Chapter 6
Hashing and Message Digests

Hash Function

A hash value h is generated by a function H of the form

h = H(M)

where M is a variable-length message and H(M) is the fixed-length hash value. The hash value is appended to the message at the source at a time when the message is assumed or known to be correct. The receiver authenticates that message by recomputing the hash value. Because the hash function itself is not considered to be secret, some means is required to protect the hash value

Requirements for a Hash Function

The purpose of a hash function is to produce a "fingerprint" of a file, message, or other block of data. To be useful for message authentication, a hash function H must have the following properties:

1. H can be applied to a block of data of any size.
2. H produces a fixed-length output.
3. H(x) is relatively easy to compute for any given x, making both hardware and software implementations practical.
4. For any given value h, it is computationally infeasible to find x such that H(x) = h. This is sometimes referred to in the literature as **the one-way property.**
5. For any given block x, it is computationally infeasible to find $y \neq x$ such that H(y) = H(x). This is sometimes referred to as **weak collision resistance**.
6. It is computationally infeasible to find any pair (x, y) such that H(x) = H(y). This is sometimes referred to as **strong collision resistance**.

Cryptographic Hash Function

A cryptographic hash function is a transformation that takes an input and returns a fixed-size string, which is called the hash value. The hash value is a concise representation of the longer message or document from which it was computed. The message digest is a sort of "digital fingerprint" of the larger document. Cryptographic hash functions are used to do message integrity checks and digital signatures in various information security applications, such as authentication and message integrity.

Hash functions are an important type of cryptographic algorithms and are widely used in cryptography such as digital signature, data authentication, e-cash and many other applications. Hash functions are at work in the millions of transactions that take place on the internet every day. The purpose of the use of hash functions in many cryptographic protocols is to ensure their security as well as improve their efficiency. The most widely used hash functions are dedicated hash functions such as MD5 and SHA-1.

A cryptographic hash function should behave as much as possible like a random function while still being deterministic and efficiently computable. A cryptographic hash function is considered insecure if either of the following is computationally feasible:

Finding a (previously unseen) message that matches a given digest

Finding "collisions", wherein two different messages have the same message digest.

An attacker who can do either of these things might, for example, use them to substitute an unauthorized message for an authorized one.

Ideally, it should not even be feasible to find two messages whose digests are substantially similar; nor would one want an attacker to be able to learn anything useful about a message given only its digest. Of course the attacker learns at least one piece of information, the digest itself, which for instance gives the attacker the ability to recognize the same message should it occur again.

A hash function must be able to process an arbitrary-length message into a fixed-length output. This can be achieved by breaking the input up into a series of equal-sized blocks, and operating on them in sequence using a one-way compression function. The compression function can either be specially designed for hashing or be built from a block cipher.

There is no formal definition which captures all of the properties considered desirable for a cryptographic hash function. These properties below are generally considered prerequisites: **Preimage resistant:** given h it should be hard to find any m such that h = hash(m). **Second preimage resistant:** given an input m_1, it should be hard to find another input, m_2 (not equal to m_1) such that $hash(m_1) = hash(m_2)$.

Applications

Message Integrity Verification: Determining whether any changes have been made to a message (or a file), for example, can be accomplished by comparing message digests calculated before, and after, transmission (or any other event).

Password Verification: Passwords are usually not stored in cleartext, for obvious reasons, but instead in digest form. To authenticate a user, the password presented by the user is hashed and compared with the stored hash. This is sometimes referred to as one-way encryption.

Digital Signatures: while generating digital signatures, the message digest is created and it is encrypted with the private key so that the signing process becomes faster.

Massage Digest Algorithms

1. MD 4 (Message Digest 4)

MD 4 Algorithm is a cryptographic hash function developed by Ronald Rivest in 1990. It produces 128 bits (four 32 bits words) message digest and is optimized for 32-bit computers.

Operations

Step 1: MD4 Message Padding

Original Message	1000……………..000	Original length in bits 64 bits

The message to be processed by MD4 computation must be multiple of 512 bits (16 32-bit words). The original message is padded by adding 1 followed by required number of 0s so that the length of the message is 64 bits less than multiple of 512. The remaining 64 bits is used for providing length of the original message i.e. unpadded message.

Step 2: MD4 Message Digest Computation

MD4 processes message in 512 bits (16 32 bits words) each time. At first message digest is initialized to a fixed value and then each stage of the algorithm takes current value of message digest and modifies it using the next block of message. The function producing 128 bits output from the 512 bits block is called **compression function**. Each stage makes three passes with different methods of mangling for each pass. Each word of mangled message digest is added to its pre-stage value to produce post-stage value that becomes the pre-stage value for the next stage.

Each stage starts with 16-words message block and 4-words message digest values. Say message words as m_0, m_1, …m_{15} and message digest words as d_0, d_1, d_2,d_3 with initial values d_0 = 67452301_{16}, d_1 = $efcdab89_{16}$, d_2 = $98badcfe_{16}$, d_3 = 10325476_{16}, these numbers seems to be random but are initialized to the following values in hexadecimal, low-order bytes first 01 23 45 67 89 ab cd ef fe dc ba 98 76 54 32 10.

Each pass modifies d_0, d_1, d_2, d_3 using m_0, m_1, … m_{15} with the following operations.

- **floor(x):** the greatest integer not greater than x.
- **~x:** bitwise complement of 32 bits quantity x.
- **$x \wedge y$/ $x \vee y$/ $x \oplus y$:** bitwise AND/OR/XOR of 32 bits quantities x and y.
- **x + y:** bitwise binary sum of 32 bits quantities x and y with carry discarded.
- **$x \hookleftarrow y$:** called left rotate generates 32 bits quantity by shifting bit position of x to the left by y times. The shifted bits occupy the right position.

Pass 1: This pass uses the **selection function** [$F(x, y, z) = (x \wedge y) \vee (\sim x \wedge z)$] that takes three 32 bits words as input and produced a 32 bits output. This function's output depending upon nth bits of x since it selects nth bit of y if nth bit of x is 1, otherwise it selects nth bit of z. Each of the 16 words of the messages is separately processed using the following relation, where i goes from 0 to 15.

$d_{(-i) \wedge 3} = (d_{(-i) \wedge 3} + F(d_{(1-i) \wedge 3}, d_{(2-i) \wedge 3}, d_{(3-i) \wedge 3}) + m_i) \hookleftarrow S_1(i \wedge 3)$, where $S_1(i) = 3 + 4i$ so $\hookleftarrow$s cycles over the values 3, 7, 11, 15.

The above relation's "$\wedge$ 3" signifies that only the bottom two bits are used since bitwise AND with 11_2 changes last two bits. In our relation $(-i) \wedge 3$ gives us cycle 0, 3, 2, 1, 0, similarly $(1-i) \wedge 3$ gives 1, 0, 3, 2, 1; $(2-i) \wedge 3$ gives 2, 1, 0, 3, 2; and so on. So expanding the above relation we get first few steps as:

$d_0 = (d_0 + F(d_1, d_2, d_3) + m_0) \hookleftarrow 3;$

$d_3 = (d_3 + F(d_0, d_1, d_2) + m_1) \hookleftarrow 7;$

$d_2 = (d_2 + F(d_3, d_0, d_1) + m_2) \hookleftarrow 11;$

$d_1 = (d_1 + F(d_2, d_3, d_0) + m_3) \hookleftarrow 15;$

$d_0 = (d_0 + F(d_1, d_2, d_3) + m_4) \hookleftarrow 3;$ and so on.

Pass 2: This pass uses **majority function** $[G(x, y, z) = (x \wedge y) \vee (x \wedge z) \vee (y \wedge z)]$ that takes three 32 bits words gives a 32 bits output. The function's output of nth bit is 1 if and only if any two of the three input's nth bits are 1. In this pass we introduce one strange constant **floor**$(2^{30}\sqrt{2}) = 5a827999_{16}$. Each of the 16 words of the messages is separately processed, not in order, using the following relation, where i goes from 0 to 15.

$d_{(-i) \wedge 3} = (d_{(-i) \wedge 3} + G(d_{(1-i) \wedge 3}, d_{(2-i) \wedge 3}, d_{(3-i) \wedge 3}) + m_{X(i)} + 5a827999_{16}) \hookleftarrow S_2(i \wedge 3)$, where X(i) is 4-bits number formed by exchanging the low order and high order pairs of bits in 4-bits number **i** [so, X(i) = 4i – 15**floor**(i/4)], and $S_2(0) = 3$, $S_2(1) = 5$, $S_2(2) = 9$, $S_2(3) = 13$, so $\hookleftarrow$s cycles over the values 3, 5, 9, 13.

$d_0 = (d_0 + G(d_1, d_2, d_3) + m_0 + 5a827999_{16}) \hookleftarrow 3;$

$d_3 = (d_3 + G(d_0, d_1, d_2) + m_4 + 5a827999_{16}) \hookleftarrow 5;$

$d_2 = (d_1 + G(d_3, d_0, d_1) + m_8 + 5a827999_{16}) \hookleftarrow 9;$

$d_1 = (d_1 + G(d_2, d_3, d_1) + m_{12} + 5a827999_{16}) \hookleftarrow 13;$

$d_0 = (d_0 + G(d_1, d_2, d_3) + m_1 + 5a827999_{16}) \hookleftarrow 3;$ and so on.

Pass 3: This pass uses the function $[H(x, y, z) = x \oplus y \oplus z]$ that takes three 32 bits words gives a 32 bits output. In this pass a different strange constant **floor**$(2^{30}\sqrt{3}) = 6ed9eba1_{16}$. Each of the 16 words of the messages is separately processed, not in order, using the following relation, where i goes from 0 to 15.

$d_{(-i) \wedge 3} = (d_{(-i) \wedge 3} + H(d_{(1-i) \wedge 3}, d_{(2-i) \wedge 3}, d_{(3-i) \wedge 3}) + m_{R(i)} + 6ed9eba1_{16}) \hookleftarrow S_3(i \wedge 3)$, where X(i) is 4-bits number formed by reversing the order of bits in 4-bits number **i** [so, R(i) = 8i – 12**floor**(i/2) – 6**floor**(i/4) - 3**floor**(i/8)], and $S_3(0) = 3$, $S_3(1) = 9$, $S_3(2) = 11$, $S_3(3) = 15$, so $\hookleftarrow$s cycles over the values 3, 9, 11, 15.

$d_0 = (d_0 + H(d_1, d_2, d_3) + m_0 + 6ed9eba1_{16}) \hookleftarrow 3;$

$d_3 = (d_3 + H(d_0, d_1, d_2) + m_8 + 6ed9eba1_{16}) \hookleftarrow 9;$

$d_2 = (d_1 + H(d_3, d_0, d_1) + m_4 + 6ed9eba1_{16}) \hookleftarrow 11;$

$d_1 = (d_1 + H(d_2, d_3, d_1) + m_{12} + 6ed9eba1_{16}) \hookleftarrow 15;$

$d_0 = (d_0 + H(d_1, d_2, d_3) + m_2 + 6ed9eba1_{16}) \hookleftarrow 3;$ and so on.

MD 5 (Message Digest 5)

MD5 was designed by Ron Rivest in 1991 to replace an earlier hash function, MD4. MD5 is a widely used hash function with a 128-bit hash value. An MD5 hash is typically expressed as a sequence of 32 hexadecimal digits.

Differences Between MD4 and MD5

- A fourth pass has been added.
- Each step now has a unique additive constant (T_i), so there are 64 constants.
- The function G in pass 2 was changed from $((x \wedge y) \vee (x \wedge z) \vee (y \wedge z))$ to $((x \wedge z) \vee (y \wedge \sim z))$ to make g less symmetric.
- Each step now adds in the result of the previous step. This promotes a faster "avalanche effect".
- The order in which input words are accessed in passes 2 and 3 is changed, to make these patterns less like each other.
- The shift amounts in each pass have been approximately optimized, to yield a faster "avalanche effect." The shifts in different passes are distinct.

MD2, MD4, and MD5 Hashes

The 128-bit (16-byte) MD2, MD4, and MD5 hashes (also termed message digests) are typically represented as 32-digit hexadecimal numbers. The following demonstrates a 43-byte ASCII input and the corresponding MD2 hash:

```
MD2("The quick brown fox jumps over the lazy dog")
                              = 03d85a0d629d2c442e987525319fc471
MD4("The quick brown fox jumps over the lazy dog")
                              = 1bee69a46ba811185c194762abaeae90
MD5("The quick brown fox jumps over the lazy dog")
                              = 9e107d9d372bb6826bd81d3542a419d6
```

Even a small change in the message will (with overwhelming probability) result in a completely different hash, due to the avalanche effect. For example, changing d to c:

MD2("The quick brown fox jumps over the lazy cog")
= 6b890c9292668cdbbfda00a4ebf31f05

MD4("The quick brown fox jumps over the lazy cog")
= b86e130ce7028da59e672d56ad0113df

MD5("The quick brown fox jumps over the lazy cog")
= 1055d3e698d289f2af8663725127bd4b

The hash of the zero-length string is:

MD2("")	= 8350e5a3e24c153df2275c9f80692773
MD4("")	= 31d6cfe0d16ae931b73c59d7e0c089c0
MD5("")	= d41d8cd98f00b204e9800998ecf8427e

Secure Hash Standard (SHS)

The Secure Hash Standard (SHS) is a set of cryptographically secure hash algorithms specified by the National Institute of Standards and Technology. The SHS standard specifies a number of Secure Hash Algorithms (SHA), for example SHA-1, SHA-256 and SHA-512.

SHA-1 (Secure Hash Algorithm 1)

When a message of length < 2^{64} bits is input, the SHA produces a 160-bit representation of the message called the message digest. The SHA is designed to have the following properties: it is computationally infeasible to recover a message corresponding to a given message digest, or to find two different messages which produce the same message digest.

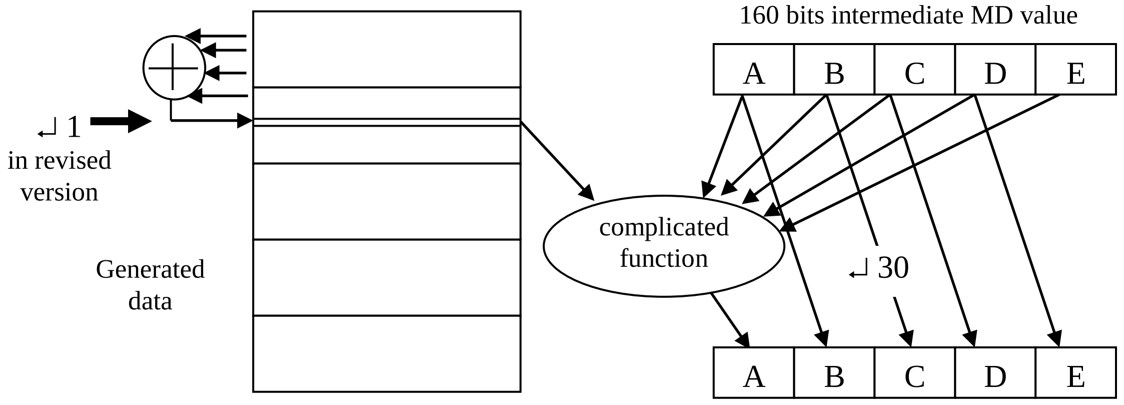
Operations

Step 1: Message Padding

The padding is same as that of MD4 or MD5 except that the length of the message is not longer than 2^{64} bits.

Step 2: SHA-1 Message Digest Computation

SHA-1 processes message in 512 bits each time where it mangles the current message block using pre-stage message digest and sequence of operations. Each word of mangled message digest is added to its pre-stage value to produce post-stage value that becomes the pre-stage value for the next stage. Initial values for five 32 bits message digest words are: A = 67452301_{16}, B = $efcdab89_{16}$, C = $98badcfe_{16}$, D = 10325476_{16}, E = $c3d2e1f0_{16}$.



SHA-1 operations on a 512-bits block (see figure above): At each stage, the 512-bits message block is used to create 5×512-bits chunk. The first 512 bits is the actual message block and the rest chunks are filled 32 bits at a time using a rule: n^{th} word (starts from 16, since 0-15 are for actual message block) is the left rotation (this rotation differs SHA-1 from SHA) of ⊕ of words (n-3), (n-8), (n-14), and (n-16). Now, we have the buffer of 80 32 bits words, lets denote them by W_0, W_1, …, W_{79}. here as from the above discussion we can find n^{th} 32 bits word W_n as: $W_n = (W_{n-3} \oplus W_{n-8} \oplus W_{n-14} \oplus W_{n-16})$

The change of A, B, C, D, and E are done as: for t = 0 to 79, B = old A; C = old B ↲ 30; D = old C; E = old D; A = E + (A ↲ 5) + W_t + K_t + f(t,B,C,D); Here at first A is calculated using old values of A, B, C, D, E (complicated function) and other calculations for B, C, D, and E are done. In the calculation of A, W_t is the t^{th} 32 bits word block and K_t is the constant depending upon the value of t given by the following relations:

K_t = **floor**($2^{30}\sqrt{2}$) = $5a827999_{16}$	if $0 \le t \le 19$.
K_t = **floor**($2^{30}\sqrt{3}$) = $6ed9eba1_{16}$	if $20 \le t \le 39$.
K_t = **floor**($2^{30}\sqrt{5}$) = $8f1bbcdc_{16}$	if $40 \le t \le 59$.

K_t = **floor**($2^{30}\sqrt{10}$) = $ca62c1d6_{16}$ if $60 \le t \le 79$.

Again f(t,B,C,D) is a function that varies according to the following relations:

$f(t,B,C,D) = (B \wedge C) \vee (\sim B \wedge D)$	if $0 \le t \le 19$.
$f(t,B,C,D) = B \oplus C \oplus D$	if $20 \le t \le 39$.
$f(t,B,C,D) = (B \wedge C) \vee (B \wedge D) \vee (C \wedge D)$	if $40 \le t \le 59$.
$f(t,B,C,D) = B \oplus C \oplus D$	if $60 \le t \le 79$.