Revisiting Dedicated and Block Cipher based Hash Functions

Anupam Pattanayak

Computer Science & Engineering Department, Haldia Institute of Technology, Haldia, India – 721657, E-mail – anupam.pk@gmail.com

Abstract: A hash function maps a variable length input into a fixed length output. The hash functions that are used in the information security related applications are referred as cryptographic hash functions. Hash functions are being used as building blocks of many complex cryptographic mechanisms and protocols. Construction of a hash function consists of two components. First component is a *compression function* and the second component is a *domain extender*. The various hash function design philosophies try to design the compression function from different angles. Two major categories of hash functions are: *dedicated hash functions*, and *block cipher-based hash functions*. These two kinds of design philosophies have been revisited in this paper. Two dedicated has functions from MD4 family - MD4, and SHA-256 constructions have been detailed in this paper. To limit the scope of this paper in this framework, discussions on attacks on hash functions, and SHA-3 finalists have been excluded here.

Keywords: Hash Function, Merkle-Damgård Design, MD4, SHA-2.

1. Introduction

We know that an ancient approach for identifying a person uniquely is to take the left thumb impression of that person. Similarly it would be of great help if we can work with a small message which represents a much longer message uniquely. Hash function provides us this facility. A hash function maps a variable-length input into a fixed-length output. This hash function output can be treated as a *fingerprint* of the input data [20]. A very simple example of hash function is modulo operation. Hash functions have been used in many fields of computer science such as hash table in data structure, checksum algorithms for error detection, digital signature in information security etc. They all depend on the fundamental property that different input values would produce different fingerprints in most of the cases. The hash functions that are used in the information security related applications are referred as cryptographic hash functions. A cryptographic hash function h takes a message with arbitrary length as input, and deterministically maps it to a bit-string with fixed length as output. That is

$$h: \{0,1\}^* \rightarrow \{0,1\}^n \qquad 1.1$$

This output bit-string of the hash function is commonly referred as “message digest” or simply “digest”, or just “hash”.

Hash functions are being used as building blocks of many complex cryptographic mechanisms and protocols. One such usage is in digital signature. Digital signature scheme (DSS) is used for authentication of data. In general, a digital signature scheme consists of three components; key generation, signature generation, and signature verification. Secure hashes are used by all of these components in DSS. Hash function is also used in authentication protocols such as Kerberos. Kerberos offers authentication, eavesdropping prevention, and integrity of data in client-server architecture. Kerberos uses hash function to calculate the hash value of the given client password and this hash value becomes the secret key of the client. Secure communication protocols such as IPSec, SSL, or SSH also use hash functions. Internet Key Exchange (IKE) protocols in IPSec use hash functions as pseudo-random functions. The handshaking protocol in SSL uses a hash function to create a message authentication code. PGP and S/MIME also use hash function to ensure the integrity of e-mail messages.

The organization of this paper is as follows. The basic properties of a cryptographic hash functions have been introduced in section 2. Next section briefly introduces various components or building blocks of hash function design. Section 4 discusses dedicated hash functions. This includes Markle-Damgård construction, Markle-Damgård alternatives, concept of domain extenders, and detail descriptions of two hash functions in the MD4 family: MD4, and SHA-256. The block cipher based hash function designs have been discussed in section 5.

2. Properties of Cryptographic Hash Functions

The algorithmic properties of hash functions differ depending upon the usage of hash functions. To introduce the three properties a cryptographic hash function (or simply, hash function) should possess, we recollect the terms *image*, and *preimage*. Consider a function $f(x) = y$ that maps x to the image y. The x is said to be preimage of y. Now the three properties of hash functions are as in the following [21].

1. *Preimage Resistance*: Given a digest y, it is computationally infeasible to find a message x that hashes to y. That is, computational cost of finding the input x must be $\geq 2^n$, where $h(x) = y$ and $|y| = n$.

2. *Second Preimage Resistance*: Given a message x, it is computationally infeasible to find a different message x', such that both messages hash to a same digest. That is, computational cost of finding the input x' ($\neq$ x) must be $\geq 2^n$, where $h(x') = y$, $h(x) = y$, and $|y|=n$.

3. *Collision Resistance*: It is computationally infeasible to find two different messages, which hash to the same digest. That is, computational cost of finding an input pair x and x' such that $h(x) = h(x')$ must be $\geq 2^{n/2}$. Here n is the length of message digest.

The preimage resistance property can be expressed as the inability to learn about the contents of the input data from its digest. The second preimage resistance property can be interpreted as the inability to learn about the second preimage from the given first preimage such that both of these preimages have same digest. The collision resistance property signifies that the digests are almost unique for each given message. If the input message is altered, almost always the hash changes as well. The word *almost* is to be noted. Because when a function maps from a larger domain to a smaller range, collisions necessarily exist. If we properly design cryptographic hash functions with digests of sufficient length then the probability that one can obtain two different messages with identical hashes is too small to be bothered in all practical applications. These three properties - preimage resistance, second preimage resistance, and collision resistance are also known as one-way, weak collision resistance, and strong collision resistance properties respectively. If a hash function satisfies the first two properties then it is referred as *one-way hash function* (OWHF). Whereas the hash function that satisfies all the three properties referred as *collision resistant hash function* (CRHF) [12]. A hash function with an output of n bits can only offer a security level of 2^n operations for pre-image and second pre-image attacks and $2^{n/2}$ operations against finding collisions. While a security level of 128-bits is typical for main stream applications, 80-bit security is often a reasonable target for RFID tag-based applications [6].

3. Components of Hash Function

To process arbitrary long input data, hash functions are generally designed by reusing small and fixed-length input functions under some composition method. The composition method a hash functions goes through is arbitrary-length domain extender of underlying building blocks with a fixed domain size. Such building blocks are known as *compression functions*. Compression function can be either *keyed* or *keyless*. So construction of a hash function consists of two components. First component is a *compression function* that maps a fixed-length input to a fixed-length output. Second component is a *domain extender* that uses a compression function and produces a function with arbitrary-length input and fixed-length output. The design of a compression function is the key design component of hash function. The various hash function design philosophies try to build the compression functions from different angles. Although most of the existing hash functions can be described as being based on a block cipher, these block cipher based hash functions can be further classified into two categories. The first category is the *block cipher-based hash functions* that use hash functions based on an existing block cipher, particularly designed for encryption/decryption purpose such as DES, AES etc. The second category is the hash functions that use block ciphers that have been designed particularly for use in hash functions. Such hash functions are referred as *dedicated hash functions*. A point about these block ciphers, which have been designed exclusively for use in hash functions, is that they are not necessarily secure and hence may be unsuitable for exclusive encryption/decryption purposes. Another approach of constructing hash functions rely on difficulty of solving some well known computational problems. It may be pointed out here that people have used stream cipher like RC4 instead of traditional approach of using block cipher in designing a hash function instead of block ciphers [7]. Compare to block-cipher-based hashes, the stream-cipher-based hashes have smaller block size and more number of rounds.

In brief, the general framework for iterated hash function to process the padded input message $M=m_1m_2\ldots m_n$ can be described as follows:
$H_0 = IV$
$H_i = f(m_i,H_{i-1})$ for $i = 1,2,\ldots,n$
$h(x) = g(H_n)$

IV is the *initial vector* or *initial value*. The function f is called the *round function* or *compression function*. H_i is called *chaining variable*. And the result of the hash function is denoted with h(x).

The function g is called the *output transformation*. In many cases, the use of output transformation is not mentioned explicitly. In that case, g is simply the identity function. That is, $g(H_n) = H_n$. In this case the output length is equal to the length of the chaining variable. Role of an output transformation is to further reduce the length of the hash result.

4. Dedicated Hash Functions

The most adopted approach by the designers to design hash functions is to use a *domain extender* on top of a *compression function* in an *iterative* manner. Iterative structures allow for a sequential message processing. One of the first examples of an iterative hash function is the Rabin hash [28]. In 1989, Merkle [22] and Damgård [9] independently introduced the concept of systematic iterative hash construction known as the *Merkle-Damgård construction*.

4.1 Merkle-Damgård construction

The building block of the Merkle-Damgård construction is the compression function $f : \{0, 1\}^n \times \{0, 1\}^b \rightarrow \{0, 1\}^n$ that accepts input - a chaining or state variable *h* of n-bits size and a message block m of b-bits size, and produces n-bits updated chaining variable as output.

Padding Rules: The message padding mechanism appends sufficient bits to the original message to make its length a multiple of the input size of the compression function f. This padding function for the Merkle-Damgård construction is *suffix-free*. The suggested suffix-free padding functions proposed by Markle, and the one proposed independently by Damgård differ. Merkle's padding rule restricts the size of the processed message to maximum 264-bits, but this is not a problem for practical message sizes. On the other hand, adding a single bit per message block as per the padding of Damgård makes it less efficient due to the overhead of bit manipulations. This disadvantage of Damgård's padding has paved the way for Merkle's mechanism to be established as the standard padding rule for the Merkle-Damgård construction.

The Merkle-Damgård design accepts an additional input parameter, *initial value* IV. The IV is a fixed constant. This inclusion of the initialization vector and the Merkle suffix-free padding to the Merkle-Damgård iterative domain extender has been referred as the Merkle-Damgård *strengthening* by Lai and Massey [16].

Now Merkle-Damgård construction is stated as follows:

Given: (i) Compression function f: $\{0, 1\}^n \times \{0, 1\}^m \rightarrow \{0, 1\}^n$ and
(ii) n-bit constant (Initialization Vector) IV.

Input: Message M
1. Divide M into $m_1, m_2, m_3, \ldots, m_k$, each of m-bit blocks such that the last block m_k is padded with the encoding of |M|.
2. Initialize $h_0 = IV$;
3. For i = 1 to k
   Compute $h_i = f(h_{i-1}, m_i)$;

Output: Message digest of M is h_{k+1}.

So, the Markle-Damgård construction iterates the compression function f. The output of f at i^{th} round is h_i. This h_i and the next message block m_{i+1} are the input to the next $i+1^{st}$ round of f. The hash of the last block, which contains the encoding of message length, is the hash of complete message. The temporary storage of the compression function's output, h_i, is referred as chaining variable or internal state. So to design a hash function we have to:

1. *choose a collision-resistant compression function*
2. *use a padding procedure*
3. *choose a good initial vector*

The main advantage of dedicated has function constructions is their high speed and low resource consumption in the software as well as hardware implementations. This is the reason behind the popularity of this class of cryptographic hash functions. Examples include such famous functions as MD5 as well as NIST standards SHA-1 and SHA-2.

4.2 Domain Extenders

The domain extenders can be classified as either Merkle-Damgård-based domain extenders or non-Merkle-Damgård-based domain extenders. In each category there are several domain extenders. A superb discussion on domain extenders can be found in the thesis by Andreeva [1]. Two major design choices for Merkle-Damgård-based domain extenders are:

(i) Wide-Pipe or Narrow-Pipe design,
(ii) Keyed or Keyless design.

Wide-Pipe Versus Narrow-Pipe Domain Extenders: The original wide pipe construction was introduced by Lucks et al [18]. It is characterized by keeping a full large (>> *n*) internal state in the iterative Merkle-Damgård portion. As final step, a distinct output transformation is employed on this "wide" state to compress it to the desired output hash length, which is shorter than the internal state size. JH and Keccak are examples in third round SHA-3 candidates that have adopted the wide-pipe strategy.

Narrow-pipe constructions, in contrast, are designed by iterating a state as large as the output hash value. BLAKE is the example in third round SHA-3 candidates that has adopted the narrow-pipe design.

Keyed Versus Keyless Domain Extenders: Another separation of domain extenders is based on the presence or lack of an explicit key input. When the key is unique for every message, it is referred as *salt*. Keyed designs are often less efficient than keyless ones but come with more security guarantees. Many designs that have advanced in the NIST competition include them as an optional input.

4.3 A Summary of Merkle-Damgård Alternatives

Prefix-free Merkle-Damgård: The basic prefix-free Merkle-Damgård designs are narrow-pipe, keyless iterative domain extenders that apply a prefix-free padding function [8]. A padding rule is called *prefix-free*, if for any distinct $M, M_{\mathbf{0}}$, there exists no bit string X such that $pad(M_{\mathbf{0}}) = pad(M)\|X$. If the prefix-free designs are not additionally suffix-free, they do not preserve the main collision security property.

Enveloped Merkle-Damgård: The enveloped Merkle-Damgård design was proposed by Bellare and Ristenpart [3]. It is a narrow-pipe, keyless domain extender. It uses two fixed initialization vectors IV and IV_0. The first vector, IV is applied in a Merkle-Damgård style as input to the first compression function. The second vector, 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.

Merkle-Damgård with permutation:.The Merkle-Damgård with permutation construction is a narrow-pipe, keyless variant of the original Merkle-Damgård design [15]. The difference with the Merkle-Damgård construction is that a permutation is applied before the processing of the last message block.

Linear hash: The linear hash function is a narrow-pipe, keyed Merkle-Damgård iteration [4]. The only difference with the Merkle-Damgård design is that it accepts an additional key input in every call of the iteration. Moreover, each key is distinct. Notice that this approach ensures a domain separation of the underlying compression function.

Linear XOR: The linear XOR is a narrow-pipe, keyed Merkle-Damgård iteration [4]. It adds a linear number of keys by XOR-ing these with the chaining values in a Merkle-Damgård style iterative hash function. The first key is XOR-ed with the initialization vector IV and the final key is XOR-ed with the final intermediate chaining value, while the final hash result is left unmodified.

Shoup's hash: The Shoup's hash function [33] derives from the linear XOR hash function and optimizes it in terms of the number of keys. It uses logarithmic rather than linear number of keys, following a specific sequence.

HAIFA: The HAsh Iterative FrAmework (HAIFA), designed by Biham and Dunkelman, is a narrow-pipe hash function [10]. HAIFA modifies Merkle-Damgård by introducing extra input parameters to the compression function: a bit counter, and an optional salt value. The bit counter keeps track of the number of bits hashed so far. And the salt value is used as a key to create families of hash functions. Salt is set to 0 if only one hash function is required.

Sponge: The Sponge design supports variable length outputs. If the output length is fixed, the Sponge construction is roughly classified as a keyless, wide-pipe, non-strengthened Merkle-Damgård construction [11]. Sponge operates on a fixed-length state $b = \{0, 1\}^{r+c}$ through transformation or permutation function $p: \{0, 1\}^{r+c} \rightarrow \{0, 1\}^{r+c}$. Here r is the bit rate and c is the capacity of the sponge. It consists of an absorbing phase and a squeezing phase. In the absorbing phase, the (padded) message is divided into r-bit blocks and each block is XOR-ed with the r part of b (initially, b = 0^{r+c}), p then iteratively processes b until all blocks are finished. In the squeezing phase, the state continues to be transformed / permuted by p but this time the r parts of the states are returned at each iteration as output blocks. A well-known sponge construction is Keccak.

4.4 Other Domain Extenders

Several non-Merkle-Damgård alternative designs are known in the literature. Often the incentives are twofold: increasing the efficiency and/or the security guarantees.

Tree-Based Hash Functions: The *tree-based* constructions, in contrast to the Merkle-Damgård based designs, allow for parallelism. An early tree-based mode of operation was proposed by Damgård [9]. Tree constructions split the message into blocks which could be processed by independent processors or machines and the final result is combined to produce the hash value.

Few other non-Merkle-Damgård alternative designs are multi-pass domain extender and multi-pipe domain extender. A *multi-pass* domain extender processes the data in more than one pass. A *multi-pipe* domain extender allows for processing the message in multiple pipes without the need to store the message.

4.5 Examples and Description of Dedicated Hash Functions

MD4, MD5, SHA are some examples of dedicated hash function. The term “MD4 family” is used for hash functions whose design principles are influenced by MD4 up to much extent. Apart from MD4, the other members of the family are hash functions such as MD5, SHA-0, SHA-1, SHA-2 etc.

4.5.1 MD4

The famous cryptographer Rivest was motivated by the works of Merkle's and Damgård's at Crypto 1989 and proposed MD4 in next year 1990 [29]. MD4 is a very efficient hash function based on the principles by Merkle and Damgård. Cryptanalysis of MD4 revealed certain unexpected properties raising concerns about its security. Rivest then proposed the successor MD5 in 1992 [30]. MD5 is based on MD4 and shares many design ideas of MD4. Focus of MD5 is much more on security than on efficiency. In the following the MD4 has been explained in moderate detail. MD4 compresses any arbitrary bit-length message into a 128-bit hash value. MD4 consists of the following five steps.

Step 1. Append Padding Bits

The input message is padded so that its bit-length is congruent to 448, modulo 512. That is, the message is extended so that it is exactly 64 bits short of being a multiple of 512 bits long. A bit "1" is appended to the message, and then "0" bits are appended so that the length in bits of the padded message becomes congruent to 448, modulo 512. At least one bit and at most 512 bits are appended. Padding operation is to be done always, even if the length of the message is already congruent to 448, modulo 512.

Step 2. Append Length

A 64-bit representation of input message is appended next to make the resultant message exactly multiple of 512-bits. If the length of message is greater than 64-bits then only the low-order 64 bits of input message length are used.

Step 3. Initialize MD Buffer

The initial value is IV = 67452301 EFCDAB89 98BADCFE 10325476 in hexadecimal. The four 32-bit state registers, A, B, C, and D are initialized from the IV as follows:

$A = 67452301_H$, $B = EFCDAB89_H$, $C = 0x98BADCFE_H$, and $D = 10325476_H$.

Step 4. Process Message in 16-word Blocks

Each message block of 512-bits is processed by a compression function. The compression function consists of three rounds, each of which has sixteen steps. The 512-bit message block is broken up in sixteen words of 32-bits and exactly one of these words is used in every step. In each round, a separate ordering of message words is used. Each of i^{th} round uses different nonlinear Boolean function F_i defined as follows:

$F_0(X, Y, Z) = (X \wedge Y) \vee (\neg X \wedge Z)$
$F_1(X, Y, Z) = (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z)$
$F_2(X, Y, Z) = X \oplus Y \oplus Z$
Here X, Y, Z are 32-bit words.

Let m_i be the i^{th} message word, $0 \leq \cdot\, i \leq 15$. Then, in round 0, the message words appear in the order $m_0, m_1, \ldots, m_{15}$. In round 1, the message words appear in the order m_0, m_4, m_8, m_{12}, m_1, m_5, m_9, m_{13}, m_2, m_6, m_{10}, m_{14}, m_3, m_7, m_{11}, m_{15}. In round 2, the message words appear in the order m_0, m_8, m_4, m_{12}, m_2, m_{10}, m_6, m_{14}, m_1, m_9, m_5, m_{13}, m_3, m_{11}, m_7, m_{15}. The expanded message word W_i serves as input for the i^{th} step in the step operation part.

A rotation operation is the circular shift of the bits in a word. The notation $x^{<<<n}$ is used to represent the operation left-rotation of x by n bit positions. The value n varies depending on the step number and round number. The value of n is 3, 7, 11, 19, … (recurring four times) in round-0, and 0, 3, 5, 9, 13, … (recurring four times) in round-1, and 3, 9, 11, 15, … (recurring four times) in round-2.

The step update function modifies the four registers A, B, C, and D into A', B', C', and D' as follows:

$B' \leftarrow (A + F_i(B, C, D) + W + k_i)^{<<<n} \equiv 2^{32}$
$C' \leftarrow B$,
$D' \leftarrow C$, and
$A' \leftarrow D$.

After a 512-bit message block is compressed, the variables A, B, C, and D are updated as follows:
$A \leftarrow A + A'$,
$B \leftarrow B + B'$,
$C \leftarrow C + C'$, and
$D \leftarrow D + D'$.
And then remaining 512-bits message blocks are processed similarly.

Step 5. Output
The message digest of the input message is generated by concatenating all the 32-bit register values of A, B, C, and D after we process all the input message blocks. The concatenation is to be done from low-order byte of A, followed by B, C, and D in the same manner.

4.5.2 Diving deep into the MD4

The four 32-bit state registers, A, B, C, and D were initialized from the IV as specified earlier. These registers are updated through 48 steps (separated into three rounds, each of 16 steps). The 512-bit message is expanded into 48 words, each of 32 bits length. Each word is updated by one of the 48 steps. After these 48 steps, the registers are added (modulo 2^{32}) to the four words of the chaining input, and the sum is returned as the output of the compression function. Each step updates only one of the registers via a step update function.

The step update function accepts the four registers (A, B, C, and D), a message word W, and a 32-bit constant k (which changes in every round) as input, and produces a single 32-bit word as output.
$B' \leftarrow (A + F_i(B, C, D) + W + k_i)^{<<<n} \equiv 2^{32}$
$C' \leftarrow B$,
$D' \leftarrow C$, and
$A' \leftarrow D$.

The step update function changes little-bit for each step. It consists of additions modulo 2^{32}, a rotation, and a Boolean function which changes in each round:
$F_0(X, Y, Z) = (X \wedge Y) \vee (\neg X \wedge Z)$
$F_1(X, Y, Z) = (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z)$
$F_2(X, Y, Z) = X \oplus Y \oplus Z$

In each bit position F_0 acts as a conditional: if X then Y else Z. That is why F_0 is also written as IF() function sometimes. In each bit position F_1 acts as a majority function: if at least two of X, Y, Z are on, then F_1 has a "1" bit in that bit position, else F_1 has a "0" bit. This the reason why F_1 is also mentioned as MAJ() function sometimes. It is interesting to note that if the bits of X, Y, and Z are independent and unbiased, the each bit of F_0 (X, Y, Z) will be independent and unbiased, and similarly each bit of F_1 (X, Y, Z) will be independent and unbiased. The function F_2 is the bit-wise XOR or parity function.

The message expansion actually describes how the values W_i which serve as inputs for the step operations are computed from the current input message block X_j. In MD4, the message expansion is done by some round-wise permutations. It is simply a permutation of the 16 message words of 32 bits each. The expanded message word W_i serves as input for the i^{th} step in the step operation part. The permutations are chosen in such a way that there are as few patterns as possible, e.g. there are no two words X_i, X_j which are applied in two consecutive steps more than once. This is done to provide a good diffusion.

The 32-bit constants k_i used are distinct for each round i. They are defined as follows:
$k_0 = 0$,
$k_1 = 5A827999_H$, and
$k_2 = 6ED9EBA1_H$.
The constant k_1 and k_2 represent the square roots of 2 and 3 respectively.

This step operation represents the core component of every compression function. It should be quite easy to implement and very efficient, but at the same time they must provide good diffusion and be difficult to analyze. To be efficient there are two main aspects in the step operations of the MD4: Only one of the registers changed in each step, and the step operations are built of very and simple basic operations (bitwise Boolean operations, modular additions, bit shifts, and rotations). These operations have been chosen, because on the one hand they can be computed very efficiently on modern computer architectures, and on the other hand the mixing of Boolean functions and addition is believed to be cryptographically strong.

4.5.3 SHA

NIST published the *Secure Hash Standard* in 1993. The hash function underlying the standard was named the *Secure Hash Algorithm* (SHA). At present, it is commonly referred to as SHA-0 [24]. SHA-0 has been developed following the same principles as MD4. SHA-0 is a 160-bit hash function, with five registers in the state: A, B, C, D, and E. The initial value is chosen as follows:
IV = 67452301 EFCDAB89 98BADCFE 10325476 C3D2E1F0.

Padding is the same as in MD4, and the chaining input is fed forward in the same way, but all SHA functions assume a big-endian byte ordering (i.e., the sequence 00 01 02 03 of bytes will be read as a single 32-bit word 00010203).
The message expansion is more complex than in MD4, and there are four rounds of 20 steps each. The four Boolean functions are defined as follows.

$F_0(X, Y, Z) = (X \wedge Y) \vee (\neg X \wedge Z)$
$F_1(X, Y, Z) = X \oplus Y \oplus Z$
$F_2(X, Y, Z) = (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z)$
$F_3(X, Y, Z) = X \oplus Y \oplus Z$

Note that the functions F_1 and F_3 are same. There are four constants, one for each round. These are:
$k_0 = 5A827999_H$
$k_1 = 6ED9EBA1_H$
$k_2 = 8F1BBCDC_H$
$k_3 = CA62C1D6_H$

The message expansion works as follows. Let $m_0, m_1, \ldots, m_{15}$ be the sixteen input words of 32-bits. The 80 words w_i, $0 \leq \cdot\ i \leq 79$, in the expanded message are defined as follows.

$w_i = m_i$ for $0 \leq i \leq 15$

$w_i = (m_{i-3} \oplus m_{i-8} \oplus m_{i-14} \oplus m_{i-16})$ for $16 \leq i \leq 79$

A SHA-0 step consists of the following operations:

$A' \leftarrow A^{<<<5} + F_i(B, C, D) + E + w_i + k_i$

$B' \leftarrow A$

$C' \leftarrow B^{<<<30}$

$D' \leftarrow C$, and

$E' \leftarrow D$.

4.5.4 SHA-1

SHA-0 was replaced by SHA-1 in 1995 [25]. SHA-1 is an widely used cryptographic hash function. It has been used in many cryptographic standards, protocols, schemes etc. SHA-1 is a minor modification of SHA-0. The only difference between the two hash functions is the additional rotation operation in the message expansion of SHA-1, which is supposed to provide more security. This is described as

$w_i = m_i$ for $0 \leq i \leq 15$

$w_i = (m_{i-3} + m_{i-8} + m_{i-14} + m_{i-16})^{<<<1}$ for $16 \leq i \leq 79$

The four round constants k_i are first 32 bits of the decimal places of the square root of 2, 3, 5 and 10.

4.5.5 SHA-2

When the *Advanced Encryption Standard* (AES) was introduced in 2001, the need of new hash functions with larger output sizes were felt to match the key sizes in the AES. This led to the development of three new hash functions, SHA-256, SHA-384, and SHA-512, collectively termed SHA-2, published in 2002 [26]. In 2004, another hash function, SHA-224, was included to the list of SHA-2 family.

The message expansion is much more complicated than earlier versions of SHA, and two registers are updated in each step. SHA-224 and SHA-256 are constructed in the same way, but they use different IVs, and in SHA-224, a 256-bit state is truncated to 224 bits in the end. Similar differences exist between SHA-384 and SHA-512. SHA-256 and SHA-512 differ in word size; SHA-256 uses 32-bit words, whereas it is 64-bits word in SHA-512. The number of steps in SHA-256 and SHA-512 are 64 and 80 respectively. Also, there are few more minor differences.

Here, we shall only describe SHA-256 in detail. SHA-256 uses the same padding technique as MD4, and the compression function is also a block cipher in Davies-Meyer mode. The initial value of SHA-256 is
IV = 6A09E667 BB67AE85 3C6EF372 A54FF53A 510E527F 9B05688C 1F83D9AB $5BE0CD19_H$.
These values are the first 32-bits of fractional parts of square roots of first eight prime numbers.
SHA-256 uses a number of Boolean functions. First, the following two functions applied on a single 32-bit word are used in the message expansion.

$\sigma_0(x) = x^{>>>2} \oplus x^{>>>13} \oplus x^{>>22}$
$\sigma_1(x) = x^{>>>6} \oplus x^{>>>11} \oplus x^{>>25}$
Here, $x^{>>n}$ means shift right by n bits (not circular shift). The n most significant bits are filled with zero bits.
Two other functions operating on a single 32-bit word are the following:

$\Sigma_0(x) = x^{>>>2} \oplus x^{>>>13} \oplus x^{>>>22}$

$\Sigma_1(x) = x^{>>>6} \oplus x^{>>>11} \oplus x^{>>>25}$

The Boolean functions F_0 and F_2 are the same as defined in SHA-0 and SHA-1. The SHA-256 compression function takes a 512-bit message (sixteen words $m_0, m_1, \ldots, m_{15}$ of 32-bits each) and expands it into a 2048-bit message (sixty four words $w_0, w_1, \ldots, w_{63}$ of 32-bits each) as follows:
$w_i = m_i$ for $0 \leq i \leq 15$
$w_i = \sigma_1(m_{i-2}) + m_{i-7} + \sigma_1(m_{i-15}) + m_{i-16}$ for $16 \leq i \leq 63$

This message expansion introduces more diffusion than the SHA-0 and SHA-1. SHA-256 maintains a state of eight registers of 32-bits length. The state is updated through sixty four steps. Two registers are updated via a function of a number of other registers. Each step involves a distinct constant k_j, $0 \leq i \leq 63$. These constants are the first 32-bits of the fractional parts of the cube roots of the first sixty four prime numbers.

In each of sixty four steps, the following operations are performed (t1 and t2 are temporary variables, and A, B, . . ., H are the eight registers of the state).

$t1 \leftarrow \sum_0(A) + F_2(A,B,C)$
$t2 \leftarrow \sum_1(E) + F_0(E,F,G) + H + w_i + k_i$
$H \leftarrow G$
$G \leftarrow F$
$F \leftarrow E$
$E \leftarrow D + t2$
$D \leftarrow C$
$C \leftarrow B$
$B \leftarrow A$
$A \leftarrow t1 + t2$

After a message block is compressed, the variables A, B, …, H are updated similar to the update of chaining variables in MD4. After all the message blocks are compressed in this fashion, the message digest input message in SHA-256 is concatenation of final values of chaining variables i.e., A||B||C||D||E||F||G||H.

5. Block Cipher based Hash Functions

Simon, in his seminal work, [34] suggested that collision-resistant hash functions cannot be constructed based on one-way functions. Instead collision-resistant hash functions can be designed based on another very well known cryptographic primitive – Block Cipher. This idea of reducing the security of the hash to the security of a block cipher has its own advantages and disadvantages. The main advantage is that using a block cipher as a primitive of hash function gives us the security based on the proven security of block ciphers. Another advantage is that existing implementations can be reused. On the other hand, relying on just one primitive may be counter productive. A disadvantage is that hash functions based on a block cipher are less efficient than the dedicated proposals.

Block cipher is a popular encryption-decryption primitive. To encrypt, the block cipher accepts a key K and a plaintext block x as input, and produces a cipher text block c = E(K, x), also written as $c = E_K(x)$. The function E is invertible when K is known. The cipher text can be decrypted to obtain plain text $x = E_k^{-1}(c)$. It is to be noted that $|m| = |c|$, e.g., $|m| = n$. If we assume that $|K| = n$ or $|K| = m$ then a block cipher, in encryption mode, can be seen as compressing the $2n$ (or $n+m$) bits constituting the key and the plaintext block to n bits of cipher text.

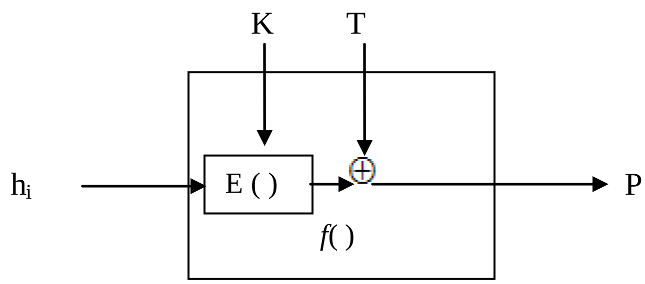

Figure 1: Round function of hash function based on block cipher

The above figure shows a round function that is used in block-cipher based hash functions. E is the block cipher primitive. P is plaintext, and K is key. The output of the round function after i^{th} round is h_i.

Essentially all modern hash functions are built by iterating a compression function following Merkle-Damgård paradigm or its variation. Again, these compression functions are almost always built from a block cipher. Even so-called “dedicated” hashing primitives like MD5 and SHA-1 are in fact based on block cipher. The SHA-1 compression function can be used as a block cipher [13]. Later on this suggestion was termed as SHACAL-1. SHACAL-1 is a 160-bit block cipher that takes variable length (0-512 bits key) and 80 rounds [14].

The earliest hash function from block ciphers was constructed by Rabin and DES block cipher was used [28]. He proposed to hash a message $X = x_1x_2 \cdot \cdot \cdot x_n$ by fixing an initial value h_0 and computing $H(M) = DESx_n(DESx_{n-1}(\ldots (DESx_1(h_0))))$. Here the input message X is divided into message blocks $x_1, x_2, x_3, \ldots, x_n$. This is in effect a Merkle-Damgård construction with block cipher based compression function $f(h_{i-1},x_i) = Ex_i(h_{i-1})$.

The general framework of iterated constructions with compression functions is of the form $f(h_{i-1},x_i) = E_a(b) \oplus c$ where $a,b,c \in \{h_{i-1}, x_i, h_{i-1} \oplus x_i, V\}$ for some fixed constant vector V. Generally, the value of V is considered to be zero. H_0 is equal to the initial value, IV. So, by choosing different possibilities of a, b, and c, we can have total 4^3 number of possibilities for round function f( ) to choose a particular block-cipher to be used in hash function. That is, there are $4^3 = 64$ different ways a block cipher can be used in compression function.

Preneel et al investigated in detail about different ways a block cipher can be used as a primitive for constructing compression function [27]. They have shown that only twelve simple constructions based on a block cipher result in collision-resistant compression functions. The following lists these compression functions. The notations used in this list are: x_i is the i^{th} message-block, h_i is the compression function used at i^{th} iteration, and E_k is the block cipher with key k.

$f_1: \quad E_{h_{i-1}}(m_i) \oplus m_i$

$f_2: \quad E_{h_{i-1}}(w_i) \oplus w_i$

$f_3: \quad E_{h_{i-1}}(m_i) \oplus w_i$

$f_4: \quad E_{h_{i-1}}(w_i) \oplus m_i$

$f_5: \quad E_{m_1}(h_{i-1}) \oplus h_{i-1}$

$f_6: \quad E_{m_1}(w_i) \oplus w_i$

$f_7: \quad E_{m_1}(h_{i-1}) \oplus w_i$

$f_8: \quad E_{m_1}(w_i) \oplus h_{i-1}$

$f_9: \quad E_{w_1}(m_i) \oplus m_i$

$f_{10}: \quad E_{w_1}(h_{i-1}) \oplus h_{i-1}$

$f_{11}: \quad E_{w_1}(m_i) \oplus h_{i-1}$

$f_{12}: \quad E_{w_1}(h_{i-1}) \oplus m_i$

We write w_i to mean $m_i \oplus h_{i-1}$. Black, Rogaway, and Srimpton showed that an additional eight of the sixty four schemes are just as collision resistant (up to a small constant) as the first group of schemes [5].

$f_{13}: \quad E_{w_1}(m_i) \oplus v$

$f_{14}: \quad E_{w_1}(m_i) \oplus w_i$

$f_{15}: \quad E_{m_1}(h_{i-1}) \oplus v$

$f_{16}: \quad E_{w_1}(h_{i-1}) \oplus v$

$f_{17}: \quad E_{m_1}(h_{i-1}) \oplus m_i$

$f_{18}: \quad E_{w_1}(h_{i-1}) \oplus w_i$

$f_{19}: \quad E_{m_1}(w_i) \oplus v$

$f_{20}: \quad E_{m_1}(w_i) \oplus m_i$

Here v denotes constant vector. From the perspective of collision resistance, one can choose any particular scheme from f_1, f_2, … , f_{20}. Among these collision-resistant compression functions the well known functions are Matyas-Meyer-Oseas (f_1), Miyaguchi-Preneel (f_3), and Davies-Meyer (f_5) [12]. Among these constructions Matyas-Meyer-Oseas and Davies-Meyer are dual constructions. In each iteration step of the hash computation, the present value of the chaining variable (h_i) and the input block to be processed (x_i) serve as key and plaintext of the encryption function (or vice versa for its dual). The plaintext input is then added with the output of encryption function, which constitutes a feed-forward operation. The result of the feed-forward serves as the new chaining value for the next iteration. They do not use an output transformation. Therefore the length of the chaining variable is equal to the output length. The feed-forward operation is based on modulo 2 addition (i.e., exclusive-OR), and its purpose is to make the compression function un-invertible.

Although these twenty schemes are provably secure, they could be viewed as inefficient from practical usability point of view. In each of these schemes, the block cipher key is changed every round. For all conventional block ciphers, changing the key in each round is not desirable. Because scheduling a new key causes a significant computational cost. Fix a small, non-empty set of block cipher keys K. A block cipher-based hash function is said to be highly-efficient if its compression function uses exactly one call to a block cipher (i.e., it is of rate 1), and if the block cipher uses only keys from K. Since we can preschedule each key in K, we enjoy a significant performance gain: key scheduling reduces to looking up a precomputed permutation.

Double block length constructions: Most of the block cipher based schemes result in hash functions with an output length that is too short from collision-resistance point of view. An alternative is the use of double block length hash functions, which produce a hash result with length equal to twice the block length of the cipher [32]. This means that DES will yield a 128-bit hash function, and AES to have a 256-bit hash function. Currently block cipher based hash functions are classified into single block length (SBL) hash functions and double block length (DBL) hash functions. For SBL hash functions, the length of output is equal to that of the block cipher, while for DBL hash functions, the length of the output is twice larger than that of the underlined block cipher.

An important parameter describing the efficiency of these constructions is the rate of the block cipher based hash function. The rate is defined as the number of b-bit input blocks that can be processed with a single encryption.

One extension of basic MD construction is "dithering". It is the technique of adding an iteration-dependent input (the dither) to the compression function to defeat certain generic attacks [31]. Aumasson et al identified methods for dithering block cipher based hash functions [2].

Permutation based design**:** Of late, researchers have started to explore to build compression functions from fixed key block ciphers, where only a small number of constants are used as keys. As each key of a block cipher defines an independent random permutation in the ideal cipher model, *such compression functions are often called permutation-based* [17, 19, and 23]. Permutation-based compression functions have an advantage over conventional block cipher based ones, since fixing the keys allows to save computational overload for key scheduling. These designs are characterized by the fact that the key input to the cipher depends on the input values. This implies that the key schedule has to be strong and that it needs to be executed for every encryption (or for every second encryption), which needs a substantial computational cost. An alternative approach is to fix one or more keys, and restrict the hash function to use the block cipher for these keys only. The usage of fixed-key block ciphers, or alternatively permutations, causes benefit that one does not need to implement an entire block cipher but only a limited number of instantiations of it. In the five finalists of the SHA-3 competition, two of them (BLAKE, and Skein) are *block cipher-based design*, the other three (Grøstl, JH, and Keccak) are *permutation-based design*.

6. Conclusion

An effort has been made to understand the prominent design rationale behind various cryptographic hash functions. The focus of discussion was on dedicated hash functions and block cipher based hash functions. Nonetheless there are other design philosophies to construct hash functions. One such approach is to construct hash functions that are provably secure in a sense that the problem of breaking it is related to some known computational problem considered very difficult. Classical examples include functions that rely on the hardness of factoring a large composite number proposed by Damg°ard [44] and Gibson [70] or on difficulty of solving discrete logarithm problem [8]. More recent constructions feature collision-resistant VSH [36] (very smooth hash) based on a number-theoretic problem related to factoring, one way and collision resistant FSB [4] related to hard problems in coding theory as well as provably one-way MQ-HASH [20] that depends on the difficulty of solving systems of multivariate quadratic equations. Unfortunately, there is also a price to pay for provable security. Most of functions designed that way need longer digests to achieve the desired level of security. They are also relatively less efficient as they usually require complex mathematical operations. NIST announced a public competition in 2007 to develop a new cryptographic hash algorithm, to choose "SHA-3". Five finalists of this SHA-3 competition are *BLAKE, Grøstl, JH, Keccak***,** and *Skein*. NIST has set the schedule to publish the winner of this competition in the last quarter of 2012. Each of these functions have received considerable amount of focus by the crypto community. With the rising use of sensor devices in many security sensitive applications, few lightweight hash functions have been proposed and many more to come in near future. This paper can serve newcomers, in its small effort, to understand the some prominent design philosophies of hash functions up to SHA-2 and then to move towards understanding SHA-3 to be announced and the lightweight hash functions.

7. References

1. Elena Andreeva. Domain Extenders for Cryptographic Hash Functions. Phd Thesis. Katholieke University Leuven. 2010.

2. Jean-Philippe Aumasson and Raphael C. -W. Phan. How (Not) to Efficiently Dither Blockcipher-Based Hash Functions?. AFRICACRYPT 2008. Springer LNCS Volume 5023, pages 308-324. Morocco. June 11-14, 2008

3. Mihir Bellare and Thomas Ristenpart. Multi-property-preserving hash domain extension and the EMD transform. ASIACRYPT 2006, Springer LNCS Volume 4284 of 2006, pages 299–314, Shanghai, December 3–7, 2006.

4. Mihir Bellare and Phillip Rogaway. Collision-resistant hashing: Towards making UOWHFs practical. CRYPTO 1997, Springer LNCS Volume 1294, pages 470–484, Santa Barbara, August 17–21, 1997.

5. J. Black, P. Rogaway, T. Shrimpto. Black-Box Analysis of the Block-Cipher-Based Hash-Function Constructions from PGV. CRYPTO 2002. Springer LNCS Volume 2442, pages 103-118. Santa Barbara, August 18-22, 2002.

6. Andrey Bogdanov. Analysis and Design of Block Cipher Constructions. Phd thesis. Ruhr University Bochum, Germany. 2009.

7. Donghoon Chang, Kishan Chand Gupta and Mridul Nandi . RC4-Hash: A New Hash Function Based on RC4. INDOCRYPT 2006. Springer LNCS Volume 4329. pages 80-94. Kolkata, December 11-13, 2006.

8. Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. Merkle-Damgård revisited: How to construct a hash function. CRYPTO 2005, Springer LNCS Volume 3621, pages 430–448, Santa Barbara, August 14–18, 2005.

9. Ivan Damgård. A Design Principle for Hash Functions. CRYPTO 1989. Springer LNCS Volume 435, pages 416-427. Santa Barbara, August 20-24, 1989.

10. Orr Dunkelman and Eli Biham. A Framework for Iterative Hash Functions: HAIFA. 2nd NIST Cryptographich Hash Workshop 2006. Santa Barbara, August 24-25, 2006.

11. Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. Sponge Functions. ECRYPT Hash Workshop 2007. Barcelona, May 24 - 25, 2007.

12. Praveen Gauravaram and Lars R. Knudsen. Cryptographic Hash Functions. In Handbook of Information and Communication Security. Peter Stavroulakis, Mark Stamp, Editors. Springer First edition, pages 59-79. 2010

13. Helena Handschuh and David Naccache. Analysis of SHA-1 in Encryption Mode, CT-RSA 2001, Springer LNCS Volume 2020, pages 70-83. San Francisco, April 8-12, 2001.

14. Helena Handschuh and David Naccache . SHACAL. Pre-proceedings of First Open NESSIE Workshop, Leuven, Belgium, November 13–14, 2000.

15. Shoichi Hirose, Je Hong Park, and Aaram Yun. A simple variant of the Merkle-Damgård scheme with a permutation. ASIACRYPT 2007, Springer LNCS Volume 4833, pages 113–129. Malaysia, December 2–6, 2007.

16. Xucjia Lai and James Massey. Hash Functions Based on Block Ciphers. Eurocrypt 1993, Springer LNCS Volume 658, pages 55-70. Norway, May 23-27, 1993.

17. Jooyoung Lee, Je Hong Park . Adaptive Preimage Resistance and Permutation-based Hash Functions. Cryptology ePrint Archive, Report 2009/66. 2009, URL - http://eprint.iacr.org/2009/66.pdf

18. Stefan Lucks. A failure-friendly design principle for hash functions. ASIACRYPT 2005, Springer LNCS Volume 3788, pages 474–494. Chennai, December 4–8, 2005.

19. Yiyuan Luo, Xuejia Lai. More Insights on Block cipher-Based Hash Functions. Cryptology ePrint Archive, Report 2010/642. 2010, URL – http://eprint.iacr.org/2010/642.pdf

20. Krystian Matusiewicz. Analysis of Modern Dedicated Cryptographic Hash Functions. phd thesis. Macquarie University. 2007

21. Menezes, Oorschot, and Vanstone. Handbook of Applied Cryptography. CRC Press, Florida. First edition, 1997:

22. Ralph Merkle. One Way Hash Functions and DES. CRYPTO 1989. Springer LNCS Volume 435, pages 428-446. Santa Barbara, August 20-24 1989.

23. Bart Mennink and Bart Preneel. Hash Functions Based on Three Permutations: A Generic Security Analysis. Cryptology ePrint Archive, Report 2011/532. 2011, URL - http://eprint.iacr.org/2011/532.pdf

24. NIST. Secure hash standard. Federal Information Processing Standard, FIPS-180, May 1993.

25. NIST. Secure hash standard. Federal Information Processing Standard, FIPS-180-1, April 1995.

26. NIST. Secure hash standard. Federal Information Processing Standard, FIPS 180-2, August 2002.

27. Preneel, B., Govaerts, R., and Vandewalle, J. Hash functions based on block ciphers: A synthetic approach. CRYPTO 1993. Springer LNCS Volume 773, pages. 368–378. Santa Barbara, August 22-26, 1993.

28. Rabin, M. Digitalized signatures. In Foundations of Secure Computation. R. DeMillo, D. Dobkin, A. Jones, and R. Lipton, Eds., Academic Press, pages. 155–168. 1978.

29. Ronald L. Rivest. The MD4 message digest algorithm. CRYPTO 1990, Springer LNCS Volume 537, pages 303–311. Santa Barbara, August 11-15, 1990.

30. Ronald L. Rivest. The MD5 message-digest algorithm, Request for Comments (RFC 1320), Internet Activities Board, Internet Privacy Task Force, 1992.

31. Ronald L. Rivest. Abelian Square-Free Dithering for Iterated Hash Functions. Cryptographic Hash workshop 2005, Gaithersburg, Maryland, November 2005.

32. van Rompay. Analysis and Design of Cryptographic Hash Functions, MAC Algorithms and Block Ciphers. Phd thesis. Katholieke University Leuven. 2004.

33. Victor Shoup. A composition theorem for universal one-way hash functions. EUROCRYPT 2000, Springer LNCS volume 1807, pages 445–452. Belgium, May 14–18, 2000.

34. Daniel R. Simon, Finding collisions on a one-way street: Can secure hash functions be based on general assumptions? EUROCRYPT 1998, Springer LNCS Volume 1403, pages. 334–345, Finland, May 31 - June 4, 1998.