INTERNATIONAL JOURNAL OF COMPUTER ENGINEERING & TECHNOLOGY (IJCET)

ISSN 0976 – 6367(Print)
ISSN 0976 – 6375(Online)
Volume 4, Issue 4, July-August (2013), pp. 01-19
© IAEME: www.iaeme.com/ijcet.asp
Journal Impact Factor (2013): 6.1302 (Calculated by GISI)
www.jifactor.com

IJCET

©IAEME

MOIM: A NOVEL DESIGN OF CRYPTOGRAPHIC HASH FUNCTION

Mohammad A. AlAhmad[1], Imad Fakhri Alshaikhli[2]

[1,2]Department of Computer Science, International Islamic University of Malaysia, 53100 Jalan Gombak Kuala Lumpur, Malaysia

ABSTRACT

A hash function usually has two main components: a compression function or permutation function and mode of operation. In this paper, we propose a new concrete novel design of a permutation based hash functions called *MOIM*. *MOIM* is based on concatenating two parallel fast wide pipe constructions as a mode of operation designed by Nandi and Paul, and presented at Indocrypt 2010 where the size of the internal state is significantly larger than the size of the output. And the permutations functions used in *MOIM* are inspired from the SHA-3 finalist Grøstl hash function which is originally inspired from Rijndael design (AES). As a consequence there is a very strong confusion and diffusion in *MOIM*. Also, we show that *MOIM* resists all the generic attacks and Joux attack in two defense security levels.

Keywords: FWP - permutation - concatenation – mixing

1. INTRODUCTION

Cryptographic hash functions have indeed proved to be the workhorses for modern cryptographic hash functions. Another name given to cryptographic hash functions is “Swiss knife army” because it can serve many different purposes such as digital signatures, conventional message authentication to secure passwords storage or forensics data identification. Cryptographic hash functions take an unfixed size of input and produce a fixed size of an output. A hash function usually built from two main components: (1) a basic primitive compression function C and (2) an iterative mode of operation H, where the symbol H^C denotes the hash function H^C based on the compression function C. Most hash functions in use today are so-called iterated hash functions, i.e. Merkle-Damgård (MD), based on iterating a compression function. Examples of iterated hash functions are MD4 [1], MD5 [2], SHA [3] and RIPEMD-160 [4]. For a cryptographic hash function H^C, if the compression function C is resistant to the following attacks, then the hash function considered secure:

1. **Preimage**: it is computationally infeasible to find x' such that $H(x') = y$, where $y = H(x)$.
2. **2nd preimage**: it is computationally infeasible to find x and $y=H(x)$ find $x' \neq x$ where $H(x') = y$.
3. **Collision**: it is computationally infeasible to find x and x' where $x' \neq x$ and $H(x) = H(x')$.

Birthday attack (collision) is applicable to all hash functions after about $2^{n/2}$. Brute force attack preimages and 2nd preimages can be found after about 2^n. It clearly stated that the existence of 2nd preimage implies the existence of collision [5]. At this point, we have to distinguish between theoretical and practical break. To illustrate the theoretical break, if an attack succeeds to prove that a hash function can be exploited (e.g. finding a collision, or pre-image, or second pre-image) with work less than required by the birthday or brute force attack, the hash function is considered broken, even if the work required to break it is still infeasible in practice. Indeed, finding such flaws in a hash function is an evidence of structural weaknesses that may be exploited at later stages to turn this theoretical break into a practical one; the primary example is MD5, which was first theoretically broken, then the attacks eventually evolved and today practical collisions can easily be found in MD5 [7]. So, everything started in 2004, when collisions were announced in SHA-0, MD4, MD5, HAVAL-128, and RIPEMD. French researcher Antoine Joux [6] presented the collision in SHA-0, and a group of collisions against MD4, MD5, HAVAL- 128, and RIPEMD were found by the Chinese researcher Xiaoyun Wang with co-authors Dengguo Feng, Lai, and Hongbo Yu [7]. After that, in February 2005, the same Xiaoyun Wang, Lisa Yiqun Yin, and Hongbo Yu found collisions in SHA-1 using 2^{69} hash computations [8]. Several strategies were developed to thwart these attacks. Stefan lucks [9] introduced the wide pipe hash construction as an intermediate version of Merkle-Damgård to improve the structural weaknesses of Merkle-Damgård design. The process is similar to Merkle-Damgård algorithm steps except of having a larger internal state size, which means the final hash digest is smaller than the internal state size of bit length. For example, the final compression function compresses the internal state length (for ex, *2n*- bit) to output a hash digest of *n*-bit. This simply can be achieved by discarding the last half of *2n*-bit output. Mridul Nandi and Souradyauti Paul [10] proposed the fast wide pipe (FWP) construction to overcome these attacks. It is twice faster than the wide pipe construction. FWP is used in this paper to construct *MOIM* keyed hash function. It is used as an operation of mode for *MOIM* (see section 3). HAsh Iterated FrAmework (HAIFA) is also a patched version Merkle-Damgård construction [11]. HAIFA design solves many of the internal collision problems associated with the classic MD construction design by adding a fixed (optional) salt of s-bits along with a (mandatory) counter C_i of *t*-bits to every message block in the iteration *i* of the hash function. Wide-pipe and HAIFA are very similar designs. Sponge construction is an iterative construction designed by Guido Bertoni, Joan Daemen, Micheal Peeter and Gilles Van Assche to replace Merkle-Damgård construction [12]. It is a construction that maps a variable length input to a variable length output. Keccak (SHA-3 winner) hash function uses sponge construction. Namely, by using a fixed-length transformation (or permutation) *f* that operates on a fixed number of $b = r + c$ bits. Where *r* is called the bitrate and *c* is called the capacity. First, the input is padded with padding algorithm and cut into blocks of *r* bits. Then, the *b* bits of the state are initialized to zero. The sponge construction operates in two phases:

- **Absorbing phase:** The *r*-bit message blocks are XORed with the first *r* bits of the state of the function *F*. After processing all the message blocks, the squeezing phase starts.

• **Squeezing phase:** The first *r* bits of the state are returned as output blocks of the function *F*. lastly, the number of output blocks is chosen by the user [13].

The sponge construction has been studied by many researchers to prove its security robustness. Bertoni et al. [13] proved that the success probability of any generic attack to a sponge function is upper bound by its success probability for a random oracle plus $N^2/2^{c-1}$ with *N* the number of queries to *f*. Aumasson and Meier [14] showed the existence of zero-sum distinguishers for 16 rounds of the underlying permutation *f* of Keccak hash function. Boura, Canteaut and De Cannière [15] showed the existence of zero-sums on the full permutation (24 rounds).

For the reasons discussed earlier, most of modern hash functions are permutation-based hash function due to the weaknesses of Merkle-Damgård construction. Permutation is a mathematical term for a function that rearranges the elements of its domain so that exactly one input is mapped to each output. Some examples are permutation-based hash functions are PHOTON [16], Quark [17], Grøstl [18], Luffa [19] and Keccak [20]. Also, Keccak is the winner of SHA-3 competition and it is permutation-based sponge construction. Our designed concrete hash function *MOIM* presented in this paper is permutation-based hash function. This paper is organized as follows, In Section 2, we give a high-level summary of the *MOIM* proposal, and state the design goals. In Section 3, we present the details of the proposal and in Section 4, we describe the features specific to *MOIM* and motivate our design choices. Section 5 introduces some alternative descriptions of *MOIM* and Section 6 describes some cryptanalysis notations of *MOIM*. Finally, we conclude in Section 7.

2. DESIGN GOALS

In this section, we give a brief motivation of the *MOIM* proposal. Particularly, we aim to have security margins at several layers of abstraction in the design.

2.1 Overall goals for the hash

Here we state overall design goals for *MOIM*. Details of these goals are discussed in the next sections.

1. Simplicity of design analysis since *MOIM* is based on a small number of two permutations *P* and *Q* in each side of the two parallel FWPs of *MOIM* keyed hash function.
2. Simplicity of proving properties of the design.
3. Cryptanalysis of the design is straightforward since a well known construction and permutation functions are used.
4. Prevention of generic attacks, Joux attack and length-extension attacks.

2.2 Design Features

MOIM is a failure tolerant design due to the following feature:

1. The internal state is *2n-bits* where the final output is *n-bits*, hence all known generic attacks are thwarted.
2. Attacks on the compression function are not transferable to *MOIM* keyed hash function.
3. Joux attack are thwarted in two levels:
   a. By using FWP construction in each side of *MOIM* design
   b. Mixing the two parallel FWPs results intermediate chaining values with the other side of *MOIM* after *five* compression functions.

3. SPECIFICATION OF MOIM

MOIM is a concrete hash function, takes an input of length up to 2^{64} bits and returns a hash digest with length *1024*-bits. We now specify the *MOIM* keyed hash function.

3.1 The hash function construction

The *MOIM* keyed hash functions iterate the compression function *C* which uses two parallel FWP constructions. The fast wide pipe (FWP) construction is the faster version of the wide pipe construction designed by Lucks as mentioned earlier. The secret key and the input message of the wide pipe compression function are *2n* and *m* bits. The initial value 1 (IV1) and the initial value 2 (IV2) poses the secret keys to FWP. Once these two values iterated into the FWP construction, then, we can call them the chaining values. The wide pipe compression function can be expressed in as $C : \{0, 1\}^{m+2n} \rightarrow \{0, 1\}^{2n}$. But since the FWP is twice faster than the wide pipe construction, its compression function can be expressed as $C : \{0, 1\}^{m+n} \square \{0, 1\}^{n}$ where the message and the chaining input to the compression function are $m+n$ and n bits; thus, speeding up the hashing operation by allowing $m+n$ bits of message instead of just m bits per compression function invocation. As Figure 1 describe *MOIM* keyed hash function design, firstly, the message M is padded and divide into *l*-bit message blocks $m_1,...,m_L$, and each message block is processed sequentially (in both sides of the two parallel FWP constructions). Two initial *l*-bit values $h_0 = iv_1$ and $h_0' = iv_2$ are defined, and subsequently the message blocks m_i are processed. Algorithm 1 describes FWP construction in more details.

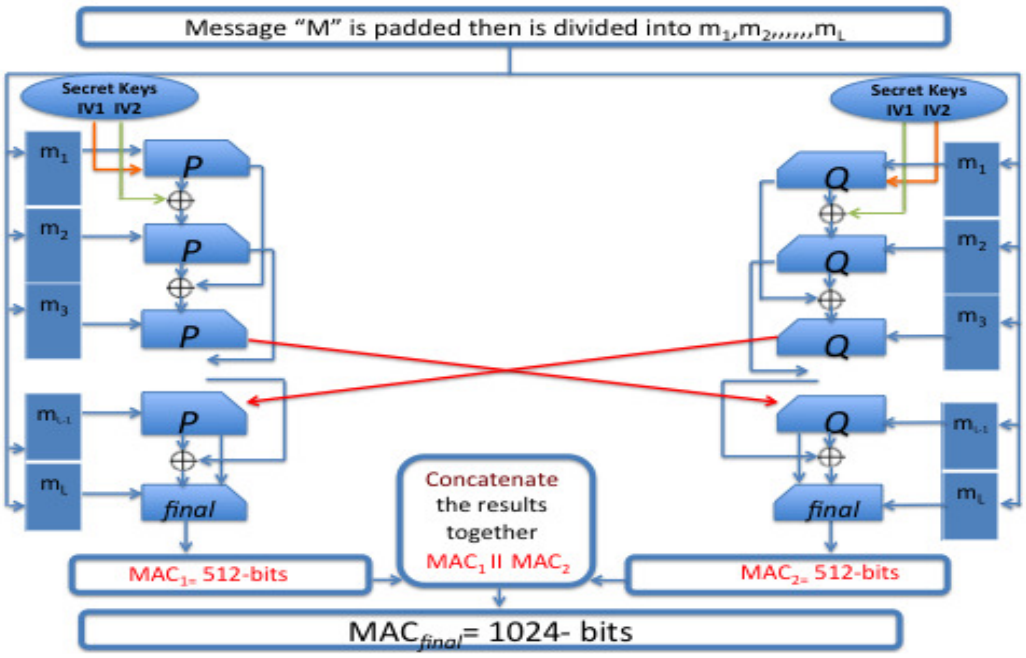

Figure 1. The *MOIM* keyed hash function

Again, algorithm 1 is performed in parallel fashion in both sides of the FWP *MOIM* keyed hash function construction but with different permutation functions in each side (see section 3.2 for the compression function construction). To illustrate algorithm 1, two initial l-bit values $h_0 = iv_1 = h_0' = iv_2 = 0^n$ are defined, then, the message M is padded by the padding rule pad(M). The padding rule pad(M) is the execution of the following operation: append t zero bits and a 64-bit encoding of |M| to the message M [18]. Select the least integer $t \geq 0$ such that $|M|+t+n+64 = 0 \bmod l$. The length of the original message M is encoded in the last block of message M. FWP_t^C denotes all the compressions functions C used in FWP except for the last compression function. In particular, the subfunction in algorithm 1 processes these values ($h_0 = h'_0, M_0, M_1, \ldots\ldots, M_{k-2}$) starting from the first compression function and ending in the one before the last compression function. Then, algorithm 1 returns the final truncated hash digest by using the last compression function denoted by *(final)* as shows in Figure 1 and line 4 of algorithm 1 (See section 3.3 for the *final* output compression function).

Algorithm 1. The FWP mode of operation with the compression function C (i.e., FWP^C) [10]

Input: Message M
Output: Hash output h of size n bits
Initialize: $h_0 = h_0' = 0^n$
1: $M_0 \| M_1 \| \ldots\ldots \| M_{k-1} = \text{pad}(M)$ where $|M_i| = l$ for all $i < k-1$ and $|M_{k-1}| = l-n$;
2: $(h_{k-2} = h'_{k-2}) = FWP_t^C\ (h_0 = h'_0, M_0, M_1, \ldots\ldots, M_{k-2})$; /***See Subfunction below***/
3: $C(h_{k-2} \| h'_{k-2} \| M_{k-1}) = h''_{k-1} \| h_{k-1}$;
4: return hash output $h = h'_{k-1}$;
Subfunction $FWP_t^C\ (h_0 = h'_0, M_0, M_1, \ldots\ldots, M_{k-2})$
5: **for** $i = 0$ *to* $k-2$ *do*
6: $C(h_{i-1} \| M_i) = h_i'' \| h_i'$;
7: $h_i = h_i'' \oplus h'_{i-1}$;
8: **end for**
9: **return** $(h_{k-2} = h'_{k-2})$;

3.2 The compression function construction

MOIM has two different compression functions C in each side of the design. These compression functions are based on the underlying l-bit permutations P and Q inspired from Grøstl hash function. Hence, the two permutations P and Q in each side of *MOIM* keyed hash function are defined as follows:

$$f_1(h,m) = P(h \oplus m) \oplus h \quad \textit{(one side of MOIM)}$$
$$f_2(h,m) = Q(h \oplus m) \oplus h \quad \textit{(one side of MOIM)}$$

The constructions of f_1 and f_2 are illustrated in Figure 2a and 2b respectively. In section 3.3, we describe how P and Q are defined.

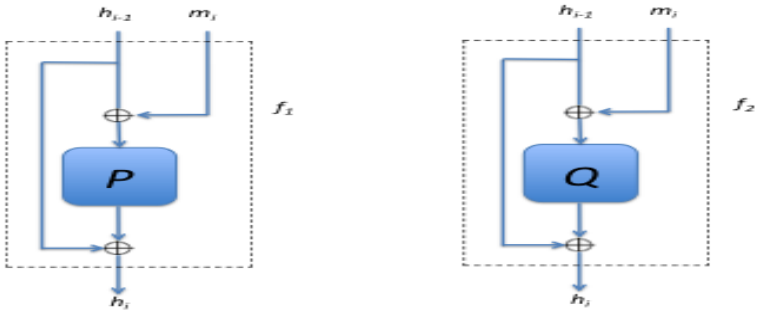

Figure 2. The two compression functions f_1 and f_2 of P and Q permutations respectively

3.3 The output transformation (*final)*

Let φ be the truncation operation that discards from *2n-bits* to *n-bits*. The output transformation MAC_1 illustrated in Figure 3 is then defined by

$$MAC_1 = trunc_n P \oplus h_{i-2}$$

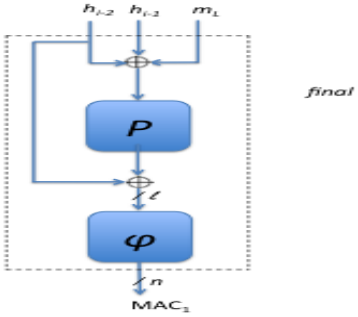

Figure 3. The output transformation φ computes $trunc_n P$

To obtain MAC_2, the same process is performed with replacing Q instead of P as Figure 3 shown.

3.4 Concatenation of *MAC*and Y_2

As Figure 3 shows that the two MAC values of MAC_1 and MAC_2 in each side of the two parallel FWP hash constructions are concatenated together to form $MAC_1 IIMAC_2$. $MAC_1 IIMAC_2$ are *1024-bits* long which imply that MAC_1 and MAC_2 are *512-bits* long each.

3.5 The design of *P* and *Q*

The design of P and Q were inspired from Grøstl [18] which originally inspired by the Rijndael block cipher algorithm [21, 22]. Rijndael design consists of a number of rounds R, which consists of a number of round transformations. Since P and Q are much larger than the 128-bit state size of Rijndael, most round transformations have been redefined. In *MOIM*, a total of four round transformations are defined for each permutation [18]. These are

- AddRoundConstant
- SubBytes
- ShiftBytesWide
- MixBytes.

A round R consists of these four round transformations applied in the above order as illustrated in Figure 4. Hence,

R = MixBytes • SubBytes • SubBytes • AddRoundConstant

We note that all rounds follow this definition. We denote by r the number of rounds. Concrete recommendations for r will be given in Section 3.5.6.

The transformations operate on a state, which is represented as a matrix *A* of bytes (of 8 bits each). The matrix has 8 rows and 16 columns. For the simplicity of exposure, we describe a matrix that has 8 rows and 8 columns, where *MOIM* is the large variant of this concept (8×16) [18].

3.5.1 Mapping from a byte sequence to a state matrix and vice versa

Since *MOIM* operates on bytes, it is generally endianness neutral. However, we need to specify how a byte sequence is mapped to the matrix *A*, and vice versa. This mapping is done in

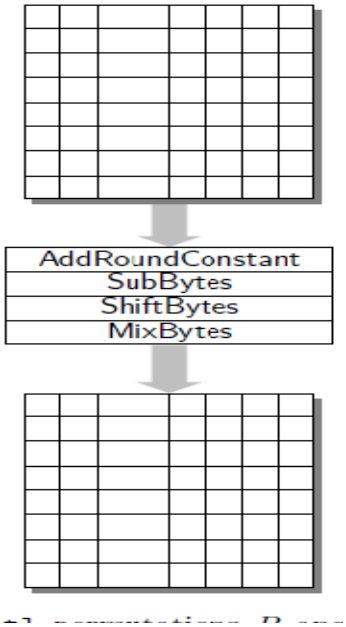

Figure 4. One round of the *MOIM* permutations for each *P* and *Q* is a composition of four basic transformations [18].

similar way as in Rijndael. Hence, the 64-byte sequence 00 01 02 ... 3f is mapped to an 8×8 matrix as [18]

$$\begin{bmatrix} 00 & 08 & 10 & 18 & 20 & 28 & 30 & 38 \\ 01 & 09 & 11 & 19 & 21 & 29 & 31 & 39 \\ 02 & 0a & 12 & 1a & 22 & 2a & 32 & 3a \\ 03 & 0b & 13 & 1b & 23 & 2b & 33 & 3b \\ 04 & 0c & 14 & 1c & 24 & 2c & 34 & 3c \\ 05 & 0d & 15 & 1d & 25 & 2d & 35 & 3d \\ 06 & 0e & 16 & 1e & 26 & 2e & 36 & 3e \\ 07 & 0f & 17 & 1f & 27 & 2f & 37 & 3f \end{bmatrix}.$$

For *MOIM*, we used 8×16 matrix, this method is extended in the natural way. Mapping from a matrix to a byte sequence is simply the reverse operation. From now on, we do not explicitly mention this mapping [18].

3.5.2 AddRoundConstant

The AddRoundConstant transformation adds a round-dependent constant to the state matrix *A*. By addition we mean exclusive-or (XOR). To be precise, the AddRoundConstant transformation in round *i* (starting from zero) updates the state *A* as

$$A \leftarrow A\ C[i];$$

where *C[i]* is the round constant used in round *i*. *P* and *Q* have different round constants. The round constants for P_{1024} and Q_{1024} are [18]

$$P_{1024}: C[i] = \begin{bmatrix} 00 \oplus i & 10 \oplus i & 20 \oplus i & 30 \oplus i & 40 \oplus i & 50 \oplus i & 60 \oplus i & 70 \oplus i & \cdots & f0 \oplus i \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & \cdots & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & \cdots & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & \cdots & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & \cdots & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & \cdots & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & \cdots & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & \cdots & 00 \end{bmatrix}$$

and

$$Q_{1024}: C[i] = \begin{bmatrix} ff & ff & ff & ff & ff & ff & ff & ff & \cdots & ff \\ ff & ff & ff & ff & ff & ff & ff & ff & \cdots & ff \\ ff & ff & ff & ff & ff & ff & ff & ff & \cdots & ff \\ ff & ff & ff & ff & ff & ff & ff & ff & \cdots & ff \\ ff & ff & ff & ff & ff & ff & ff & ff & \cdots & ff \\ ff & ff & ff & ff & ff & ff & ff & ff & \cdots & ff \\ ff & ff & ff & ff & ff & ff & ff & ff & \cdots & ff \\ ff \oplus i & ef \oplus i & df \oplus i & cf \oplus i & bf \oplus i & af \oplus i & 9f \oplus i & 8f \oplus i & \cdots & 0f \oplus i \end{bmatrix}$$

where *i* is the round number viewed as an 8-bit value.

3.5.3 SubBytes

The SubBytes transformation substitutes each byte in the state matrix by another value, taken from the s-box S. This s-box is the same as the one used in Rijndael as Figure 5 shown. Hence, if $a_{i,j}$ is the element in row *i* and column *j* of *A*, then SubBytes performs the following transformation [18]:

$$a_{i,j} \leftarrow S(a_{i,j}),\ 0 \le i < 8,\ 0 \le j < v.$$

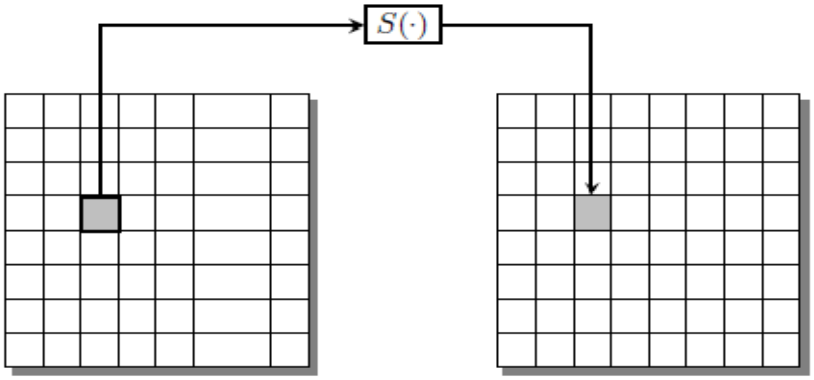

Figure 5. SubBytes substitutes each byte of the state by its image under the s-box S [18].

3.5.4 ShiftBytesWide

ShiftBytesWide cyclically shift the bytes within a row to the left by a number of positions. Let [σ_0, σ_1,..., σ_7] be a list of distinct integers in the range from *0* to *v*-1. Then, ShiftBytesWide moves all bytes in row *i* of the state matrix σ_i positions to the left, wrapping around as necessary. For ShiftBytes in *P* and *Q*, we use σ=[0,1,2,3,4,5,6,11] and σ=[1,3,5,11,0,2,4,6] respectively as illustrated in Figure 6 [18].

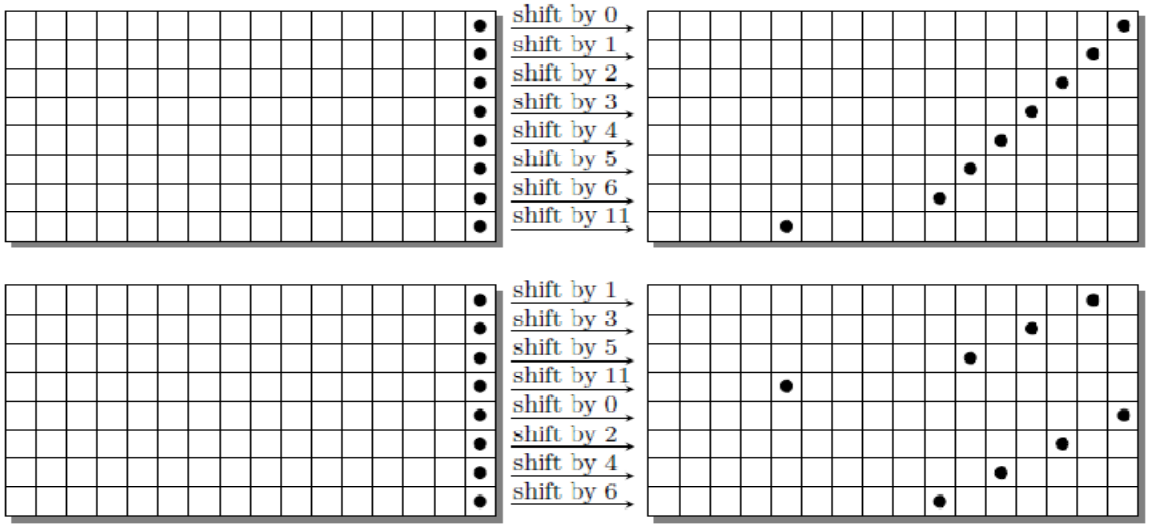

Figure 6. The ShiftBytesWide transformation of permutation P_{1024} (top) and Q_{1024} (bottom) [18].

3.5.5 MixBytes

In the MixBytes transformation, each column in the matrix is transformed independently. To describe this transformation we first need to introduce the finite field F_{256}. This finite field is defined in the same way as in Rijndael via the irreducible polynomial $x^8 \oplus x^4 \oplus x^3 \oplus x \oplus 1$ over F_2. The bytes of the state matrix *A* can be seen as elements of F_{256}, i.e., as polynomials of degree at most 7 with coefficient in {0,1}. The least significant bit of each byte determines the coefficient of x^0, etc [18].

MixBytes multiplies each column of A by a constant 8×8 matrix B in F_{256}. Hence, the transformation on the whole matrix A can be written as the matrix multiplication

$$A \leftarrow B \times A.$$

The matrix B is specified as [18]

$$B = \begin{bmatrix} 02 & 02 & 03 & 04 & 05 & 03 & 05 & 07 \\ 07 & 02 & 02 & 03 & 04 & 05 & 03 & 05 \\ 05 & 07 & 02 & 02 & 03 & 04 & 05 & 03 \\ 03 & 05 & 07 & 02 & 02 & 03 & 04 & 05 \\ 05 & 03 & 05 & 07 & 02 & 02 & 03 & 04 \\ 04 & 05 & 03 & 05 & 07 & 02 & 02 & 03 \\ 03 & 04 & 05 & 03 & 05 & 07 & 02 & 02 \\ 02 & 03 & 04 & 05 & 03 & 05 & 07 & 02 \end{bmatrix}.$$

This matrix is circulant, which means that each row is equal to the row above rotated right by one position. In short, we may write B = circ(02, 02, 03, 04, 05, 03, 05, 07) instead. See also Figure 7 [18].

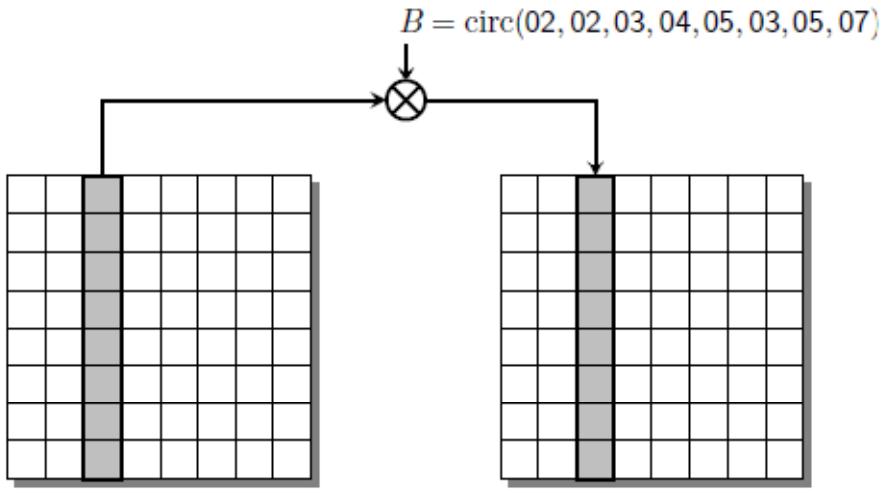

Figure 7. The MixBytes transformation left-multiplies each column of the state matrix treated as a column vector over F_{256} by a circulant matrix B [18].

3.5.6 Number of rounds

Table 4.1 shows the recommended value for the two permutations P_{1024} and Q_{1024}.

Table 1. Recommended value of number of round for *MOIM*

Permutations	Digest Size	Recommended value of r
P_{1024} and Q_{1024}	512	14

3.6 Initial values

The initial value iv_n of *MOIM*-n is the l-bit which is *1024-bits* long.

3.7 Padding

As mentioned earlier, the length of each message block is *l*. To be able to operate on inputs of varying length, a padding function pad is defined. This padding function takes a string x of length N bits and returns a padded string x* = pad(x) of a length which is a multiple of *l*.

The padding function does the following. First, it appends the bit '1' to x. Then, it appends w = -N-65mod l '0' bits, and finally, it appends a 64-bit representation of $(N + w + 65)/l$. This number is an integer due to the choice of w, and it represents the number of message blocks in the final, padded message.

Since it must be possible to encode the number of message blocks in the padded message within 64 bits, the maximum message length is 65 bits short of 2^{64}-1 message blocks. So, the maximum message length in bits is therefore is $1024 \cdot (2^{64}-1) - 65 = 2^{74}-1089$ [18].

3.8 Summary

This section summarizes the complete process of *MOIM* keyed hash function. First, a message M which is to be digested by *MOIM* is padded using the padding function pad. Then, the message M is sent into two parallel FWP construction functions. In one side of the two FWPs, the hash function iterates a compression function $C : \{0, 1\}^l \times \{0, 1\}^l \rightarrow \{0, 1\}^l$ (where *l=1024 -bits)*, which is based on one permutation *P*. The last compression function *C* is called *final* which truncates the output MAC_1 from *1024-bits* to *512-bits*. On the other side of *MOIM* keyed hash function, the same process done as the first half of *MOIM* to produce MAC_2 with different permutation *Q*. Finally, the outputs of MAC_1 and MAC_2 are concatenated together to form MAC_{final} which is *1024-bits* long.

4. Design decisions and design features

In this section, we explain the design decisions made for *MOIM* and some design features of the *MOIM*. Some advantages of *MOIM* keyed hash function compared to other hash functions are listed below.

• FWP construction used in *MOIM* is proved to be indifferentiable from a random oracle model [10]. Also, the compression functions used in *MOIM* is provably collision resistant and preimage resistant assuming that the permutations *P* and *Q* are ideal [23].

• Flexibility of *MOIM* is due to the fact that the algorithm can be implemented in different application. The parameters used in *MOIM*, i.e. security parameters r, the number of rounds can be adjusted to fit application's purposes.

• Familiarity of *MOIM* due to the familiarity of Rijndael design. Rijndael design proved its advantageous through cryptographers analysis. Since, *MOIM* compression function is inspired from Grøstl which is originally inspired by Rijndael design, consequently, *MOIM* gained the familiarity feature from Rijndael design.

4.1 The security of the construction

Generally, the estimation of the security level of hash functions can be measured with respect to the standard properties such as collision resistance and (second) preimage resistance. Also, this estimation includes the indifferentiability from the random oracle or the random sponge. The compression functions used in *MOIM* is provably collision resistant and (second) preimage resistant assuming that the permutations *P* and *Q* are ideal [23]. The

security proof states that at least $2^{l/4}$ evaluations of P and/or Q are required to find a collision for the hash function that iterates C, and that at least $2^{l/2}$ evaluations are required to find a preimage. Note that these levels are the square root of the security levels for an ideal compression function. However, since $l \geq n/2$ internal collision and preimage attacks on the hash functions have complexities of at least $2^{n/2}$ and $2n$. This analysis assumes that the l output bits of the last call to C are the final output of the hash function [18]. However, in *MOIM*, an output transformation is applied. We discuss this output transformation in Section 4.8.

The *MOIM* construction (FWP) was also proved to be indifferentiable from random oracle up to bound to 2n/3 bits (up to an additive constant in one side of *MOIM*) [10]. This result states that when permutations P and Q are assumed ideal and independent from each other, *MOIM* behaves like a random oracle up to O(2n/3+2n/3) queries.

4.2 Mixing between FWP_1 and FWP_2 constructions

MOIM mixes the intermediate chaining values between the two parallel FWP constructions as Figure 2 shows. After obtaining the fifth intermediate chaining value in each side, *MOIM* exchange these intermediate chaining values with each other. This means, the fifth intermediate chaining value obtained by FWP_1 compression function is exchanged with the fifth intermediate chaining value obtained by FWP_2 compression function. The idea behind mixing (or exchanging) process is to thwart Joux attack. More particularly, Joux stated that finding multicollisions, i.e. r-tuples of messages that all hash to the same value, is not much harder than finding ordinary collisions, i.e. pairs of collisions, even for extremely large values of r [6]. To illustrate the idea of *MOIM* mixing process, we assume the exchange of the two intermediate chaining values occurs after the fifth output of the compression function in each side of *MOIM*. At this point, Joux attack in not applicable since we cannot obtain multicollisions more than five consecutive messages due to the mixing process performed between the two parallel FWP constructions. This is the first defense line of *MOIM* against Joux attack, where another defense line for the same attack is presented in section 4.3.

4.3 Concatenation of MAC_1 and MAC_2

In the past few years, hash functions designers does not prefer to design concatenated hash functions due to Joux attack presented in Crypto 2004 [6]. Basically, Joux stated that multicollisions attack in one of hash functions of a concatenated hash function can be extended to a collision on the overall design, i.e. FIIG is not really secure than F or G itself. *MOIM* design is based on concatenating two parallel FWP constructions. FWP construction adopts the idea of widening the size of the internal state, i.e. $2n$, of hash functions and truncates the final output with output transformation function, i.e. n as stated in section 3.3. Accordingly, the design of FWP thwarts Joux attack [10]. So, this is the second defense line of *MOIM* against Joux attack. However, the final output of *MOIM* is obtained by concatenating two independent outputs of the two parallel FWP constructions which is *1024-bits* long. In cryptography folklore, the longer hash digests the more secure hash function. But, unfortunately, there is a price to pay for this security. The tradeoff between security and efficiency is the most important issue in the design of cryptographic algorithms. Cryptographic algorithms should be suitable to implement in a variety of platforms and have reasonable performance with an adequate security margin. Most of functions designed that way need longer digests to achieve the desired level of security. With *MOIM* design, we

preferred to increase the security margin by having a longer hash digest, at the same time, we used two constructions of the fast wide pipe (FWP) to balance the gap between the performance and security margin. As Figure 2 shows that the two MACs values of MAC_1 and MAC_2 in each side of the two parallel FWP hash constructions are concatenated together to form MAC_1II MAC_2. MAC_1II MAC_2 are *1024-bits* long which imply that MAC_1 and MAC_2 are *512-bits* long each.

4.4 AddRoundConstant

The purpose of adding round constants is to make each round different and at the same time this provides a natural opportunity to make *P* and *Q* independent from each other [18]. Hence, *MOIM* uses each of these permutations in each side of the design to achieve the independency of the two parallel FWP constructions. In addition, by having different round constants for AddRoundConstant in *P* and *Q*, the internal differential attack, which considers differences between the permutations *P* and *Q*, can be made infeasible [18].

4.5 SubBytes

The SubBytes transformation is the only non-linear transformation in *MOIM*. It uses the same s-box as used in Rijndael. For a walk-through of its properties, we refer to one of [18, 21, 22]. The choice for this particular transformation was driven by the following reasoning:

- Size: 8-bit s-boxes are a convenient trade-off between implementation aspects (smallest word size on popular platforms) and cryptanalytic considerations. On the other hand, there are 2^8! different permutations to choose from [18].
- Single s-box rather than many different s-boxes: this is again a trade-off between implementation and cryptanalytic considerations [18].
- No random s-box: A structured s-box allows for significantly more efficient hardware implementation than random s-box [18].
- The particular structure of the chosen s-box was already proposed in 1993 [24] and has therefore undergone a long period of study [18].
- Since the s-box is inherited from the AES, implementation aspects (especially in hardware) are well studied [18].

4.6 ShiftBytesWide

The ShiftBytesWide used in *MOIM* is the same used in Grøstl hash function. To illustare the idea of this transformation, we needed shift values which result in optimal diffusion. Let $v_{t,c}(a_{i,j})$ be the number of times that a state byte $a_{i,j}$ affects every state byte of column *c* after *t* rounds. In detail, $v_{t,c}(a_{i,j})$ defines how often (or in how many ways) every state byte of column *c* depends on $a_{i,j}$. Hence, we have full diffusion after *t* rounds if $v_{t,c}(a_{i,j}) \geq$ 1for all columns *c* and state bytes $a_{i,j}$. In other words, each state byte is affected by every state byte $a_{i,j}$ at least once. Let t^* be the value of *t* for which this happens. Then we get optimal diffusion, if $\min(v_{t^*,c}(a_{i,j}))$ is maximal for a specific geometry [18]. Second, to make *P* and *Q* more independent form each other, they use different shift values in *P* and *Q*. In more detail, we use shift values in *Q* such that no row is shifted by the same amount as in *P*, and such that the resulting states in *P* and *Q* are not simply shifted versions of each other. This way, it becomes much more differences or any other pattern in *P* and *Q* may line-up or cancel each other. We achieve this property using shift values in *Q* with a different (halved or doubled) slope than in *P*. For P_{1024} and Q_{1024} (ShiftBytesWide) we have searched for shift values with

optimal diffusion after three rounds (two rounds is not possible) and get optimal diffusion if $\min(v_{3,c}(a_{i,j})) = 2$. For P_{1024}, we have chosen the first set of such values when sorted in lexicographical order. Again for Q_{1024}, we used the same shift values as in P_{1024} in a different order to get optimal independence (see Figure 6) [18].

4.7 MixBytes

The main design goal of the MixBytes transformation is to follow the wide trail strategy. Hence, the MixBytes transformation is based on an error-correcting code with the MDS (maximum distance separable) property. This ensures that both the differential and linear branch number is 9. In other words, a difference in $k > 0$ bytes of a column will result in a difference of at least 9-*k* bytes after one MixBytes application [18].

4.8 The output transformation (*final)*

The output transformation is the last compression function denoted with *final* which truncates the large size of the chaining variables, i.e. *1024-bits*, to the required output size, i.e. *512*-bits. Inside the *final* compression function, let φ be the operation that discards all but the trailing *n* bits. The output transformation MAC_1 illustrated in Figure 3 is then defined by

$$MAC_1 = trunc_n P$$

On the other side of *MOIM,* the same aforementioned process will occur in parallel fashion to produce MAC_2 except of using *Q* permutation on this half of *MOIM* where we used *P* permutation on the first half. The output transformation MAC_2 illustrated in Figure 3 is then defined by

$$MAC_2 = trunc_n Q$$

4.9 Number of rounds

The choice of the number of rounds is primarily based on the cryptanalysis results described in section 6. These results were originally obtained from the cryptanalysis of *P* and *Q* permutations of Grøstl hash function. The square/integral attack indicates that the permutations might be distinguishable from ideal if the number of rounds is 9 or less in the *MOIM* keyed hash function. The final choice of number of rounds to be 14 provides a reasonably large security margin for *MOIM* keyed hash function.

5. ALTERNATIVE DESCRIPTIONS OF *MOIM*

This section provides alternative descriptions of *MOIM* which serve multiple purposes. It helps cryptanalysists to analysis *MOIM* to lead for better implementation. In the standard description of *MOIM*, the hash function iterates a permutation-based compression function *P* and *Q* in two parallel FWP constructions, and then applies an output transformation in each side to form the final hash of a message MAC_1II MAC_2. However, in next sections, we describe *MOIM* in different manner.

5.1 The mixing process of FWP_1 and FWP_2

Mixing (or exchanging) the two output chaining variables after every *five* outputs of the two parallel FWPs construction increases the avalanche effect property. For example, if a one single bit changes in the input that will change the output significantly, i.e. half the output bits flip. As explained in section 4.2, also the mixing process thwarts Joux attack since an adversary cannot generate a multicollisions attack more than *five* consecutive messages.

5.2 The output transformations

The output transformations are defined as $MAC_1 = trunc_n\ P \oplus h_{i-2}$ and $MAC_2 = trunc_n\ Q \oplus h_{i-2}$. The truncation process performed in *MOIM* clearly protects against length extension attack as described in section 6.7. Basically, this attack is based on the observation of the truncation from l to n bits. Since at least n bits are dropped in each side of *MOIM*, the probability of correctly guessing those bits is about $2^{-n}+2^{-n}$. The alternative description can also be seen as an indication that *MOIM* is in fact an instance of the FWP construction, which prevents length extension attacks [10]. Indeed, we can strictly state that the output transformations of *MOIM* improves the security of the hash function.

6. CRYPTANALYSIS NOTATIONS

In this section, we describe some cryptanalysis notations on *MOIM*, and we state our security claims.

6.1 Attacks exploiting properties of the permutations

We first consider well known attack methods that aim to exploit potential weaknesses in the permutations P and Q.

6.1.1 Differential cryptanalysis

The permutations P and Q have diffusion properties according to the wide trail design strategy. Since the MixBytes transformation has branch number 9, and ShiftWideBytes is diffusion optimal (moves the bytes in each column to sixteen different columns), it is guaranteed that for *MOIM* in each side, there are at least $9^2 = 81$ active s-boxes in any four-round differential trail [22, Theorem 9.5.1]. Hence, there are at least $3 \cdot 81 = 243$ active s-boxes in any twelve-round differential trail. This, combined with the maximum difference propagation probability of the s-box of 2^{-6}, means that the probabilities of any differential trail (assuming independent rounds) over twelve rounds (for either P or Q) are expected to be at most $2^{-6.243} = 2^{-1458}$ [18]. Therefore, in a classical differential attack where one specifies a differential trail for every round for both P and Q, there is only a very small chance that this would lead to a successful attack for *MOIM* [18].

6.1.2 Rebound attacks

The rebound attack [23, 24] is a new attack method for the cryptanalysis of hash functions. It gives the best known results for a number of AES-based hash functions and many SHA-3 candidates [25, 26, 27, 28, 29, 30, 31]. In general, the rebound attack works with any differential or truncated differential. However, the diffusion properties of AES based hash functions allow a very simple construction of good truncated differential paths, which facilitates the analysis. The rebound attack is most successful if a high number of degrees of freedom is available. Therefore, attacks on hash functions with a key schedule to the underlying block cipher or other sources of freedom are more likely to succeed (see the attacks on ECHO [32], LANE [33] or Whirlpool [34]). However, *MOIM* has been designed to limit the degrees of freedom available to an attacker. Moreover, in attacks on the hash function, much fewer degrees of freedom are available (compared to an attack on the compression function or permutation). The best attack on the hash function for *MOIM* is for 3 rounds 14 [18].

6.1.3 Linear cryptanalysis

Linear and differential trails propagate in a very similar way. Since the MixBytes transformation has linear branch number 9, it is guaranteed that for *MOIM* in each side, there are at least $9^2 = 81$ active s-boxes in any four-round linear trail [22, Theorem 9.5.1]. Hence, there are at least $3 \cdot 81 = 243$ active s-boxes in any twelve-round linear trail. Since the s-box has maximum correlation of 2^{-3}, the maximum correlation for any four-round linear trial is $2^{-3.81} = 2^{-243}$. This means that the correlation of any linear trail over twelve rounds (for either *P* or *Q*) is expected to be at most 2^{-279} [18].

6.2 Collision attacks on the compression function

This attack has complexity $2^{l/3}$, and hence is faster than a birthday attack on the compression function. Note that this is still above the proven bound of $2^{l/4}$ and above the complexity of a birthday attack on the hash function, since $n \leq l/2$. The attack does not provide the attacker with much control over the chaining input, and hence we do not see any methods to extend the attack to the full hash function [18].

6.3 Collision attacks on the hash function

The construction of Figure 1 is provably collision resistant up to the level of $2^{l/4}$ permutation calls. Still, no collision attack of this complexity is known when the permutations are assumed to be ideal. The best known collision attack requires $2^{3l/8}$ permutation calls, but the true complexity in terms of compression function call equivalents is higher than $2^{l/2}$. Hence, a large security margin remains [18].

6.4 Generic attacks on the iteration

The internal state being at least twice the size of the hash value for *MOIM*, generic attacks applying to the Merkle-Damgård construction cannot be applied to *MOIM* directly via brute force or birthday attacks. However, since the construction used for *MOIM* does not achieve security comparable to an ideal iterated hash function with the same internal state size, we do not claim that generic attacks do not apply using some other methods than the standard brute force and birthday attacks [18].

6.5 Multicollision attack

Joux stated that finding multicollisions, i.e. *r*-tuples of messages that all hash to the same value, is not much harder than finding ordinary collisions, i.e. pairs of collisions, even for extremely large values of *r* [6]. Joux attack is not applicable in *MOIM* since we cannot obtain multicollisions more than *five* consecutive messages due to the mixing process performed between the two parallel FWP constructions. This is the first defense line of *MOIM* against Joux attack. Also, FWP construction adopts the idea of widening the size of the internal state, i.e. *2n*, of hash functions and truncates the final output with output transformation function, i.e. *n* as stated in section 3.3. Accordingly, the design of FWP thwarts Joux attack.

6.6 Second preimage attack

The second preimage attack of Kelsey and Schneier [35] on the Merkle-Damgård construction also seems to be complicated by the large internal state size. For an *n-bit* iterated hash function based on an *n-bit* compression function, given a first preimage of length 2^k message blocks this attack finds a second preimage of the same length in 2^{n-k} evaluations of the compression function. A variant of this attack was published in [36]. Using the techniques of [35, 36], the complexity of carrying out the second preimage attack on *MOIM* given a 2^{64}-

block first preimage is about 2^{l-64}. For all the message digest sizes of *MOIM*, this complexity is well above 2^{n-k}. Hence, our claimed security level for the second preimage resistance is at least 2^{n-k} for any first message of at most 2^k blocks. However, we do not know of an attack with complexity below 2^n [18].

6.7 Length extension attack

The length extension attack on Merkle-Damgård hash functions works as follows. Let (M, M^*) be a collision for the hash function H, with $|M| = |M^*|$. H pads M and M^* to $\bar{M}$ and $\bar{M}^*$ before hashing, and by choosing any message suffix y, we have that $B = \bar{M}\text{II}y$ and $B = \bar{M}\text{II}y$ also collide. Hence, a single collision gives rise to many new collisions that "come for free". The length extension method is not trivial to carry out in *MOIM*, unless the messages collide before the output transformation. Finding a collision before the output transformation takes time $2^{l/2} \geq 2^n$ by the birthday attack. A related weakness of the Merkle-Damgård transformation is the following. Assume the two values $H(M)$ and $|M|$ are known, but M itself is not. Knowing $|M|$, one also knows how M was padded, and hence for any suffix y, one may compute $H(\bar{M}\text{II}y)$, where $\bar{M}$ is the padded version of M, without knowing M. This weakness leads to attacks when Merkle-Damgård hash function underlies a secret prefix MAC. In *MOIM*, this attack does not seem possible due to the output transformation [18].

7. CONCLUSION

This paper gives a proposal for new concrete novel design based on permutation hash function named *MOIM*. *MOIM* is based on concatenating two FWP constructions and the permutations P and Q used in SHA-3 finalist Grøstl hash function. We claimed that it is hard to attack *MOIM* with complexities significantly less than brute force. *MOIM* has two defense level of security to thwarts generic and Joux attacks. We leave the software implementation of *MOIM* as a future work.

8. REFERENCES

1. R.L.Rivest.TheMD4messagedigestalgorithm.InS.Vanstone,editor,AdvancesinCryptology - CRYPTO'90, Lecture Notes in Computer Science 537, pages 303–311. Springer Verlag, 1991.
2. R.L. Rivest. The MD5 message-digest algorithm. Request for Comments (RFC) 1321, Internet Activities Board, Internet Privacy Task Force, April 1992.
3. NIST. Secure hash standard. FIPS 180-1, US Department of Commerce, Washington D.C., April 1995.
4. H. Dobbertin, A. Bosselaers, and B. Preneel. RIPEMD-160: A strenghened version of RIPEMD. In Gollmann D., editor, Fast Software Encryption, Third International Workshop, Cambridge, UK, February 1996, Lecture Notes in Computer Science 1039, pages 71–82. Springer Verlag, 1996.
5. D. R. Stinson. Cryptography : Theory and Practice, Second Edition, CRC Press, Inc.
6. Antoine Joux. Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions. In Matt Franklin, editor, Advances in Cryptology- CRYPTO 2004, volume 3152 of Lecture Notes in Computer Science, pages 306–316. Springer, August 15–19 2004.

7. Wang, Xiaoyun, Hongbo Yu, and Yiqun Lisa Yin. "Efficient collision search attacks on SHA-0." *Advances in Cryptology–CRYPTO 2005*. Springer Berlin Heidelberg, 2005.
8. XiaoyunWang, Yiqun Lisa Yin, and Hongbo Yu, 2005."Finding Collisions in the Full SHA-1, Lecture Notes in Computer Science, Volume 3621, Advances in Cryptology – CRYPTO 2005 Proceedings, pp. 17–36.
9. Lucks, S. (2004). Design principles for iterated hash functions, Cryptology ePrint Archive, Report 2004/253, 2004, http://eprint. iacr. org.
10. Nandi, M. and S. Paul (2010). "Speeding up the wide-pipe: Secure and fast hashing." Progress in Cryptology-INDOCRYPT 2010: 144-162.
11. Eli Biham and Orr Dunkelman, "A Framework for Iterative Hash Functions - HAIFA," Cryptology ePrint Archive, 2007. [Online]. http://eprint.iacr.org/2007/278
12. Guido Bertoni, Joan Daemon, Michael Peeters, and Gilles Van Assche, "Sponge Functions," in *ECRYPT Hash Function Workshop,* 2007.
13. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. On the indifferentiability of the sponge construction. *Advances in Cryptology - EUROCRYPT 2008, 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, April 13-17, 2008. Proceedings,* 4965:181–197, 2008.
14. J.-P. Aumasson and W. Meier. Zero-sum distinguishers for reduced Keccak-f and for the core functions of Luffa and Hamsi. *NIST mailing list,* 2009.
15. C. Boura, A. Canteaut, and C. D. Canniére. Higher-order differential properties of Keccak and Luffa. Cryptology ePrint Archive, Report 2010/589, 2010. http://eprint.iacr.org/2010/589.pdf.
16. Guo, Jian, Thomas Peyrin, and Axel Poschmann. "The PHOTON family of lightweight hash functions." *Advances in Cryptology–CRYPTO 2011*. Springer Berlin Heidelberg, 2011. 222-239.
17. Aumasson, Jean-Philippe, et al. "Quark: A lightweight hash." *Cryptographic Hardware and Embedded Systems, CHES 2010*. Springer Berlin Heidelberg, 2010. 1-15.
18. Gauravaram, P., Knudsen, L. R., Matusiewicz, K., Mendel, F., Rechberger, C., Schläffer, M., & Thomsen, S. S. (2008). Grøstl–a SHA-3 candidate. *Submission to NIST*.
19. De Canniere, C., Sato, H., & Watanabe, D. (2009). Hash function Luffa: specification. *Submission to NIST (Round 2)*.
20. Bertoni, G., Daemen, J., Peeters, M., & Van Assche, G. (2009). Keccak sponge function family main document. *Submission to NIST (Round 2)*, *3*.
21. J. Daemen and V. Rijmen. AES Proposal: Rijndael. AES Algorithm Submis- sion, September 1999. Available: http://csrc.nist.gov/archive/aes/rijndael/ Rijndael-ammended.pdf (2011/01/15).
22. J. Daemen and V. Rijmen. The Design of Rijndael. Springer, 2002.
23. P.-A. Fouque, J. Stern, and S. Zimmer. Cryptanalysis of Tweaked Versions of SMASH and Reparation. In R. Avanzi, L. Keliher, and F. Sica, editors, Selected Areas in Cryptography 2008, Proceedings, volume 5381 of Lecture Notes in Computer Science, pages 136–150. Springer, 2009.
24. K. Nyberg. Differentially uniform mappings for cryptography. In T. Helleseth, editor, Advances in Cryptology – EUROCRYPT '93, volume 765 of Lecture Notes in Computer Science, pages 55–64. Springer, 1994.
25. M. Lamberger, F. Mendel, C. Rechberger, V. Rijmen, and M. Schl↵er. The Rebound Attack and Subspace Distinguishers: Application to Whirlpool. Cryptology ePrint Archive, Report 2010/198, 2010. http://eprint.iacr.org/2010/198(2011/01/15).

26. F. Mendel, C. Rechberger, M. Schl □a↵er, and S. S. Thomsen. The Rebound Attack: Cryptanalysis of Reduced Whirlpool and Grøstl. In O. Dunkelman, editor, Fast Software Encryption 2009, Proceedings, volume 5665 of Lecture Notes in Computer Science, pages 260–276. Springer, 2009.
27. D. Khovratovich, I. Nikolic, and C. Rechberger. Rotational Rebound Attacks on Reduced Skein. In M. Abe, editor, Advances in Cryptology – ASIACRYPT 2010, Proceedings, volume 6477 of Lecture Notes in Computer Science, pages 1–19. Springer, 2010.
28. K. Matusiewicz, M. Naya-Plasencia, I. Nikoli´c, Y. Sasaki, and M. Schl □a↵er. Rebound Attack on the Full LANE Compression Function. In M. Matsui, editor, Advances in Cryptology – ASIACRYPT 2009, Proceedings, volume 5912 of Lecture Notes in Computer Science, pages 106–125. Springer, 2009.
29. F. Mendel, T. Peyrin, C. Rechberger, and M. Schl □a↵er. Improved Cryptanalysis of the Reduced Grøstl Compression Function, ECHO Permutation and AES Block Cipher. In M. J. Jacobson, V. Rijmen, and R. Safavi-Naini, editors, Selected Areas in Cryptography 2009, Proceedings, volume 5867 of Lecture Notes in Computer Science, pages 16–35. Springer, 2009.
30. M. Schla □↵er. Improved Collisions for Reduced ECHO-256. Cryptology ePrint Archive, Report 2010/588, 2010. http://eprint.iacr.org/.
31. K. Matusiewicz, M. Naya-Plasencia, I. Nikoli´c, Y. Sasaki, and M. Schl □a↵er. Rebound Attack on the Full LANE Compression Function. In M. Matsui, editor, Advances in Cryptology – ASIACRYPT 2009, Proceedings, volume 5912 of Lecture Notes in Computer Science, pages 106–125. Springer, 2009.
32. J. Kelsey and B. Schneier. Second Preimages on n-Bit Hash Functions for Much Less than 2n Work. In R. Cramer, editor, Advances in Cryptology – EUROCRYPT 2005, Proceedings, volume 3494 of Lecture Notes in Computer Science, pages 474–490. Springer, 2005.
33. E. Andreeva, C. Bouillaguet, P.-A. Fouque, J. J. Hoch, J. Kelsey, A. Shamir, and S. Zimmer. Second Preimage Attacks on Dithered Hash Functions. In N. Smart, editor, Advances in Cryptology – EUROCRYPT 2008, Proceedings, volume 4965 of Lecture Notes in Computer Science, pages 270–288. Springer, 2008.
34. Imad Fakhri Al-shaikhli, Mohammad A. Alahmad and Khansaa Munther. The "Comparison and analysis study of sha-3 finallists." International Conference on Advanced Computer Science Applications and Technologies(26-28 Nov 2012): 7.
35. Mohammad A. Ahmad, I. F. A. S., Hanady Mohammad Ahmad (2012). "Protection of the Texts Using Base64 and MD5." JACSTR Vol 2, No 1 (2012)(1): 12.
36. Imad F. Alshaikhli, M. A. Ahmad. (2011). "Security Threats of Finger Print Biometric in Network System Environment." Advanced Computer Science and Technology Research **1**(1): 15.
37. Mohammad A. Ahmad, I. F. A. S., Hanady Mohammad Ahmad (2012). "Protection of the Texts Using Base64 and MD5." JACSTR **Vol 2, No 1 (2012)**(1): 12.
38. Mahmoud M. Maqableh, "Secure Hash Functions Based on Chaotic Maps for E-Commerce Applications", International Journal of Information Technology and Management Information Systems (IJITMIS), Volume 1, Issue 1, 2010, pp. 12 - 22, ISSN Print: 0976 – 6405, ISSN Online: 0976 – 6413.
39. Varun Shukla and Abhishek Choubey, "A Comparative Analysis of the Possible Attacks on Rsa Cryptosystem", International Journal of Electronics and Communication Engineering &Technology (IJECET), Volume 3, Issue 1, 2012, pp. 92 - 97, ISSN Print: 0976- 6464, ISSN Online: 0976 –6472