key addition for 128 bit block length.

The round transformation of AES is given in pseudo C notation as

```
AESRound(State,RoundKey)
{
  SubBytes(State);
  ShiftRows(State);
  MixColumns(State);
  AddRoundKey(State, RoundKey);
}
```

in the standard FIPS 197, "ADVANCED ENCRYPTION STANDARD(AES)" by NIST [21]. We use the same transformation except running the

```
  AddRoundKey(State, RoundKey)
```

function. More precisely, the ARF function is given in pseudo C notation as

```
ARF(State)
{
  SubBytes(State);
  ShiftRows(State);
  MixColumns(State);
}
```

In other words, ARF is the AES round function with the RoundKey which is equal to zero.