Cryptography: An Introduction (3rd Edition)

Nigel Smart

Chapter 10 — Edited for CIS 331 (cut short)

CHAPTER 10

Hash Functions and Message Authentication Codes

Chapter Goals

- To understand the properties of cryptographic hash functions.
- To understand how existing deployed hash functions work.
- To examine the workings of message authentication codes.

1. Introduction

In many situations we do not wish to protect the confidentiality of information, we simply wish to ensure the integrity of information. That is we want to guarantee that data has not been tampered with. In this chapter we look at two mechanisms for this, the first using cryptographic hash functions is for when we want to guarantee integrity of information after the application of the function. A cryptographic hash function is usually used as a component of another scheme, since the integrity is not bound to any entity. The other mechanism we look at is the use of a message authentication code. These act like a keyed version of a hash function, they are a symmetric key technique which enables the holders of a symmetric key to agree that only they could have produced the authentication code on a given message.

Hash functions can also be considered as a special type of manipulation detection code, or MDC. For example a hash function can be used to protect the integrity of a large file, as used in some virus protection products. The hash value of the file contents is computed and then either stored in a secure place (e.g. on a floppy in a safe) or the hash value is put in a file of similar values which is then digitally signed to stop future tampering.

Both hash functions and MACs will be used extensively later in other schemes. In particular cryptographic hash functions will be used to compress large messages down to smaller ones to enable efficient digital signature algorithms. Another use of hash functions is to produce, in a deterministic manner, random data from given values. We shall see this application when we build elaborate and "provably secure" encryption schemes later on in the book.

2. Hash Functions

A cryptographic hash function h is a function which takes arbitrary length bit strings as input and produces a fixed length bit string as output, the output is often called a hashcode or hash value. Hash functions are used a lot in computer science, but the crucial difference between a standard hash function and a cryptographic hash function is that a cryptographic hash function should at least have the property of being one-way. In other words given any string y from the range of h, it should be computationally infeasible to find any value x in the domain of h such that

$$h(x) = y.$$

Another way to describe a hash function which has the one-way property is that it is preimage resistant. Given a hash function which produces outputs of n bits, we would like a function for which finding preimages requires $O(2^n)$ time.

In practice we need something more than the one-way property. A hash function is called collision resistant if it is infeasible to find two distinct values x and x' such that

$$h(x) = h(x').$$

It is harder to construct collision resistant hash functions than one-way hash functions due to the birthday paradox. To find a collision of a hash function f, we can keep computing

$$f(x_1), f(x_2), f(x_3), \ldots$$

until we get a collision. If the function has an output size of n bits then we expect to find a collision after $O(2^{n/2})$ iterations. This should be compared with the number of steps needed to find a preimage, which should be $O(2^n)$ for a well-designed hash function. Hence to achieve a security level of 80 bits for a collision resistant hash function we need roughly 160 bits of output.

But still that is not enough; a cryptographic hash function should also be second preimage resistant. This is the property that given m it should be hard to find an $m' \neq m$ with $h(m') = h(m)$. Whilst this may look like collision resistance, it is actually related more to preimage resistance. In particular a cryptographic hash function with n-bit outputs should require $O(2^n)$ operations before one can find a second preimage.

In summary a cryptographic hash function needs to satisfy the following three properties:

(1) **Preimage Resistant:** It should be hard to find a message with a given hash value.
(2) **Collision Resistant:** It should be hard to find two messages with the same hash value.
(3) **Second Preimage Resistant:** Given one message it should be hard to find another message with the same hash value.

But how are these properties related. We can relate these properties using reductions.

LEMMA 10.1. *Assuming a function is preimage resistant* for every element of the range of h *is a weaker assumption than assuming it either collision resistant or second preimage resistant.*

PROOF. Suppose h is a function and let $\mathcal{O}$ denote an oracle which on input of y finds an x such that $h(x) = y$, i.e. $\mathcal{O}$ is an oracle which breaks the preimage resistance of the function h.

Using $\mathcal{O}$ we can then find a collision in h by pulling x at random and then computing $y = h(x)$. Passing y to the oracle $\mathcal{O}$ will produce a value x' such that $y = h(x')$. Since h is assumed to have infinite domain, it is unlikely that we have $x = x'$. Hence, we have found a collision in h.

A similar argument applies to breaking the second preimage resistance of h. □

However, one can construct hash functions which are collision resistant but are not one-way for some of the range of h. As an example, let $g(x)$ denote a collision resistant hash function with outputs of bit length n. Now define a new hash function $h(x)$ with output size $n+1$ bits as follows:

$$h(x) = \begin{cases} 0\|x & \text{If } |x| = n, \\ 1\|g(x) & \text{Otherwise.} \end{cases}$$

The function $h(x)$ is clearly collision resistant, as we have assumed $g(x)$ is collision resistant. But the function $h(x)$ is not preimage resistant as one can invert it on any value in the range which starts with a zero bit. So even though we can invert the function $h(x)$ on some of its input we are unable to find collisions.

LEMMA 10.2. *Assuming a function is second preimage resistant is a weaker assumption than assuming it is collision resistant.*

PROOF. Assume we are given an oracle $\mathcal{O}$ which on input of x will find x' such that $x \neq x'$ and $h(x) = h(x')$. We can clearly use $\mathcal{O}$ to find a collision in h by choosing x at random. □

3. Designing Hash Functions

To be effectively collision free a hash value should be at least 128 bits long, for applications with low security, but preferably its output should be 160 bits long. However, the input size should be bit strings of (virtually) infinite length. In practice designing functions of infinite domain is hard, hence usually one builds a so called compression function which maps bits strings of length s into bit strings of length n, for $s > n$, and then chains this in some way so as to produce a function on an infinite domain. We have seen such a situation before when we considered modes of operation of block ciphers.

We first discuss the most famous chaining method, namely the Merkle–Damgård construction, and then we go on to discuss designs for the compression function.

3.1. Merkle–Damgård Construction. Suppose f is a compression function from s bits to n bits, with $s > n$, which is believed to be collision resistant. We wish to use f to construct a hash function h which takes arbitrary length inputs, and which produces hash codes of n bits in length. The resulting hash function should be collision resistant. The standard way of doing this is to use the Merkle–Damgård construction described in Algorithm 10.1.

Algorithm 10.1: Merkle–Damgård Construction

$l = s - n$
Pad the input message m with zeros so that it is a multiple of l bits in length
Divide the input m into t blocks of l bits long, $m_1, \ldots, m_t$
Set H to be some fixed bit string of length n.
for $i = 1$ **to** t **do**
  $H = f(H \| m_i)$
end
return (H)

In this algorithm the variable H is usually called the *internal state* of the hash function. At each iteration this internal state is updated, by taking the current state and the next message block and applying the compression function. At the end the internal state is output as the result of the hash function.

Algorithm 10.1 describes the basic Merkle–Damgård construction, however it is almost always used with so called *length strengthening*. In this variant the input message is preprocessed by first padding with zero bits to obtain a message which has length a multiple of l bits. Then a final block of l bits is added which encodes the original length of the unpadded message in bits. This means that the construction is limited to hashing messages with length less than 2^l bits.

To see why the strengthening is needed consider a "baby" compression function f which maps bit strings of length 8 into bit strings of length 4 and then apply it to the two messages

$$m_1 = 0b0, \ m_2 = 0b00.$$

Whilst the first message is one bit long and the second message is two bits long, the output of the basic Merkle–Damgård construction will be

$$h(m_1) = f(0b00000000) = h(m_2),$$

i.e. we obtain a collision. However, with the strengthened version we obtain the following hash values in our baby example

$$\begin{aligned} h(m_1) &= f(f(0b00000000) \| 0b0001), \\ h(m_2) &= f(f(0b00000000) \| 0b0010). \end{aligned}$$

These last two values will be different unless we just happen to have found a collision in f.

Another form of length strengthening is to add a single one bit onto the data to signal the end of a message, pad with zeros, and then apply the hash function. Our baby example in this case would become

$$\begin{aligned} h(m_1) &= f(0b01000000), \\ h(m_2) &= f(0b00100000). \end{aligned}$$

Yet another form is to combine this with the previous form of length strengthening, so as to obtain

$$\begin{aligned} h(m_1) &= f(f(0b01000000)\|0b0001), \\ h(m_2) &= f(f(0b00100000)\|0b0010). \end{aligned}$$

3.2. The MD4 Family. A basic design principle when designing a compression function is that its output should produce an avalanche affect, in other words a small change in the input produces a large and unpredictable change in the output. This is needed so that a signature on a cheque for 30 pounds cannot be altered into a signature on a cheque for 30 000 pounds, or vice versa. This design principle is typified in the MD4 family which we shall now describe.

Several hash functions are widely used, they are all iterative in nature. The three most widely deployed are MD5, RIPEMD-160 and SHA-1. The MD5 algorithm produces outputs of 128 bits in size, whilst RIPEMD-160 and SHA-1 both produce outputs of 160 bits in length. Recently NIST has proposed a new set of hash functions called SHA-256, SHA-384 and SHA-512 having outputs of 256, 384 and 512 bits respectively, collectively these algorithms are called SHA-2. All of these hash functions are derived from an earlier simpler algorithm called MD4.

The seven main algorithms in the MD4 family are

- **MD4**: This has 3 rounds of 16 steps and an output bitlength of 128 bits.
- **MD5**: This has 4 rounds of 16 steps and an output bitlength of 128 bits.
- **SHA-1**: This has 4 rounds of 20 steps and an output bitlength of 160 bits.
- **RIPEMD-160**: This has 5 rounds of 16 steps and an output bitlength of 160 bits.
- **SHA-256**: This has 64 rounds of single steps and an output bitlength of 256 bits.
- **SHA-384**: This is identical to SHA-512 except the output is truncated to 384 bits.
- **SHA-512**: This has 80 rounds of single steps and an output bitlength of 512 bits.

We discuss MD4 and SHA-1 in detail, the others are just more complicated versions of MD4, which we leave to the interested reader to look up in the literature.

In recent years a number of weaknesses have been found in almost all of the early hash functions in the MD4 family, for example MD4, MD5 and SHA-1. Hence, it is wise to move all application to use the SHA-2 algorithms.

3.3. MD4. In MD4 there are three bit-wise functions of three 32-bit variables

$$\begin{aligned} f(u,v,w) &= (u \wedge v) \vee ((\neg u) \wedge w), \\ g(u,v,w) &= (u \wedge v) \vee (u \wedge w) \vee (v \wedge w), \\ h(u,v,w) &= u \oplus v \oplus w. \end{aligned}$$

Throughout the algorithm we maintain a current hash state

$$(H_1, H_2, H_3, H_4)$$

of four 32-bit values initialized with a fixed initial value,

$$\begin{aligned} H_1 &= \texttt{0x67452301}, \\ H_2 &= \texttt{0xEFCDAB89}, \\ H_3 &= \texttt{0x98BADCFE}, \\ H_4 &= \texttt{0x10325476}. \end{aligned}$$

There are various fixed constants (y_i, z_i, s_i), which depend on each round. We have

$$y_j = \begin{cases} 0 & 0 \leq j \leq 15, \\ \texttt{0x5A827999} & 16 \leq j \leq 31, \\ \texttt{0x6ED9EBA1} & 32 \leq j \leq 47, \end{cases}$$

and the values of z_i and s_i are given by following arrays,

$$\begin{aligned} z_{0\ldots15} &= [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15], \\ z_{16\ldots31} &= [0,4,8,12,1,5,9,13,2,6,10,14,3,7,11,15], \\ z_{32\ldots47} &= [0,8,4,12,2,10,6,14,1,9,5,13,3,11,7,15], \\ s_{0\ldots15} &= [3,7,11,19,3,7,11,19,3,7,11,19,3,7,11,19], \\ s_{16\ldots31} &= [3,5,9,13,3,5,9,13,3,5,9,13,3,5,9,13], \\ s_{32\ldots47} &= [3,9,11,15,3,9,11,15,3,9,11,15,3,9,11,15]. \end{aligned}$$

The data stream is loaded 16 words at a time into X_j for $0 \leq j < 16$. The length strengthening method used is to first append a one bit to the message, to signal its end and then to pad with zeros to a multiple of the block length. Finally the number of bits of the message is added as a seperate final block.

We then execute the steps in Algorithm 10.2 for each 16 words entered from the data stream.

Algorithm 10.2: MD4 Overview

$(A, B, C, D) = (H_1, H_2, H_3, H_4)$
Execute Round 1
Execute Round 2
Execute Round 3
$(H_1, H_2, H_3, H_4) = (H_1 + A, H_2 + B, H_3 + C, H_4 + D)$

After all data has been read in, the output is the concatenation of the final value of

$$H_1, H_2, H_3, H_4.$$

The details of the rounds are given by Algorithm 10.3 where $\lll$ denotes a bit-wise rotate to the left:

3.4. SHA-1. We use the same bit-wise functions f, g and h as in MD4. For SHA-1 the internal state of the algorithm is a set of five, rather than four, 32-bit values

$$(H_1, H_2, H_3, H_4, H_5).$$

These are assigned with the initial values

$$\begin{aligned} H_1 &= \texttt{0x67452301}, \\ H_2 &= \texttt{0xEFCDAB89}, \\ H_3 &= \texttt{0x98BADCFE}, \\ H_4 &= \texttt{0x10325476}, \\ H_5 &= \texttt{0xC3D2E1F0}, \end{aligned}$$

Algorithm 10.3: Description of the MD4 round functions

Round 1
for $j = 0$ **to** 15 **do**
  $t = A + f(B, C, D) + X_{z_j} + y_j$
  $(A, B, C, D) = (D, t \lll s_j, B, C)$
end
Round 2
for $j = 16$ **to** 31 **do**
  $t = A + g(B, C, D) + X_{z_j} + y_j$
  $(A, B, C, D) = (D, t \lll s_j, B, C)$
end
Round 3
for $j = 32$ **to** 47 **do**
  $t = A + h(B, C, D) + X_{z_j} + y_j$
  $(A, B, C, D) = (D, t \lll s_j, B, C)$
end

We now only define four round constants y_1, y_2, y_3, y_4 via

$$y1 = \texttt{0x5A827999},$$
$$y2 = \texttt{0x6ED9EBA1},$$
$$y3 = \texttt{0x8F1BBCDC},$$
$$y4 = \texttt{0xCA62C1D6}.$$

The data stream is loaded 16 words at a time into X_j for $0 \leq j < 16$, although note the internals of the algorithm uses an expanded version of X_j with indices from 0 to 79. The length strengthening method used is to first append a one bit to the message, to signal its end and then to pad with zeros to a multiple of the block length. Finally the number of bits of the message is added as a seperate final block.

We then execute the steps in Algorithm 10.4 for each 16 words entered from the data stream. The details of the rounds are given by Algorithm 10.5.

Algorithm 10.4: SHA-1 Overview

$(A, B, C, D, E) = (H_1, H_2, H_3, H_4, H_5)$
/* Expansion */
for $j = 16$ **to** 79 **do**
  $X_j = ((X_{j-3} \oplus X_{j-8} \oplus X_{j-14} \oplus X_{j-16}) \lll 1)$
end
Execute Round 1
Execute Round 2
Execute Round 3
Execute Round 4
$(H_1, H_2, H_3, H_4, H_5) = (H_1 + A, H_2 + B, H_3 + C, H_4 + D, H_5 + E)$

Note the one bit left rotation in the expansion step, an earlier algorithm called SHA(now called SHA-0) was initially proposed by NIST which did not include this one bit rotation. This was however soon replaced by the new algorithm SHA-1. It turns out that this single one bit rotation improves the security of the resulting hash function quite a lot.

Algorithm 10.5: Description of the SHA-1 round functions

Round 1
for $j = 0$ **to** 19 **do**
  $t = (A \lll 5) + f(B, C, D) + E + X_j + y_1$
  $(A, B, C, D, E) = (t, A, B \lll 30, C, D)$
end
Round 2
for $j = 20$ **to** 39 **do**
  $t = (A \lll 5) + h(B, C, D) + E + X_j + y_2$
  $(A, B, C, D, E) = (t, A, B \lll 30, C, D)$
end
Round 3
for $j = 40$ **to** 59 **do**
  $t = (A \lll 5) + g(B, C, D) + E + X_j + y_3$
  $(A, B, C, D, E) = (t, A, B \lll 30, C, D)$
end
Round 4
for $j = 60$ **to** 79 **do**
  $t = (A \lll 5) + h(B, C, D) + E + X_j + y_4$
  $(A, B, C, D, E) = (t, A, B \lll 30, C, D)$
end

After all data has been read in, the output is the concatenation of the final value of

$$H_1, H_2, H_3, H_4, H_5.$$

3.5. Hash Functions and Block Ciphers. One can also make a hash function out of an n-bit block cipher, E_K. There are a number of ways of doing this, all of which make use of a constant public initial value IV. Some of the schemes also make use of a function g which maps n-bit inputs to keys.

We first pad the message to be hashed and divide it into blocks

$$x_0, x_1, \ldots, x_t,$$

of size either the block size or key size of the underlying block cipher, the exact choice of size depending on the exact definition of the hash function being created. The output hash value is then the final value of H_i in the following iteration

$$H_0 = IV,$$
$$H_i = f(x_i, H_{i-1}).$$

The exact definition of the function f depends on the scheme being used. We present just three, although others are possible.

- **Matyas–Meyer–Oseas hash**
$$f(x_i, H_{i-1}) = E_{g(H_{i-1})}(x_i) \oplus x_i.$$
- **Davies–Meyer hash**
$$f(x_i, H_{i-1}) = E_{x_i}(H_{i-1}) \oplus H_{i-1}.$$
- **Miyaguchi–Preneel hash**
$$f(x_i, H_{i-1}) = E_{g(H_{i-1})}(x_i) \oplus x_i \oplus H_{i-1}.$$