CHAPTER 5

第5章 散列函数与消息鉴别

5.1 散列函数的概念

密码学中的散列函数又称为哈希函数(Hash 函数)、杂凑函数,它是一种单向密码体制,是一个从明文到密文的不可逆映射,只有加密过程,没有解密过程。散列函数是可将任意长度的输入消息(Message)压缩为某一固定长度的消息摘要(Message Digest,MD)的函数,输出的消息摘要也称为散列码。散列函数的这种单向特性和输出数据长度固定的特征,使得可利用它生成文件或其他数据块的"数字指纹",因此在数据完整性保护、数字签名等领域得到广泛应用。

5.1.1 散列函数的性质

设散列函数为 $h(m)$,具有以下基本特性:

(1) $h(m)$算法公开,不需要密钥。

(2) 具有数据压缩功能,可将任意长度的输入数据转换成一个固定长度的输出。

(3) 对任何给定的 m,$h(m)$易于计算。

除此之外,散列函数还必须满足以下安全性要求:

(1) 具有单向性。给定消息的散列值 $h(m)$,要得到消息 m 在计算上不可行;

(2) 具有弱抗碰撞性(Weak Collision Resistance,也称弱抗冲突性)。对任何给定的消息 m,寻找与 m 不同的消息 m',使得它们的散列值相同,即 $h(m')=h(m)$,在计算上不可行。

(3) 具有强抗碰撞性(Strong Collision Resistance,也称强抗冲突性)。寻找任意两个不同的消息 m 和 m',使得 $h(m)=h(m')$ 在计算上不可行。

所谓散列函数的碰撞是指若两个消息 m 与 m',$m\neq m'$,但它们的散列值相等,即 $h(m)=h(m')$,那么,则把这种情况称为发生了碰撞或冲突。由于输

入消息的长度可以是任意的，但散列函数输出的散列值的长度是固定的，如对于散列值长度为 160 位的散列函数而言，可能的散列值总数为 2^{160}。显然，不同的消息就有可能会产生相同的散列值，即散列函数具有碰撞的不可避免性。但是，散列算法的安全性要求找到一个碰撞在计算上是不可行的。也就是说，要求碰撞是不可预测的，攻击者不能指望对输入消息的预期改变可以得到一个相同的散列值。

5.1.2　散列函数的应用

散列函数的应用主要有以下三个方面。

1. 保证数据的完整性

例如，为保证数据或文件的完整性，可以使用散列函数对数据或文件生成其散列码，并加以安全保存，然后，每当使用数据或文件时，用户可使用散列函数重新计算其散列码，并与保存的散列码进行比较。如果相等，说明数据是完整的，没有经过改动；否则，数据表示已经被篡改过。这样，可发现病毒或入侵者对程序或文档的非授权篡改。

2. 单向数据加密

应用于诸如用户口令加密等场合。将用户口令的散列码存放到一个口令表中，使用时将用户输入的口令进行相应散列运算后，再与口令表中对应用户的口令散列码比较，从而完成口令的有效性验证，这种方式可避免以明文的形式保存用户口令，也无须解密运算，增强了用户口令的安全性，这种单向数据加密至今仍然在许多信息系统中得到广泛使用。

3. 数字签名

将散列函数应用于数字签名可以提高签名的速度，不泄露数字签名所对应的原始消息，而且还可以将消息的签名变换与加密变换分开处理，因此散列函数在数字签名中得到普遍应用。

下面看看应用散列函数如何为消息提供鉴别和完整性保护。假定用户 A 希望给用户 B 发送一条消息 m，如果没有完整性保护，在网络上完全有可能受到第三方的恶意篡改，导致消息 m 的失真。为了提供消息的完整性保护，假设用户 A 与用户 B 共享了一个秘密密钥 k。那么，A 和 B 可以协商采用一个散列函数，对消息 m 作如下计算：

$$d = h(m \| k)$$ （这里 h 为散列函数，“‖”表示两个消息序列的连接）

由此产生一个定长的消息摘要 d。用户 A 发送给用户 B 的消息不仅是 m，而是 $m \| d$。现在，若某攻击者截获消息 m 后，企图将 m 修改为 m'，由于 m'必须连接消息摘要 $h(m' \| k)$ 才是合法的消息，而由于攻击者不知道 k，且散列函数 $h(x)$具有强抗碰撞特性，因此难以通过计算构造出有效的 $h(m' \| k)$。这就意味着攻击者难以修改 m 而不被用户 B 发现。而用户 B 通过计算 $h(m \| k)$，很容易验证 m 是否是真实的。

用图 5.1 可以表示出应用散列函数实现数据完整性保护的模型：

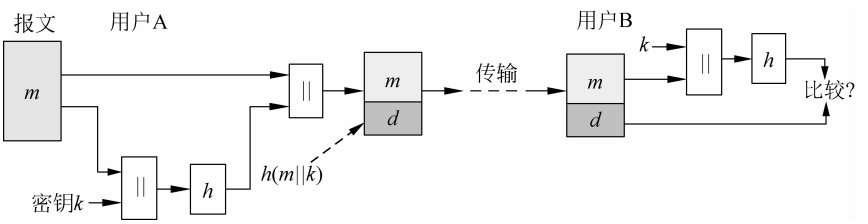


图 5.1　散列函数实现数据完整性保护模型

注：实际应用中，未必一定是如 $h(m \| k)$ 的计算方式，明文与密钥 k 的组合方式因不同的实现可不同，如 HMAC 就给出了另一种计算方式。

5.2 散列函数的构造与设计

5.2.1 迭代型散列函数的一般结构

目前使用的大多数散列函数，如 MD5、SHA 系列等，其结构都是迭代型的，如图 5.2 所示。散列函数将输入的消息 M，分为 t 个分组($m_1, m_2, m_3, \cdots, m_t$)，每组固定长度为 b 位。如果最后个分组的长度不够的话，需对其他填充。通常的填充方法是：在最后一个分组之后进行填充，保证填充后的分组的最后 64 位为整个输入消息 M 的总长度(以位为单位)，然后在中间进行填充。填充的方式有两种：一种是全部填充 0，另一种是填充序列的最高位为 1，其余为 0。这样一来，将使攻击者的攻击更为困难，即攻击者若想成功地产生假冒的消息，就必须保证假冒消息的散列值与原消息的散列值相同，而且假冒消息的长度也要与原消息的长度相等。

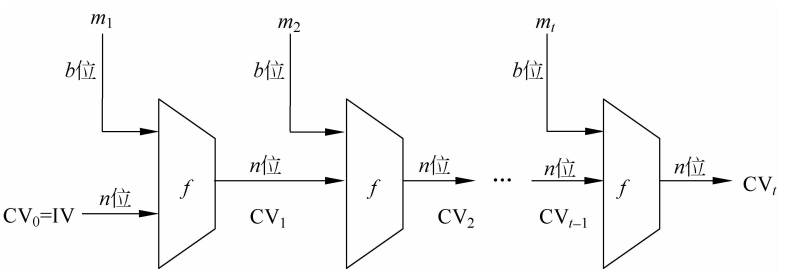


图 5.2 迭代型散列函数的一般结构

算法中重复使用一个函数 f。函数 f 的输入有两项：一项是上一轮(第 $i-1$ 轮)的输出 CV_{i-1}，称为链接变量；另一项是算法在本轮(第 i 轮)b 位的输入分组 m_i。函数 f 的输出为 n 位的 CV_i，它又作为下一轮的输入。算法开始时需要指定一个初始变量 IV，最后一轮输出的链接变量 CV_i 即为最终产生的散列值。通常有 $b>n$，这样函数 f 将一个固定长度为 b 位的输入，变换成较短的 n 位输出，因此函数 f 称为压缩函数(Compression Function)。

整个散列函数的逻辑关系可表示为

$$CV_0 = IV$$
$$CV_i = f(CV_{i-1}, m_i); \quad 1 \leqslant i \leqslant t$$
$$h(M) = CV_t$$

散列算法的核心技术是设计抗碰撞的压缩函数 f，而攻击者对散列算法的攻击重点是 f 的内部结构，由于函数 f 和分组密码一样是由若干轮处理过程组成，所以对函数 f 的攻击需通过对各轮之间的位模式的分析来进行，分析过程常常需要先找出函数 f 的碰撞。由于散列函数的碰撞具有不可避免性，因此在设计函数 f 时就要使其碰撞在计算上是不可行的。

5.2.2　散列函数的设计方法

散列函数的原理比较简单，而且它并不要求可逆，因此，散列函数的设计自由度比较大。散列函数的基本设计方法有：基于公开密钥密码算法的设计、基于对称分组密码算法的设计以及直接设计法。

1. 基于公开密钥密码算法设计散列函数

以 CBC 模式利用公开密钥算法，使用公钥 PK 以及初始变量 IV 对消息分组进行加密，并输出最后一个密文分组 c_t 作为散列函数输出值，如图 5.3 所示。

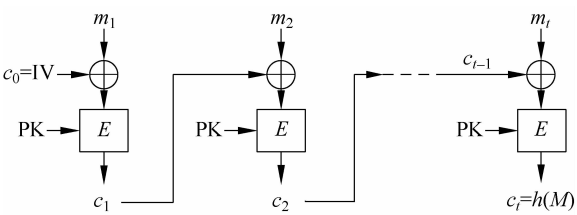


图 5.3　基于公开密钥密码算法 CBC 工作模式的散列函数

对于消息 $M=(m_1,m_2,m_3,\cdots,m_t)$，这时散列函数的逻辑关系可表示为

$$c_0 = \mathrm{IV}$$
$$c_i = E_{\mathrm{Pk}}(m_i \oplus c_{i-1}) \quad ; \quad 1 \leqslant i \leqslant t$$
$$h(M) = c_t$$

如果丢弃用户的私钥 SK，这时产生的散列值将无法解密，它满足了散列函数的单向性要求。

虽然在合理的假设下，可以证明这类散列函数是安全的，由于它的计算效率太低，所以这一类散列函数并没有什么实用价值。

2. 基于对称分组密码算法设计散列函数

通常，可以使用对称密钥分组密码算法的 CBC 模式或 CFB 模式来产生散列值，如图 5.4 和图 5.5 所示。它将使用一个对称密钥 k 及初始变量 IV 加密分组消息，并将最后的密文分组作为散列值输出。这时，如果分组算法是安全的，那么散列函数也将是安全的。

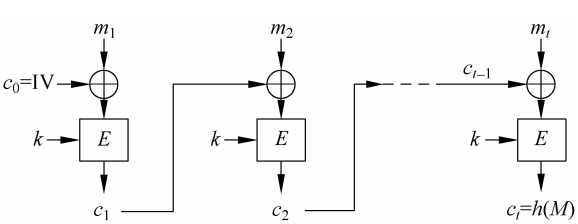


图 5.4　基于对称分组密码算法 CBC 工作模式的散列函数

对于消息 $M=(m_1,m_2,m_3,\cdots,m_t)$，这时 CBC 模式的散列函数逻辑关系可表示为

$$c_0 = \mathrm{IV}$$
$$c_i = E_k(m_i \oplus c_{i-1}) \quad ; \quad 1 \leqslant i \leqslant t$$

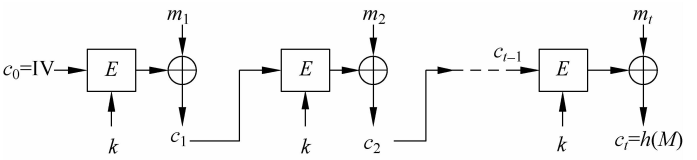


图 5.5　基于对称分组密码算法 CFB 工作模式的散列函数

$$h(M) = c_t$$

CFB 模式的散列函数逻辑关系可表示为

$$c_0 = \mathrm{IV}$$
$$c_i = m_i \oplus E_k(c_{i-1}) \; ; \quad 1 \leqslant i \leqslant t$$
$$h(M) = c_t$$

基于分组密码的 CBC 工作模式和 CFB 工作模式的散列函数中，密钥 k 不能公开。如果密钥 k 公开，则会使得攻击者构造消息碰撞十分容易，下面以 CBC 模式为例进行分析。

如图 5.6 所示，在 CBC 工作模式中，消息 M 被分为 t 个分组($m_1, m_2, m_3, \cdots, m_t$)，然后分别用密钥 k 加密，并对每次输出结果进行链接得到最后的散列值 $h(M)$。如果攻击者知道密钥 k，便可以解密计算得到最后一次加密的输入 x_t。于是攻击者可以将消息 M 的前 $t-1$ 个分组 $m_1, m_2, m_3, \cdots, m_{t-1}$ 任意篡改成 $m'_1, m'_2, m'_3, \cdots, m'_{t-1}$，并计算得到 c'_{t-1}，再构造 $m'_t = c'_{t-1} \oplus x_t$，形成篡改后的消息 $M' = m'_1 \| m'_2 \| m'_3 \| \cdots \| m'_{t-1} \| m'_t$。

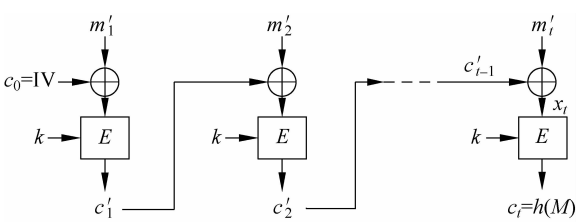


图 5.6　密钥 k 公开时对 CBC 工作模式的散列函数攻击

虽然 $M' \neq M$，但是由于异或运算的性质，在散列函数计算的最后一步得到的 $m'_t \oplus c'_{t-1} c'_{t-1} \oplus x_t \oplus c'_{t-1} = x_t$ 未发生变化，所以仍然有 $c'_t = c_t$，即 $h(M') = h(M)$，这就通过计算造成了碰撞。对于 CFB 模式，也可以使用相同的方法篡改密文造成碰撞。因此，在密钥公开的情况下，基于分组密码的 CBC 工作模式和 CFB 工作模式的 Hash 函数是不安全的，它们甚至不是弱抗碰撞的。在实际使用中密钥 k 必须保密，这种带密钥的散列函数常用于产生消息鉴别码。

3. 直接设计散列函数

这类散列函数并不基于任何假设和密码体制，它是通过直接构造复杂的非线性关系达到单向性要求来设计单向散列函数。这类散列算法典型的有 MD2、MD4、MD5、SHA 系列等散列算法。目前，直接设计散列函数的方法受到了人们的广泛关注和重视，是较普遍采用的一种设计方法，下面以 SHA-1 为例进行介绍。

5.3　安全散列算法 SHA

5.3.1　SHA-1

安全散列算法(Secure Hash Algorithm,SHA)是(美国)联邦信息处理标准(Federal Information Processing Standards Publication,FIPS)所认证的五种安全散列算法,分别是SHA-1、SHA-224、SHA-256、SHA-384 和 SHA-512,由美国国家安全局(NSA)所设计,并由美国国家标准与技术研究院(NIST)发布,后四者有时被并称为 SHA-2。

SHA-1 是指 1995 年 NIST 发布的联邦信息处理标准修订版本 FIPS PUB 180-1 数字签名标准 DSS(Digtial Signature Standard)中使用的散列算法,它能够处理最大长度为 2^{64} 位的输入数据,输出为 160 位的散列函数值,SHA-1 的输出正好适合作为数字签名算法(Digtial Signature Algorithm,DSA)的输入。

SHA-1 已在许多安全协议中广为使用,包括 TLS 和 SSL、PGP、SSH、S/MIME 和 IPSec,曾被视为是更早之前被广为使用的散列函数 MD5 的后继者。

1. 基本操作和元素

1) 逐位逻辑运算

设 X、Y 为两个二进制数,

$X \wedge Y$:表示 X、Y 的逐位逻辑“与”。

$X \vee Y$:表示 X、Y 的逐位逻辑“或”。

$X \oplus Y$:表示 X、Y 的逐位逻辑“异或”。

$\overline{X}$:表示 X 的逐位逻辑“反”。

2) 加法运算

令字长 $w=32$,若 $X \equiv x \bmod 2^w$,$Y \equiv y \bmod 2^w$,定义:$Z \equiv X+Y \bmod 2^w$

3) 移位操作

设 x 为任意二进制数,

$x<<n$ 表示 x 向左移 n 位,右边空出的 n 位用 0 填充。

$x>>n$ 表示 x 向右移 n 位,左边空出的 n 位用 0 填充。

设 X 是一个字(字长为 w),$0 \leqslant n < w$ 是一个整数,定义:

$X<<<n=(X<<n) \vee (X>>w-n)$,即 $<<<$ 是 w 位循环左移位操作。

2. SHA 的散列过程

1) 消息分割与填充

SHA-1 算法是以 512 位的数据块(或称分组)为单位来处理消息的。当消息长度大于 512 位时,需要对消息进行分割与填充。

(1) 将消息 x 按照每块 512 位,分割成消息块 $x_1, x_2, \cdots, x_n$,最后一个块填充为

$$x_n = \overbrace{\underbrace{\cdots\cdots\cdots}_{\text{数据部分}}\ 10\cdots0\ \underbrace{\text{length}(x)}_{64\text{位}}}^{512\text{位}}$$

其中 length(x)表示消息 x 的总长度的二进制形式,其长度为 64 位。如果 x 的长度不

足 64 位时,在其左边补 0,使其达到 64 位。填充部分和数据部分应该用界符"1"来分割开。

(2) 当消息 x 被分割成 512 位的消息块 $x_1, x_2, \cdots, x_n$ 后,每个 x_i 又被分为 16 个字,每个字为 32 位,记作 $M_0, M_1, \cdots, M_{15}$,M_0 是最左边的字。记为 $x_i = M_0^{(i)} M_1^{(i)} M_2^{(i)} \cdots M_{14}^{(i)} M_{15}^{(i)}$。

下面具体分析第 i 个消息块 x_i 的处理过程。

2) 初始化缓冲区

SHA-1 用一个 160 位的缓冲区来存放上一个消息块散列处理后得到的结果,它可表示为 5 个 32 位的数据块(H_0, H_1, H_2, H_3, H_4)。第 i 个消息块 x_i 散列结果可表示为 $H^{(i)} = (H_0^{(i)}, H_1^{(i)}, H_2^{(i)}, H_3^{(i)}, H_4^{(i)})$。

在实现时还需要 5 个 32 位的寄存器(A, B, C, D, E),来保存散列处理的中间结果。

在对整个消息进行散列操作进行之前,这 5 个寄存器的初始化值为

$$A = H_0^{(0)} = 67452301$$
$$B = H_1^{(0)} = \text{EFCDAB89}$$
$$C = H_2^{(0)} = \text{98BADCFE}$$
$$D = H_3^{(0)} = 10325476$$
$$E = H_4^{(0)} = \text{C3D2E1F0}$$

3) 处理第 i 个数据块 x_i

如图 5.7 所示,SHA-1 算法的核心包含 4 个循环模块,每个循环模块由 20 个处理步骤组成,共包含 80 个处理步骤(用 t 表示)。

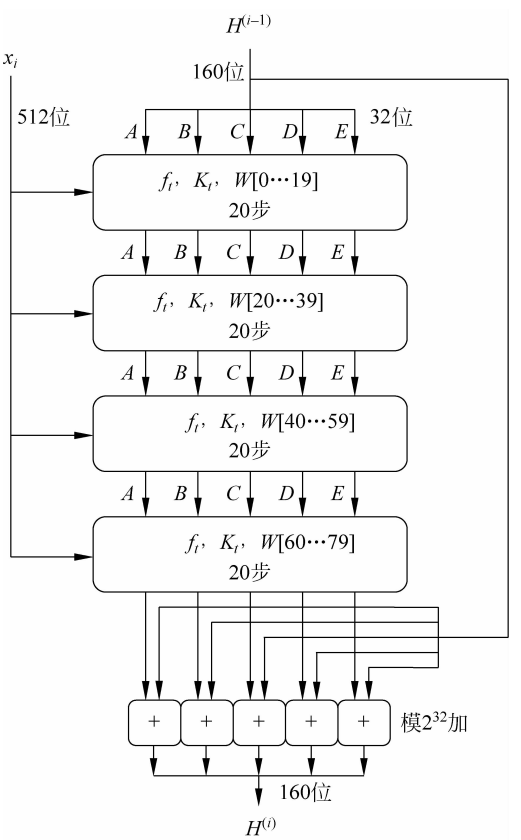


图 5.7 SHA-1 对单个 512 位分组的处理过程

每一循环都以当前正在处理的 512 位数据块(x_i)和 160 位的缓存值 $ABCDE$ 为输入，然后更新缓存 $ABCDE$ 的内容，缓存 $ABCDE$ 的初始值为：$A=H_0^{(i-1)}$，$B=H_1^{(i-1)}$，$C=H_2^{(i-1)}$，$D=H_3^{(i-1)}$，$E=H_4^{(i-1)}$。每个循环还使用一个额外的常数值 K_t，其中 $t(0\leqslant t\leqslant 79)$ 表示 4 个循环总共 80 步的某一步。实际上，80 个常量中只有 4 个不同取值，如表 5.1 所示，其中 $\lfloor x \rfloor$ 为 x 取整。

表 5.1　常数 K_t 的值

步　　数	常量 K_t(十六进制)	常量 K_t(十进制)
$0\leqslant t\leqslant 19$(第一轮)	5A827999	$\lfloor 2^{30}\times\sqrt{2}\rfloor$
$20\leqslant t\leqslant 39$(第二轮)	6ED9EBA1	$\lfloor 2^{30}\times\sqrt{3}\rfloor$
$40\leqslant t\leqslant 59$(第三轮)	8F1BBCDC	$\lfloor 2^{30}\times\sqrt{5}\rfloor$
$60\leqslant t\leqslant 79$(第四轮)	CA62C1D6	$\lfloor 2^{30}\times\sqrt{10}\rfloor$

每一个循环模块在处理步骤中重复使用一个基本逻辑函数，一共使用了 4 个基本逻辑函数。这些基本函数的输入都为 3 个 32 位的字，如表 5.2 所示。

表 5.2　SHA-1 的基本逻辑函数

步　　骤	函　　数	表　达　式
$0\leqslant t\leqslant 19$	$f_t(A,B,C)$	$(A\wedge B)\vee(\overline{B}\wedge C)$
$20\leqslant t\leqslant 39$	$f_t(A,B,C)$	$A\oplus B\oplus C$
$40\leqslant t\leqslant 59$	$f_t(A,B,C)$	$(A\wedge B)\vee(A\wedge C)\vee(B\wedge C)$
$60\leqslant t\leqslant 79$	$f_t(A,B,C)$	$A\oplus B\oplus C$

SHA-1 的 80 个处理步骤中，每个步骤还分别需要 80 个不同的 32 位字 W_t 作为输入参数，其计算方法如图 5.8 所示。

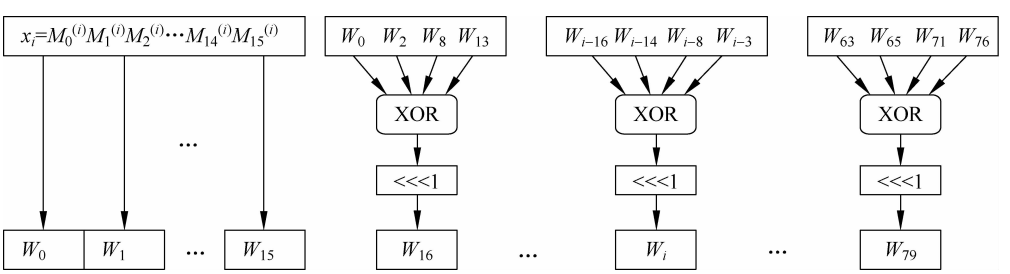


图 5.8　SHA-1 生成字 W_t 的方法

$W_0\sim W_{15}$：直接取自当前数据块 x_i 的 16 个字的值，即

$$W_t=M_t^{(i)};\quad 0\leqslant t\leqslant 15$$

$W_{17}\sim W_{79}$：按下面公式导出(共 64 个)

$$W_t=(W_{t-16}\oplus W_{t-14}\oplus W_{t-8}\oplus W_{t-3})<<<1;\quad 16\leqslant t\leqslant 79$$

也就是说，在前 16 步处理中，W_t 的值等于当前数据块 x_i 中对应字的值，而对余下的 64 步，其值由 4 个前面的 W_t 值异或后，再循环左移 1 位得出。

4) 4 轮循环，80 步操作完成后，保存散列中间结果，再与第一轮的输入相加(模 2^{32})

$$H_0^{(i)}=H_0^{(i-1)}+A,\quad H_1^{(i)}=H_1^{(i-1)}+B,\quad H_2^{(i)}=H_2^{(i-1)}+C,$$

$$H_3^{(i)} = H_3^{(i-1)} + D\ , \quad H_4^{(i)} = H_4^{(i-1)} + E$$

5) 然后,以 $H_0^{(i)}, H_1^{(i)}, H_2^{(i)}, H_3^{(i)}, H_4^{(i)}$ 作为寄存器初值,用于对分组 x_{i+1} 进行散列处理

当对最后一个数据分组 x_n 处理完成后,即得到整个输入消息 x 的 160 位散列值

$$\text{SHA-1}(x) = H^{(n)} = (H_0^{(n)} \parallel H_1^{(n)} \parallel H_2^{(n)} \parallel H_3^{(n)} \parallel H_4^{(n)})$$

3. SHA-1 的压缩操作

SHA-1 压缩函数操作过程,如图 5.9 所示,它就是处理一个 512 位分组的 4 次循环中每一循环的基本压缩操作流程。

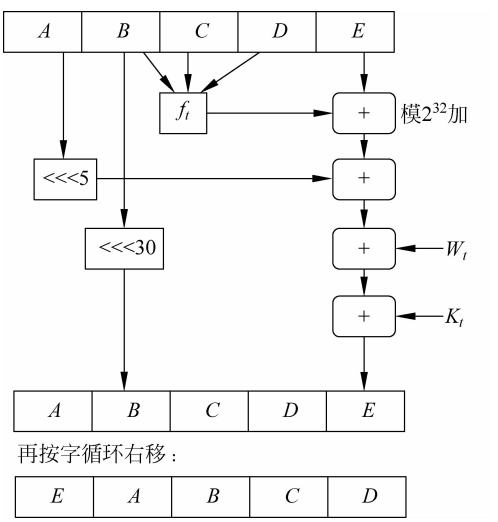


图 5.9 SHA-1 的基本压缩操作

表示为

$$(A,B,C,D,E) \leftarrow (E + f_t(B,C,D) + A <<< 5 + W_t + K_t), A, B <<< 30, C, D$$

即

$$A \leftarrow (E + f_t(B,C,D) + A <<< 5 + W_t + K_t)$$
$$B \leftarrow A$$
$$C \leftarrow B <<< 30$$
$$D \leftarrow C$$
$$E \leftarrow D$$

4. 示例

【例 5.1】 计算字符串“abc”的 SHA-1 散列值。

字符串“abc”的二进制表示为:01100001 01100010 01100011,共有 24 位的长度。按照 SHA-1 的填充要求,应该填充一个“1”(界符)和 423 个“0”,最后有两个字“00000000 00000018”(十六进制),表明原始消息的长度为 24 位。本例中只有一个分组。

五个寄存器的初始值为

$$H_0^{(0)} = 67452301, \quad H_1^{(0)} = \text{EFCDAB89}, \quad H_2^{(0)} = 98\text{BADCFE},$$
$$H_3^{(0)} = 10325476, \quad H_4^{(0)} = \text{C3D2E1F0}$$

进行迭代计算，前 16 个 32 位字的值刚好取自这个分组的所有字，即 W_0=61626380(即 01100001 01100010 01100011 10000000)，$W_1=W_2=\cdots=W_{14}$=00000000，W_{15}=00000018。对 t=0～79，计算得到各个寄存器中的值如表 5.3 所示。

表 5.3　寄存器 *A*、*B*、*C*、*D*、*E* 的值

步数	*A*	*B*	*C*	*D*	*E*
t=0	0116FC33	67452301	7BF36AE2	98BADCFE	10325476
t=1	8990536D	0116FC33	59D148C0	7BF36AE2	98BADCEF
t=2	A1390F08	8990536D	C045BF0C	59D148C0	7B36AE2
t=3	CDD8E11B	A1390F08	626414DB	C045BF0C	59D148C0
t=4	CFD499DE	CDD8E11B	284E43C2	626414DB	C045BF0C
t=5	3FC7CA40	CFD499DE	F3763846	284E43C2	626414DB
t=6	993E30C1	3FC7CA40	B3F52677	F3763846	284E43C2
t=7	9E8C07D4	993E30C1	0FF1F290	B3F52677	F3763846
t=8	4B6AE328	9E8C07D4	664F8C30	0FF1F290	B3F52677
t=9	8351F929	4B6AE328	27A301F5	664F8C30	0FF1F290
t=10	FBDA9E89	8351F929	12DAB8CA	27A301F5	664F8C30
t=11	63188FE4	FBDA9E89	60D47E4A	12DAB8CA	27A301F5
t=12	4607B664	63188FE4	7EF6A7A2	60D47E4A	12DAB8CA
t=13	9128F695	4607B664	18C623F9	7EF6A7A2	60D47E4A
t=14	196BEE77	9128F695	1181ED99	18C623F9	7EF6A7A2
t=15	20BDD62F	196BEE77	644A3DA5	1181ED99	18C623F9
t=16	4E925823	20BDD62F	C65AFB9D	644A3DA5	1181ED99
t=17	82AA6728	4E925823	C82F758B	C65AFB9D	644A3DA5
t=18	DC64901D	82AA6728	D3A49608	C82F758B	C65AFB9D
t=19	FD9E1D7D	DC64901D	20AA99CA	D3A49608	C82F758B
t=20	1A37B0CA	FD9E1D7D	77192407	20AA99CA	D3A49608
t=21	33A23BFC	1A37B0CA	7F67875F	77192407	20AA99CA
t=22	21283486	33A23BFC	868DEC32	7F67875F	77192407
t=23	D541F12D	21283486	0CE88EFF	868DEC32	7F67875F
t=24	C7567DC6	D541F12D	884A0D21	0CE88EFF	868DEC32
t=25	48413BA4	C7567DC6	75507C4B	884A0D21	0CE88EFF
t=26	BE35FBD5	48413BA4	B1D59F71	75507C4B	884A0D21
t=27	4AA84D97	BE35FBD5	12104EE9	B1D59F71	75507C4B
t=28	8370B52E	4AA84D97	6F8D7EF5	12104EE9	B1D59F71
t=29	C5FBAF5D	8370B52E	D2AA1365	6F8D7EF5	12104EE9
t=30	1267B407	C5FBAF5D	A0DC2D4B	D2AA1365	6F8D7EF5
t=31	3B845D33	1267B407	717EEBD7	A0DC2D4B	D2AA1365
t=32	046FAA0A	3B845D33	C499ED01	717EEBD7	A0DC2D4B
t=33	2C0EBC11	046FAA0A	CEE1174C	C499ED01	717EEBD7
t=34	21796AD4	2C0EBC11	811BEA82	CEE1174C	C499ED01
t=35	DCBBB0CB	21796AD4	4B03AF04	811BEA82	CEE1174C
t=36	0F511FD8	DCBBB0CB	085E5AB5	4B03AF04	811BEA82

续表

步数	A	B	C	D	E
$t=37$	DC63973F	0F511FD8	F72EEC32	085E5AB5	4B03AF04
$t=38$	4C986405	DC63973F	03D447F6	F72EEC32	085E5AB5
$t=39$	32DE1CBA	4C986405	F718E5CF	03D447F6	F72EEC32
$t=40$	FC87DEDF	32DE1CBA	53261901	F718E5CF	03D447F6
$t=41$	970A0D5C	FC87DEDF	8CB7872E	53261901	F718E5CF
$t=42$	7F193DC5	970A0D5C	FF21F7B7	8CB7872E	53261901
$t=43$	EE1B1AAF	7F193DC5	25C28357	FF21F7B7	8CB7872E
$t=44$	40F28E09	EE1B1AAF	5FC64F71	25C28357	FF21F7B7
$t=45$	1C51E1F1	40F28E09	FB86C6AB	5FC64F71	25C28357
$t=46$	A01B846C	1C51E1F1	503CA382	FB86C6AB	5FC64F71
$t=47$	BEAD02CA	A01B846C	8714787C	503CA382	FB86C6AB
$t=48$	BAF29337	BEAD02CA	2806E11B	8714787C	503CA382
$t=49$	120731C5	BAF29337	AFAB40B2	2806E11B	8714787C
$t=50$	641DB2CE	120731C5	EEBCE4CD	AFAB40B2	2806E11B
$t=51$	3847AD66	641DB2CE	4481CC71	EEBCE4CD	AFAB40B2
$t=52$	E490436D	3847AD66	99076CB3	4481CC71	EEBCE4CD
$t=53$	27E9F1D8	E490436D	8E11EB59	99076CB3	4481CC71
$t=54$	7B71F76D	27E9F1D8	792410DB	8E11EB59	99076CB3
$t=55$	5E6456AF	7B71F76D	09FA7C76	792410DB	8E11EB59
$t=56$	C846093F	5E6456AF	5EDC7DDB	09FA7C76	792410DB
$t=57$	D262FF50	C846093F	D79915AB	5EDC7DDB	09FA7C76
$t=58$	09D785FD	D262FF50	F211824F	D79915AB	5EDC7DDB
$t=59$	3F52DE5A	09D785FD	3498BFD4	F211824F	D79915AB
$t=60$	D756C147	3F52DE5A	4275E17F	3498BFD4	F211824F
$t=61$	548C9CB2	D756C147	8FD4B796	4275E17F	3498BFD4
$t=62$	B66C929B	548C9CB2	F5D5B051	8FD4B796	4275E17F
$t=63$	6B61C9E1	B66C929B	9523272C	F5D5B051	8FD4B796
$t=64$	19DFA7AC	6B61C9E1	ED9B0082	9523272C	F5D5B051
$t=65$	101655F9	19DFA7AC	5AD87278	ED9B0082	9523272C
$t=66$	0C3DF2B4	101655F9	0677E9EB	5AD87278	ED9B0082
$t=67$	78DD4D2B	0C3DF2B4	4405957E	0677E9EB	5AD87278
$t=68$	497093C0	78DD4D2B	030F7CAD	4405957E	0677E9EB
$t=69$	3F2588C2	497093C0	DE37534A	030F7CAD	4405957E
$t=70$	C199F8C7	3F2588C2	125C24F0	DE37534A	030F7CAD
$t=71$	39859DE7	C199F8C7	8FC96230	125C24F0	DE37534A
$t=72$	EDB42DE4	39859DE7	F0667E31	8FC96230	125C24F0
$t=73$	11793F6F	EDB42DE4	CE616779	F0667E31	8FC96230
$t=74$	5EE76897	11793F6F	3B6D0B79	CE616779	F0667E31
$t=75$	63F7DAB7	5EE76897	C45E4FDB	3B6D0B79	CE616779
$t=76$	A079B7D9	63F7DAB7	D7B9DA25	C45E4FDB	3B6D0B79
$t=77$	860D21CC	A079B7D9	D8FDF6AD	D7B9DA25	C45E4FDB
$t=78$	5738D5E1	860D21CC	681E6DF6	D8FDF6AD	D7B9DA25
$t=79$	42541B35	5738D5E1	21834873	681E6DF6	D8FDF6AD

最后得到结果为

$$H_0 = 67452301 + 42541B35 = A9993E36$$
$$H_1 = EFCDAB89 + 5738D5E1 = 4706816A$$
$$H_2 = 98BADCFE + 21834873 = BA3E2571$$
$$H_3 = 10325476 + 681E6DF6 = 7850C26C$$
$$H_4 = C3D2E1F0 + D8FDF6AD = 9CD0D89D$$

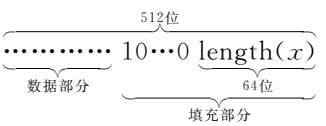
于是：SHA-1("abc")= A9993E36 4706816A BA3E2571 7850C26C 9CD0D89D，共 160 位，20 个字节。

5.3.2　其他 SHA 算法

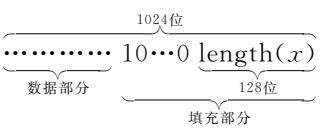
2002 年，NIST 在 FIPS 180-1 的基础上做了修改，发布了推荐的修订版本 FIPS 180-2。在这个标准中，除了 SHA-1 外，还新增了 SHA-256、SHA-384 和 SHA-512 三个散列算法标准，它们的消息摘要长度分别为 256 位、384 位和 512 位，以便与 AES 的使用相匹配。SHA 系列散列算法的基本运算结构很相似。

SHA-1 和 SHA-256 的数据分组都是 512 位，SHA-384 和 SHA-512 的数据分组则是 1024 位。

SHA-1 和 SHA-256 的数据分组、填充格式为



SHA-384 和 SHA-512 的数据分组、填充格式为



1. SHA-256

(1) SHA-256 需要进行 64 步操作，使用了 64 个 32 位字的常数，这些常数定义如下(十六进制表示)：

428a2f98 71374491 b5c0fbcf e9b5dba5 3956c25b 59f111f1 923f82a4 ab1c5ed5
d807aa98 12835b01 243185be 550c7dc3 72be5d74 80deblfe 9bdc06a7 c19bf174
e49b69c1 efbe4786 0fc19dc6 240ca1cc 2de92c6f 4a7484aa 5cb0a9dc 76f988da
983e5152 a831c66d b00327c8 bf597fc7 c6e00bf3 d5a79147 06ca6351 14292967
27b70a85 2e1b2138 4d2c6dfc 53380d13 650a7354 766a0abb 81c2c92e 92722c85
a2bfe8a1 a81a664b c24b8b70 c76c51a3 d192e819 d6990624 f40e3585 106a8070
19a4c116 le376c08 2748774c 34b0bcb5 391c0cb3 4ed8daa4a 5b9cca4f 682e6ff3
748f82ee 78a5636f 84c87814 8cc70208 90befffa a4506ceb bef9a3f7 c67178f2

(2) SHA-256 的初始散列值 $H_0^{(0)}, H_1^{(0)}, \cdots, H_7^{(0)}$ 为 8 个 32 位字：

6a09e667 bb67ae85 3c6ef372 a54ff53a
510e527f 9b05688c 1f83d9ab 5be0cd19

(3) 6 个基本函数为

$$\mathrm{Ch}(x,y,z)=(x\wedge y)\oplus(\bar{x}\wedge z)$$

$$\mathrm{Maj}(x,y,z)=(x\wedge y)\oplus(x\wedge z)\oplus(y\wedge z)$$

$$\sum_{1}^{(256)}(x)=(x<<<2)\oplus(x<<<13)\oplus(x<<<22)$$

$$\sum_{1}^{(256)}(x)=(x<<<6)\oplus(x<<<11)\oplus(x<<<22)$$

$$\sigma_0^{(256)}=(x<<<7)\oplus(x<<<18)\oplus(x>>3)$$

$$\sigma_1^{(256)}=(x<<<17)\oplus(x<<<19)\oplus(x>>10)$$

其中,x,y,z 都是 32 位。

(4) 迭代过程。

设 $X+Y\equiv X+Y \bmod 2^{32}$,对所有消息块做如下操作:

① 分割消息块 $x_i=M_0^{(i)}M_1^{(i)}M_2^{(i)}\cdots M_{14}^{(i)}M_{15}^{(i)}$;

② 计算 W_t

$$W_t=\begin{cases}M_t^{(i)}, & 0\leqslant t\leqslant 15\\ \sigma_1^{(256)}(W_{t-2})+W_{t-7}+\sigma_0^{(256)}(W_{t-15})+W_{t-16}, & 16\leqslant t\leqslant 63\end{cases}$$

③ 用上一轮的散列值结果初始化 8 个工作变量(a,b,c,d,e,f,g,h),令

$$(a,b,c,d,e,f,g,h)=(H_0^{(i-1)},H_1^{(i-1)},H_2^{(i-1)},H_3^{(i-1)},H_4^{(i-1)},H_5^{(i-1)},H_6^{(i-1)},H_7^{(i-1)})$$

④ 对 $t=0\sim63$ 做如下操作:

$$T_1=h+\sum_{1}^{(256)}(e)+\mathrm{Ch}(e,f,g)+K_t^{(256)}+W_t$$

$$T_2=\sum_{0}^{(256)}(a)+\mathrm{Maj}(a,b,c)$$

$$h=g,\quad g=f,\quad f=e,\quad e=d+T_1,\quad d=c,\quad c=b,\quad b=a,\quad a=T_1+T_2$$

⑤ 输出结果再与第一轮的输入相加(模 2^{32}):

$$H_0^{(i)}=H_0^{(i-1)}+a,\quad H_1^{(i)}=H_1^{(i-1)}+b,\quad H_2^{(i)}=H_2^{(i-1)}+c,\quad H_3^{(i)}=H_3^{(i-1)}+d$$

$$H_4^{(i)}=H_4^{(i-1)}+e,\quad H_5^{(i)}=H_5^{(i-1)}+f,\quad H_6^{(i)}=H_6^{(i-1)}+g,\quad H_7^{(i)}=H_7^{(i-1)}+h$$

(5) 当对最后一个数据分组 x_n 处理完成后,即得到整个输入消息 x 的散列值:

$$\mathrm{SHA}_{256}(x)=H_0^{(n)}\|H_1^{(n)}\|H_2^{(n)}\|H_3^{(n)}\|H_4^{(n)}\|H_5^{(n)}\|H_6^{(n)}\|H_7^{(n)}$$

2. SHA-384 和 SHA-512

(1) SHA-384 和 SHA-512 都需要进行 80 步操作,具有相同的 80 个常数:

428a2f98d728ae22　7137449123ef65cd　b5c0fbcfec4d3b2f　e9b5dba58189dbbc
3956c25bf348b538　59f111f1b605d019　923f82a4af194f9b　ab1c5ed5da6d8118
d807aa98a3030242　12835b0145706fbe　243185be4ee4b28c　550c7dc3d5ffb4e2
72be5b74f27b896f　80deb1fe3b1696b1　9bdc06a725c71235　c19bf174cf692694
e49b69c19ef14ad2　efbe4786384f25e3　0fc19dc68b8cd5b5　240ca1cc77ac9c65
2de92c6f592b0275　4a7484aa6ea6e483　5cb0a9dcbd41fbd4　76f988da831153b5
983e5152ee66dfab　a831c66d2db43210　b00327c898fb213f　bf597fc7beef0ee4

c6e00bf33da88fc2	d5a79147930aa725	06ca6351e003826f	142929670a0e6e70
27b70a8546d22ffc	2e1b21385c26c926	4d2c6dfc5ac42aed	53380d139d95b3df
650a73548baf63de	766a0abb3c77b2a8	81c2c92e47edaee6	92722c851482353b
a2bfe8a14cf10364	a81a664bbc423001	c24b8b70d0f89791	c76c51a30654be30
d192e819d6ef5218	d69906245565a910	f40e35855771202a	l06aa07032bbd1b8
19a4c116b8d2d0c8	1e376c085141ab53	2748774cdf8eeb99	34b0bcb5e19b48a8
391c0cb3c5c95a63	44ed8aa4ae3418acb	5b9cca4f7763e373	682e6ff3d6b2b8a3
748f82ee5defb2fc	78a5636f43172f60	84c87814a1f0ab72	8cc702081a6439ec
90befffa23631e28	a4506cebde82bde9	bef9a3f7b2c67915	c67178f2e372532b
ca273eceea26619c	d186b8c721c0c207	eada7dd6cde0eb1e	f57d4f7fee6ed178
06f067aa72176fba	0a637dc5a2c898a6	113f9804bef90dae	1b710b35131c471b
28db77f523047d84	32caab7b40c72493	3c9ebe0a15c9bebc	431d67c49c100d4c
4cc5d4becb3e42b6	597f299cfc657e2a	5fcb6fab3ad6faec	6c44198c4a475817

(2) 初始散列值 $H_0^{(0)}, H_1^{(0)}, \cdots, H_7^{(0)}$ 为 8 个 64 位字。

SHA-384 的 8 个初始散列值为

cbbb9d5dc1059ed8，629a292a367cd507，9159015a3070dd17，152fecd8f70e5939，67332667ffc00b31，8eb44a8768581511，db0c2e0d64f98fa7，47b5481dbefa4fa4

SHA-512 的 8 个初始散列值为

6a09e667f3bcc908，bb67ae8584caa73b，3c6ef372fe94f82b，a54ff53a5f1d36f1，510e527fade682d1，9b05688c2b3e6c1f，1f83d9abfb41bd6b，5be0cd19137e2179

(3) SHA-384、SHA-512 具有相同的 6 个基本函数：

$$\mathrm{Ch}(x,y,z) = (x \wedge y) \oplus (\bar{x} \wedge z)$$

$$\mathrm{Maj}(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$$

$$\sum_0^{\binom{384}{512}}(x) = (x <<< 28) \oplus (x <<< 34) \oplus (x <<< 39)$$

$$\sum_1^{\binom{384}{512}}(x) = (x <<< 14) \oplus (x <<< 18) \oplus (x <<< 41)$$

$$\sigma_0^{\binom{384}{512}} = (x <<< 1) \oplus (x <<< 8) \oplus (x >> 7)$$

$$\sigma_1^{\binom{384}{512}} = (x <<< 19) \oplus (x <<< 61) \oplus (x >> 6)$$

其中，x、y、z 都是 64 位。

(4) SHA-384 和 SHA-512 具有相同的迭代算法。

设 $X+Y \equiv X+Y \bmod 2^{64}$，对所有消息块做如下操作：

① 分割消息块 $x_i = M_0^{(i)} M_1^{(i)} M_2^{(i)} \cdots M_{14}^{(i)} M_{15}^{(i)}$；

② 计算 W_t

$$W_t = \begin{cases} M_t^{(i)}, & 0 \leqslant t \leqslant 15 \\ \sigma_1^{\binom{384}{512}}(W_{t-2}) + W_{t-7} + \sigma_0^{\binom{384}{512}}(W_{t-15}) + W_{t-16}, & 16 \leqslant t \leqslant 63 \end{cases}$$

③ 用上一轮的散列值结果初始化 8 个工作变量(a,b,c,d,e,f,g,h),令

$(a,b,c,d,e,f,g,h)=(H_0^{(i-1)},H_1^{(i-1)},H_2^{(i-1)},H_3^{(i-1)},H_4^{(i-1)},H_5^{(i-1)},H_6^{(i-1)},H_7^{(i-1)})$

④ 对 $t=0\sim79$ 做如下操作:

$$T_1=h+\sum_{1}^{\binom{384}{512}}(e)+\mathrm{Ch}(e,f,g)+K\binom{384}{512}_t+W_t$$

$$T_2=\sum_{0}^{\binom{384}{512}}(a)+\mathrm{Maj}(a,b,c)$$

$$h=g,\quad g=f,\quad f=e,\quad e=d+T_1,\quad d=c,\quad c=b,\quad b=a,\quad a=T_1+T_2$$

⑤ 输出结果再与第一轮的输入相加(模 2^{64}):

$$H_0^{(i)}=H_0^{(i-1)}+a,\quad H_1^{(i)}=H_1^{(i-1)}+b,\quad H_2^{(i)}=H_2^{(i-1)}+c,\quad H_3^{(i)}=H_3^{(i-1)}+d$$

$$H_4^{(i)}=H_4^{(i-1)}+e,\quad H_5^{(i)}=H_5^{(i-1)}+f,\quad H_6^{(i)}=H_6^{(i-1)}+g,\quad H_7^{(i)}=H_7^{(i-1)}+h$$

(5) 当对最后一个数据分组 x_n 处理完成后,即得到整个输入消息 x 的散列值:

$$\mathrm{SHA}_{384}(x)=H_0^{(n)}\|H_1^{(n)}\|H_2^{(n)}\|H_3^{(n)}\|H_4^{(n)}\|H_5^{(n)}$$

$$\mathrm{SHA}_{512}(x)=H_0^{(n)}\|H_1^{(n)}\|H_2^{(n)}\|H_3^{(n)}\|H_4^{(n)}\|H_5^{(n)}\|H_6^{(n)}\|H_7^{(n)}$$

SHA-384 的处理与 SHA-512 的处理基本一致,只是有两个方面的不同:

① 初始散列值的设定不同(如前所述);

② SHA-384 的散列值取自于最终输出的 512 位散列值的左边的 384 位。

3. SHA-1、SHA-256、SHA-384 和 SHA-512 的比较

SHA-1、SHA-256、SHA-384 和 SHA-512 这些散列函数的主要差别在于填充方法、字长、初始常数、基本函数等部分,它们的基本运算结构大体是相同的。表 5.4 是它们基本参数的比较。

表 5.4 SHA-1、SHA-256、SHA-384 和 SHA-512 参数比较(长度单位为位)

算法	消息长度	分组长度	字长	散列值长	使用常数	基本函数	处理步骤
SHA-1	$<2^{64}$	512	32	160	4×32	4	80
SHA-256	$<2^{64}$	512	32	256	64×32	6	64
SHA-384	$<2^{128}$	1024	64	384	80×32	6	80
SHA-512	$<2^{128}$	1024	64	512	80×32	6	80

5.4 对散列函数的攻击

对散列函数的攻击是指攻击者寻找一对产生碰撞的消息的过程。评价散列函数的有效方法就是看一个攻击者找到一对产生碰撞的消息所花的代价有多高。遵循柯克霍夫斯(Kerckhoffs)原则,在讨论散列函数的安全性时,都假设攻击者已经知道了散列函数的算法。

目前对于散列函数的攻击方法可以分为两类:

第一类称为穷举攻击(或暴力攻击),它能对任何类型的散列函数进行攻击,其中最典型

的方法就是“生日攻击”。采用这类攻击的攻击者将产生许多的消息，计算其散列值，并进行比较。当消息足够多时，根据概率论的有关结论，消息将以某种较大的概率发生碰撞，这时可以认为散列函数已经被攻破。这种攻击可行性的关键在于究竟需要多少消息的输入才能使发生碰撞的概率足够大。如果消息数目大到使计算上是不可行的，那么这种攻击就是不可行的；否则，可以认为该散列函数是不安全的。

第二类称为密码分析法，这类攻击方法依赖于对散列函数的结构和代数性质分析，采用针对散列函数弱性质的方法进行攻击。这类攻击方法有中间相遇攻击、修正分组攻击和差分分析等等。另外，使用了其他密码算法构造的散列函数，还可能因为所使用的密码算法本身存在的弱点而引起攻击。例如，针对 DES 密码算法的一些安全性弱点，如互补性、弱密钥与半弱密钥等，都可用来攻击基于 DES 构造的散列函数。

在 2004 年美国密码会议上，我国山东大学的王小云教授，公开了自己多年来对散列函数的系列研究结果。她给出了计算 MD5 等散列算法碰撞的方法，以及 SHA-1 的理论破解，并证明了 160 位 SHA-1，只需要大约 2^{69} 次计算就能找出来，而理论值是 2^{80} 次。而她的寻找 MD5 碰撞的方法是极端高效的。这项研究成果引发了密码学界的轰然大波，著名密码学家 Rivest 说：“MD5 函数十几年来经受住了众多密码学专家的攻击，而王小云教授却成功地破解了它，这实在是一种令人印象极深的卓越成就，是高水平的世界级研究成果。”

第二类方法涉及的知识已超出本书内容范围，下面重点介绍对散列函数的“生日攻击”。

5.4.1　生日悖论

下面来考虑这样一个有趣的问题：在一个教室中最少应有多少学生才使得至少有两个学生的生日在同一天的概率大于 0.5？计算与此相关的概率被称为生日悖论问题。

首先定义下述概率：设有 k 个整数，每一个都在 $1\sim n$ 之间等概率取值。则 k 个整数中至少有两个取值相同的概率为 $P(n,k)$。因而生日悖论就是求使 $n=365$ 时，$P(n,k)>0.5$ 的最小 k 值。为此首先考虑 k 个数中任意两个取值都不相同的概率，记为 $Q(365,k)$。如果 $k>365$，则使得任意两个数都不相同是不可能的，因此假设 $k<365$。k 个数中任意两个都不相同的所有取值方式数量为

$$365\times 364\times\cdots\times(365-k+1)=\frac{365!}{(365-k)!}$$

即第 1 个数可从 365 个值中任取一个，第 2 个数可从剩余的 364 个值中任取一个，以此类推，最后一个数可从 $365-k+1$ 个值中任取一个。而 k 个数任意取两个值的方式总数为 356^k。所以可得

$$Q(365,k)=\frac{365!}{(365-k)!365^k}$$

$$P(365,k)=1-Q(365,k)=1-\frac{365!}{(365-k)!365^k}$$

当 $k=23$ 时，$P(365,k)=0.5073$，即生日悖论问题只需要 23 人，人数如此之少！而且，如果 k 取 100 时，$P(365,k)=0.999\,999\,7$，近似于 1 的概率！之所以称这一问题是悖论，是因为当人数 k 给定时，得到的至少有两个人的生日相同的概率比想象的要大得多。从引起逻辑矛盾的角度来说生日问题并不是一种悖论，只是从这个数学事实与一般直觉相抵触的

意义上,它才称得上是一个悖论。因为大多数人会认为,23 人中有 2 人生日相同的概率应该远远小于 0.5。

可以将生日问题归纳为这样一个问题:设一个在 1～n 之间均匀分布的整数型随机变量,若使该变量的 k 个取值中至少有两个取值相同的概率大于 0.5,则 k 至少多大?

根据上述分析,可得

$$P(n,k) = 1 - Q(n,k) = 1 - \frac{n!}{(n-k)!n^k} \tag{5.1}$$

令 $P(n,k)>0.5$,可以解得

$$k = 1.18\sqrt{n} \approx \sqrt{n} \tag{5.2}$$

与此类似的一个相交集合问题是:设取整数的随机变量,它服从 1～n 的均匀分布,另有两个含有 k 个如此随机变量的集合,若使两个集合中至少有一个元素取值相同的概率大于 0.5,则 k 至少多大?

用以上类似分析方法,可得

$$k \approx \sqrt{n} \tag{5.3}$$

对此类问题分析感兴趣的读者还可参考文献[2],此处不再赘述。

5.4.2 生日攻击

与散列函数相关的类似问题可表述如下:给定一个散列函数 h 的输出长度为 m 位,共有 2^m 个可能的散列值输出,如果让散列函数 h 接收 k 个随机输入产生集合 X,再使用另外 k 个随机输入产生集合 Y,问 k 必须为多大才能使两个集合产生相同散列值输出的概率大于 0.5?

这时,$n = 2^m$,由式(5.3)有

$$k \approx \sqrt{2^m} = 2^{m/2}$$

这种寻找散列函数 h 的具有相同输出的两个任意输入的攻击方式称为生日攻击。

下面举一个简单的示例来说明这种针对散列函数的生日攻击过程。

还是假设 Alice 是一家计算机公司的总经理,要从 Bob 的公司购买一批计算机。经过双方协商,确定了 5000 元/台的价格,于是 Bob 发来合同的电子文本征得 Alice 的同意,Alice 确认后计算出这一合同文本的散列值,用自己的私钥进行数字签名并发回给 Bob,以此作为 Alice 对合同样本的确认。

但是,Bob 在发给 Alice 合同样本前,首先写好一份正确的合同,然后标出这份合同中无关紧要的地方。由于合同总是由许许多多的句子构成的,而这些句子往往可以有很多不同的表达方式,所以一份合同总可以有很多种不同的说法,却都能表达同样的意思。现在 Bob 只要把这些意思相同的合同都列出来作为一组,然后再把每一份合同当中标明的价格从 5000 元/台改为 10 000 元/台,并且把修改过的合同也集中起来作为另外一组,这样他的手中就有了两组合同:一组的价格条款都是 5000 元/台的,而另一组的价格都是 10 000 元/台。然后,Bob 只要把这两组合同的散列值都计算一遍,从当中挑出一对散列值相同的,把这一对当中的那份写明 5000 元/台的合同作为合同样本给 Alice,并交由 Alice 进行签名,而自己则偷偷把那份 10 000 元/台的合同藏起来,以便在将来进行欺诈。

从生日攻击的理论上来讲,如果假设上述事例使用的散列函数输出为 64 位,那么 Bob

只要找到合同中 32 个无关紧要的地方，来分别构成 5000 元/台和 10 000 元/台两组合同，有 0.5 以上的概率能在这两组合同中找到碰撞，来实现他的诈骗行为。

需要指出的是，就典型散列算法 MD5 和 SHA-1 而言，除去常见的穷举攻击或生日攻击，我国密码学家已经发现了效率更高的使 MD5 和 SHA-1 产生碰撞的办法，其研究成果引起了密码学界的高度关注。

5.5　消息鉴别

在消息传递的过程中，需要考虑两个方面问题：一方面，为了可以抵抗窃听等被动攻击，需要对传输的消息进行加密保护；另一方面，就是使用消息鉴别来防止攻击者对系统进行主动攻击，如伪造、篡改消息等。消息鉴别是一个过程，用于验证接收消息的真实性（的确是由它所声称的实体发来的）和完整性（未被篡改、插入、删除），同时还用于验证消息的顺序性和时间性（未被重排、重放、延迟等）。除此之外，在考虑信息安全时还需考虑业务的不可否认性，也称抗抵赖性，即防止通信双方中的某一方对所传输消息的否认或抵赖。实现消息的不可否认性可采用数字签名，数字签名也是一种鉴别技术，它可用于对抗主动攻击。消息鉴别对于开放的网络中的各种信息系统的安全性具有重要作用。

例如，发送者通过一个公开的信道将消息发送给接收者的过程，由于信道是公开的、不安全的，攻击者可以在传送过程中进行窃听，或者是篡改消息内容，甚至是伪造消息发送给接收者。因此，发送者应当在发送之前，使用消息鉴别，对待发送的信息生成一个鉴别标志，与消息捆绑在一起发送给接收者，以便于接收者在收到消息之后，通过附加的鉴别标志来验证消息是否来自合法的发送者，以及消息是否受到过篡改。

大体来说，实现消息鉴别的手段可以分为两类：基于加密技术的消息鉴别和基于散列函数的消息鉴别。

5.5.1　基于加密技术的消息鉴别

从消息鉴别的目的出发，无论是对称密码体制，还是公钥密码体制，对于消息本身的加密都可以看作是一种鉴别的手段。

1. 利用对称加密体制实现消息鉴别

如图 5.10 所示，发送方 A 和接收方 B 共享密钥 k。A 用密钥 k 对消息 M 加密后通过公开信道传送给 B。B 接收到密文消息之后，通过是否能用密钥 k 将其恢复成合法明文，来判断消息是否来自 A，信息是否完整。

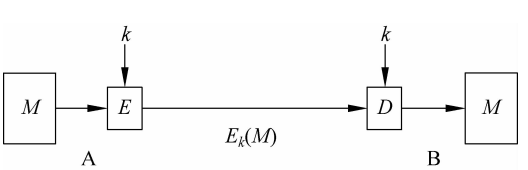


图 5.10　对称加密实现报文鉴别

这种方法处理的局限性在于需要接收方有某种方法能判定解密出来的明文是否合法。

因此在处理中,可以规定合法的明文只能是属于在可能位模式上有微小差异的一个小子集,这使得任何伪造的密文解密恢复出来后能够成为合法明文的几率十分微小。而且为了确定消息的真实性,可要求传递的内容具有某种可识别的结构,如在加密前,对每个消息附加一个帧校验序列(Frame Check Sequence,FCS)。接收方可以按发送方同样的方法生成 FCS,与收到的 FCS 进行比较,从而确定消息的真实性。

该方法的特点是:

(1) 它能提供机密性——只有 A 和 B 知道密钥 k;

(2) 提供鉴别——只能发自 A,传输中未被改变;

(3) 不能提供数字签名——接收方可以伪造消息,发送方可以抵赖消息的发送。

2. 利用公钥密码体制实现消息鉴别

1) 提供消息鉴别

如图 5.11 所示,发送方 A 用自己的私钥 SK_A对消息进行加密运算(但并不能提供机密性保护,请思考为什么?),再通过公开信道传送给接收方 B。接收方 B 用 A 的公钥 PK_A对得到的消息进行解密运算并完成鉴别。因为只有发送方 A 拥有自己的私钥,只有发送方 A 才能产生用公钥 PK_A可解密的密文,所以消息一定来自于拥有私钥 SK_A的发送方 A。这种机制也要求明文具有某种内部结构以使接收方易于确定得到明文的真实性。

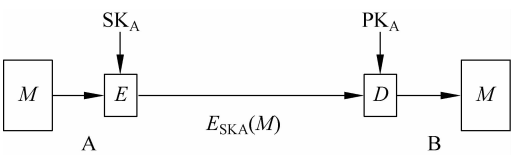


图 5.11　公钥密码体制实现消息鉴别

这种方法的特点是:能实现数字签名的功能,可以抗抵赖,并提供鉴别。

2) 提供消息鉴别和机密性保护

前一种方法只能提供数字签名,不能提供机密性,因为任何具有发送方 A 公钥 PK_A的人都可以解密密文。而如图 5.12 所示的方法,就是一种既能提供消息鉴别,还能提供机密性的方案。如图 5.12 所示,在发送方 A 用自己的私钥 SK_A进行加密运算(实现数字签名)之后,还要用接收方 B 的公钥 PK_B进行加密,从而实现机密性。这种方法的缺点:一次完整的通信需要执行公钥算法的加密、解密操作各两次。

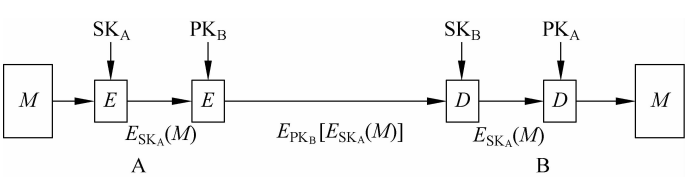


图 5.12　公钥密码体制实现签名、加密和鉴别

这种方法的特点是:提供机密性、数字签名和鉴别。

5.5.2　基于散列函数的消息鉴别

1. 消息鉴别码的概念

消息鉴别码(Message Authentication Code,MAC)或报文鉴别码,是用于提供数据原发鉴别和数据完整性的密码校验值。MAC 是使用一个特定的密钥将消息通过一种鉴别算法处理所得出的一串代码。一个 MAC 算法是由一个秘密密钥 k 和参数化的一簇函数 h_k 构成,这簇函数具有如下特性:

(1) 容易计算——对于一个已知函数 h,给定一个值 k 和一个输入 x,$h_k(x)$是容易计算的。这个计算的结果被称为消息鉴别码值或消息鉴别码。

(2) 压缩——h_k把任意具有有限长度(比特数)的一个输入 x 映射成一个具有固定长度的输出 $h_k(x)$。

(3) 强抗碰撞性——要找到两个不同消息 x 和 y,使得 $h_k(x)=h_k(y)$是计算上不可行的。

利用 MAC 进行消息鉴别的基本方法是:假定通信双方(A 和 B)共享密钥 k,且 h_k函数公开,发送方 A 对要发送的消息 M,使用一个密钥 k 计算得到 MAC= $h_k(M)$,MAC 的值只与消息 M 和密钥 k 有关。然后将它附在消息后面得到 $M'=M\|\text{MAC}$,一起发送给接收方 B。接收者 B 收到 M'后,使用共享的密钥 k 对其中的消息部分 M 执行相同的计算 $\text{MAC}'=h_k(M)$,然后将收到的 MAC 与计算的 MAC′进行比较(如图 5.13 所示),如果两者相等,由于只有 A 和 B 知道密钥 k,故可以判断:

(1) 接收者 B 收到的消息 M 未被篡改过;

(2) 消息 M'确实来自发送者 A。

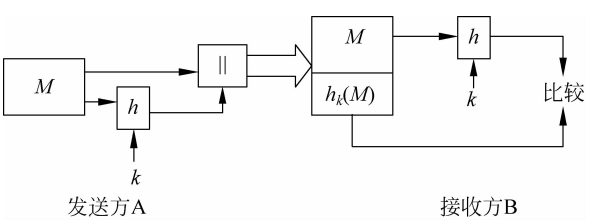


图 5.13　消息鉴别

MAC 需要使用密钥 k,这类似于加密,但其区别是 MAC 函数无须可逆。因为它不需要解密,这个性质使得 MAC 比加密函数更难于破解。同时 h_k函数具有压缩、强抗碰撞等特性,使它更相似于散列函数。因此在应用中往往使用带密钥的散列函数作为 h_k函数来实现 MAC。由于收发双方使用的是相同的密钥,因此单纯使用 MAC 无法提供数字签名。

图 5.13 中的方法只能提供消息鉴别,不能提供机密性,因为消息 M 是以明文的形式传送的。如果在生成 MAC 之后(如图 5.14 所示)或者之前(如图 5.15 所示)使用加密机制,则可以获得机密性。这两种方法生成的 MAC 或者基于明文,或者基于密文,因此相应的鉴别或者与明文有关,或者与密文有关。一般来说,基于明文生成 MAC 的方式在实际应用中会更方便一些。

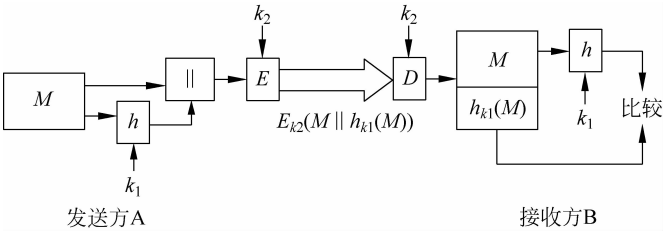


图 5.14　消息鉴别与机密性(与明文相关)

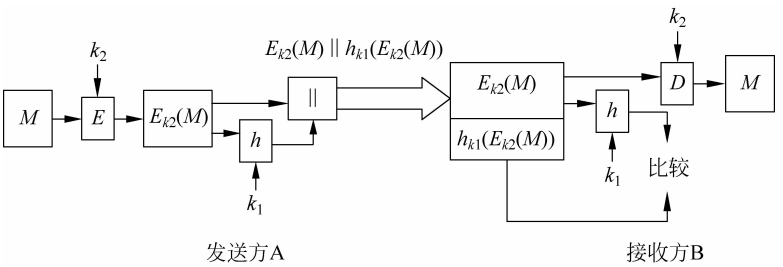


图 5.15　报文鉴别与机密性(与密文相关)

2. 基于散列函数的消息鉴别

散列函数具有以下特点：输入是可变大小的消息 M，输出固定长度的散列值(即消息摘要)；计算简单，不需要使用密钥，具有强抗碰撞性。散列值只是输入消息的函数，只要输入消息有任何改变，就会导致不同的散列值输出，因此散列函数常常用于实现消息鉴别。

但必须注意的是，由于散列函数没有密钥，在散列函数公开的情况下，任何人都可以根据输入的消息计算其散列值。故在安全通信中，常常需要将一个密钥或秘密信息与散列函数结合起来产生鉴别码 MAC。

基于散列函数的消息鉴别方法如图 5.16 所示，有以下几种情况：

(1) 用对称密码体制加密消息及其散列值，即 A→B：$E_k[M \| h(M)]$，如图 5.16(a)所示。由于只有发送方 A 和接收方 B 共享加密密钥 k，因此通过对 $h(M)$的比较鉴别可以确定消息一定来自于 A，并且未被修改过。散列值在方法中所起的作用是提供用于鉴别的冗余信息，同时 $h(M)$受到加密的保护。该方法实现了对消息的明文加密保护，因此可以提供机密性。

(2) 用对称密码体制只对消息的散列值进行加密，即 A→B：$M \| E_k[h(M)]$，如图 5.16(b)所示。这种方法中消息 M 是以明文形式传递，因此不能提供机密性，适合于对消息提供完整性保护，而不要求机密性的场合，有助于减少处理代价。

(3) 用公钥密码体制只对散列值进行加密运算，即 A→B：$M \| E_{SK_A}[h(M)]$，如图 5.16(c)所示。这种方法与方法(2)的区别是对散列值的加密运算是使用公钥密码体制，即用发送方 A 的私钥对 $h(M)$进行加密运算(实现数字签名)，接收方 B 用发送方 A 的公钥 PK_A进行解密运算以实现鉴别。这种方案可提供消息鉴别和数字签名，因为只有 A 能生成 $E_{SK_A}[h(M)]$。

(4) 结合使用公钥密码体制和对称密码体制，这种方法用发送方的私钥对散列值进行数字签名，用对称密码体制加密消息 M 和得到的数字签名，即 A→B：$E_k[M \| E_{SK_A}[h(M)]]$，如

图 5.16(d)所示。因此，这种方法既提供鉴别和数字签名，也提供机密性，在实际应用中较为常见。

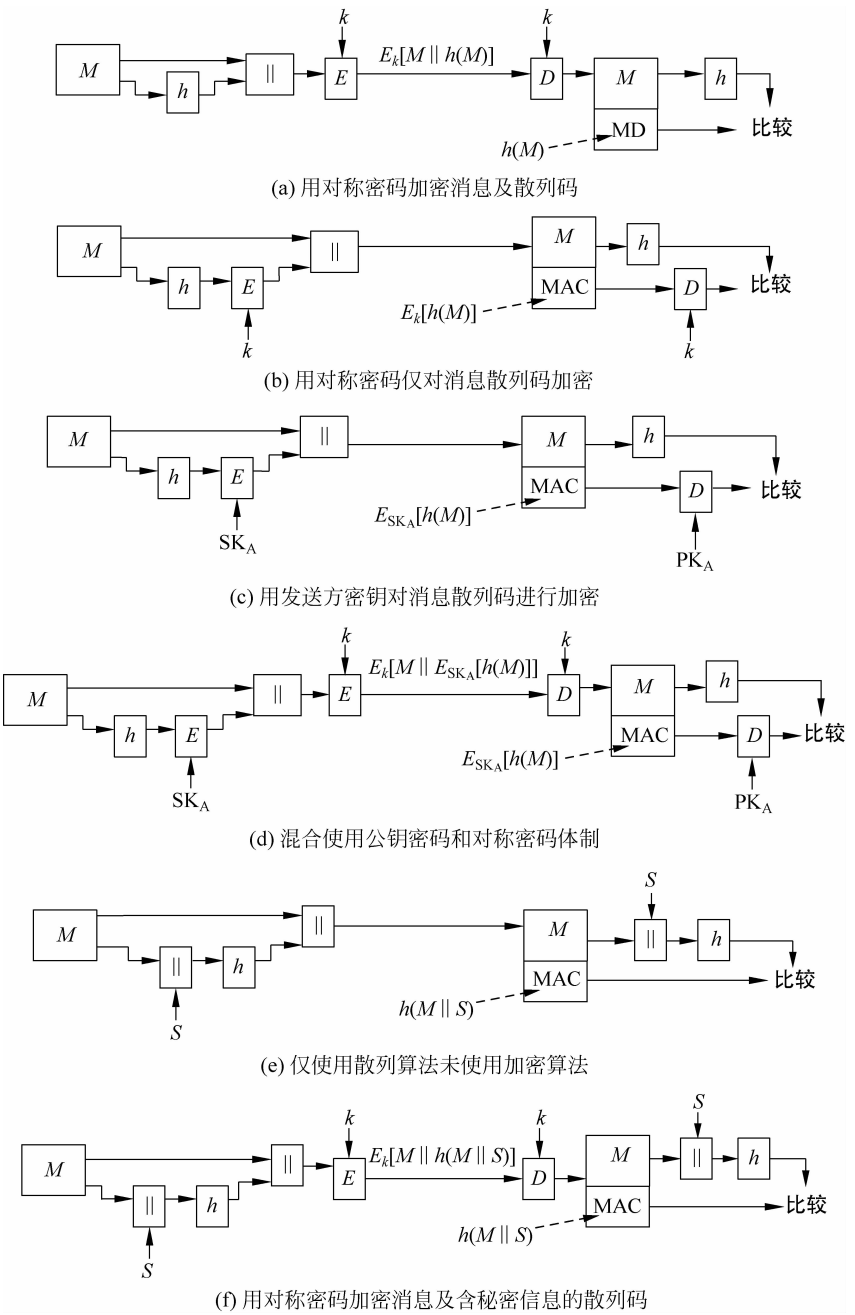


图 5.16　散列函数用于鉴别的基本方法

(5) 这种方法使用了散列算法，但未使用加密算法。为了实现鉴别，要求发送方 A 和接收方 B 共享一个秘密信息 S，发送方生成消息 M 和秘密信息 S 的散列值，然后与消息 M 一起发送给对方，即 A→B：$M \| h(M \| S)$，如图 5.16(e)所示；接收方 B 按照发送方相同的处

理方式生成消息 M 和秘密信息 S 的散列值,两者进行比较,从而实现鉴别。这种方法中,秘密信息 S 并不参与传递,因此可以保证攻击者无法伪造。

(6) 在方法(5)的基础上,使用对称密码体制对消息 M 和生成的散列值进行保护,即 A→B: $E_k[M \| h(M \| S)]$,如图 5.16(f)所示。因此这种方法除了提供消息鉴别还提供机密性保护。

5.5.3 HMAC 算法

早期构造消息鉴别码(MAC)的方法常常是使用分组密码的 CBC 模式,即基于分组密码的构造方法。而构造消息鉴别码(MAC)的另一类方法是基于散列函数的构造方法。这类方法的好处是散列算法(如 MD5、SHA-1 等)的软件实现快于分组密码的软件实现,函数的库代码来源广泛、限制较少。基于散列函数的消息鉴别码构造的基本思想就是将某个散列函数嵌入到消息鉴别码的构造过程中。HMAC 作为这种构造方法的代表,已经作为 RFC 2104 公布,并在 IPSec 和其他网络协议(如 SSL)中得到应用。

1. HMAC 算法的设计要求

按照 RFC 2104,HMAC 希望达到以下的设计要求:

(1) 可不经修改而使用现有的散列函数,特别是那些易于软件实现的、源代码可方便获取且免费使用的散列函数。

(2) 其中嵌入的散列函数可以易于替换为更快或更安全的散列函数,以适应不同的安全需求。

(3) 保持嵌入的散列函数的原有性能,不因用于 HAMC 而使其性能降低。

(4) 密钥的使用和处理简单方便。

(5) 以嵌入的散列函数安全假设为基础,易于分析 HMAC 用于鉴别时的安全强度。

其中前两个要求是 HMAC 被公众普遍接受的主要原因。这两个要求是将散列函数当作一个黑盒使用,其好处是散列算法模块化,容易实现和更新。散列函数可作为 HMAC 的一个模块,HMAC 代码中很大一块就可事先准备好,无须修改就可使用;而且如果 HMAC 要求使用更快或更安全的散列函数,则只需用新模块代替旧模块。最后一条设计要求则是保证 HMAC 的安全强度易于分析和确认。HMAC 在其嵌入的散列函数具有合理密码强度的假设下,可证明是安全的。

2. HMAC 算法描述

图 5.17 是 HMAC 算法的实现框图。其中 h 为嵌入的散列函数(如 MD5、SHA-1),输入消息 $M=(Y_0,Y_1,\cdots,Y_{L-1})$,$Y_i(0\leqslant i\leqslant L-1)$ 是 b 位的一个分组,M 包含了散列函数的填充位。n 为嵌入的散列函数输出值的长度,K 为密钥,如果密钥长度大于 b,则将密钥输入散列函数中产生一个 n 位的密钥。K^+ 是左边填充了 0 后的 K,其长度为 b 位,ipad 为 $b/8$ 个 00110110 序列串,opad 为 b/8 个 01011010 序列串。

HMAC 算法可表示为

$$\mathrm{HMAC}(M,K) = h\left((K^+ \oplus \mathrm{opad}) \| h\left((K^+ \oplus \mathrm{ipad}) \| M\right)\right)$$

对 HMAC 算法的处理过程描述如下:

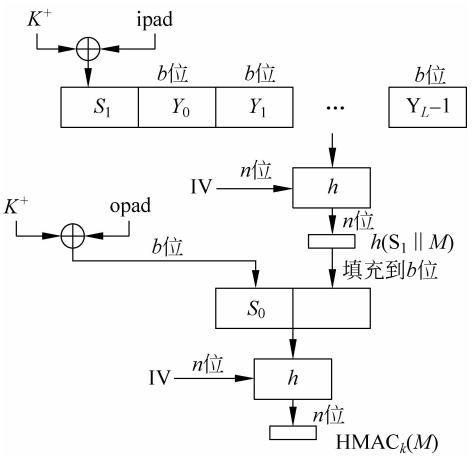


图 5.17　HAMC 的算法框图

(1) 对密钥 K 左边填充若干个 0，以生成 b 位长的 K^+。例如，K 的长度为 160 位，$b=512$，则需要填充 44 个 0 字节 0x00。

(2) 产生 b 位的分组 $S_1=K^+\oplus$opad。

(3) 将消息 M 链接到 S_1之后，得到 $S_1 \| M$。

(4) 将步骤(3)得到的结果输入散列函数 h，得到 $h(S_1 \| M)$。

(5) 产生 b 位的分组 $S_0=K^+\oplus$ipad。

(6) 将步骤(4)得到的结果链接到 S_0之后。

(7) 将步骤(6)的结果输入散列函数 h，得到最后的 HMAC 值。

可以看到，K^+分别与 ipad 与 opad 逐位异或得到 S_0和 S_1，其结果是将 K 中的一半比特取反，但两次取反的比特位置不同。所以可以认为 S_0和 S_1是由 K 产生的两个伪随机密钥。而且 S_0和 S_1的引入，仅相当于增加了两个数据分组散列处理的开销。因此，当消息 M 充分大时 HMAC(M,K)的执行时间与所嵌入的散列函数 $h(M)$执行时间大致相等。

仔细分析 HMAC 算法可以发现，($K^+\oplus$ opad) 和($K^+\oplus$ ipad) 以及有关散列函数对第一个分组初始化向量的两个压缩函数运算，即 f(IV,($K^+\oplus$ipad))和 f(IV,($K^+\oplus$opad))，还可以预先计算出来，如图 5.18 虚线以左部分。其中 f(cv,block) 是散列函数中的压缩函数，其输入是 n 位的链接变量和 b 位的分组，输出是 n 位的链接变量。这两个量的预先计算只在每次更改密钥时才需进行。事实上这两个预先计算的量用于作为散列函数的初值 IV。

3. HMAC 的安全性

由 HMAC 的结构可以发现，其安全性依赖于所嵌入散列函数的安全性。对 HMAC 的攻击等价于对内嵌散列函数的下列攻击之一：

(1) 攻击者能够计算压缩函数的一个输出，即使 IV 是机密的或者随机的。

(2) 攻击者能够找到散列函数的碰撞，即使 IV 是机密的或者随机的。

第一种攻击可以认为压缩函数与散列函数等价，n 位的初始链接变量 IV 可以看作 HMAC 的密钥。因此攻击可以通过对密钥的穷举来进行攻击。对密钥穷举的计算复杂度

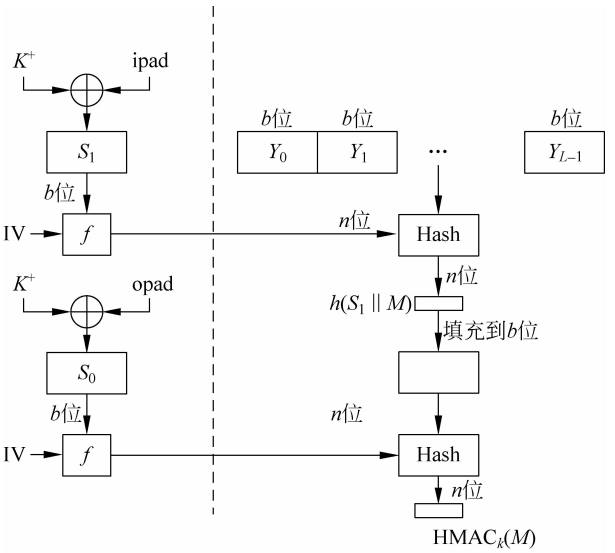


图 5.18　HMAC 的有效实现

为 $O(2^n)$。

第二种攻击要求寻找两个具有相同散列值的消息,即实施生日攻击,对于具有 n 位长散列值的散列函数而言,其计算复杂度为 $O(2^{n/2})$。例如,对于散列值长度为 128 位的散列函数,第二种攻击的复杂度为 $O(2^{64})$,按照现有技术这是可能的。但是这不影响将该散列函数用于 HMAC。因为攻击者对散列函数实施生日攻击时,由于已经知道算法和默认 IV 值,因此在选择消息集合后可以离线地寻找碰撞。但是在攻击 HMAC 时,由于攻击者不知道密钥 K,从而不能离线产生消息和散列码对。所以攻击者必须监听并得到 HMAC 在同一密钥下产生的一系列消息,并对得到的消息序列进行攻击。对于长度为 128 位的散列值来说,要求得到同一密钥下产生的 2^{64} 个分组(2^{73} 位)。在 1Gbps 的链路上这大约需要 25 万年。因此就速度和安全性而言,采用该散列函数或者计算复杂度更高的散列函数嵌入 HMAC 中是可以接受的。

思考题与习题

5-1　什么是散列函数?对散列函数的基本要求和安全性要求分别是什么?

5-2　安全散列函数的一般结构是什么?

5-3　为什么要进行散列填充?

5-4　散列函数的主要应用有哪些?

5-5　什么是消息鉴别?为什么要进行消息鉴别?消息鉴别的实现方法有哪些?

5-6　有很多散列函数是由 CBC 模式的分组加密技术构造的,其中的密钥为消息分组。例如将消息 M 分成分组 $M_1, M_2, \cdots, M_n$,H_0 = 初值,迭代关系为 $H_i = E_{M_i}(H_{i-1}) \oplus H_{i-1}$ ($i = 1, 2, \cdots, n$),散列值为 H_n,其中 E 是分组加密算法。

(1) 设 E 为 DES,已证明如果对明文分组和加密密钥都逐位取补,那么得到的密文也

是原密文的逐位取补，即 $\mathrm{DES}_{\overline{K}}(\overline{M})=\overline{\mathrm{DES}_K(M)}$。利用这一结论证明在上述散列函数中可对消息进行修改但却保持散列值不变。

(2) 若迭代关系改为 $H_i=E_{\mathrm{H}_{i-1}}(M_i)\oplus M_i(i=1,2,\cdots,n)$，证明仍可对其进行上述攻击。

5-7　对本章中 SHA-1 的示例，计算 W_{16}、W_{17}、W_{18}和 W_{19}。

5-8　设 $a_1\ a_2\ a_3\ a_4$是 32 位长的字中的 4 个字节，每一个 a_i可看作由二进制表示 0～255 的整数，在 big-endian 结构中，该字表示整数 $a_1 2^{24}+a_2 2^{16}+a_3 2^8+a_4$。而在 little-endian 结构中，该字表示整数 $a_4 2^{24}+a_3 2^{16}+a_2 2^8+a_1$。SHA 使用 big-endian 结构。试说明 SHA 中如何对以 little-endian 结构存储的两个字 $X=x_1\ x_2\ x_3\ x_4$和 $Y=y_1\ y_2\ y_3\ y_4$进行模 2 加法运算。

5-9　证明：如果散列函数 h_1和 h_2是强抗碰撞的，则 $h(x)=h_1(x)\parallel h_2(x)$定义的散列函数也是强抗碰撞的，其中“‖”表示比特串的连接。

5-10　设 H_1是一个从$(Z_2)^{2n}$到$(Z_2)^n$具有强抗碰撞的散列函数，H_1的输入为 Z_2上的长度为 $2n$ 的字符串，而输出为 Z_2上的长度为 n 的字符串。这里 $n\geqslant 1$，$Z_2=\{1,0\}$。

从$(Z_2)^{4n}$到$(Z_2)^n$的散列函数 H_2按下述方式定义：

对任意 $x\in(Z_2)^{4n}$，设 $x=x_1\parallel x_2$，有 $H_2(x)=H_1(H_1(x_1)\parallel H_1(x_2))$，其中“‖”表示比特串的连接。

试证明散列函数 H_2也是强抗碰撞的。

5-11　用公钥密码体制 RSA 来构造一个散列函数。将输入 m 分为 k 个分组，即 $m=m_1\ m_2\cdots\ m_k$，RSA 的公钥为(e,n)，定义散列函数如下：

$$h_i=(h_{i-1}^e \bmod n)\oplus m_i,\quad i=2,3,\cdots,k$$

其中 $h_1=m_1$，最后一个分组的输出 h_k，即为输出的散列码。

试问如此构造的散列函数安全吗？为什么？