UDC 004.056.55
Survey Paper

Merkle-Damgård Construction Method and Alternatives: A Review

Harshvardhan Tiwari *tiwari.harshvardhan@gmail.com*
Centre for Incubation, Innovation, Research and Consultancy (CIIRC)
Jyothy Institute of Technology, Bangalore, Karnataka, India

Abstract

Cryptographic hash function is an important cryptographic tool in the field of information security. Design of most widely used hash functions such as MD5 and SHA-1 is based on the iterations of compression function by Merkle-Damgård construction method with constant initialization vector. Merkle-Damgård construction showed that the security of hash function depends on the security of the compression function. Several attacks on Merkle-Damgård construction based hash functions motivated researchers to propose different cryptographic constructions to enhance the security of hash functions against the differential and generic attacks. Cryptographic community had been looking for replacements for these weak hash functions and they have proposed new hash functions based on different variants of Merkle-Damgård construction. As a result of an open competition NIST announced Keccak as a SHA-3 standard. This paper provides a review of cryptographic hash function, its security requirements and different design methods of compression function.

Keywords: Cryptographic hash function, Information security, Merkle-Damgård construction, MD5, SHA-1, Differential attacks, Generic attacks.

1. Introduction

Cryptographic hash function is a one-way and compression function that converts an arbitrary length message to a fixed length hash value. This hash value of a message is also known as the fingerprint of the message. Any small change or modification in the input data causes the drastic change in the hash value. Cryptographic hash function is widely used in security applications and protocols [1]. Hash functions are targeted heavily by cryptanalysts as they are a fundamental building block for many security applications. Cryptographic hash function ensures the integrity and authentication in the communication. There are various applications of cryptographic hash function such as pseudo-random string generation, digital signature and MAC.

The basic operation of cryptographic hash function has been shown in Figure 1. A cryptographic hash function is like a deterministic and computationally efficient random function. Cryptographic hash function has to satisfy requirements of onewayness and collision resistance. Onewayness means that the method to calculate a hash value from a given message is easy, but it is computationally infeasible to generate any message that yields a given hash value. Collision resistance means it is extremely difficult to find two messages that have the same hash value. Cryptographic hash functions are classified into unkeyed hash functions and keyed hash functions. Unkeyed hash functions, also known as modification detection codes (MDCs), use message as a single input whereas keyed hash functions, also known as message authentication codes (MACs), can be viewed as hash functions which take two functionally distinct inputs, a message of arbitrary finite length and a fixed length secret key.

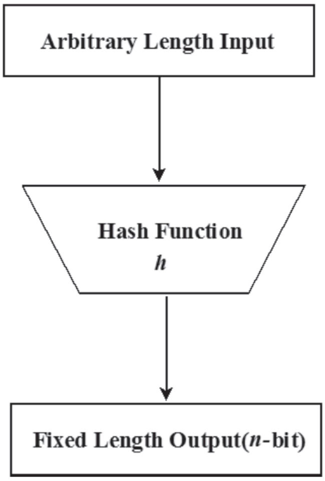

Figure 1. Cryptographic hash function

Formally, a hash function can be shown as: $h:\{0,1\}^* \rightarrow \{0,1\}^n$. It presents that an input is arbitrary length of any binary string, and the output is n bits of binary string. We usually call n as the size of hash value. A hash function must satisfy the following properties;

Compression: h maps an input M of an arbitrary bit length (up to a predefined very long maximum length) to an output of a fixed bit length n.

Ease of computation: for a given input M and a hash function h, the process of computing $h(M)$ should be easy and fast.

(1) Preimage resistance: it is computationally infeasible to find any input which hashes to any pre-specified output i.e., given a hash value H, it is computationally infeasible to find an input M such that $h(M) = H$.

(2) Second preimage resistance: it is computationally infeasible to find any second input which has the same output as any specified input. That is, given an input M, it is computationally infeasible to find another input M' such that $h(M) = h(M')$ and $M \neq M'$. This is also known as weak collision resistance.

(3) Collision resistance: it is computationally infeasible to find two different inputs with the same output. That is, it is computationally infeasible to find a

pair of inputs, M and M', such that $h(M) = h(M')$ and $M \neq M'$. This is also known as strong collision resistance.

(4) Near-collision resistance: it is computationally difficult to find any two different inputs M and M' , that have a low Hamming weight between their hash values, i.e., $h(M)$ differs from $h(M')$by a few number of bits.

(5) Partial preimage resistance: given a hash value, it is computationally difficult to recover any part of the message.

(6) Non-correlation: the input bits of an input M should not be correlated to the output bits of $h(M)$.

(7) Random behaviour: hash function should have random behaviour. That is, given a particular input M it should be infeasible to predict any output bits of $h(M)$ without actually applying the function h.

(8) Deterministic nature: hash function h should be deterministic, i.e. given a particular input M, the function always computes the same output $h(M)$.

Properties preimage, second preimage and collision resistance are ground properties of a hash function. These are NIST core requirements for a cryptographic hash algorithm and are the requirements which are generally of most practical importance. It is always important to achieve the first five properties as much as possible. Preimage resistance is important in some authentication scenarios and password storage where one does not send plain messages with their hash values, so if adversary can reverse the hash function he/she will be able to find the original message. Second preimage is for preventing the adversary from changing the original message in a way that the hash value remains unchanged. Collision resistance is stronger notion than preimage and second preimage resistance. Collision resistance always implies property second preimage resistance but does not imply preimage resistance. Collision resistance is easy to breach, so most cryptanalysis target collision attack. Collision resistance is important for digital signatures. The properties of second preimage resistance and collision resistance may seem similar but the difference is that in the case of second preimage resistance, the attacker is given a message to start with, but for collision resistance no message is given; it is simply up to the attacker to find any two messages that yield the same hash value. The term computationally infeasible or computationally difficult means that the complexity of an algorithm to break any of these properties is not less than that of the generic attack required to break that property.

For a n-bit hash function, we have a generic collision attack with complexity $2^{n/2}$, while brute force preimage or second preimage attacks have complexity 2^{n}. In case of collision attack, birthday attack is popularly used exhaustive search. The term computational easiness might mean polynomial time and space; or more practically, within a certain number of machine operations or time units [2]. Unkeyed hash function is further classified into oneway hash function (OWHF) and collision resistant hash function (CRHF). A hash function that satisfies first four properties mentioned above is termed as an oneway hash function (OWHF). A hash function that satisfies the first five properties mentioned above is sometime called a collision resistant hash function (CRHF). The construction of CRHF is hard than

OWHF. CRHF usually deals with longer length hash values. Other then these functions universal one way hash function (UOWHF) also exist [1], [13].

Paper discussed basic hash function design and MD construction in section 2. Section 3 gives iterative processing of the messages by different alternative hash construction methods. In this section weaknesses and security of compression function have also been highlighted. Paper is concluded in section 4.

2. Basic hash function design

Iterated hash functions have been the most successful method for constructing fast and secure hash functions. Usually, hash functions are built upon two components: a compression function and a domain extension algorithm.

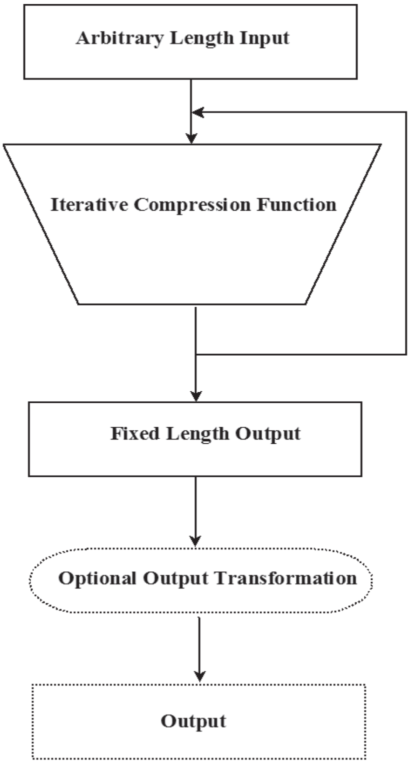
Arbitrary Length Input
Iterative Compression Function
Fixed Length Output
Optional Output Transformation
Output

Figure 2. Iterative hash construction

The compression function has the same security requirements that a hash function but takes fixed length inputs. The domain extension algorithm defines how to use the compression function in order to handle arbitrary length inputs. Almost all hash functions are iterative processes which hash inputs of arbitrary length by processing successive fixed-size blocks of input. In this section, we will discuss some popular known iterative hashing constructions. Common iterative structure is shown in the Figure 2.

2.1. Merkle-Damgård Construction

From the early beginning of hash functions in cryptography, designers relied on the Merkle-Damgård (abbreviated to MD) construction. The MD construction was discovered by Merkle [3] and Damgård [4] in 1989 independently. Majority of famous hash functions such as MD4 [5], MD5 [6], SHA-0 [7], SHA-1 [8], RIPEMD-160 [9] etc., follow the iterative MD method. A compression function which takes a fixed input length value and outputs a fixed length hash value is core component of this construction.

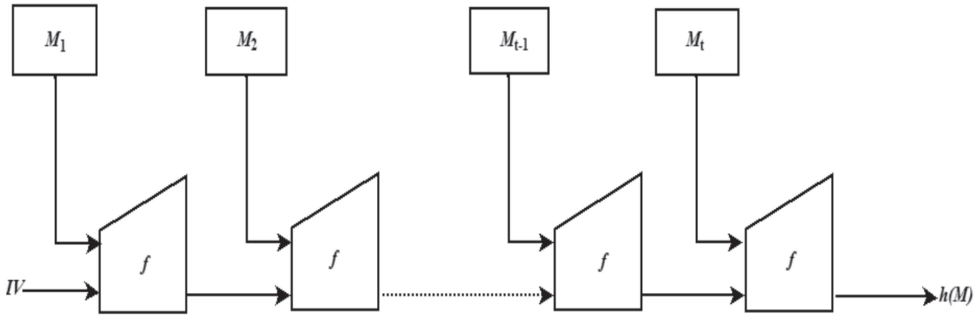

Figure 3. Merkle-Damgård construction

A compression function accepts two inputs: a chaining variable and a block of message. A compression function accepts two inputs: a chaining variable and a block of message. Let $f:\{0,1\}^b \times \{0,1\}^n \rightarrow \{0,1\}^n$ be a compression function which takes a *b*-bit message block and an *n*-bit chaining value. Let $h:\{0,1\}^* \rightarrow \{0,1\}^n$ be a MD construction built by iterating the compression function *f* in order to process a message of arbitrary length. A message *M* to be processed using *h* is always padded in a manner such that the length of the padded message is a multiple of the block length *b* of *f*. Bit-length *b* corresponds to input length of desired compression function *f*. The padding is done by adding after the last bit of the last message block a single 1-bit followed by the necessary number of 0-bits. Let $|M|$ be abinary representation of the length of the message *M*. The binary encoding of the message length is also be added to complete the padding. This is called a Merkle-Damgård strengthening.Then input *M* subsequently divided into *t* blocks, each of bit-length *b*. The hash function *h* can then be described as follows:

$$\left.\begin{aligned} &H_0 = IV, \\ &H_i = f(H_{i-1}, M_i), i = 1 \ldots t, \\ &h(M) = H_t \end{aligned}\right\} \qquad \ldots(1)$$

Where *f* is the compression function of *h*, $H_i$is the intermediate chaining variable between stage *i*-1 and stage *i*, and H_0 is a pre-defined starting value or the initial value *IV*. The block diagram of the iterative hash function using the compression function is shown in the Figure 3. The computation of the hash value is dependent on the chaining variable. At the start of hashing, this chaining variable has a fixed initial value which is specified as part of the algorithm.

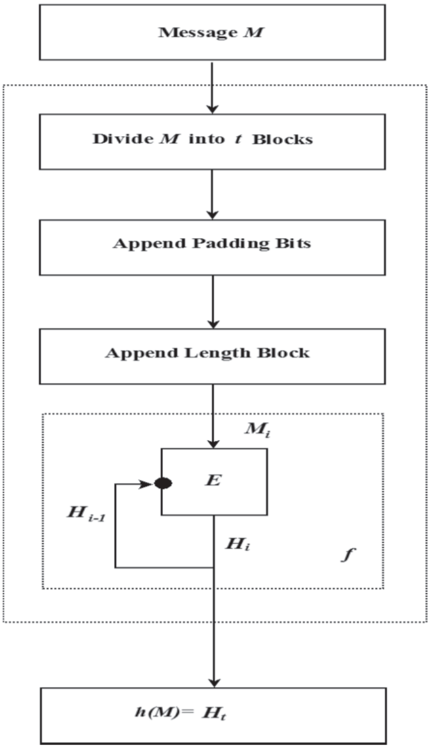

Figure 4. Detailed view of Merkle-Damgård construction

This process continues recursively, with the chaining variable being updated under the action of different part of the message, until the entire message has been used. The final value of the chaining variable is then output as the hash value corresponding to that message. One of its distinctive features is that it promotes the collision resistance and preimage resistance of the compression function to the full hash function: for instance, a collision on the compression function can be deduced efficiently from a collision on the full hash function. The inclusion of the length at the end of the message is important for this situation, and is also important for preventing a number of attacks, including long-message attacks.

Merkle-Damgård construction proves that the security of hash function relies on the security of the compression function. Thus, in order to build a collision resistant hash function, it is sufficient to design a collision resistant compression function. Recent results, however, highlight some intrinsic limitations of the MD approach [27]. This includes being vulnerable to multicollision attacks [10], long second preimages attacks [11], and herding attack [12]. Figure 4 shows detailed view of MD construction.

3. Alternative construction methods

3.1. Tree Construction

This is the most parallelizable class of constructions and is mainly suited for multi core platforms where multiple processors can independently operate on different parts of the message simultaneously.

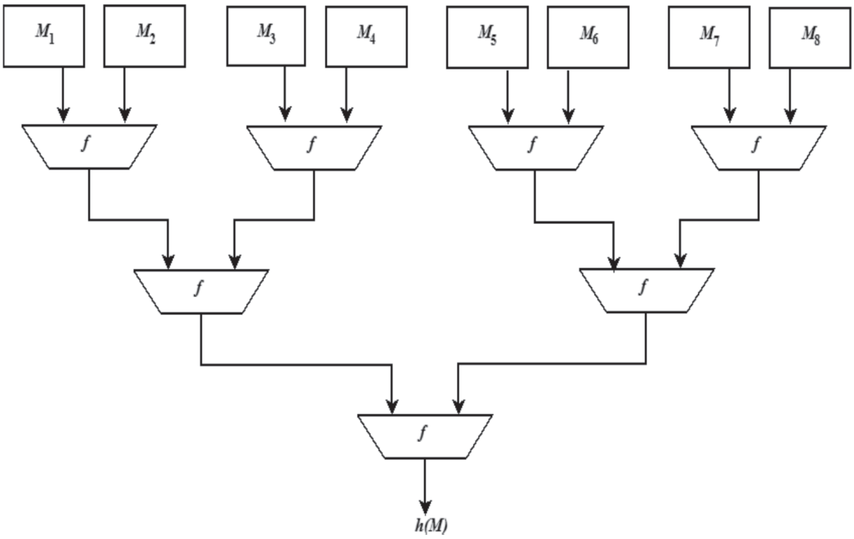

Figure 5.Tree construction

The compression function of tree construction is of the form $f:\{0,1\}^{2n} \rightarrow \{0,1\}^{n}$. However, the Damgård tree construction is not practical since the size of the binary tree grows with the length of the message. Figure 5 illustrates a typical tree based hashing construction. Damgård tree construction was later optimized by Sarkar and Scellenberg [14]. Sarkar and Scellenberg construction (SS construction) was a parallel version of MD construction. The main difference between SS construction and previous constructions is that authors consider the number of available processors to be fixed while the length of the message can be arbitrarily long.

Thus SS construction considered a fixed processor tree and used it to hash arbitrarily long messages. Each processor simply computes the base hash function. Similarly, Carter and Wegman [15] used tree hashing techniques to build universal hash functions. This was followed up by Naor and Yung [16] and Bellare and Rogaway [17] in the context of UOWHFs (Universal One Way Hash Functions). In [18] Bellare and Micciancio proposed the randomize-then-combine paradigm, where the message is split into blocks, each block is processed via randomizing function (derived from some standard hash function) and finally combined by an operation such as XOR. Although this structure was originally proposed to build incremental functions, it can be thought of as a 2- level tree and can still be parallelized since the randomization process of the individual blocks are independent. Tree-based constructions are slightly less popular than the iterative ones.

3.2. Sponge Construction

Sponge construction [19] is an iterative hash function construction, builds upon a fixed length transformation or permutation instead of a compression function and can generate output strings of infinite length. Sponge construction can be used to build both hash functions and stream ciphers.

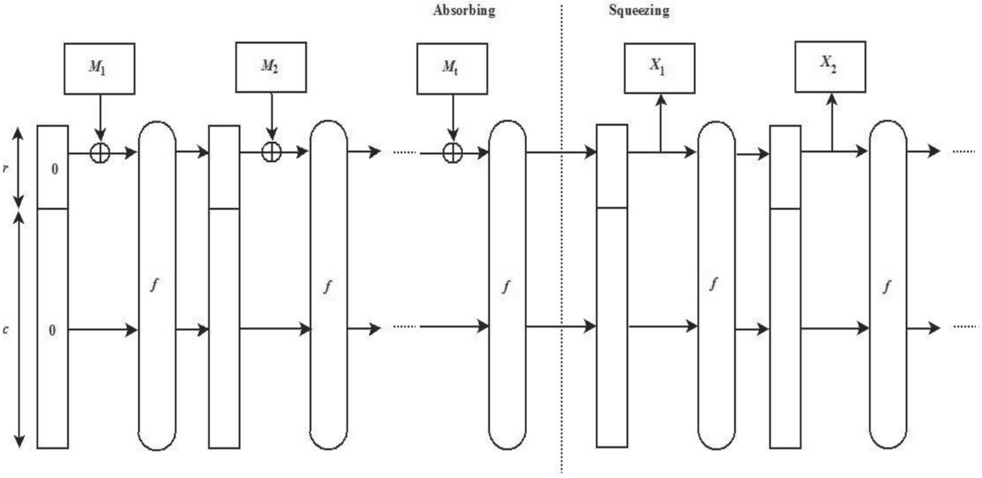

Figure 6.Sponge construction

Basically, sponge hashing proceeds in two phases, the absorbing phase and the squeezing phase. The sponge operates on a fixed length state $s=\{0,1\}^{r+c}$ composed of *r* bits (called bit-rate) and *c* bits (called capacity), through a function $f:\{0,1\}^{r+c}\rightarrow\{0,1\}^{r+c}$ which produces a transformation or permutation of *s*. In the absorbing phase, the message is divided into *r*-bit blocks (padded if necessary) and each block is XORed with the *r* part of *s* (initially, $s=\{0\}^{r+c}$), *f* then iteratively processes *s* until all blocks are exhausted. In the squeezing phase, the state continues to be transformed or permuted by *f* but this time the *r* parts of the states are returned at every iterations as output blocks. Since the sponge construction supports variable length output, the user chooses the length of the final hash value which determines how many of the returned blocks in the squeezing phase need to be returned. Optionally, between two phases, some number of blank rounds can be applied. In the blank rounds, there is no input to or output from the state. Only, *f* is applied to the state *s*. If *f* is expressed as a random function, the construction is called a T-sponge, otherwise if it is expressed as a permutation then the construction is called a P-sponge. The security of a sponge construction depends on its capacity *c*, hash size *n* and function *f*. For P-sponge construction the complexity of a collision is $Min(2^{c/2}, 2^{n/2})$ and complexity of preimage and second preimage is $Min(2^{c/2}, 2^{n})$. Collision complexity of T-sponge construction is equal to the collision complexity of P-sponge construction. Finding a preimage costs $Min(2^{c}, 2^{n})$ and finding a second preimage costs $Min(2^{c}/N, 2^{n})$ for a T-sponge, where *N* is the length of the original message. Figure 6 illustrates the sponge construction.

Hash functions such as Keccak [20] and PHOTON [21] are based on the sponge construction. Keccak has recently been selected as the winner of SHA-3 competition. Although still considered an iterative construction, the sponge is completely different from the Merkle-Damgård construction. When iterated hash functions are considered, there always exist inner collisions which can be defined as if two message pair $M_1$and M_2give the same chaining value, then concatenation of $M_1$and $M_2$with collide suffix M^*collide. In the sponge function construction, there also exist inner collisions and this is the only weaknesses of sponge functions so far.

3.3. Wide and Double Pipe Construction

Lucks has proposed a wide pipe and double pipe hash function construction [22] which provides an enhancement of the Merkle-Damgård construction. The wide pipe construction intended to increase the size of the internal state of *n*-bit hash function and *w*-bit compression function, where $w > n$. This means that the wide pipe design obtains a greater internal state than message digest length by using a larger compression function.

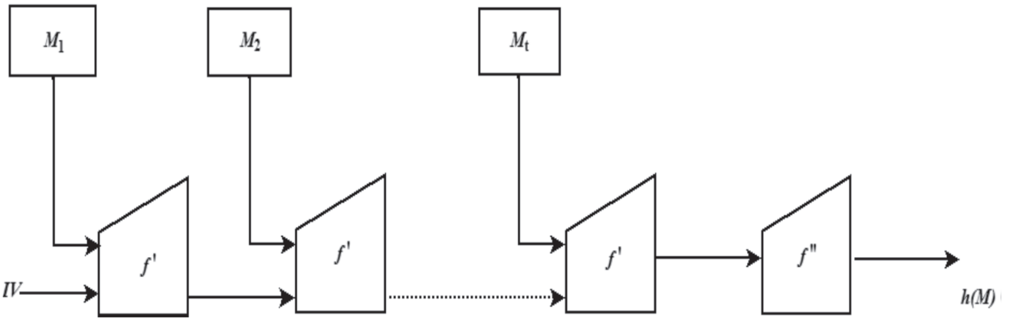

Figure 7.Wide-pipe construction

Constructing a collision-resistant compression function with $w > n$ output bits may be simpler than constructing an *n*-bit compression function with the same level of

collision resistance. Let $H_0 \in \{0,1\}^w$ be a random initial value. Using two compression functions:
$f' : \{0,1\}^w \times \{0,1\}^m \to \{0,1\}^w$ and $f'' : \{0,1\}^w \to \{0,1\}^n$. Wide pipe hash is computed as:

$$\left.\begin{aligned} H_i &= f'(H_{i-1}, M_i), i = 1 \ldots t, \\ h(M) &= f''(H_t) \end{aligned}\right\} \qquad \ldots(2)$$

Wide-pipe construction is shown in the Figure 7.

On the other hand, the double pipe design maintains twice the hash size using the $w = 2n$ compression function in parallel to process each message block. Using one compression function $f : \{0,1\}^n \times \{0,1\}^{n+m} \to \{0,1\}^n$, with $m \geq n$ and two distinct random initial values $H_0' \neq H_0'' \in \{0,1\}^n$ double pipe hash is computed as:

$$\left.\begin{aligned} H_i' &= f(H_{i-1}', H_{i-1}'' \| M_i), i = 1 \ldots t-1 \\ H_i'' &= f(H_{i-1}'', H_{i-1}' \| M_i), i = 1 \ldots t-1 \\ h(M) &= f(H_{t-1}', H_{t-1}'' \| M_t) \end{aligned}\right\} \qquad \ldots(3)$$

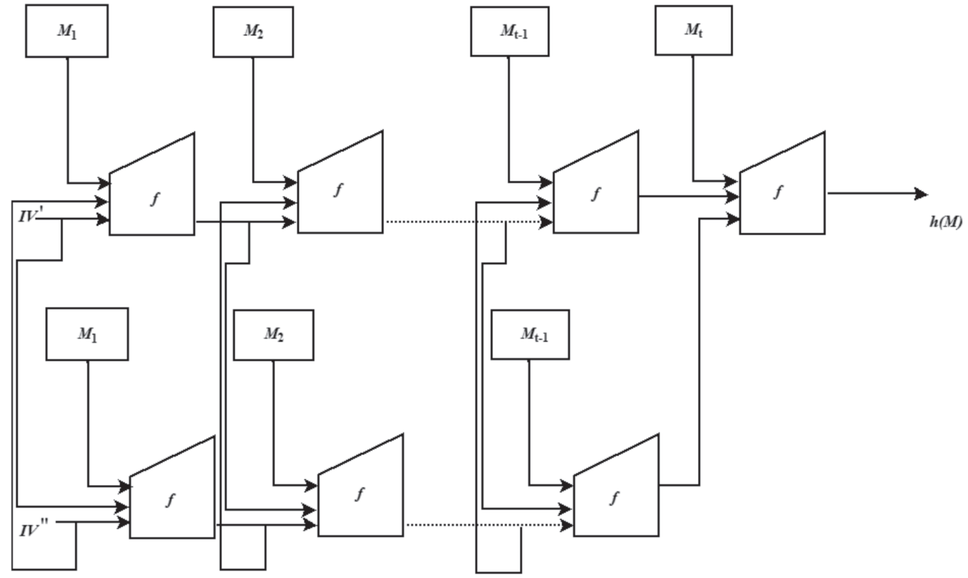

Figure 8. Double-pipe construction

Double-pipe construction is shown in the Figure 8. From these designs Lucks showed that increasing the size of the internal state (i.e. the chaining variable) to become larger than the size of the final hash value, would significantly improve the security of the hash function. This modification clearly thwarts the extension attack since in the wide and double pipe construction the final hash value is truncated, so in order to append an extension, the unknown discarded bits have to be guessed, which is clearly difficult if the number of the discarded bits is non-trivial. Furthermore, by increasing the size of the internal state, finding collisions for the compression function becomes harder, which complicates the other generic attacks. An obvious drawback of the wide and double pipe construction, however, is a degraded efficiency as the compression function now has larger input and output while keeping the hashing rate constant (the size of the compression function input corresponding to a message block is fixed) since the chaining variable input is increased. Also, adapting existing hash functions for the wide and double pipe construction may be difficult since it might be the only reasonable way to increase the internal state is to use multiple compression function calls in parallel for every iteration. Recently, Yasuda [23] adopted a slightly modified variant of the double pipe construction and proved its unforgeability beyond the birthday barrier.

3.4. 3C Construction

The 3C construction is the simplest variant of the MD construction that one can obtain to improve its security against multi block collision attack [24]. The 3C hash function processes the intermediate chaining values of the MD construction by maintaining a second internal chaining variable containing a value produced by repeatedly XORing the chaining variables while hashing a message; this variable is then processed in an extra finalisation call to the compression function. There are two chains in 3C construction: the accumulation chain and cascading chain. The accumulation chain and the compression function have an accumulator XOR function that works iteratively in the cascade chain, similarly to the MD construction. The processing in the 3C divides the message into *t*-blocks with IV_0 representing the initial value. a_i and c_i are the chaining variables in the accumulation chain and cascade chain. The compression functions are executed three times for each block: the processing data block, padding block and forming the block *Z* in the accumulation chain. The 3C is as secure as the MD construction.

The 3C hash is computed as:

$$\left.\begin{array}{l} c_0 = IV_0, \\ c_i = f\left(c_{i-1}, M_i\right), i = 1\ldots t, \\ a_1 = c_1 \\ a_i = a_{i-1} \oplus c_i, i = 2\ldots t, Z = a_t, \\ h(M) = g(Z, c_t). \end{array}\right\} \qquad \ldots(4)$$

To increase the security level of 3C, 3C+ design has been proposed. In the 3C+ hash construction, there is an additional chain called the final chain. The final chain is added to the cascade and accumulation chains of the 3C hash construction. The final compression function *g* at the last block takes the series of the result of the accumulation and final chains after padding. Due to this enhancement, the security level of 3C+ is higher than that of 3C and MD construction. 3C+ uses extra memory, but makes finding multi-block collisions more difficult. However, the 3C and 3C+ structures are slower because the processors have to process data sequentially, where every block takes its input from the previous block, causing sequential delay. However, in [25], it was shown that both 3C and 3C+ are indeed susceptible for multi-block attack. The designers of 3C claimed that while it is susceptible to the multi-collision attack, it resists the long messages 2nd pre-image and herding attacks. However, it was shown in [26] that 3C is also indeed susceptible to both the second preimage and herding attacks.

3.5. The Prefix Free, Chop Constructions, NMAC and HMAC Constructions

Coron et al. proposed, prefix-free, NMAC, and HMAC constructions as secure variants for the MD construction [28]. Later it is found that even though these constructions are indifferentiable from RO, they are not collision resistant. The prefix-free construction does not modify the Merkle-Damgård construction, instead it modifies the padding algorithm to make sure that the message is prefix free. One way to do this is by prepending or appending the length of the whole message to every message block. PFMD construction uses a padding function *g* which ensures that for any two messages M, M' with $M \neq M'$, $g(M)$ cannot be a prefix of $g(M')$. Let *N* is the length of the message *M*. Three variants of PFMD are:

Varient1: PF_{g1}

$$\left.\begin{array}{l} y_0 = IV, \\ M = (M_1, \ldots, M_t), |M_i| = b \\ g_1(M) = (N \| M_1 \| \ldots \| M_t), \\ y = f(y_0, g_1(M)) \end{array}\right\} \quad \ldots(5)$$

Varient2: PF_{g2}

$$\left.\begin{array}{l} y_0 = IV, \\ M = (M_1, \ldots, M_t), |M_i| = b - 1 \\ g_2(M) = ((0 \| M_1) \| (0 \| M_2) \| \ldots \| (0 \| M_{t-1}) \| (1 \| M_t)), \\ y = f(y_0, g_2(M)) \end{array}\right\} \quad \ldots(6)$$

Varient3: PF_{g3}

$$\left.\begin{array}{l} y_0 = IV, \\ M = (M_1,\ldots,M_t), |M_i| = b \\ y_i = f(y_{i-1}, M_i, N, i)), i = 1,\ldots,t \end{array}\right\} \quad \ldots(7)$$

The Chop construction is an *n* bit MD hash where *r* out of *n* bits of the hash value are chopped, thus producing an *(n−r)*-bit hash value. The chop construction basically removes a non-trivial number of bits from the final hash value. This, while it solves the indifferentiability issue, unfortunately lowers the security bounds of the hash function.

In NMAC, an independent function *g* is applied to the output of the last application of the compression function, while HMAC is a special case of the NMAC in which an extra compression function call is introduced. The HMAC hash construction hashes a message by applying the same *f* function twice, using the same *IV*. The NMAC and HMAC are computed as:

NMAC:

$$\left.\begin{array}{l} y_0 = IV, \\ y_i = f(y_{i-1}, M_i), i = 1,\ldots,t \\ y_l = g(y_t) \end{array}\right\} \quad \ldots(8)$$

HMAC:

$$\left.\begin{array}{l} M_0 = 0^b, \\ y_0 = f(IV, M_0) \\ y_i = f(y_{i-1}, M_i), i = 1,\ldots,t \\ y_l = f(y_t \| 0^{b-n}, IV) \end{array}\right\} \quad \ldots(9)$$

3.6. Linear Hash and Linear XOR Hash

The linear hash function is described by Bellare and Rogaway [28]. It accepts an additional key input in every call of the iteration. Moreover, each key is distinct and therefore LH requires number of key inputs that is a linear in the message size. It employs distinct compression functions for each message block evaluation. In the same paper another approach linear XOR (XLH) by Bellare and Rogaway [28] was discussed. In contrast to the LH hash function it adds the same number of distinct keys by XORing these with the chaining values resulting from each iteration of the Merkle-Damgård style hash function. The first key is XORed with the initialization vector IV and the final key is XORed with the final intermediate chaining value, while the final hash result is left unmodified.

3.7. Enveloped Merkle-Damgård

The enveloped Merkle-Damgård [29] (EMD) was proposed by Bellare and Ristenpart and resembles the design of HMAC [30]. EMD uses two fixed initialization vectors IV_0 and IV_1. The first vector is applied in a Merkle-Damgård style as input to the first compression function. The second IV_0 is provided as input to the final compression function together with the chaining variable and the final input message bits and this step is known as the “enveloping” step of the construction. Bellare and Ristenpart showed that EMD preserves collision resistance, indifferentiability from random oracle and indistinguishability from Pseudorandom Function (PRF).

$$\left.\begin{array}{l} y_0 = IV_0, \\ y_i = f(y_{i-1}, M_i), i = 1, \ldots, t-1 \\ y_t = f(y_{t-1} \| M_t, IV_1) \end{array}\right\} \qquad \ldots(10)$$

3.8. Merkle-Damgård with permutation

The Merkle-Damgård with permutation, due to Hirose et al. is a simple variant of the original Merkle-Damgård design [31]. The only difference with the Merkle-Damgård construction is that a permutation is applied before the processing of the last message block. The permutation masks the internal Merkle-Damgård style processing, similarly to the idea of EMD, and MDP is proven indifferentiable from a random oracle when the underlying compression function is an ideal function.

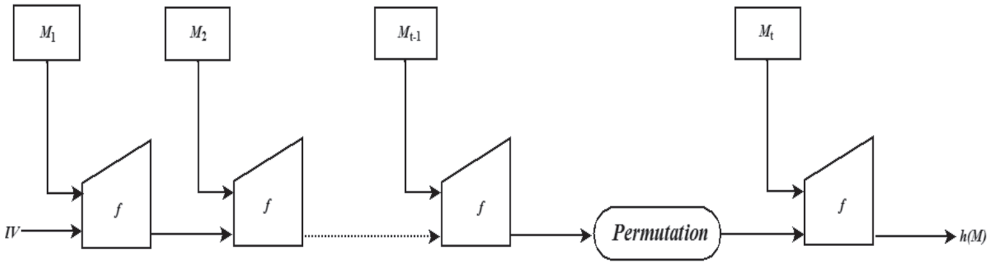

Figure 9. Merkle-Damgård with permutation (MDP) construction

MDP construction is shown in the Figure 9. The authors proved that the collision resistance of MDP follows trivially from the collision resistance of the Merkle-Damgård construction as the former introduces minimal changes to the latter. The authors also discussed the security of possible simple MAC constructions based on MDP. However, although with such a simple modification, the authors succeeded in proving a significant security gain, MDP seems to be able to thwart only the extension attack, but not other Merkle-Damgård generic attacks. Also, recently it was shown that MDP is neither pre-image nor second preimage resistant [31].

$$\left.\begin{aligned} & y_0 = IV_0, \\ & y_i = f(y_{i-1}, M_i), i = 1, \ldots, t-1 \\ & y_t = f(\Pi(y_{t-1}), M_t) \end{aligned}\right\} \qquad \ldots(11)$$

3.9. Zipper Hash Construction

Zipper hash construction was developed by Liskov [32] that makes an ideal hash function from weak ideal compression function. Zipper hash structure was developed as a strengthen structure against multicolission attack. Let $f:\{0,1\}^b \times \{0,1\}^n \to \{0,1\}^n$ and $g:\{0,1\}^b \times \{0,1\}^n \to \{0,1\}^n$ becompression functions. On input message *M* the following procedure is executed:

$$\left.\begin{aligned} & H_0 = IV, \\ & H_i = f(H_{i-1}, M_i), i = 1, \ldots, t \\ & H_0^{'} = H_t \\ & H_i^{'} = g(H_{i-1}^{'}, M_{t-i+1}), i = 1, \ldots, t \end{aligned}\right\} \qquad \ldots(12)$$

3.10. RMX Construction

Randomized hashing was proposed by Halevi and Krawczyk [33]. It is somewhat different from other typical variants of Merkle-Damgård, instead it is a generic fix that can be applied on any construction. The RMX transform is in its essence a message modification technique. It prepends a random string s to the message as a first message block to be processed and then the same random string is XORed with each message block. The idea is to randomize the message inputs by XORing a salt input into the message.

RMX was proposed as a general transform that is particularly well-suited for digital signature applications of hash functions, where a message *M* is first randomised with a salt *s* to produce a randomized message M'. A digital signature *sign* is then generated from M'. The original message *M*, the salt *s* and the signature *sign* are then sent to the verifier. When the verifier receives these parameters, it first randomises *M* with *s* to produce M' and carries out standard signature verification using M' and *sign*. It aims the provision of security guarantees even when the compression function is compromised with respect to collision security. It was formally showed that just finding collisions on the compression function is not sufficient in order to break the resultant signatures: instead, the attacker needs to solve a much harder cryptanalytical problem, closer to finding second preimages. The authors claim that randomized hashing will strengthen any hash function, even the weakest ones.

$$\left.\begin{array}{l} y_0 = f(IV, s) \\ y_i = f(y_{i-1}, M_i \oplus s), i = 1, \ldots, t \\ h(M) = y_t \end{array}\right\} \quad \ldots(13)$$

3.11. Dither Hash Construction

The dither hash function by Rivest is another variant of MD construction which includes an additional counter-like input [34]. The design intension behind the dither construction is to add an iteration-dependent input to the compression function in order to defeat certain generic attacks. The additional input, called the "dithering" input, to the compression function is formed by the consecutive elements of a fixed sequence. This gives the attacker less control over the input of the compression function, and makes the hash of a message block dependent on its position in the whole message. In particular, its goal is to prevent attacks based on expandable messages.

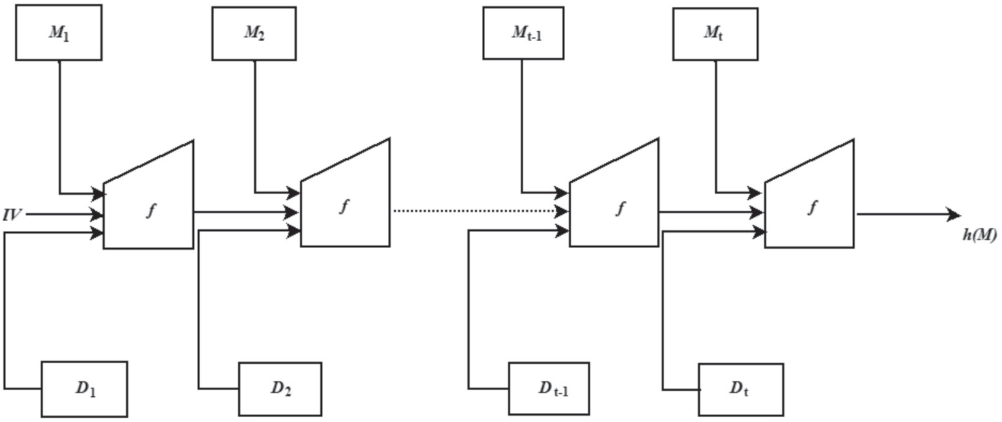

Figure 10.Dither construction

In the dither hash function, every call to the compression function f has the three inputs: the dithering sequence $D = D_1, \ldots, D_t$ which depends on the iteration, the chaining value, and the next message block from $M = M_1,\ldots,M_t$ and generates hash as follows:

$$\left.\begin{array}{l} H_0 = IV, \\ H_i = f(H_{i-1}, M_i, D_i), i = 1,\ldots,t \\ h(M) = H_t \end{array}\right\}$$

…(14)

Figure 10 shows the Dither construction. The dither value can be selected in many ways: one of the ways by following the suggestion of Kelsey and Schneier the dither value can be selected as the index, $D_i = i$. This approach is called the dithering by counter but this approach requires that compression function accept an arbitrary large input. Another suggestion for selecting the dither value can be a sequence of alternative 0's and 1's.

A pseudorandom sequence can also be used as a dither value. This provides protection against message block repetition. In his proposal Rivest suggested the use the infinite abelian square-free sequence. The abelian square-free sequence is an aperiodic sequence over a finite alphabet with the property that no sub-word is repeated. However, the method proposed for integrating the dither value into concrete hash functions is inefficient, in the sense that it increases the number of calls to the compression function. No indifferentiability result is known for the Dither hash function.

3.12. HAIFA Construction

HAsh Iterative FrAmework (HAIFA) is a modified Merkle-Damgård construction proposed by Dunkelman and Biham [35]. It preserved all the good properties of Merkle-Damgård construction. HAIFA modifies Merkle-Damgård by introducing extra input parameters to the compression function. These are: a salt value and the number of bits hashed so far, which thwarts many of the generic attacks against the plain Merkle-Damgård construction since the input to every compression function call becomes unique and highly dependent on where the compression function call is made through the hashing chain. The inclusion of a bit counter ensures the suffix and prefix properties of the design and helps to prove it indifferentiable from a random oracle In fact, HAIFA can be considered a dedicated-key hash function.

The idea of adding additional input parameters to the compression function has been previously proposed by Rivest through a process called dithering. The HAIFA is built by iterating a compression function:

$$f : \{0,1\}^n \times \left\{0,1^b\right\} \times \left\{0,1^m\right\} \times \{0,1\}^s \rightarrow \{0,1\}^n$$

…(15)

The padding in HAIFA is very similar to the padding of Merkle-Damgård construction.Moreover, the padding is done by appending a single '1' bit followed by as many '0' bit as needed to complete an *b*-bit block after the message length and the digest size are appended. Then input *M* subsequently divided into *t* blocks

$M = M_1, \ldots, M_t$, each of bit-length *b*. H_i is found by computing $f(H_{i-1}, M_i, bc, s)$ where *bc* denotes the bit counter i.e. number of hashed bits so far and *s* denotes the salt, and this operation is repeated until message blocks ends in the iteration part. There does not exist any difference between iteration method of Merkle-Damgård and HAIFA. Only difference is between the compression functions. The hash function *h* can then be described as follows:

$$\left.\begin{array}{l} H_0 = IV, \\ H_i = f(H_{i-1}, M_i, bc, s), i = 1, \ldots, t \\ h(M) = H_t \end{array}\right\} \quad \ldots(16)$$

An obvious drawback of HAIFA is efficiency degradation since the compression function now has more input parameters to process. Furthermore, HAIFA cannot be (easily) used to patch existing Merkle-Damgård based hash functions because a compression function designed for the Merkle-Damgård construction would not naturally accommodate the extra HAIFA parameter inputs. The idea is incorporated also in few SHA-3 candidates: BLAKE, ECHO and SHAvite-3.

3.13. BCM

The backwards chaining mode was proposed by Andreeva and Preneel. It uses three keys k_1, k_2, k_3 and of fixed length (*b* + 2*n*) bits, where $|k_2| = b$ and $|k_1| = |k_3| = n$ where *n* is the state and *b* is the block size. It XORs the key k_1 and the most significant *n* bits of block M_2 with the fixed initial chaining variable *IV*. The message block M_1 together with the resulting value from the XOR computation form the input to the first application of *f*. In the iteration the message block M_i and the chaining variable H_{i-1} in-line are XORed with the most significant *n* bits of the next-in-line message block M_{i+1} and form the inputs to the *i*th compression function *f*. The one but last block M_{t-1} is interpreted differently than the rest of the message blocks. Here the difference is that the least significant *n* bits of M_{t-1} are XORed with the key k_1, the chaining variable H_{t-2} is XORed with the first significant bits of k_2 and M_t. The final input to the last compression function is provided by the last message block M_t and the chaining variable H_{t-1} XORed with keys k_2 and k_3, respectively.

3.14. Nested Iteration

NI is basically a keyed variant of the Merkle-Damgård construction making use of two keys $k', k'' \in \{0,1\}^k$. Beside being unforgeable, Bellare and Ristenpart later proved in [36] that NI is also indistinguishable from PRF, indifferentiable from RO, and if strengthening was used, NI is also collision resistant.

$$\left.\begin{array}{l} H_0 = IV, \\ H_i = f(H_{i-1}, M_i, k'), i = 1, \ldots, t-1 \\ H_t = f(H_{t-1}, M_t, k'') \end{array}\right\} \quad \ldots(17)$$

3.15. Shoup Construction

In [37], Shoup proposed an elegant keyed construction. Shoup's hash function (SH) derives from the linear XOR hash function and optimizes it in terms of the number of keys. It uses logarithmic number of keys (instead of linear), following a specific sequence. In addition to the key input of the compression function, the chaining variables of every compression function iteration in SH is further XORed with a key mask. A variant of the SH construction has been proposed by Bellare and Ristenpart in [37] that makes the last compression function call a wrapping call (this last application of the compression function is called an envelope). Thus, this variant is called the Envelop Shop (ESH).

$$\left.\begin{array}{l} H_0 = IV, \\ H_i = f(H_{i-1} \oplus k^{mask}, M_i), i = 1, \ldots, t \end{array}\right\} \quad \ldots(18)$$

3.16. Chaining Shift

The Chaining Shift (CS) construction was proposed by Maurer and Sjodin in [38] as a more efficient solution than the NI construction. The CS construction was shown to be unforgeable, indistinguishable from PRF, indifferentiable from RO, and the strengthened variant of it (with strengthened padding) is collision resistant.

$$\left.\begin{array}{l} H_0 = IV_1, \\ H_i = f(H_{i-1}, M_i), i = 1, \ldots, t-1 \\ H_t = f(IV_2 \| H_{t-1}, M_t) \end{array}\right\} \quad \ldots(19)$$

4. Conclusion

A cryptographic hash function plays a vital role in many security applications and protocols such as digital signatures and authentication schemes. Among several security requirements collision resistance is an important property of a cryptographic hash function. The security of hash function depends on the collision resistance property of the underlying compression functions. Merkle-Damgård construction method failed to preserve this important security property. In recent years, it has been shown that hash functions based on weak Merkle-Damgård construction are vulnerable to different attacks. As a result researchers have given different construction methods to design the hash functions such as Haifa, Dither and Tree. This paper reviews different popular alternative construction methods to Merkle-Damgård method and also discusses how these methods have strengthened the weak MD method. The use of hash functions based on Merkle-Damgård construction in different security products, services, algorithms and protocols makes them vulnerable to different cryptanalytic attacks.

References

[1] A.J. Menezes, P.C. vanOorschot, and S.A. Vanstone, “Handbook of applied cryptography,” 1997.

[2] W. Diffie and M. Hellman, 1976, “New Directions in Cryptography,” IEEE Transaction on Information Theory, 1976, vol. 22, pp. 644-654.

[3] R. C. Merkle, “One Way Hash Functions and DES,” Crypto’89, 1989, LNCS, vol. 435, pp. 428-446.

[4] I. Damgård, “A Design Principle for Hash Functions,” Crypto’89, 1989, LNCS, vol. 435, pp. 416-427.

[5] R. Rivest, “The MD4 Message Digest Algorithm,” Request for Comments (RFC) 1320, Internet Engineering Task Force, 1992. http://www.rfceditor.org/rfc/pdfrfc/rfc1320.txt.pdf.

[6] R. Rivest, “The MD5 Message Digest Algorithm,” Request for Comments (RFC) 1321, Internet Engineering Task Force, 1992.

[7] NIST, “Secure Hash Standard (SHS),” Federal Information Processing Standards 180. 1993.

[8] NIST, “Secure Hash Standard (SHS),” Federal Information Processing Standards 180-1, 1995.

[9] B. Preneel, A. Bosselaers and H. Dobbertin, “RIPEMD-160: A Strengthened Version of RIPEMD,” FSE’96, 1996, LNCS, vol. 1039, pp. 71–82.

[10] A. Joux, “Multicollisions in Iterated Hash Functions: Application to Cascaded Constructions,” Crypto’04, 2004, LNCS, vol. 3152, pp. 306-316.

[11] J. Kelsey and B. Schneier, “Second Preimages on n-bit Hash Functions for Much Less than 2^n work,” Eurocrypt’05, 2005, LNCS, vol. 3494, pp. 474-490.

[12] J. Kelsey and T. Kohno, “Herding Hash Functions and the Nostradamus Attack,” Eurocrypt’06, 2006, LNCS, vol. 4004, pp. 183–200.

[13] X. Lai and J. Massey, "Hash functions based on block ciphers," Eurocrypt'92, 1992, vol. 92, pp. 55-70.

[14] P. Sarkar and P. Scellenberg, "A Parallel Algorithm for Extending Cryptographic Hash Functions," Indocrypt'01, 2001, LNCS, vol. 2247, pp. 40-49.

[15] J. Carter and M. Wegman, "Universal Classes of Hash Functions," Journal of computer and system sciences, 1979, vol. 18, pp. 143-154.

[16] M. Naor and M. Yung, "Universal One-way Hash Functions and their Cryptographic Applications," Proceedings of the twenty-first annual ACM symposium on Theory of computing, 1989, pp. 33-43.

[17] M. Bellare and P. Rogaway, "Collision-resistant Hashing: Towards Making UOWHF's Practical," Crypto'97, 1997, LNCS, vol. 1294, pp. 470-484.

[18] M. Bellare and D. Micciancio, "A New Paradigm for Collision-free Hashing: Incrementality at Reduced Cost," Eurocrypt'97, 1997, LNCS, vol. 1233, pp. 163-192.

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

[20] Keccak Design Team, "The Keccak Sponge Function Family," http://keccak.noekeon.org/.

[21] J. Guo, T. Peyrin and A. Poschmann, "The PHOTON Family of Lightweight Hash Functions,"Crypto'11, 2011, LNCS, vol. 6841, pp. 222-239.

[22] S. Lucks, "A Failure-Friendly Design Principle for Hash Functions," Asiacrypt'05, 2005, LNCS, vol. 3788, pp. 474-494.

[23] K. Yasuda, "A Double-Piped Mode of Operation for MACs, PRFs and PROs: Security beyond the Birthday Barrier," Eurocrypt '09, 2009, LNCS, vol. 5479, pp. 242-259.

[24] P. Gauravaram, W. Millan, E. Dawson, and K. Viswanathan, "Constructing Secure Hash Functions by Enhancing Merkle-Damgård Construction," Information Security and Privacy, 2006, LNCS, vol. 4058, pp. 407-420.

[25] D. Joscak and J. Tuma, "Multi-block Collisions in Hash Functions Based on 3C and 3C+ Enhancements of the Merkle- Damgård Construction," ICISC'06, 2006, LNCS, vol. 4296, pp. 257-266.

[26] P. Gauravaram and J. Kelsey, "Linear-XOR and Additive Checksums Dont Protect Damgård-Merkle Hashes from Generic Attacks," CT-RSA'08, 2008, LNCS, vol. 4964, pp. 36-51.

[27] J. Coron,Y. Dodis, C. Malinaud and P. Puniya, "Merkle-Damgård Revisited: How to Construct a Hash Function," Crypto'05, 2005, LNCS, vol. 3621, pp. 430-448.

[28] M. Bellare and P. Rogaway, "Collision-resistant Hashing: Towards Making UOWHF's Practical," Crypto'97, 1997, LNCS, vol. 1294, pp. 470-484.

[29] M. Bellare and T. Ristenpart, "Multi-Property-Preserving Hash Domain Extension and the EMD Transform," Asiacrypt'06, 2006, LNCS, vol. 4284, pp. 299-314.

[30] H. Krawczyk, M. Bellare and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication," RFC 2104, Feb. 1997.

[31] S. Hirose, J. H. Park and A. Yun, "A Simple Variant of the Merkle-Damgård Scheme with a Permutation," Asiacrypt'08, 2008, LNCS, vol. 4833, pp. 113-129.

[32] M. Liskov, "Constructing an Ideal Hash Function from Weak Ideal Compression Functions," Selected Areas in Cryptography, 2006, LNCS, vol. 4356, pp. 358-375.

[33] S. Halevi and H. Krawczyk, "Strengthening Digital Signatures via Randomized Hashing," Crypto'06, 2006, LNCS, vol. 4117, pp. 41-59.

[34] R. Rivest, "Abelian Square-free Dithering for Iterated Hash Functions", Ecrypt Hash Function Workshop, 2005, vol.21.

[35] E. Biham and O. Dunkelman, "A Framework for Iterative Hash Functions-HAIFA," Cryptology ePrint Archive, Report 2007/278, 2007.

[36] M. Bellare and T. Ristenpart, "Hash Functions in the Dedicated-Key Setting: Design Choices and MPP Transforms," ICALP '07, 2007, LNCS, vol. 4596, pp. 399-410.

[37] V. Shoup, "A Composition Theorem for Universal One-Way Hash Functions," Eurocrypt'00, 2000, LNCS, vol. 1807, pp. 445-452.

[38] U. Maurer and J. Sjodin, "Single-key AIL-MACs from any FIL-MAC," ICALP'05, 2005, LNCS, vol. 3580, pp. 472-484.