UNIT 6 HASH FUNCTIONS

6.1 INTRODUCTION

Hash functions have many applications in Computer Science. Loosely speaking, hash functions are functions that take a variable length input and give an output of fixed length, i.e. it compresses the input of arbitrary size to a fixed size. From the point of view of applications in cryptography, we need hash functions satisfying some extra conditions. Such hash functions, called **one-way hash functions** or **cryptographic hash functions**, are the main topic of discussion in this Unit. In Sec. 6.2 we discuss the considerations that go into the design of cryptographic hash functions. In Sec. 6.3, we will discuss two examples of hash functions, namely, MD5 and SHA-II. In the Sec. 6.4, we will discuss some attacks on hash functions.

Objectives

After studying this unit, you should be able to

- define a cryptographic hash function;
- define a compression function;
- explain how to construct compression functions from block ciphers using Davies-Meyer, Matyas-Meyer-Oseas and Miyaguchi-Preneel methods;
- explain the Merkle-Damgård method for constructing hash functions from compression functions;
- explain the working of MD5 and SHA-II algorithms; and
- explain the birthday attack on hash functions.

6.2 DESIGN OF HASH FUNCTIONS

Let us begin our discussion of hash functions with an example of application of hash functions. Many useful softwares are available for free download from the internet. Sometimes, the file may not download properly for the following reason: All the data is transferred over the internet using the TCP/IP protocol. In this protocol, the data is divided into packets and sent over the network. All the packets are put together again at the destination. In this process, some packets may get lost. If this happens, you would have a damaged version of the file. How can you check if the software has downloaded correctly.

In some sites, you would have noticed that a number having 32 hexadecimal digits, called MD5 sum, is also given.(See Fig. 1 on the next page.) We will see later in this

Unit what this MD5 sum is. For the moment, we will discuss only the reason for giving the MD5 sum. There are softwares available on the internet to find the MD5 sums of files. You can find the MD5 sum for the file you have downloaded using any of these software and compare it with the MD5 sum given in the website. Even if the file you downloaded is slightly different from the file available on the internet, the MD5 sums will not match. So, you can check if your software has downloaded correctly by comparing the hash values. The MD5 sum is actually the **hash value** of the file

Fig. 1: Software and its MD5 sum.

calculated using a **hash function**.

The term hash functions comes from Computer Science where it denotes a function, not necessarily one-way, that compresses an input string of arbitrary length to a string of fixed length. There, the hash functions are used in the data structure called dictionary used for sorting. Depending on the application to which it is put, one-way hash function has many names in cryptographic literature like hashcode, hash total, checksum, Message Integrity Code(MIC), finger print, MDC, etc.

Let us now formally define a cryptographic hash function. Before we do that let us set up some notation. Note that there is a bijection between $\{0,1\}^n$, $n \geq 1$, and the set of all binary strings of length n. Similarly, there is bijection between the set of all possible finite binary strings and the set $\cup_{n=1}^{\infty} \{0,1\}^n$. Let us denote the set of all possible finite binary strings by $\{1,0\}^*$.

Definition 7: A function $h\colon \{0,1\}^* \longrightarrow \{0,1\}^n$, $n \geq 1$, is called a **a cryptographic hash function** if h has the following properties:

1) We should be able to calculate $h(\mathcal{M})$ easily from a given a message $\mathcal{M}$.
2) For a given y in the image of h, it should computationally infeasible to find an input $\mathcal{M}'$ with $h(\mathcal{M}') = y$. (One way)
3) It is computationally infeasible to find two inputs $\mathcal{M}_1$ and $\mathcal{M}_2$, $\mathcal{M}_1 \neq \mathcal{M}_2$, with $h(\mathcal{M}_1) = h(\mathcal{M}_2)$.(Collision resistance.)
4) Given $\mathcal{M}$ and $h(\mathcal{M})$, it is difficult to find $\mathcal{M}'$ such that $h(\mathcal{M}) = h(\mathcal{M})$.(Second pre-image resistant)

For reasons that will be clear later, we call the input to a hash function a **message** and the output the **message digest**. We call a function that satisfies condition 2) and 4) a **one way function**. We call a function that satisfies condition 3) a **collision resistant** hash function.

Note that, in condition 2), if $y = h(\mathcal{M})$, we are not trying to find $\mathcal{M}$. Rather, we are trying to find an $\mathcal{M}'$ with image y. Instead of condition 3), we often settle for the following weaker condition:

Definition 8: We call a function h **weakly collision free** if, given $\mathcal{M}$, it is computationally infeasible, to find another $\mathcal{M}' \neq \mathcal{M}$ such that $h(\mathcal{M}) = h(\mathcal{M}')$.

Recall that, the pigeon hole principle says that if the number of pigeons is greater than the number of pigeon holes, some pigeon holes must have more than one pigeon. In the case of hash functions, by mapping a large set to a small set, we are trying to put a large number of pigeons in a small number of pigeon holes. The set of possible messages is much larger than the set of possible message digests. So, there should be many instances of messages $\mathcal{M}_1$ and $\mathcal{M}_2$ with $h(\mathcal{M}_1) = h(\mathcal{M}_2)$. The requirement 3) only says that it should be computationally infeasible to such pairs $\mathcal{M}_1$ and $\mathcal{M}_2$.

In the example we discussed at the beginning of the unit we saw how we can detect if a file has changed using MD5 sum. We call the MD5 sum a **Modification Detection Code(MDC)** or **Manipulation Detection Code** because it helps us to check if a file has been modified. However, if a malicious attacker gains access to the server where the software is made available he/she can do the following:

1) Replace the software file with another file containing a harmful software.
2) Calculate the MD5 sum for this modified file and replace the hash sum on the website with the hash sum of the malicious program.

The attacker is able to do this because any one can calculate the MD5 sum. To prevent this, we use another kind of hash functions called **keyed hash-functions**. In this case a secret key is required to calculate the hash. Such keyed hash functions are also called **Message Authentication Codes(MACs)** for the following reason: Suppose Alice and Bob share a secret key k. Also, suppose that Bob wants to send a message to Alice and Alice wants to be sure that the message was actually sent by Bob. Then, if $\mathcal{M}$ is the message, Bob can calculate the hash $h_k(\mathcal{M})$ and send $(\mathcal{M}, h_k(\mathcal{M}))$ to Alice. Here, $h_k(\mathcal{M})$ is a hash function that uses the secret key k for calculating the hash. Since Alice also has the key k, she can calculate $h_k(\mathcal{M})$ and compare it with the value sent by Bob and convince herself that the message was actually sent by Bob. If Eve modifies the message to $\mathcal{M}'$, she will not be able to calculate the hash $h_k(\mathcal{M}')$ since she doesn't know the secret key k. So, she will not be able to replace the hash value Bob has sent with $h_k(\mathcal{M}')$. Thus, she cannot modify the message without Alice's knowledge. Let us now define keyed hash functions formally:

Definition 9: By a family of **keyed hash functions** we mean a set of hash functions $\{h_k\}_{k\in K}$ parametrised by a finite set $\mathcal{K}$ called the keyspace. In other words, for each $k \in \mathcal{K}$, we have a hash function $h_k : \{0,1\}^* \to \{0,1\}^n$.

Definition 10: A **HMAC(Hashed Message Authentication Code)** takes a message $\mathcal{M}$of arbitrary length and a secret key k as input and gives an output of fixed length. It is defined as follows:

$$\text{HMAC}_k(\mathcal{M}) = h\left(k' \oplus \text{opad}, h\left(k' \oplus \text{ipad}, \mathcal{M}\right)\right)$$

where:

1) h is a usual hash function that acts on blocks of size m.
2) k' is got by padding k with sufficient number of zeros on the right so that k' has size m.
3) We form ipad by repeating the byte 0x36 as many times as necessary so that the length is m. Similarly, we form opad by repeating the byte 0x5c.

An **NMAC(Nested Message Authentication Code)** is a generalised version of HMAC which uses two secret keys k_1 and k_2 and is defined as follows:

$$\text{NMAC}_{K_1,k_2}(\mathcal{M}) = h(k_2, h(k_1, \mathcal{M}))$$

Here h is a usual hash function.

Note that, in the definition of the NMAC, if we take $k_1 = k' \oplus \text{ipad}$ and $k_2 = k' \oplus \text{opad}$, we get a HMAC.

A method for constructing hash functions that is used in the construction of many hash functions is the **Merkle-Damgård** method. The basic idea behind the method is to use a **one-way compression function** repeatedly.

Definition 11: A **collision resistant function** f is a map

$$f:\ \{0,1\}^{n+m} \longrightarrow \{0,1\}^{n},\ m,n \in \mathbf{N},\ m > n,$$

which is collision resistant, i.e. it satisfies condition 3) in the definition of a cryptographic hash function.

The nice thing about the Merkle-Damgård method is that we can prove that the hash function constructed using this method satisfies conditions 1), 2) and 3) if the one-way compression function satisfies them. The Merkle-Damgård method is as follows:

Let f be a compression function. Suppose, we want to find the hash of a message $\mathcal{M}$. Let us call the number of bits in the message, the **length** of the message. We first pad it with zeroes so that the number of bits in the message is a multiple of m. However, there is a problem in this. Suppose for simplicity that $m = 64$ and the message $\mathcal{M}$ is the string "Hashexample". If we assume that we use 8 bits to store a character, then this string is of length 88 bits. Then, dividing into blocks of length 64 each and padding it we will get two blocks "Hashexam" and "ple$\underbrace{000\ldots00}_{40\text{ zeroes}}$". (Note here, that we consider the zeroes added as bits and not characters and so each zero is of length 1 bit.) However, if we take the string "Hashexample00",the phrase "Hashexample" followed by two zero bits, we will again get the same two blocks and so the strings "Hashexample" and "Hashexample00" will have the same hash value. To avoid this we insert a one bit at the beginning of the zeroes. For example, using this method, "Hashexample" will yield the blocks "Hashexam" and "ple1$\underbrace{00\ldots00}_{39\text{ zeroes}}$" while "Hashexample00" will yield the strings "Hashexam" and "ple001$\underbrace{000\ldots00}_{37\text{ zeroes}}$" which are different.

We further pad the message by adding one more block which we construct as follows: Suppose the length of the message $\mathcal{M}$ before padding is ℓ, in binary. Here, we assume that $\ell < 2^m$. We add as many zeroes to the left of ℓ as necessary to get a block of size m. For example, suppose we want to apply this to the string "Hashexample". Then, after padding it to get "Hashexam" and "ple$\underbrace{000\ldots00}_{40\text{ zeroes}}$", we add another block "$\underbrace{00\ldots00}_{57\text{ zeroes}}$1011000". Here the length of the string "Hashexample" is 88 and the binary representation of 88 is 1011000. We add 57 zeroes to the left of the bitstring 1011000 so that the total length is 64. This technique is called **length padding** or **Merkle-Damgård** strengthening.

After padding the message so that its length is a multiple of m, we divide the message into blocks $\mathcal{M}_1$, $\mathcal{M}_2$, …, $\mathcal{M}_t$ where each block is of size m. We then use the one-way compression function f to create the hash in t steps. Let 0^n denote a string consisting of n zeroes. We set

$$\mathcal{H}_0 = f(0^n||\mathcal{M}_1), \mathcal{H}_i = f\left(\mathcal{H}_{i-1}||\mathcal{M}_i\right) \text{ for } 1 \leq i \leq t$$

Here 0^n is an example of **initial vector**. In the hash algorithms, we use some other initial vectors also instead of 0^n. Note that each $\mathcal{H}_i$ is m bits long. The last output $\mathcal{H}_t$ is the hash of the message $\mathcal{M}$. (See Fig. 2 on the facing page.) The values $\mathcal{H}_0$, $\mathcal{H}_1$, …, $\mathcal{H}_{t-1}$ are called **chaining variables** because they are chained to the next stage of application of the compression function, as shown in Fig. 2 on the next page.

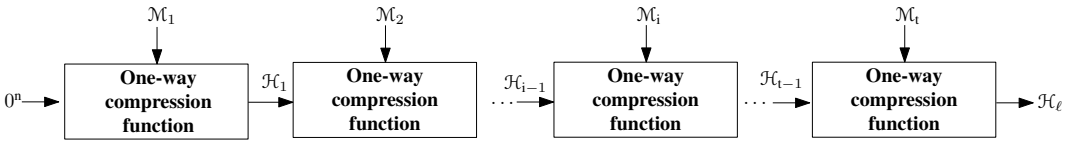

Fig. 2: The Merkle-Damgård method.

Before we move on to the next topic of discussion, here is an exercise for you.

E1) Assuming a block size of 64 and that we use 8 bits to store a character, what will be the string you will get by applying the Merkle-Damgård strengthening to the string "tohashornottohash"?

Let us now address the next issue, namely the construction of one-way compression function. One method is to use a block cipher to construct a one-way compression function. In this Unit, we will discuss the methods for this due to Davies-Meyer, Matyas-Meyer-Oseas, Miyaguchi-Preneel. For a discussion of other methods, you can see the Wikipaedia article [23] or [11].

In our discussion, let $E_k(\mathcal{M})$ denote ciphertext we get by encrypting a message $\mathcal{M}$ with a block cipher E and key k.

Davies-Meyer method: Here, we assume that the block cipher takes as input a string of size n and outputs a string of size n and the size of the key is m. In this method, the one way compression function is defined as follows: Let X be a string of length $n+m$. Let X_L denote first n bits on the left of X and let X_R denote the remaining m bits so that $X = X_L||X_R$. Then, the compression function is

$$f(X) = f(X_L||X_R) = E_{X_R}(X_L) \oplus X_L.$$

So, we have

$$\mathcal{H}_i = E_{\mathcal{M}_i}\left(\mathcal{H}_{i-1}\right) \oplus \mathcal{H}_{i-1}.$$

See Fig. 3.

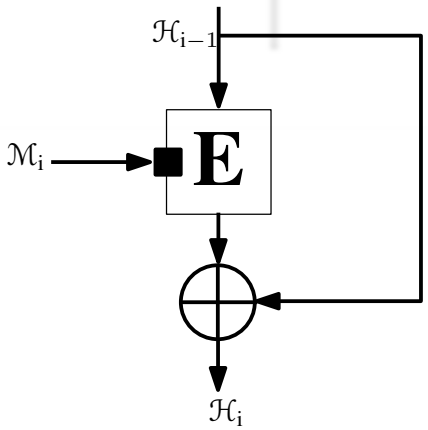

Fig. 3: Davies-Meyer Method.

Matyas-Meyer-Oseas method: In this method, we use a block cipher E which takes an input of size m, uses a key of size n and produces an output of size m. Note the interchange of the roles of the message and the hash in this method. If the block cipher has different block and key sizes, the length of the previous hash may not be of the correct size for a key. Also, the key for the block cipher may have some special requirements. So, the previous hash $\mathcal{H}_{i-1}$ has to be processed with another function g

to produce a key from the hash $\mathcal{H}_{i-1}$. In this case we define the compression function as follows:

$$f(X) = f(X_L||X_R) = E_{g(X_L)}(X_R) \oplus X_R.$$

So,

$$\mathcal{H}_i = E_{g(\mathcal{H}_{i-1})}(\mathcal{M}_i) \oplus \mathcal{M}_i.$$

See Fig. 4.

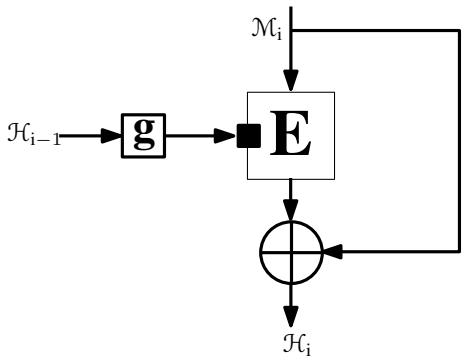

Fig. 4: Matyas-Meyer-Oseas method.

Miyaguchi-Preneel method: This was proposed independently by Miyaguchi and Preneel. It is an extended version of Matyas-Meyer-Oseas method. In this method, in addition to XORing with $\mathcal{M}_i$, the previous hash value is also XORed with the output of the block cipher. In this case we define the compression function as follows:

$$f(X) = f(X_L||X_R) = E_{g(X_L)}(X_R) \oplus X_R \oplus X_L.$$

So,

$$\mathcal{H}_i = E_{g(\mathcal{H}_{i-1})}(\mathcal{M}_i) \oplus \mathcal{M}_i \oplus \mathcal{H}_{i-1}.$$

See Fig. 5.

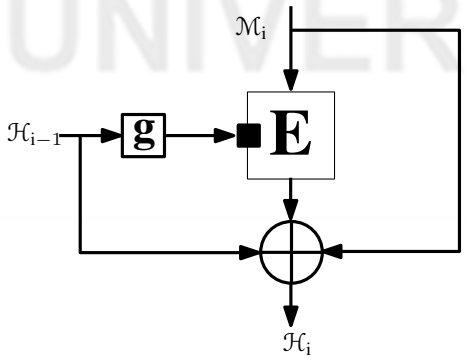

Fig. 5: Miyaguchi-Preneel method.

We close this section here. In the next section, we will discuss some popular hash functions.

6.3 EXAMPLES OF HASH FUNCTIONS

In the previous section, we saw some methods of designing hash functions. In this section, we are going to see some hash functions based on these design principles. We start our discussion with MD5 hash function. This was invented by Ronald Rivest in the year 1992 to replace MD4, a hash function he invented earlier, after cryptographers found some flaws in it.

6.3.1 MD5

MD5 takes as input messages of arbitrary size and outputs a hash of size 128 bits. We describe this process now. The block size of the function is 512 bits, i.e. the compression function acts on 512 bits of the message at a time.

Step 1: Padding. We pad the message with 1 followed by as many zeroes as necessary so that the message length is $\equiv 448 \pmod{512}$, i.e. 64 less than a multiple of 512. We do this even if the length of the message is already a multiple of 512. Then, we add the 64 bit representation of the length of the message. For example, suppose the length of the message is 234. Then, $234 = 11101010$ has eight binary digits. We add 56 zeros to the left to get a 64 bit representation of 234 and append this to the message. If the length of the message is greater than 2^{64} we reduce the length $\bmod 2^{64}$. At the end of this step, let the message be $\mathcal{M}_1\mathcal{M}_2\cdots\mathcal{M}_\ell$ where 512ℓ is the total length of the message after padding and each $\mathcal{M}_i$ is 512 bits long.

Step 2: Initialise the Buffer. The intermediate results of the process of creating the hash are stored in a 128 bit buffer. We initialise these registers to the following values written in hexadecimal:

$$\begin{aligned}A &= 67452301\\ B &= EFCDAB89\\ C &= 98BADCFE\\ D &= 10325476\end{aligned}$$

These values are stored in **little-endian** format. What is this format? Note that we can divide 32 bits into four bytes of length 8 bits each. In the little-endian format, the lower bytes are stored at the lower addresses and the higher order bytes at higher addresses. For example, the four bytes in 67452381 will be stored as follows:

Word A : 81 23 45 67

We store the least significant byte, 01 in the left most byte of the 32 bit word which has the smallest address. We store the bytes of higher significance at higher addresses. In the other format, which is called **big-endian**, we store the most significant byte at the lowest address and bytes of progressively higher significance are stored in progressively higher addresses.

Step 3: Process the message in 512 bit blocks. This involves the use of a compression function that processes the 512 bit input in four steps or rounds. In each of these rounds, it uses four different functions F, G, H and I which take three 32-bit words X, Y and Z and output one 32 bit word:

$$\begin{aligned}F(X,Y,Z) &= (X\wedge Y)\vee(\neg X\wedge Z)\\ G(X,Y,Z) &= (X\wedge Z)\vee(Y\wedge\neg Z)\\ H(X,Y,Z) &= X\oplus Y\oplus Z\\ I(X,Y,Z) &= Y\oplus(X\vee\neg Z)\end{aligned}$$

Let us see what these functions are. Here, $\wedge$, $\vee$ and $\oplus$ are the familiar functions from logic. Let us quickly recall what they are. Suppose $x, y \in \{0,1\}$. Recall that:

1) We have $x \vee y = 0$ if and only if both x and y are zero. For all other values of x and y, $x \vee y$ is 1.
2) We have $x \wedge y = 1$ if and only if both x and y are one. For all other values of x and y, $x \wedge y$ is 0.
3) We have $x \oplus y = 1$ if $x \neq y$ and 0 if $x = y$.
4) We have $\neg x = 0$ if $x = 1$ and $\neg x = 1$ if $x = 0$.

Here, X, Y and Z are 32 bit words. Suppose $X = x_1x_2\cdots x_{32}$ and $Y = y_1y_2\cdots y_{32}$ where x_i and y_i are 0 or 1. What do we mean by $(X \vee Y)$, for example? It has the following obvious meaning:

$$X = (x_1x_2\cdots x_{32}) \vee (y_1y_2\cdots y_{32}) = a_1a_2\cdots a_{32} \qquad \text{where } a_i = x_i \vee y_i, 1 \leq i \leq 32$$

Similarly, we apply all the other operations bit by bit.

Addition Modulo 2^{32}: Suppose X and Y are 32 bit representations of two integers x and y, where $0 \leq x < 2^{32}$ and $0 \leq y < 2^{32}$. Then, we define $X + Y$ as follows: Let $z = x + y \pmod{32}$, $0 \leq z \leq 2^{32}$. Let Z be the 32 bit word corresponding to z. Then $Z = X + Y$. For example, suppose $x = 2^{32} - 3$ and $y = 5$. Then,

$$X = \underbrace{111\cdots1}_{30\ 1s}01, Y = \underbrace{000\cdots0}_{29\ 0s}101.$$

Then $x + y = 2^{32} + 2 \equiv 2 \pmod{2^{32}}$. Therefore, $z = 2$ and

$$Z = \underbrace{000\cdots0}_{29\ 0s}10.$$

Another function that we are going to use is the **right and left circular shifts**. We have already discussed this in the the appendix to unit 4.

Step 4 This step uses a table T. Let T[i] be the i-th element of the table. Then, $T[i] = \lfloor |2^{32} \sin i| \rfloor$, i.e. the absolute value of the integer part of $2^{32} \sin i$, where i is in radians. Note that $\mathcal{M}_i$ is 512 bits long. Let $\mathcal{M}_i^{(j)}$ denote the chunk of bits from $32j + 1$th bit to $32(j + 1)$th bit, $0 \leq j \leq 15$. Now, we do as in Algorithm 3 on the next page.

Step 5: A, B, C, D is the 128 bit message digest produced. (Note that the length of each of A, B, C and D is 32 bits.)

We conclude our discussion with some remarks on the current status of MD5. In 1996, a flaw was found by Dobbertin. See [6]. Although the flaw was not considered very serious, cryptographers started recommending other hash functions like SHA-1. However, in [22], Wang, Feng, Lai and Yu found many examples of collisions in many popular hash functions including MD5. See also [3]. In 2007, Marc Stevens, Arjen Lenstra and Benne de Weger MD5 showed how to create a pair of files with the same hash sum. In 2008, Alexander Sotirov, Marc Stevens, Jacob Appelbaum, Arjen Lenstra, David Molnar, Dag Arne Osvik, Benne de Weger used the technique to fake certificate validity. Also, according to US-CERT of the U. S. Department of Homeland Security MD5 "should be considered cryptographically broken and unsuitable for further use," and most U.S. government applications will be required to move to the SHA-2 family of hash functions after 2010. As of now, MD5 seems useful only for the application we mentioned in the introduction, namely checking files for accidental damages. In 2005, security flaws were found in 2005. Because of these reasons, we discuss SHA-2 in the next subsection.

6.3.2 SHA-2

The SHA-2 is actually a family of hash functions consisting of SHA-224, SHA-256, SHA-384, SHA-512. NSA designed these functions and NIST published them in 2001. In 2005, some attacks were identified in SHA-1, indicating that a stronger hash function is needed. Although SHA-2 is similar to SHA-1, these attacks don't seem feasible for SHA-2 at present. See [17] and [18] for some recent work on SHA-2. A competition organised by NIST is underway to develop SHA-3 and this will end in 2012 with the selection of a winning hash function. Our discussion will follow [14] although our notation is different from [14] in some ways. We will restrict ourselves to SHA-256 only. SHA-224 is very similar to SHA-256 and we will mention how SHA-224 differs from SHA-256.

Algorithm 3 MD5 Algorithm

procedure MD5($\mathcal{M}$) ▷ $\mathcal{M}$ is the message after padding.
  for i $\leftarrow 1, \ell$ **do** ▷ ℓ is the number of blocks in $\mathcal{M}$ of length 512
    for j $\leftarrow 0, 15$ **do** ▷ Copy $\mathcal{M}_i$ into X.
      $X[j] \leftarrow \mathcal{M}_i^{(j)}$
    end for ▷ End of the loop over j.
    ▷ Save A as AA, B as BB, C as CC and D as DD.
    AA $\leftarrow$ A
    BB $\leftarrow$ B
    CC $\leftarrow$ C
    DD $\leftarrow$ D
    ▷ Round 1
    ▷ Let [abcd k s i] denote the operation $a = b + ((a + F(a,b,c,d) + X[k] + T[i]) \lll s)$. Do the following 16 operations.
    [ABCD 0 7 1] [DABC 1 12 2] [CDAB 2 17 3] [BCDA 3 22 4]
    [ABCD 4 7 5] [DABC 5 12 6] [CDAB 6 17 7] [BCDA 7 22 8]
    [ABCD 8 7 9] [DABC 9 12 10] [CDAB 10 17 11] [BCDA 11 22 12]
    [ABCD 12 7 13] [DABC 13 12 14] [CDAB 14 17 15] [BCDA 15 22 16]
    ▷ Round 2
    ▷ Let [abcd k s i] denote the operation $a = b + ((a + G(b,c,d) + X[k] + T[i]) \lll s)$. Do the following 16 operations.
    [ABCD 1 5 17] [DABC 6 9 18] [CDAB 11 14 19] [BCDA 0 20 20]
    [ABCD 5 5 21] [DABC 10 9 22] [CDAB 15 14 23] [BCDA 4 20 24]
    [ABCD 9 5 25] [DABC 14 9 26] [CDAB 3 14 27] [BCDA 8 20 28]
    [ABCD 13 5 29] [DABC 2 9 30] [CDAB 7 14 31] [BCDA 12 20 32]
    ▷ Round 3.
    ▷ Let [abcd k s i] denote the operation $a = b + ((a + H(b,c,d) + X[k] + T[i]) \lll s)$. Do the following 16 operations.
    [ABCD 5 4 33] [DABC 8 11 34] [CDAB 11 16 35] [BCDA 14 23 36]
    [ABCD 1 4 37] [DABC 4 11 38] [CDAB 7 16 39] [BCDA 10 23 40]
    [ABCD 13 4 41] [DABC 0 11 42] [CDAB 3 16 43] [BCDA 6 23 44]
    [ABCD 9 4 45] [DABC 12 11 46] [CDAB 15 16 47] [BCDA 2 23 48]
    ▷ Round 4.
    ▷ Let [abcd k s t] denote the operation $a = b + ((a + I(b,c,d) + X[k] + T[i]) \lll s)$. Do the following 16 operations.
    [ABCD 0 6 49] [DABC 7 10 50] [CDAB 14 15 51] [BCDA 5 21 52]
    [ABCD 12 6 53] [DABC 3 10 54] [CDAB 10 15 55] [BCDA 1 21 56]
    [ABCD 8 6 57] [DABC 15 10 58] [CDAB 6 15 59] [BCDA 13 21 60]
    [ABCD 4 6 61] [DABC 11 10 62] [CDAB 2 15 63] [BCDA 9 21 64]
    ▷ Then perform the following additions. (That is, increment each of the four registers by the value it had before this block was started.)
    A = A + AA
    B = B + BB
    C = C + CC
    D = D + DD
  end for ▷ End of loop over i.
end procedure

Table 1: The 64 constants used in SHA-256.

$K_0^{\{256\}} = 428a2f98$	$K_1^{\{256\}} = 71374491$	$K_2^{\{256\}} = b5c0fbcf$	$K_3^{\{256\}} = e9b5dba5$
$K_4^{\{256\}} = 3956c25b$	$K_5^{\{256\}} = 59f111f1$	$K_6^{\{256\}} = 923f82a4$	$K_7^{\{256\}} = ab1c5ed5$
$K_8^{\{256\}} = d807aa98$	$K_9^{\{256\}} = 12835b01$	$K_{10}^{\{256\}} = 243185be$	$K_{11}^{\{256\}} = 550c7dc3$
$K_{12}^{\{256\}} = 72be5d74$	$K_{13}^{\{256\}} = 80deb1fe$	$K_{14}^{\{256\}} = 9bdc06a7$	$K_{15}^{\{256\}} = c19bf174$
$K_{16}^{\{256\}} = e49b69c1$	$K_{17}^{\{256\}} = efbe4786$	$K_{18}^{\{256\}} = 0fc19dc6$	$K_{19}^{\{256\}} = 240ca1cc$
$K_{20}^{\{256\}} = 2de92c6f$	$K_{21}^{\{256\}} = 4a7484aa$	$K_{22}^{\{256\}} = 5cb0a9dc$	$K_{23}^{\{256\}} = 76f988da$
$K_{24}^{\{256\}} = 983e5152$	$K_{25}^{\{256\}} = a831c66d$	$K_{26}^{\{256\}} = b00327c8$	$K_{27}^{\{256\}} = bf597fc7$
$K_{28}^{\{256\}} = c6e00bf3$	$K_{29}^{\{256\}} = d5a79147$	$K_{30}^{\{256\}} = 06ca6351$	$K_{31}^{\{256\}} = 14292967$
$K_{32}^{\{256\}} = 27b70a85$	$K_{33}^{\{256\}} = 2e1b2138$	$K_{34}^{\{256\}} = 4d2c6dfc$	$K_{35}^{\{256\}} = 53380d13$
$K_{36}^{\{256\}} = 650a7354$	$K_{37}^{\{256\}} = 766a0abb$	$K_{38}^{\{256\}} = 81c2c92e$	$K_{39}^{\{256\}} = 92722c85$
$K_{40}^{\{256\}} = a2bfe8a1$	$K_{41}^{\{256\}} = a81a664b$	$K_{42}^{\{256\}} = c24b8b70$	$K_{43}^{\{256\}} = c76c51a3$
$K_{44}^{\{256\}} = d192e819$	$K_{45}^{\{256\}} = d6990624$	$K_{46}^{\{256\}} = f40e3585$	$K_{47}^{\{256\}} = 106aa070$
$K_{48}^{\{256\}} = 19a4c116$	$K_{49}^{\{256\}} = 1e376c08$	$K_{50}^{\{256\}} = 2748774c$	$K_{51}^{\{256\}} = 34b0bcb5$
$K_{52}^{\{256\}} = 391c0cb3$	$K_{53}^{\{256\}} = 4ed8aa4a$	$K_{54}^{\{256\}} = 5b9cca4f$	$K_{55}^{\{256\}} = 682e6ff3$
$K_{56}^{\{256\}} = 748f82ee$	$K_{57}^{\{256\}} = 78a5636f$	$K_{58}^{\{256\}} = 84c87814$	$K_{59}^{\{256\}} = 8cc70208$
$K_{60}^{\{256\}} = 90befffa$	$K_{61}^{\{256\}} = a4506ceb$	$K_{62}^{\{256\}} = bef9a3f7$	$K_{63}^{\{256\}} = c67178f2$

We can use SHA-256 to hash a message $\mathcal{M}$ of length N bits where $0 \leq N < 2^{64}$. The algorithm uses the following:

1) A message schedule of 64 words of length 32 bits, W_0, W_1, …, W_{63}.

2) Eight working variables **a**, **b**, **c**, **d**, **e**, **f**, **g** and **h**.

3) A hash value of eight 32 bit words. During the i^{th} round of the processing(there are ℓ rounds in all, where ℓ is the number of blocks of length 512 in the message after padding) $\mathcal{H}_0^{(i)}$, $\mathcal{H}_1^{(i)}$, …, $\mathcal{H}_7^{(i)}$ hold the intermediate hash values. The final hash values $\mathcal{H}_\ell^{(0)}$, $\mathcal{H}_\ell^{(1)}$, …, $\mathcal{H}_\ell^{(7)}$ are concatenated to give the output of the algorithm.

4) The 64 constants in Table 1 which are the first thirty-two bits of the fractional parts of the cube roots of the first sixty-four prime numbers.

Step 1: Preprocessing. We first pad the message exactly the way we did in the case of MD5 algorithm. We then divide the resulting message into ℓ blocks $\mathcal{M}_1$, $\mathcal{M}_2$, …, $\mathcal{M}_\ell$ of 512 bits each.

Step 2: Initialise the Hash values. We set the initial hash values $\mathcal{H}_0^{(0)}$, $\mathcal{H}_1^{(0)}$, …, $\mathcal{H}_7^{(0)}$.

$$\mathcal{H}_0^{(0)} = 6a09e667$$
$$\mathcal{H}_1^{(0)} = bb67ae85$$
$$\mathcal{H}_2^{(0)} = 3c6ef372$$
$$\mathcal{H}_3^{(0)} = a54ff53a$$
$$\mathcal{H}_4^{(0)} = 510e527f$$
$$\mathcal{H}_5^{(0)} = 9b05688c$$
$$\mathcal{H}_6^{(0)} = 1f83d9ab$$
$$\mathcal{H}_7^{(0)} = 5be0cd19$$

Step 3: Hash computation. Before we describe this step, we have to discuss some new notations that we need in our description.

Let us now define the functions used in SHA-256. They are:

$$\mathrm{Ch}(X,Y,Z) = (X \wedge Y) \oplus (\neg X \wedge X) \tag{1}$$

$$\mathrm{Maj}(X,Y,Z) = (X \wedge Y) \oplus (X \wedge Z) \oplus (Y \wedge Z) \tag{2}$$

$$\sum_0^{\{256\}}(X) = (X \ggg 2) \oplus (X \ggg 13) \oplus (X \ggg 22) \tag{3}$$

$$\sum_1^{\{256\}}(X) = (X \ggg 6) \oplus (X \ggg 11) \oplus (X \ggg 25) \tag{4}$$

$$\sigma_0^{\{256\}}(X) = (X \ggg 7) \oplus (X \ggg 18) \oplus (X \gg 3) \tag{5}$$

$$\sigma_1^{\{256\}}(X) = (X \ggg 17) \oplus (X \ggg 19) \oplus (X \gg 10) \tag{6}$$

Here $\gg$ is the right shift in the C language. This is discussed in page 139 of Block 2, MMT-001.

Now, we process the message as in Algorithm 4 on the next page.

Apart from SHA-2, another algorithm that is still usable is RIPMED-160. This was developed by Dobbertin, Bosselaers and Preneel in the framework of the European Union Project RIPE(Race Integrity Primitives Evaluation). This is based on the MD4 algorithm due to Rivest. It gives an output of length 160 bits. We will not discuss this algorithm in our course.

So far we have discussed the design of hash functions. In the next section, we will discuss the analysis of hash functions.

6.4 BIRTHDAY ATTACKS

In this section, we will discuss one of the techniques for cryptanalysis of hash functions, namely the Birthday Attack. This attack is based on the Birthday Problem in probability theory. So, let us first discuss the Birthday Problem.

Suppose there are n people in a room. What is the probability that two of them have the same birthday? We will ignore leap years and assume that the year has 365 days. Let us assign the n persons n numbers and call them $p_1, p_2, p_3, \ldots, p_n$. We can assume that $n \leq 365$. If $n > 365$, pigeon principle tells us that at least two of the persons should have the same birthday, so the probability is one in this case.

Suppose there are only 2 persons. Then, birthday of the first person falls in a particular day of the year. The probability that the birthday of the second person p_2 also falls on the same day as p_1 is $\frac{1}{365}$. So, the probability that p_2 doesn't have the same birthday as p_1 is $\left(1 - \frac{1}{365}\right)$. If there are three persons, the probability that p_3 doesn't have the same birthday as p_1 and p_2 is $\left(1 - \frac{2}{365}\right)$. So, the probability that all the three have different birthdays is

$$\left(1 - \frac{1}{365}\right)\left(1 - \frac{2}{365}\right)$$

A simple inductive argument tells us that the probability that the n persons have different birthdays is

$$\prod_{i=1}^{n-1}\left(1 - \frac{i}{365}\right) = \left(1 - \frac{1}{365}\right)\left(1 - \frac{2}{365}\right)\cdots\left(1 - \frac{n-1}{365}\right) \tag{7}$$

Therefore, the probability of at least two having the same birthday is

$$p(n) = 1 - \prod_{i=1}^{n-1}\left(1 - \frac{i}{365}\right) \tag{8}$$

Here is the table of probabilities:

Algorithm 4 SHA-256 Algorithm

procedure SHA-256($\mathcal{M}$) ▷ $\mathcal{M}$ is the message after padding.
  for i $\leftarrow 1, \ell$ **do**
    for j $\leftarrow$ 1, 15 **do** ▷ Prepare the message schedule.
      $W_i \leftarrow \mathcal{M}_j^{(i)}$
    end for ▷ End of the loop over j.
    for k $\leftarrow$ 16, 63 **do**
      $W_k \leftarrow \sigma_1^{\{256\}}(W_{k-2}) + W_{k-7} + \sigma_0^{\{256\}}(W_{k-15}) + W_{k-16}$
    end for ▷ End of the loop over k.
  ▷ Initialise the eight working variables **a**, **b**, **c**, **d**, **e**, **f**, **g** and **h** with the (i-1)st hash value.
    $\mathbf{a} \leftarrow \mathcal{H}_0^{(i-1)}$
    $\mathbf{b} \leftarrow \mathcal{H}_1^{(i-1)}$
    $\mathbf{c} \leftarrow \mathcal{H}_2^{(i-1)}$
    $\mathbf{d} \leftarrow \mathcal{H}_3^{(i-1)}$
    $\mathbf{e} \leftarrow \mathcal{H}_4^{(i-1)}$
    $\mathbf{f} \leftarrow \mathcal{H}_5^{(i-1)}$
    $\mathbf{g} \leftarrow \mathcal{H}_6^{(i-1)}$
    $\mathbf{h} \leftarrow \mathcal{H}_7^{(i-1)}$
    for t $\leftarrow$ 0,63 **do**
      $T_1 \leftarrow \mathbf{h} + \sum_1^{\{256\}}(\mathbf{e}) + Ch(\mathbf{e},\mathbf{f},\mathbf{g}) + K_t^{\{256\}} + W_t$
      $T_2 \leftarrow \sum_0^{\{256\}}(\mathbf{a}) + Maj(\mathbf{a},\mathbf{b},\mathbf{c})$
      $\mathbf{h} \leftarrow \mathbf{g}$
      $\mathbf{g} \leftarrow \mathbf{f}$
      $\mathbf{f} \leftarrow \mathbf{e}$
      $\mathbf{e} \leftarrow \mathbf{d} + T_1$
      $\mathbf{d} \leftarrow \mathbf{c}$
      $\mathbf{c} \leftarrow \mathbf{b}$
      $\mathbf{b} \leftarrow \mathbf{a}$
      $\mathbf{a} \leftarrow T_1 + T_2$
    end for ▷ End of the loop over t.
  ▷ Compute the ith intermediate hash value $\mathcal{H}^{(i)}$.
    $\mathcal{H}_0^{(i)} \leftarrow \mathbf{a} + \mathcal{H}_0^{(i-1)}$
    $\mathcal{H}_1^{(i)} \leftarrow \mathbf{b} + \mathcal{H}_1^{(i-1)}$
    $\mathcal{H}_2^{(i)} \leftarrow \mathbf{c} + \mathcal{H}_2^{(i-1)}$
    $\mathcal{H}_3^{(i)} \leftarrow \mathbf{d} + \mathcal{H}_3^{(i-1)}$
    $\mathcal{H}_4^{(i)} \leftarrow \mathbf{e} + \mathcal{H}_4^{(i-1)}$
    $\mathcal{H}_5^{(i)} \leftarrow \mathbf{f} + \mathcal{H}_5^{(i-1)}$
    $\mathcal{H}_6^{(i)} \leftarrow \mathbf{g} + \mathcal{H}_6^{(i-1)}$
    $\mathcal{H}_7^{(i)} \leftarrow \mathbf{h} + \mathcal{H}_7^{(i-1)}$
  end for ▷ End of the loop over i.
end procedure

Table 2: Probability of at least two among n persons having the same birthday.

n	p(n)
10	0.1388
15	0.2816
20	0.4422
21	0.4743
22	0.5059
23	0.5371
24	0.5675
25	0.5971
30	0.7297
35	0.8317
40	0.9029
45	0.9481
50	0.9744
52	0.9811
54	0.9862
56	0.9901
57	0.9916

From Table 2, we see that, if there are 23 people in a room, the probability is slightly more than 50% that two of them have the same birthday. If there are 30, the probability is around 70%. If there are 57 people, the probability is nearly 99%! This might seem surprising; this phenomenon is called the **Birthday Paradox**.

Note that $1+x \approx e^x$. So, $1-x \approx e^{-x}$. Using this approximation in Eqn. (8) on page 59, we get

$$p(n) = 1 - \prod_{i=1}^{n-1}\left(1 - \frac{i}{365}\right) \approx 1 - \prod_{i=1}^{n} e^{\frac{i}{365}} = 1 - e^{\frac{n(n-1)}{2\times 365}} \approx 1 - e^{n^2/730} \tag{9}$$

Let us consider a more general situation. Suppose there are N objects and r persons. Each person selects an object with replacement so that more than one person can select the same object. What is the probability that two people select the same object?

We can use the same argument to get the probability as

$$1 - \prod_{i=1}^{r-1}\left(1 - \frac{i}{N}\right) \approx 1 - e^{r^2/N}. \tag{10}$$

More generally, if there are N objects and two groups of sizes r_1 and r_2, the probability that two persons from different groups pick the same object is approximately

$$1 - e^{-\frac{r_1 r_2}{N}} \tag{11}$$

If $r_1 = r_2 = \sqrt{N}$, then the probability is approximately $1 - e^{-1} \approx 0.6321 > \frac{1}{2}$. If we take $r_1 = r_2 = 2\sqrt{N}$, then the probability is $1 - e^{-2} \approx 0.8647$ which is much higher.

These ideas can be used to find collisions for hash functions. Suppose a hash function h produces an output that is n-bits long. Then, the number of possible values of the hash function is 2^n. We make a list of values of $h(x)$ of length $\sqrt{N} = 2^{n/2}$. If any of the two values among $2^{n/2}$ match, we have found a collision. Here, we are in the situation of $N = 2^n$ objects and $r = 2^{n/2}$ persons. As we saw earlier, the probability that two hash values match is ≈ 0.6321. The probability of finding a match becomes much higher if we take $r = 10\sqrt{N}$.

If the output of the hash function is 60 bits, for example, the above attack has a high chance of finding a collision. We need to make a list of size approximately

$2^{n/2} = 2^{30} \approx 10^9$ and to store them. This is possible on most computers. However, if the hash function outputs 128-bit values, the list should have a size of $2^{64} \approx 10^{20}$ which is large in terms of time and memory needed at present.

6.5 SUMMARY

In this Unit we have discussed the following:

1. What is a cryptographic hash function;
2. What is a compression function;
3. How to construct compression functions from block ciphers using Davies-Meyer, Matyas-Meyer-Oseas and Miyaguchi-Preneel methods;
4. The Merkle-Damgård method for constructing hash functions from compression functions;
5. The working of MD5 and SHA-2 algorithms; and
6. The birthday attack on hash functions.

6.6 SOLUTIONS/ANSWERS

E1) The length of the string is 136. We split up the string into three strings "tohashor", "nottohas" and "h". We add a one to the end of "h1" to get "h1". This is of length nine, so we add 55 zeros to get a block of size 64. The length of the string, 136, in binary is 10001000. The length of 136, written in binary, is 8. So, we add 56 zeros to the left of this string to get $\underbrace{00\ldots00}_{56\text{ zeros}}10001000$ So, we get the three blocks "tohashor", "nottohas", "h1$\underbrace{00\ldots00}_{55\text{ zeros}}$" and "$\underbrace{00\ldots00}_{56\text{ zeros}}10001000$".

[1] E. Biham and A. Shamir, *Differential Cryptanalysis of Snefru, Khafre, REDOC-II, LOKI and Lucifer*, Proc. CRYPTO 91 (J. Feigenbaum, ed.), Springer, 1992, Lecture Notes in Computer Science No. 576, pp. 156–171.

[2] _____, *A Differential Cryptanalysis of the Data Encryption Standard*, Springer-Verlag, 1993.

[3] J. Black, M. Cochran, and T. Highland, *A Study of the MD5 Attacks: Insights and Improvements*, 2006, `http://www.cs.colorado.edu/~jrblack/papers/md5e-full.pdf`.

[4] Don Coppersmith, *The data encryption standard (DES) and its strength against attacks*, Tech. Report RC 18613(81421), IBM T.J. Watson Research Center, December 1992.

[5] W. Diffie and M. E. Hellman, *Exhaustive cryptanalysis of the NBS data encryption standard*, Computer **10** (1977), 74–84.

[6] Hans Dobbertin, *The Status of MD5 After a Recent Attack.*, CryptoBytes **2** (1996), no. 2, 1–6.

[7] Electronic Frontier Foundation, *Cracking des: Secrets of encryption research, wiretap politics and chip design*, O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1998.

[8] D. E. Knuth, *Seminumerical algorithms*, 3rd ed., The Art of Computer Programming, vol. 2, Addison Wesley, Reading MA, 1997.

[9] M. Matsui, *Linear cryptanalysis method for DES cipher*, Advances in Cryptology — Eurocrypt '93 (Berlin) (T. Helleseth, ed.), Lecture Notes in Computer Science, vol. 765, Springer-Verlag, 1994, pp. 386–397.

[10] Ueli M. Maurer, *A universal statistical test for random bit generators*, Proc. CRYPTO 90 (A. J. Menezes and S. A. Vanstone, eds.), Springer-Verlag, 1991, Lecture Notes in Computer Science No. 537, pp. 409–420.

[11] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1997.

[12] R. C. Merkle and M. E. Hellman, *On the security of multiple encryption*, Communications of the ACM **24** (1981), 465–467.

[13] National Bureau of Standards, *Random and pseudo random number generators for cryptographic applications*, Tech. Report Special Publication 800-22, Revision 1, National Bureau of Standards, 2008.

[14] National Bureau of Standard(SHS), *Secure hash standard*, Tech. Report FIPS Publication 180-3, National Bureau of Standards, October 2009.

[15] M. J. B. Robshaw, *Block ciphers*, Tech. Report TR - 601, RSA Laboratories, July 1994.

[16] M.J.B. Robshaw, *Security of RC4*, Tech. Report TR-401, RSA Laboratories, revised July 1995.

[17] Somitra Kumar Sanadhya and Palash Sarkar, *New collision attacks against up to 24-step sha-2*, INDOCRYPT '08: Proceedings of the 9th International Conference on Cryptology in India (Berlin, Heidelberg), Springer-Verlag, 2008, pp. 91–103.

[18] Yu Sasaki, Lei Wang, and Kazumaro Aoki, *Preimage attacks on 41-step sha-256 and 46-step sha-512*, Cryptology ePrint Archive, Report 2009/479, 2009, http://eprint.iacr.org/.

[19] Bruce Schneier, *Applied cryptography (second edition)*, John Wiley & Sons, 1996.

[20] C. E. Shannon, *Communication theory of secrecy systems*, Bell Sys. Tech. J. **28** (1949), 657–715.

[21] Wade Trappe and Lawrence C. Washington, *Introduction to cryptography with coding theory*, second ed., Pearson, 2006.

[22] Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu, *Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD*, Cryptology ePrint Archive, Report 2004/199, 2004, http://eprint.iacr.org/.

[23] Wikipedia, *One-way compression function — Wikipedia, The Free Encyclopedia*, 2010, http://en.wikipedia.org/w/index.php?title=One-way_compression_function&oldid=359260715.

[24] ———, *Rc4 — Wikipedia, The Free Encyclopedia*, 2010, http://en.wikipedia.org/w/index.php?title=RC4&oldid=385372983.