第 5 章　消息认证与数字签名

本章导读

- 消息认证是用来防止主动攻击的重要技术，用以保证消息的完整性。常见的消息认证密码技术包括消息认证码（MAC）和安全散列函数。另外，消息加密也可以提供一种形式的认证。
- MAC 是需要使用密钥的算法，其输入是可变长度的消息和密钥，其输出是一个定长的认证码。只有拥有密钥的消息，发送方和接收方才可以生成消息认证码和验证消息的完整性。
- 散列函数和 MAC 算法类似，也是一个单向函数，但是无需密钥，其输入是可变长度的消息，其输出是固定长度的散列值，也叫消息摘要。
- 数字签名是基于公钥密码技术的认证技术。它和手写签名类似，使得消息的发送者可以使用自己的私钥为初始消息生成一个有签名作用的签名码，接收者接收到初始消息和相应的签名码，可以使用消息发送者的公钥对该消息的签名码进行验证。数字签名可以保证消息的来源和消息本身的完整性。
- 使用数字签名，通常需要和散列函数配合使用。

5.1　认　　证

加密通常用于保密，对某个信息的加密操作使得其内容对于未授权的人而言是保密的、安全的。但是，在某些情况下，完整性比保密性更重要。例如，在某个医院的医疗系统中检索到的病人的医疗记录，或者银行系统中检索到的某个人的信用记录，检索到的这些信息和所存储的正本是否一致是非常重要的。在传统的或者没有考虑完整性的系统中，文件的组成部分或者消息的组成部分，即每个字节、位或者字符都是彼此独立的，由于缺乏彼此的绑定，使得攻击者对于信息的修改无法被发现。在目前的电子商务系统中的各种应用中，这类问题造成的后果更加严重。能否为文件或者网络中通信的消息打上一个标签，当文件或消息出现了任何改变，即便只修改了一位信息时，我们都可以从标签和信息的关系上知道有内容被修改了。这种想法和中世纪在信封上使用蜡封类似。在密码技术中，提供这样的蜡封技术或标签技术的就是为信息提供一种认证，这样的蜡封或者标签在密码技术中称为认证码，如后文中讨论的哈希值、校验和等都是某种形式的认证码。

同样，在网络通信环境中，保密的目的是防止攻击者破译系统中的机密信息，但在大多数网络应用中，仅提供保密性是远远不够的。网络安全的威胁来自于两个方面：一是被动攻击，攻击者只是通过侦听和截取等手段被动地获取数据，并不对数据进行修改；二是主动攻击，攻击者通过伪造、重放、篡改、改变顺序等手段改变数据。认证则是防止主动攻击的重要技术，它对于开放环境中的各种信息系统的安全性有重要作用，可以防止如下一些攻击。

- 伪装：攻击者生成一个消息并声称这条消息是来自某个合法实体，或者攻击者冒充消息接收方向消息发送方发送的关于收到或未收到消息的欺诈应答。
- 内容修改：对消息内容的修改，包括插入、删除、转换和修改。
- 顺序修改：对通信双方消息顺序的修改，包括插入、删除和重新排序。
- 计时修改：对消息的延迟和重放。在面向连接的应用中，攻击者可能延迟或重放以前某合法会话中的消息序列，也可能会延迟或重放消息序列中的某一条消息。

认证的目的主要有两个：第一，验证消息的发送者是合法的，不是冒充的，这称为实体认证，包括对信源、信宿等的认证和识别；第二，验证信息本身的完整性，这称为消息认证，验证数据在传送或存储过程中没有被篡改、重放或延迟等。

保密和认证是信息系统安全的两个重要方面，但它们又是不同的。认证不能自然地提供保密性，而保密性也不能自然地提供认证功能。但从某个层面上而言，我们也可以说保密性提供了某种认证功能，因为攻击者如果无法获得用于加密的密钥，而消息接收方收到了密文，并使用密钥进行解密，同时可以确认解密后得到的信息是正确的（如根据解密后的信息的含义），在这种情况下，整个密文就提供了认证功能。但如果发送者发送的信息是无意义的字符，消息接收者即便正确解密了，也无法通过字符的含义来判定所收到的消息是否是正确的。

因此，如果考虑加密函数的某种认证功能，我们考虑的可用于提供认证功能的认证码的函数则可以分为以下 3 类。

- 加密函数：使用消息发送方和消息接收方共享的密钥对整个消息进行加密，则整个消息的密文将作为认证符。
- 消息认证码（Message Authentication Code）：它是消息和密钥的函数，用于产生定长度值，该值将作为消息的认证符。
- 散列函数：它是将任意长的消息映射为定长的 hash 值的函数，以该 hash 值作为认证符。

一个基本的认证系统模型如图 5.1 所示。

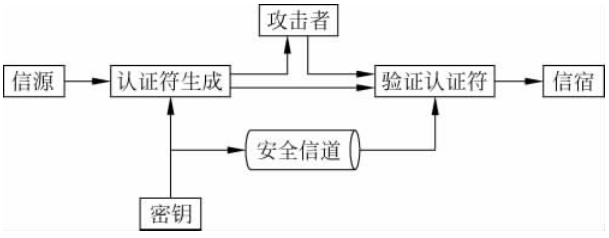

图 5.1　基本的认证系统模型

5.2　消息认证码

消息认证码（MAC）是一种使用密钥的认证技术，它利用密钥来生成一个固定长度的短数据块，并将该数据块附加在消息之后。在这种方法中假定通信双方 A 和 B 共享密钥 K。

若A向B发送消息M时,则A使用消息M和密钥K,计算MAC=$C(K,M)$,其中:

- M=输入消息,可变长
- C=MAC函数
- K=共享的密钥
- MAC=消息认证码

消息认证码MAC为消息M的认证符,MAC也称为密码校验和。

如图5.2所示,发送方将消息M和MAC一起发送给接收方。接收方收到消息后,假设该消息为M',使用相同的密钥K进行计算得出新的MAC$'$=$C(K,M')$,比较MAC$'$和所收到的MAC。假设双方共享的密钥没有被泄露,则比较计算得出的MAC$'$和收到的MAC的结果,如果两者是相同的话,则可以认为:

(1) 接收方可以相信消息未被修改。因为若攻击者篡改了消息,他必须同时相应地修改MAC值。而我们已假定攻击者不知道密钥,所以他不知道应如何改变MAC才能使其与修改后的消息相一致。

(2) 接收方可以相信消息来自真正的发送方。因为其他各方均不知道密钥,他们不能产生具有正确MAC的消息。

(3) 如果消息中含有消息序列号,那么接收方可以相信消息的顺序是正确的,因为攻击者无法成功地修改序列号。

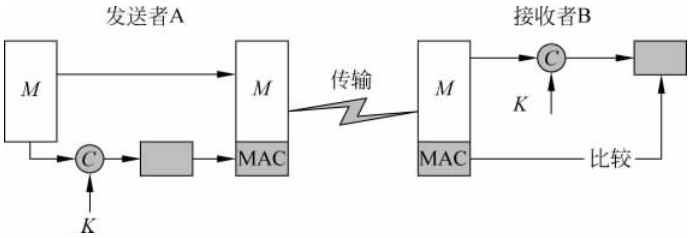

图5.2 消息认证码的使用

从使用密钥上看,MAC函数与加密函数类似,需要生成MAC方和验证MAC方共享一个密钥。但它们又存在本质的区别,区别之一为MAC算法不要求可逆性,而加密算法必须是可逆的。一般而言,MAC函数是多对一函数,其定义域由任意长的消息组成,而值域由所有可能的MAC和密钥组成。若使用n位长的MAC,则有2^n个可能的MAC,而有m条可能的消息,其中$m \gg 2^n$,而且若密钥长为k,则有2^k种可能的密钥。

例如,假定使用100位的消息和10位的MAC,那么总共有2^{100}不同的消息,但仅有2^{10}种不同的MAC。所以平均而言,同一MAC可以由$2^{100}/2^{10}$条不同的消息产生。若使用的密钥长为5位,则从消息集合到MAC值的集合有$2^5=32$种不同的映射。可见密钥的位数太短,很容易通过穷举进行攻击,但只要位数足够长,则可以保证其安全性。

图5.2给出的消息认证码的使用只是对传送消息提供单纯的认证性。它还可以和加密函数一起提供消息认证和保密性。如图5.3所示,发送方在加密消息M之前,先计算M的认证码,然后使用加密密钥将消息及其认证码一起加密;接收方收到消息后,先解密得到消息及其认证码,再验证解密得到的消息和验证码是否匹配,如果匹配则表示消息在传输中没有被改动。

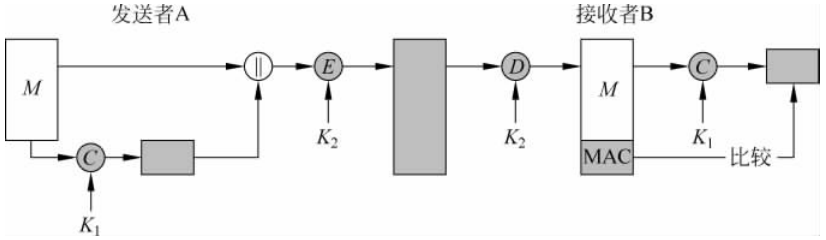

图 5.3　结合加密函数的消息认证码的使用方法

5.2.1　MAC 的安全要求

MAC 中使用了密钥，这点和对称密钥加密一样，如果密钥泄露了或者被攻击了，则无法保证 MAC 的安全性。在基于算法的加密函数中，攻击者可以尝试所有可能的密钥以进行穷举攻击，一般对 k 位的密钥，穷举攻击需要 $2^{(k-1)}$ 步。对于仅依赖于密文的攻击，若给定密文 C，攻击者要对所有可能的 K_i 计算 $P_i=D_{K_i}(C)$，直到产生的某 P_i 具有适当的明文结构为止（前提是这样的明文结构是可以判断的）。

MAC 函数是多对一函数，这就意味消息的取值空间比 MAC 的取值空间大，则一定存在着不同的消息会对应于相同的 MAC。假设 MAC 所使用的密钥位数为 k，计算所得的 MAC 位数为 n。若 $k>n$，即假定密钥位数比 MAC 长，则对满足 $\mathrm{MAC}_1=C_{K_1}(M_1)$ 的 M_1 和 MAC_1，密码分析者要对所有可能的密钥值 K_i 计算 $\mathrm{MAC}_i=C_{K_i}(M_1)$，那么至少有一个密钥会使得 $\mathrm{MAC}_i=\mathrm{MAC}_1$。因为 k 个密钥总共会产生 2^k 个 MAC，但只有 $2^n(2^n<2^k)$ 个不同的 MAC 值，所以不同密钥都会产生正确的 MAC，而攻击者却不知其中哪一个是正确的密钥。平均来说，有 $2^k/2^n=2^{(k-n)}$ 个密钥会产生正确的 MAC，因此攻击者必须重复下述攻击。

（1）第 1 轮。

- 给定 M_1，$\mathrm{MAC}_1=C_K(M_1)$。
- 对所有 2^k 个密钥判断 $\mathrm{MAC}_i=C_{K_i}(M_1)$。
- 匹配数$\approx 2^{(k-n)}$。

（2）第 2 轮。

- 给定 M_2，$\mathrm{MAC}_2=C_K(M_2)$。
- 对循环 1 中找到的 $2^{(k-n)}$ 个密钥判断 $\mathrm{MAC}_i=C_{K_i}(M_2)$。
- 匹配数$\approx 2^{(k-2n)}$。

攻击者可以按此方法不断对密钥进行测试，直到将匹配数缩小到足够小的范围。平均来讲，若 $k=a\times n$，则需 a 次循环。例如，如果使用 80 位的密钥和长为 32 位的 MAC，那么第 1 次循环会得到约 2^{48} 个可能的密钥，第 2 次循环会得到约 2^{16} 个可能的密钥，第 3 次循环则得到唯一一个密钥，这个密钥就是发送方所使用的密钥。这样看来，若密钥的长度小于或等于 MAC 的长度，则很可能在第 1 次循环中就得到一个密钥。

由此可见，如果密钥足够长，用穷举方法来确定 MAC 的密钥就不是一件容易的事。

当然，以上的穷举攻击是建立在算法安全强度可信的前提下。针对不同的 MAC 算法，攻击者可能不需要使用穷举攻击即可找到密钥。攻击者针对下面的 MAC 算法，则不需要使用穷举攻击即可获得密钥信息。

设消息 $M=(X_1\parallel X_2\parallel\cdots\parallel X_m)$，即由 64 位分组 X_i 连接而成。定义：

$$\Delta(M) = X_1 \oplus X_2 \oplus \cdots \oplus X_m$$
$$C_k(M) = E_K[\Delta(M)]$$

其中⊕是异或(XOR)运算,加密算法是电码本方式实现的DES算法,则密钥长为56位,MAC长为64位。若攻击者获取$\{M \| C_k(M)\}$,并使用穷举攻击,则确定K须执行至少2^{56}次加密,但是攻击者可以用任何期望的$Y_1 \sim Y_{m-1}$替代$X_1 \sim X_{m-1}$,用Y_m替代X_m来进行攻击,其中Y_m的计算方法如下。

$$Y_m = Y_1 \oplus Y_2 \oplus \cdots \oplus Y_{m-1} \oplus \Delta(M)$$

攻击者可以将$Y_1 \sim Y_{m-1}$与原来的MAC连接成一个新的消息M',接收方收到$(M', C_k(M))$时,由于$\Delta(M')=Y_1 \oplus Y_2 \oplus \cdots \oplus Y_m = \Delta(M)$,因此$C_k(M)=E_K[\Delta(M')]$,接收者会认为该消息是真实的。用这种办法,攻击者可以随意插入任意的、长为$64\times(m-1)$位的消息。

因此,一个安全的MAC函数应具有下列性质。

- 若攻击者知道M和$C_k(M)$,则他构造满足$C_k(M')=C_k(M)$的消息M'在计算上是不可行的。
- $C_k(M)$应是均匀分布的,即对任何随机选择的消息M和M',$C_k(M)=C_k(M')$的概率是2^{-n},其中n是MAC的位数。
- 设M'是M的某个已知的变换,即$M'=f(M)$,则$C_k(M)=C_k(M')$的概率为2^{-n}。

5.2.2 基于DES的消息认证码

数据认证算法(FIPS PUB 113)是使用最广泛的MAC算法之一,它也是一个ANSI标准(X9.17)。该算法建立在DES算法之上,利用了密文链接模式(CBC)对消息进行加密处理。该算法在实际中的应用很广泛,特别是在银行系统中。

如图5.4所示,数据认证算法取初始值为0,这个初始值没有实际意义,只是用于第1次计算,需要认证的消息被划分成64位的分组$D_1, D_2, \cdots, D_N$,若最后分组不足64位,则在其后填0直至成为64位的分组。利用DES加密算法E和密钥K,计算认证码的过程如图5.4所示。

$$O_1 = E_K(D_1)$$
$$O_2 = E_K([D_2 \oplus O_1])$$
$$O_3 = E_K([D_3 \oplus O_2])$$
$$\vdots$$
$$O_N = E_K([D_N \oplus O_{N-1}])$$

不输出最后一个分组的加密结果,取其最左边的n位作为认证码。

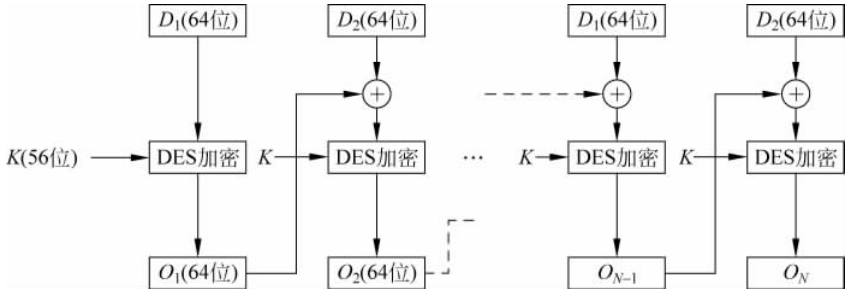

图5.4 数据认证算法

5.3　Hash 函数

Hash 函数(也称散列函数或杂凑函数)是将任意长的输入消息作为输入生成一个固定长的输出串的函数,即 $h=H(M)$。这个输出串 h 称为该消息的散列值(或消息摘要、杂凑值)。散列函数通常和一个安全的 Hash 函数 H 应该至少满足以下几个条件。

(1) H 可以应用于任意长度的数据块,产生固定长度的散列值。

(2) 对每一个给定的输入 m,计算 $H(m)$是很容易的。

(3) 给定 Hash 函数的描述,对于给定的散列值 h,找到满足 $H(m)=h$ 的 m 在计算上是不可行的。

(4) 给定 Hash 函数的描述,对于给定的消息 m_1,找到满足 $m_2 \neq m_1$ 且 $H(m_2)=H(m_1)$ 的 m_2 在计算上是不可行的。

(5) 找到任何满足 $H(m_1)=H(m_2)$且 $m_1 \neq m_2$ 的消息对(m_1,m_2)在计算上是不可行的。

条件(1)和条件(2)指得的是 Hash 函数的“单向”(One-Way)特性,条件(3)和条件(4)是对使用散列值的数字签名方法所做的安全保障。否则攻击者可以由已知的明文及相关数字签名任意伪造对其他明文的数字签名。条件(5)的主要作用是防止后文将要提到的“生日攻击”。通常我们称满足条件(1)～条件(4)的散列函数为“弱散列函数”,若能同时满足条件(5),则称其为“强散列函数”。

Hash 函数主要用于完整性校验和提高数字签名的有效性,目前已有很多方案。这些算法都是伪随机函数。早在 1978 年,Rabin 就利用 DES 算法,使用密文分组链接(CBC)方式,提出一种简单快速的散列函数,方法如下所示。

将明文 M 分成固定长度的 64 位的分组:$m_1,m_2,\cdots,m_k$。使用 DES 的 CBC 操作模式,对每个明文分组进行加密,令 $h_0=$初始值,$h_i=E_{mi}[h_{i-1}]$,最后散列值为 h_k,这个方法和第 5.2.2 节中的基于 DES 的 MAC 算法类似,但不同的是该算法中的加密没有使用任何密钥。但需要指出的是,使用 Rabin 散列函数的数字签名是不安全的,已经发现可以在有限的计算范围内,不通过获得签名私钥的方法即可实现伪造签名。

好的散列函数的输出以不可辨别的方式依赖于输入。任何输入串中单个位的变化,将会导致输出位串中大约一半的位发生变化。其处理思想是先要将明文分成固定长度的明文分组,再对每个分组做相同的处理,比较有名的有 MD5、Ripend160、SHA、Whirlpool 等算法。所有的散列函数都具有图 5.5 中的处理结构,这种结构称为迭代 Hash 函数,它是由 Merkle 提出的。其中的 f 算法即是散列函数中对分组进行迭代处理的压缩函数。散列函数重复使用压缩函数 f,它的输入是前一步得出的 n 位输出(称为链接值)和一个 b 位消息分组,输出为一个 n 位分组。链接值的初始值由算法在开始时指定,其终值即为散列值。这样,一般结构的 Hash 函数可归纳如下:

$$\begin{aligned} &CV_0 = IV = n\text{ 位初始值} \\ &CV_i = f(CV_{i-1}, Y_{i-1}) \quad 1 \leqslant i \leqslant L \\ &H(M) = CV_L \end{aligned}$$

其中 Hash 函数的输入为消息 M,经填充后的消息分成 L 个分组,分别是 $Y_0, Y_1, \cdots, Y_{L-1}$。

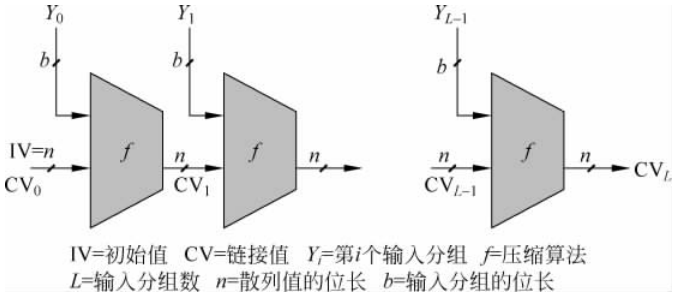

图 5.5 Hash 函数的一般结构

Hash 函数和 MAC 函数不同,不需要使用密钥,因此也觉得了 Hash 函数无法像图 5.1 中所示的那样单独提供对消息的认证,通常它和数字签名结合使用来提供认证性。在网络安全通信中,Hash 函数和对称密码、非对称密码结合使用以提供不同的安全服务。图 5.6 给出了几种基本应用。

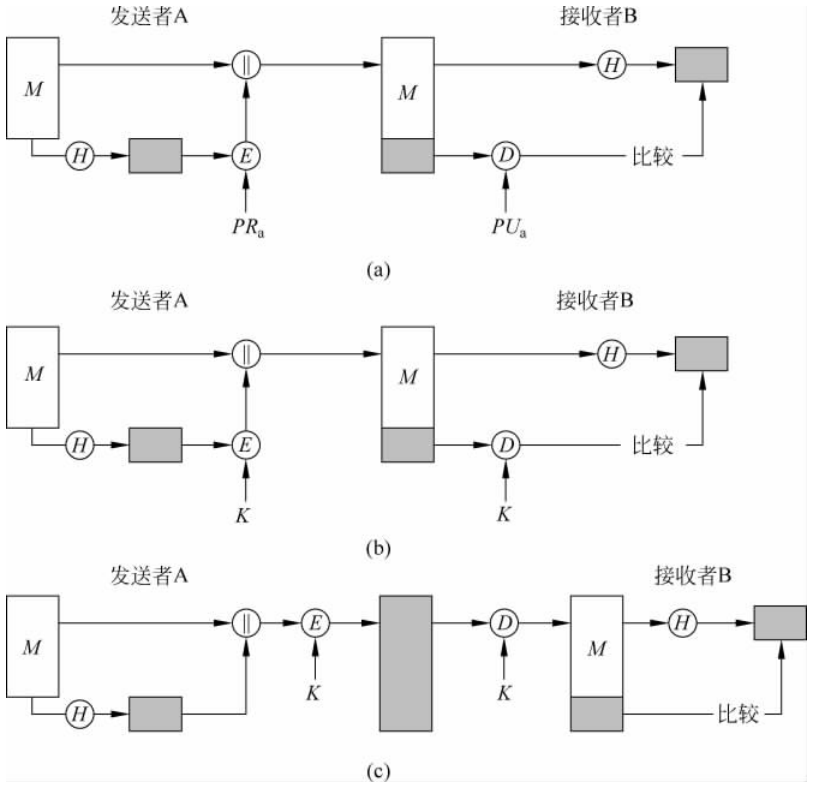

图 5.6 Hash 的基本应用

图 5.6(a)给出了 Hash 函数和数字签名的典型使用方法。对消息 M 的数字签名通常不是直接对消息进行计算,而是先使用 Hash 函数得到消息的散列值,再使用发送方的签名私钥 PR_a 对代表消息的散列值进行签名,这样既可以提供消息的认证性,又可以保证效率。图 5.6(b)所给出的消息认证方式和 MAC 有点类似,因为 Hash 函数不使用密钥,如果直接

对消息进行散列值计算，并和消息进行连接传送，则攻击者很容易篡改消息并相应地重新计算散列值。因此，对于计算出来的散列值，使用密钥加密的方法则可以避免发生上述问题，攻击者即便篡改了消息，也因为没有相应的加密密钥伪造消息散列值的密文。图 5.6(c)中提供的安全服务是保密性和认证性。除此之外，根据应用需求提供，Hash 函数还可以和其他密码函数结合使用提供不同的应用模式。

5.3.1　散列函数的安全要求

将第 5.3 节中给出的安全 Hash 函数需要满足的后面 3 个条件重新描述如下(即是 Hash 函数的安全要求)。

(1) 单向性：对任何给定的散列码 h，找到满足 $H(x)=h$ 的 x 在计算上是不可行的。

(2) 抗弱碰撞性：对任何给定的消息 x，找到满足 $y\neq x$ 且 $H(x)=H(y)$的 y 在计算上是不可行的。

(3) 抗强碰撞性：找到任何满足 $H(x)=H(y)$的偶对(x,y)在计算上是不可行的。

在图 5.5 所示的一般结构的 Hash 函数中，其输入消息被划分成 L 个固定长度的分组，每一分组长为 b 位，最后一个分组不足 b 位时需填充为 b 位，最后一个分组包含输入的总长度。由于输入中包含长度，所以攻击者必须找出具有相同散列值且长度相等的两条消息，或者找出两条长度不等但加入消息长度信息后散列值相同的消息，从而增加了攻击的难度。Merkle 和 Damgard 发现，如果压缩函数具有抗碰撞能力，那么迭代 Hash 函数也具有抗碰撞能力，因此 Hash 函数常使用上述迭代结构，这种结构可用于对任意长度的消息产生安全 Hash 函数。

1. 生日攻击(Birthday Attack)

如果攻击者希望伪造消息 M 的签名来欺骗接收者，则他需要找到满足 $H(M')=H(M)$ 的 M'来替代 M。对于生成 64 位散列值的散列函数，平均需要尝试 2^{63}次以找到 M'。但是建立在生日悖论上的生日攻击法，则会更有效。

对于上述问题换种说法：假设一个函数有 n 个函数值，且已知一个函数值 $H(x)$。任选 k 个任意数作为函数的输入值，则 k 必须为多大才能保证至少找到一个输入值 y 且 $H(x)=H(y)$的概率大于 0.5?

对于任意的 y，能够满足 $H(x)=H(y)$的概率是 $1/n$，反之，满足 $H(x)\neq H(y)$的概率为 $1-1/n$。对于所选的 k 个任意的输入值 y 都没有一个满足 $H(x)=H(y)$的概率则是 $(1-1/n)^k$。这样，至少有一个 y 满足 $H(x)=H(y)$的概率是 $1-(1-1/n)^k$。

二项式定理可描述如下：

$$(1-a)^k = 1-ka+\frac{k(k-1)}{2!}a^2-\frac{k(k-1)(k-2)}{3!}a^3+\cdots$$

当 a 趋近 0 时，$(1-a)^k$ 趋近 $1-ka$，因此至少有一个 y 满足 $H(x)=H(y)$的概率约为 $1-(1-1/n)^k\approx 1-(1-k/n)=k/n$。当 $k>n/2$ 时，这个概率将超过 0.5。

若散列值为 m 位，则可能有 2^m 个散列值，使上述概率为 0.5 的 k 为 $k=2^{(m-1)}$。即对于生成 64 位散列值的散列函数而言，攻击者至少需要尝试 2^{63}对明文，才能有大约 0.5 的成功概率。这个结果似乎表明选择 64 位的散列函数是安全的，但事实并非如此。Yuval 提出的“生日悖论”对于之前提到的 Rabin 散列函数的数字签名攻击，只需要 2^{32}次的运算。

在讨论 Yuval 的方法之前,先来解释一下“生日悖论”的数学背景。我们可以这样描述这类问题:k 为多大时,在 k 个人中至少找到两个人的生日相同的概率大于 0.5?不考虑 2 月 29 日,并且假定每个生日出现的概率相同。

首先 k 个人的生日排列的总数目是 365^k。这样,k 个人有不同生日的排列数为:

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

因此,k 个人有不同生日的概率为不重复的排列数除以总数目,得到:

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

则,k 个人中,至少找到两个人同日出生的概率是:

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

当 $k=100$ 时,$P(365,100)=0.9999997$,这意味着只有一个重复的概率接近于百分百。如果只考虑同日出生的概率超过 0.5 时,则根据 $P(365,23)=0.5037$,k 只需要为 23。

Yuval 的生日攻击法描述如下。

(1) 合法的签名方对于其认为合法的消息愿意使用自己的私钥对该消息生成的 m 位的散列值进行数字签名。

(2) 攻击者为了伪造一份有(1)中的签名者签名的消息,首先产生一份签名方将会同意签名的消息,再产生出该消息的 $2^{m/2}$ 种不同的变化,且每一种变化表达相同的意义(如:在文字中加入空格、换行字符)。然后,攻击者再伪造一条具有不同意义的新的消息,并产生出该伪造消息的 $2^{m/2}$ 种变化。

(3) 攻击者在上述两个消息集合中找出可以产生相同散列值的一对消息。根据“生日悖论”理论,能找到这样一对消息的概率是非常大的。如果找不到这样的消息,攻击者将再产生一条有效的消息和伪造的消息,并增加每组中的明文数目,直至成功为止。

(4) 攻击者用第一组中找到的明文提供给签名方要求签名,这样,这个签名就可以被用来伪造第二组中找到的明文的数字签名。这样,即使攻击者不知道签名私钥也能伪造签名。

生日攻击表明 Hash 值的长度必须达到一定的值,如果过短,则容易受到穷举攻击,如一个 40 位的 Hash 值,只需要穷举一百万次。一般建议 Hash 值需要 160 位,SHA-1 的最初选择是 128 位,后来改为 160 位,这就是为了防止利用生日攻击原理穷举 Hash 值。

2. 中间相遇攻击法(Meet in the Middle Attack)

中间相遇攻击是生日攻击的一种变形,它不比较散列值,而是比较处理链中的中间变量。这种攻击主要适用于攻击具有分组链接结构的 Hash 函数。其基本原理为:将消息分成两部分,对伪造消息的第一部分从初始值开始逐步向中间阶段产生 r_1 个变量;对伪造消息的第二部分从 Hash 结果开始逐步退回中间阶段产生 r_2 个变量。在中间阶段有一个匹配的概率与生日攻击成功的概率一样。

这种攻击方法让攻击者可以仅根据已知的明文及其数字签名,来任意伪造其他明文的数字签名。

(1) 根据已知数字签名的明文,先产生散列函数值 h。

(2) 再根据意图伪造签名的明文，将其分成每个 64 位长的明文分组 $Q_1, Q_2, \cdots, Q_{N-2}$。Hash 函数的压缩算法为 $h_i = E_{Qi}[h_{i-1}]$，$1 \leqslant i \leqslant N-2$。

(3) 任意产生 2^{32} 个不同的 X，对每个 X 计算 $E_X[h_{N-2}]$。同样地，任意产生 2^{32} 个不同的 Y，对每个 Y 计算 $D_Y[G]$，D 是相对应 E 的解密函数。

(4) 根据“生日悖论”，有很大的概率可以找到很多 X 及 Y 满足 $E_X[h_{N-2}] = D_Y[G]$。

(5) 如果找到了这样的 X 和 Y，攻击者重新构造一个明文：$Q_1, Q_2, \cdots, Q_{N-2}, X, Y$。这个新的明文的散列值也为 h，因此攻击者可以使用已知的数字签名为这个构造的明文伪造新的明文的签名。

5.3.2　SM3

SM3 是国家商用密码管理局于 2010 年 12 月 17 日发布的商用散列算法。本标准适用于商用密码应用中的数字签名和验证、消息认证码的生成与验证以及随机数的生成，可满足多种密码应用的安全需求。同时，本标准还可为安全产品生产商提供产品和技术的标准定位以及标准化的参考，提高安全产品的可信性与互操作性。SM3 是将长度为 $L(L < 2^{64})$ 比特的消息 m，压缩输出为 256 比特的摘要值，如图 5.7 所示。

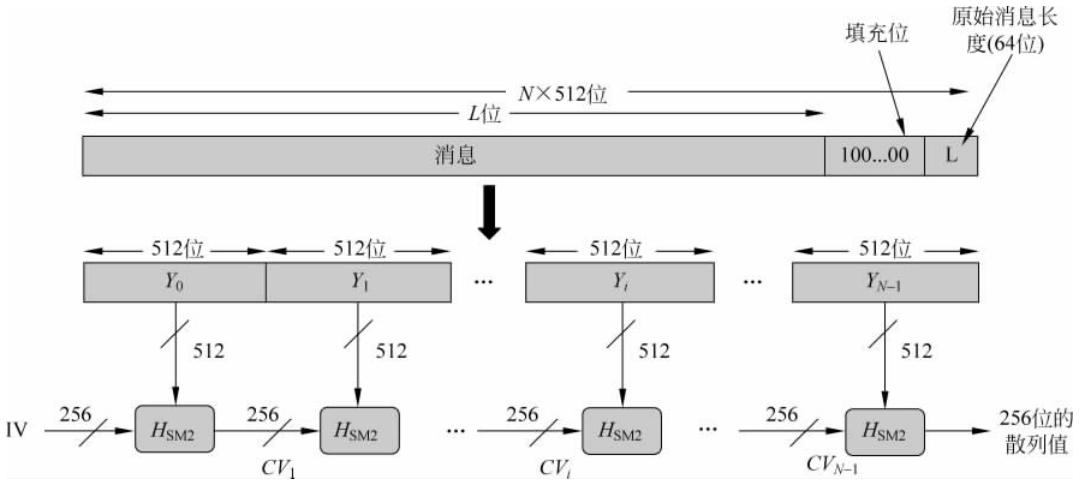

图 5.7　SM2 主要处理过程

SM3 算法描述如下。

1) 填充与分组

SM3 对输入消息先填充，再分组，分组的长度为 512 比特，消息填充后的比特长度为 512 的倍数。填充方法如下：假设消息 m 的长度为 L 比特。首先将比特“1”添加到消息的末尾，再添加 k 个“0”，k 是满足 $L+1+k \equiv 448 \bmod 512$ 的最小的非负整数。然后再添加一个 64 位比特串，该比特串是长度 L 的二进制表示。填充后的消息 m' 的比特长度为 512 的倍数。

例如：对消息 01100001 01100010 01100011，其长度 $L=24$，经填充得到比特串：

$$\underbrace{01100001\ 01100010\ 01100011\ 1\ \ 00\cdots00}_{423\text{比特}}\ \ \underbrace{00\cdots011000}_{64\text{比特}}$$

1的二进制表示

2) 初始化

SM3 采用 8 个寄存器 ABCDEFGH 存储散列运算中间结果和最终结果,首先对寄存器值进行初始化为 IV=7380166f 4914b2b9 172442d7 da8a0600 a96f30bc 163138aa e38dee4d b0fb0e4e。这些字节采用高端(big-endian)格式存储,高端格式存储规定:左边为高有效位,右边为低有效位。数的高阶字节放在存储器的低地址,数的低阶字节放在存储器的高地址。

3) 压缩过程

(1) 迭代过程。

以 512 位的分组为单位进行迭代压缩计算,将填充后的消息 m' 按 512 比特进行分组,

$$m' = Y_0 Y_1 \cdots Y_{n-1}$$

其中 $n=(L+k+65)/512$。

对 m' 按下列方式迭代:

$$\begin{aligned}&\text{FOR } i = 0 \text{ TO } n-L\\ &V^{i+1} = \mathrm{CF}(V_I, Y_i)\\ &\text{ENDFOR}\end{aligned}$$

其中 CF 是压缩函数,V_0 为 256 比特初始值 IV,Y_i 为填充后的消息分组,迭代压缩的结果为 V_n。

(2) 消息扩展。

将消息分组 Y_i 按以下方法扩展生成 132 个字 $W_0, W_1, \cdots, W_{67}, W'_0, W'_1, \cdots, W'_{63}$,用于压缩函数 CF:

a) 将消息分组 Y_i 划分为 16 个字,即 $W_0, W_1, \cdots, W_{15}$。

b) FOR $j=16$ TO 67

$W_j \leftarrow P_1(W_{j-16} \oplus_{j-9} \oplus (W_{j-3} \lll 15)) \oplus (W_{j-13} \lll 7) \oplus W_{j-6}$

ENDFOR

c) FOR $j=0$ TO 63

$W'_j = W_j \oplus W_{j+4}$

ENDFOR

其中,⊕表示 32 比特异或运算,$\lll k$ 表示循环左移 k 比特运算,P_1 为置换函数,计算方式见下面部分。

(3) 压缩函数。

令 A、B、C、D、E、F、G、H 为字寄存器,SS1、SS2、TT1、TT2 为中间变量,压缩函数 $V_{i+1}=\mathrm{CF}(V_i; Y_i)$,$0 \leqslant i \leqslant n-1$。计算过程描述如下:

ABCDEFGH$\leftarrow V_i$

FOR $j=0$ TO 63

SS1$\leftarrow$((A$\lll$12)+E+($T_j \lll j$))$\lll$7

SS2$\leftarrow$SS1 ⊕(A$\lll$12)

TT1$\leftarrow$$FF_j$(A,B,C)+D+SS1+$W'_j$

TT2$\leftarrow$$GG_j$(E,F,G)+H+SS1+$W_j$

D$\leftarrow$C

C$\leftarrow$B$\lll$9

B←A

A←TT1

H←G

G←F<<<19

F←E

E←P_0(TT2)

ENDFOR

V_{i+1}←ABCDEFGH ⊕ V_i

最后输出即为摘要值。

其中，T_j 为常量，其值为：

$$T_j = \begin{cases} 79cc4519 & 0 \leqslant j \leqslant 15 \\ 7a879d8a & 16 \leqslant j \leqslant 63 \end{cases}$$

FF_j 和 GG_j 是布尔函数，计算如下：

$$FF_j(X,Y,Z) = \begin{cases} X \oplus Y \oplus Z & 0 \leqslant j \leqslant 15 \\ (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z) & 16 \leqslant j \leqslant 63 \end{cases}$$

$$GG_j(X,Y,Z) = \begin{cases} X \oplus Y \oplus Z & 0 \leqslant j \leqslant 15 \\ (X \wedge Y) \vee (\neg X \wedge Z) & 16 \leqslant j \leqslant 63 \end{cases}$$

其中的一些符号含义表示为：

∧：32 比特与运算

∨：32 比特或运算

¬：32 比特非运算

＋：mod2^{32}算术加运算

P_0 为压缩函数部分的置换函数，P_1 消息扩展部分的置换函数，按照下面方式进行运算。

$$P_0(X) = X \oplus (X <<< 9) \oplus (X <<< 17)$$

$$P_1(X) = X \oplus (X <<< 15) \oplus (X <<< 23)$$

式中 X 为字。

5.3.3　MD5

MD5(Message-Digest Algorithm 5)是由 Ronald L. Rivest(RSA 算法中的 R)在 20 世纪 90 年代初开发出来的，经 MD2、MD3 和 MD4 发展而来。它比 MD4 复杂，但设计思想类似，同样生成一个 128 位的信息散列值。其中，MD2 是为 8 位计算机做过设计优化的，而 MD4 和 MD5 却是面向 32 位的计算机。

Osrschot 和 Wiener 曾为了攻击 MD5，使用穷举攻击搜寻碰撞的函数，并耗费一千万美元设计了一台碰撞搜寻计算机，它能在 24 天内找到一个碰撞。但即便如此，从 1991 年起的十多年里，并没有出现替代 MD5 的算法或 MD6，并且，由于使用 MD5 算法无须支付任何专利费，所以在实际中的应用非常广泛。直到 2004 年 8 月，在美国召开的国际密码学会议(Crypto'2004)上，王小云教授给出破解 MD5、HAVAL-128、MD4 和 RIPEMD 算法的报告。她给出了一个非常高效的寻找碰撞的方法，可以在数个小时内找到 MD5 的碰撞。

MD5 算法的描述：

MD5 的输入可以是任意长度的消息，其输出是 128 位的消息散列值。由于 MD5 的设计是针对 32 位处理器的，因此 MD5 内的所有基本运算都是针对 32 位运算单元的。其算法的主要过程如下。

(1) 填充消息：任意长度的消息首先需要进行填充处理，使得填充后的消息总长度与 448 模 512 同余(即填充后的消息长度≡448 mod 512)。填充的方法是在消息后面添加一位 1，后续都是 0。

(2) 添加原始消息长度：在填充后的消息后面再添加一个 64 位的二进制整数表示在填充前原始消息的长度。这时经过处理后的消息长度正好是 512 位的倍数。

(3) 初始值(IV)的初始化：MD5 中有 4 个 32 位缓冲区，用(A,B,C,D)表示，用来存储散列计算的中间结果和最终结果，缓冲区中的值被称为链接变量(Chaining Variable)。首先将其分别初始化为 A=0x01234567，B=0x89abcdef，C=0xfedcba98，D=0x76543210。这些值以高端格式存储，即字节的最高有效位存于低地址字节位置。

(4) 以 512 位的分组为单位对消息进行循环散列计算，如图 5.8 所示，经过处理的消息，以 512 位为单位，分成 N 个分组，为 Y_0,Y_1,…,Y_{N-1}。MD5 对每个分组进行散列处理。每一轮的处理会对(A,B,C,D)进行更新。

(5) 输出散列值：所有的 N 个分组消息都处理完后，最后一轮得到的 4 个缓冲区的值即为整个消息的散列值。

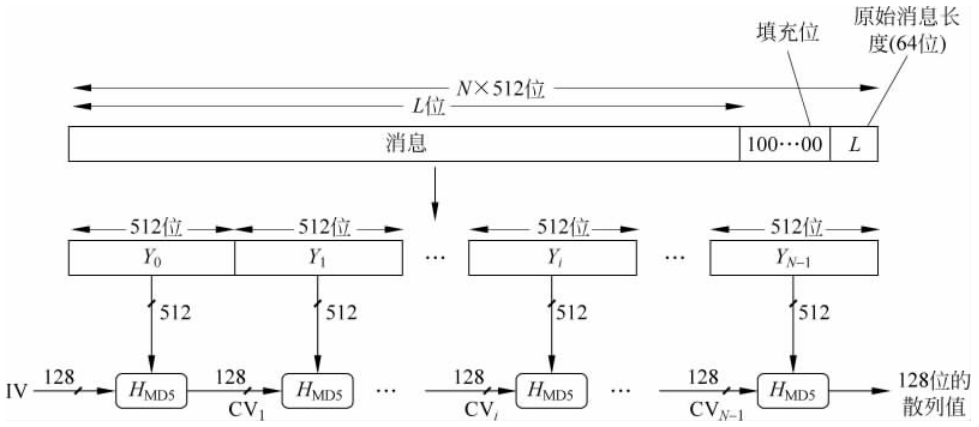

图 5.8 MD5 主要处理过程

在第(4)步中的循环散列计算共有 4 轮(见图 5.9)，每轮循环都很相似，进行 16 次操作。在第 1 轮的第 1 个步骤开始处理时，将 A、B 和 C、D 的值保存在另外的单元，假设为 AA、BB、CC 和 DD 中。然后每次操作对 A、B、C 和 D 中的其中 3 个做 1 次非线性函数运算，然后将所得结果加上第 4 个变量、消息的一个子分组和一个常数，再将所得结果向右环移一个不定的数。最后得到的结果再加上之前保存在 AA、BB、CC 或 DD 中的值，这里的“+”指的是 mod 2^{32} 的模加运算。得到的新的 4 个 32 字作为 A、B、C 和 D 的新的值。然后继续使用下一分组进行运算，最后输出的 A、B、C 和 D 的级联即是整个消息的 128 位散列值。

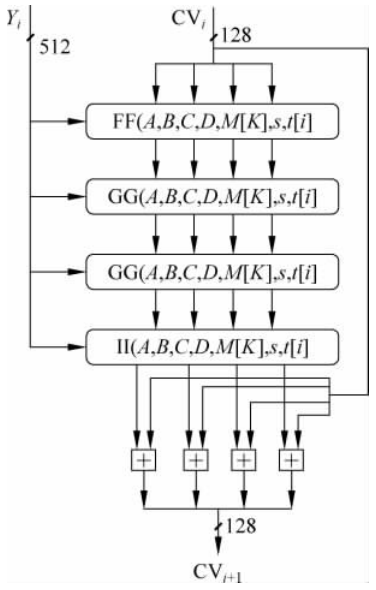

图 5.9　MD5 算法一次循环的处理过程

每次操作中用到的 4 个非线性函数为：

$$F(X,Y,Z) = (X\&Y) \mid ((\sim X)\&Z)$$
$$G(X,Y,Z) = (X\&Z) \mid (Y\&(\sim Z))$$
$$H(X,Y,Z) = X\hat{}Y\hat{}Z$$
$$I(X,Y,Z) = Y\hat{}(X \mid (\sim Z))$$

其中，& 是与，| 是或，～是非，^是异或。

将每次处理的 512 位的消息分组再分成 32 位一组，共 16 组，表示为 M_k，$k=0,\cdots,15$，则在每次的 4 轮运算中的计算方法是：

FF(A,B,C,D,M_j,s,t_i) 表示 $A = B + ((A + (F(B,C,D) + M_j + t_i) \lll s)$

GG(A,B,C,D,M_j,s,t_i) 表示 $A = B + ((A + (G(B,C,D) + M_j + t_i) \lll s)$

HH(A,B,C,D,M_j,s,t_i) 表示 $A = B + ((A + (H(B,C,D) + M_j + t_i) \lll s)$

II(A,B,C,D,M_j,s,t_i) 表示 $A = B + ((A + (I(B,C,D) + M_j + t_i) \lll s)$

其中“$\lll s$”表示循环左移 s 位。则 4 轮(每轮 16 步，共 64 步)的操作如下所示。

1) 第 1 轮

FF$(A,B,C,D,M_0,7,0xd76aa478)$

FF$(D,A,B,C,M_1,12,0xe8c7b756)$

FF$(C,D,A,B,M_2,17,0x242070db)$

FF$(B,C,D,A,M_3,22,0xc1bdceee)$

FF$(A,B,C,D,M_4,7,0xf57c0faf)$

FF$(D,A,B,C,M_5,12,0x4787c62a)$

FF$(C,D,A,B,M_6,17,0xa8304613)$

FF$(B,C,D,A,M_7,22,0xfd469501)$

FF(A,B,C,D,M_8,7,0x698098d8)

FF(D,A,B,C,M_9,12,0x8b44f7af)

FF(C,D,A,B,M_{10},17,0xffff5bb1)

FF(B,C,D,A,M_{11},22,0x895cd7be)

FF(A,B,C,D,M_{12},7,0x6b901122)

FF(D,A,B,C,M_{13},12,0xfd987193)

FF(C,D,A,B,M_{14},17,0xa679438e)

FF(B,C,D,A,M_{15},22,0x49b40821)

2) 第2轮

GG(A,B,C,D,M_1,5,0xf61e2562)

GG(D,A,B,C,M_6,9,0xc040b340)

GG(C,D,A,B,M_{11},14,0x265e5a51)

GG(B,C,D,A,M_0,20,0xe9b6c7aa)

GG(A,B,C,D,M_5,5,0xd62f105d)

GG(D,A,B,C,M_{10},9,0x02441453)

GG(C,D,A,B,M_{15},14,0xd8a1e681)

GG(B,C,D,A,M_4,20,0xe7d3fbc8)

GG(A,B,C,D,M_9,5,0x21e1cde6)

GG(D,A,B,C,M_{14},9,0xc33707d6)

GG(C,D,A,B,M_3,14,0xf4d50d87)

GG(B,C,D,A,M_8,20,0x455a14ed)

GG(A,B,C,D,M_{13},5,0xa9e3e905)

GG(D,A,B,C,M_2,9,0xfcefa3f8)

GG(C,D,A,B,M_7,14,0x676f02d9)

GG(B,C,D,A,M_{12},20,0x8d2a4c8a)

3) 第3轮

HH(A,B,C,D,M_5,4,0xfffa3942)

HH(D,A,B,C,M_8,11,0x8771f681)

HH(C,D,A,B,M_{11},16,0x6d9d6122)

HH(B,C,D,A,M_{14},23,0xfde5380c)

HH(A,B,C,D,M_1,4,0xa4beea44)

HH(D,A,B,C,M_4,11,0x4bdecfa9)

HH(C,D,A,B,M_7,16,0xf6bb4b60)

HH(B,C,D,A,M_{10},23,0xbebfbc70)

HH(A,B,C,D,M_{13},4,0x289b7ec6)

HH(D,A,B,C,M_0,11,0xeaa127fa)

HH(C,D,A,B,M_3,16,0xd4ef3085)

HH(B,C,D,A,M_6,23,0x04881d05)

HH(A,B,C,D,M_9,4,0xd9d4d039)

HH(D,A,B,C,M_{12},11,0xe6db99e5)

HH(C,D,A,B,M_{15},16,0x1fa27cf8)

HH(B,C,D,A,M_2,23,0xc4ac5665)

4) 第 4 轮

II(A,B,C,D,M_0,6,0xf4292244)

II(D,A,B,C,M_7,10,0x432aff97)

II(C,D,A,B,M_{14},15,0xab9423a7)

II(B,C,D,A,M_5,21,0xfc93a039)

II(A,B,C,D,M_{12},6,0x655b59c3)

II(D,A,B,C,M_3,10,0x8f0ccc92)

II(C,D,A,B,M_{10},15,0xffeff47d)

II(B,C,D,A,M_1,21,0x85845dd1)

II(A,B,C,D,M_8,6,0x6fa87e4f)

II(D,A,B,C,M_{15},10,0xfe2ce6e0)

II(C,D,A,B,M_6,15,0xa3014314)

II(B,C,D,A,M_{13},21,0x4e0811a1)

II(A,B,C,D,M_4,6,0xf7537e82)

II(D,A,B,C,M_{11},10,0xbd3af235)

II(C,D,A,B,M_2,15,0x2ad7d2bb)

II(B,C,D,A,M_9,21,0xeb86d391)

常数 t_i 可以如下选择：在第 i 步中，t_i 是 $2^{32}\times\text{abs}(\sin(i))$的整数部分，$i$ 的单位是弧度。

5.3.4　SHA-512

美国国家标准局(NIST)为了配合数字签名标准(DSA)，在 1993 年对外公布了安全散列函数(SHA)，并公布为联邦信息处理标准(FIPS 180)，其设计的方法是依据已有的 MD4 算法，所以其基本框架与 MD4 类似。1995 年 NIST 发布了 SHA 的修订版(FIPS 180-1)，通常称之为 SHA-1，SHA-1 产生 160 位的散列值。2002 年，NIST 再次发布了修订版(FIPS 180-2)，其中给出了 3 种新的 SHA 版本，散列值长度依次为 256、384 和 512 位，分别称作 SHA-256、SHA-384 和 SHA-512(见表 5.1)。这些新的版本和 SHA-1 具有相同的基础结构，使用了相同的模算术和二元逻辑运算。2005 年，NIST 宣布了将逐步废除 SHA-1，到 2010 年，逐步转而使用 SHA 的其他更高位长的版本。2005 年，王小云带领的研究小组研究出了一种攻击，用 2^{69} 次操作可以找到两个独立的消息使它们有相同的 SHA-1 值，而以前认为要找到一个 SHA-1 碰撞需要 2^{80} 次的操作，所需操作大为减少。这意味着，对于 SHA 的使用需要选择更高位数的版本。

表 5.1 SHA 参数比较

	SHA-1	SHA-256	SHA-384	SHA-512
散列值长度	160	256	384	512
消息长度	$<2^{64}$	$<2^{64}$	$<2^{128}$	$<2^{128}$
分组长度	512	512	1024	1024
字长度	32	32	64	64
步骤数	80	64	80	80
安全性	80	128	192	256

注:

(1) 所有的长度以比特为单位。

(2) 安全性是指对输出长度为 n 位 Hash 函数的生日攻击产生碰撞的工作量大约为 $2^{n/2}$。

1. SHA-512 算法

该算法的输入是最大长度小于 2^{128} 位的消息,输出是 512 位的散列值,输入消息以 1024 位的分组为单位进行处理。输出摘要的总体过程遵循图 5.10 所示的一般结构。和 MD5 类似,其过程包含下列步骤。

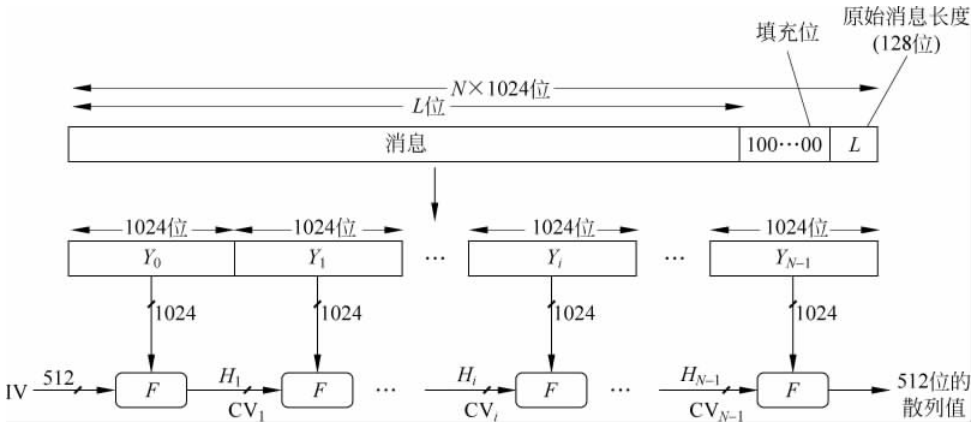

图 5.10 SHA-512 主要处理过程

(1) 对消息进行填充。对原始消息进行填充使其长度与 896 模 1024 同余(即填充后的消息长度≡896 mod 1024)。即使原始消息已经满足上述长度要求,仍然需要进行填充,因此填充位数在 1～1024 之间。填充部分由一个 1 和后续的 0 组成。

(2) 添加消息长度信息。在填充后的消息后添加一个 128 位的块,用来说明填充前消息的长度,表示为一个无符号整数(最高有效字节在前)。至此,产生了一个长度为 1024 位整数倍的扩展消息。如图 5.10 所示,扩展的消息被表示为一串长度为 1024 位的消息分组 $Y_1, Y_2, \cdots, Y_N$,因此扩展消息的长度为 $N \times 1024$ 位。

(3) 初始化 Hash 缓冲区。Hash 函数计算的中间结果和最终结果保存在 512 位的缓冲区中,分别用 64 位的寄存器(A,B,C,D,E,F,G,H)表示,并将这些寄存器初始化为下列 64 位的整数(十六进制值)。

A=0x6A09E667F3BCC908　　E=0x510E527FADE682D1

B=0xBB67AE8584CAA73B　　F=0x9B05688C2B3E6C1F

C＝0x3C6EF372FE94F82B　　G＝0x1F83D9ABFB41BD6B

D＝0xA54FF53A5F1D36F1　　H＝0x5BE0CD19137E2179

这些值以高端格式存储，即字的最高有效字节存于低地址字节位置（最左边）。这些字的获取方式如下：前 8 个素数取平方根，取分数部分的前 64 位。

（4）以 1024 位分组（16 个字）为单位处理消息。处理算法的核心是需要进行 80 轮运算的模块，即图 5.9 中的 F。图 5.11 给出了 F 的逻辑原理：每一轮都把 512 位缓冲区的值 ABCDEFGH 作为输入，并更新缓冲区的值。第 1 轮时，缓冲区的值是中间的 Hash 值 H_{i-1}。每一轮使用一个 64 位的值 W_t（$0 \leqslant t \leqslant 79$）该值由当前被处理的 1024 位消息分组 Y_i 导出，导出算法是后面将要讨论的消息调度算法。每一轮还将使用常数 K_t（$0 \leqslant t \leqslant 79$）。这些常数的获得方法：前 80 个素数取 3 次根，取小数部分的前 64 位。这些常数提供了 64 位随机串集合，可以消除输入数据里的任何规则性。第 80 轮的输出和第 1 轮的输入 H_{i-1} 相加产生 H_i。缓冲区里的 8 个字和 H_{i-1} 里的相应字独立进行模 2^{64} 的加法运算。

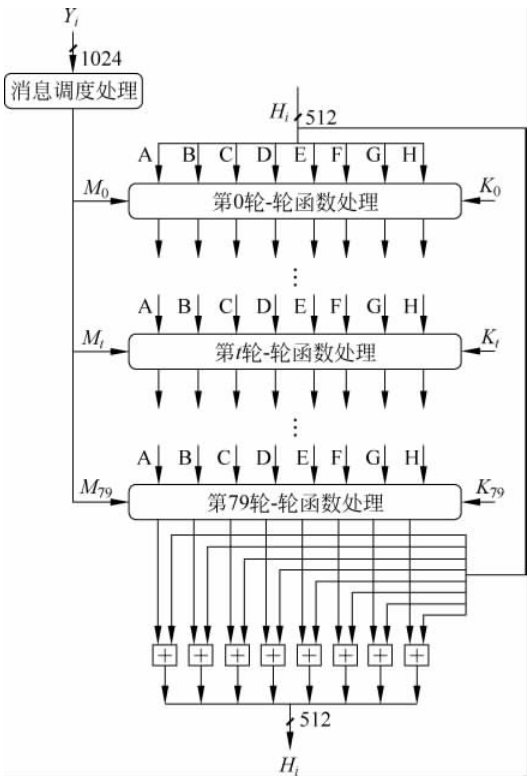

图 5.11　SHA-512 每一步的核心处理

（5）输出。所有的 N 个 1024 位分组都处理完以后，最后输出的即是 512 位的消息散列值。

从总体上看，SHA-512 的运算如下所示。

$$H_0 = \mathrm{IV}$$

$$H_i = \mathrm{SUM}_{64}(H_{i-1}, \mathrm{ABCDEFGH}_i)$$

$$\mathrm{MD} = H_N$$

其中，IV 为上述算法第（3）步里定义的 ABCDEFGH 缓冲区的初始值；$\mathrm{ABCDEFGH}_i$ 为第 i

个消息分组处理的最后一轮的输出；N 为消息(包括填充和长度域)里的分组数；SUM_{64} 为对输入对里的每个字进行独立的模 2^{64} 加；MD 为最后的消息散列值。

2. 消息调度处理

每一轮的 $W_t(0\leqslant t\leqslant 79)$ 的值由当前被处理的 1024 位消息分组 Y_i 导出。其导出算法如图 5.12 所示，前 16 个 $W_t(0\leqslant t\leqslant 15)$ 直接取自当前消息分组的 16 个字。余下的值按如下方式导出：

$$W_t = \sigma_1^{512}(W_{t-2}) + W_{t-7} + \sigma_0^{512}(W_{t-15}) + W_{t-16}$$

其中，$\sigma_0^{512}(x)=\mathrm{ROTR}^1(x)+\mathrm{ROTR}^8(x)+\mathrm{SHR}^7(x)$。

$\sigma_1^{512}(x) = \mathrm{ROTR}^{19}(x)+\mathrm{ROTR}^{61}(x)+\mathrm{SHR}^6(x)$。

$\mathrm{ROTR}^n(x)$=对 64 位的变量 x 循环右移 n 位。

$\mathrm{SHR}^n(x)$=对 64 位变量 x 向左移动 n 位，右边填充 0。

因此，在前 16 步处理过程中，W_t 的值等于消息分组里的相应字。对于余下的 64 步，W_t 的值由其前面的 4 个值的异或形成的值构成，要对 4 个值中的两个进行移位和循环移位操作。

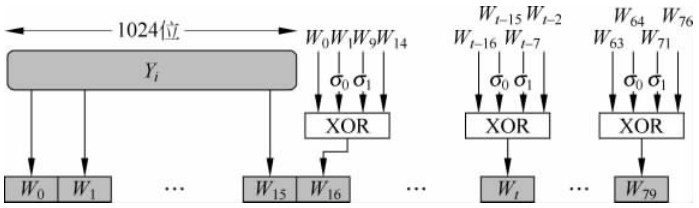

图 5.12 SHA-512 每步操作中的消息调度处理

3. SHA-512 的轮函数

SHA-512 中最核心的处理就是对单个 512 分组处理的 80 轮的每一轮的处理，其运算方法如下所示。

$$T_1 = h + \mathrm{Ch}(e,f,g) + \left(\sum_1^{512} e\right) + W_t + K_t$$

$$T_2 = \left(\sum_0^{512} a\right) + \mathrm{Maj}(a,b,c)$$

$$a = T_1 + T_2$$

$$b = a$$

$$c = b$$

$$d = c$$

$$e = d + T_1$$

$$f = e$$

$$g = f$$

$$h = g$$

其中：

- t 为步骤数，$0\leqslant t\leqslant 79$。

- Ch(e,f,g)=(e AND f)⊕(NOT e AND g)条件函数,如果 e,则 f,否则 g。
- Maj(a,b,c)=(a AND b)⊕(a AND c)⊕(b AND c),函数为真,仅当变量的多数(2 或 3)为真。
- $\left(\sum_{0}^{512} \mathrm{a}\right) = \mathrm{ROTR}^{28}(\mathrm{a}) \oplus \mathrm{ROTR}^{34}(\mathrm{a}) \oplus \mathrm{ROTR}^{39}(\mathrm{a})$。
- $\left(\sum_{1}^{512} \mathrm{e}\right) = \mathrm{ROTR}^{14}(\mathrm{e}) \oplus \mathrm{ROTR}^{18}(\mathrm{e}) \oplus \mathrm{ROTR}^{41}(\mathrm{e})$。
- W_t 为 64 位,从当前的 512 位消息分组导出。
- K_t 为 64 位常数。
- “+”为模 2^{64} 加。

5.3.5　HMAC

Hash 函数是不使用密钥的,不能像 MAC 一样使用。正如第 5.2 节所讨论的,MAC 是主要基于对称密码算法设计的,如第 5.2.2 节中讨论的 MAC 算法。但目前,研究者提出了一些将 Hash 函数用于构建 MAC 算法的方案。HMAC 是其中之一,并已经成为 FIPS 标准发布(FIPS 198),后被使用于 IPSec 和 SSL 协议中。

1. HMAC 的设计目标

HMAC 的设计目标如下。

- 可以直接使用现有的 Hash 函数。
- 不针对于某一个 Hash 函数,可以根据需要更换 Hash 函数模块。
- 可保持 Hash 函数的原有性能,不能过分降低其性能。
- 对密钥的使用和处理应较简单。
- 如果已知嵌入的 Hash 函数的强度,则可以知道认证机制抵抗密码分析的强度。

因此,HMAC 中使用的 Hash 函数并不局限于某一种 Hash 函数,所以使用不同的 Hash 函数,HMAC 将有不同的实现,如 HMAC-SHA、HMAC-MD5 等。

2. HMAC 算法

图 5.13 给出了 HMAC 的总体结构,其中的符号定义如下所示。

- H:嵌入的 Hash 函数(如 MD5、SHA-1、RIPEMD-160)。
- IV:作为 Hash 函数输入的初始值。
- M:HMAC 的消息输入(包括使用的 Hash 函数中定义的填充位)。
- Y_i:M 的第 i 个分组,$0 \leqslant i \leqslant (L-1)$。
- L:M 中的分组数。
- b:每一分组所含的位数。

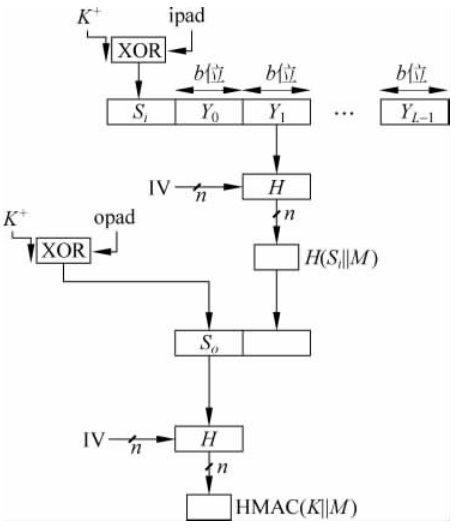

图 5.13　HMAC 的结构

- n：使用的 Hash 函数所产生的散列值的位长。
- K：密钥，建议密钥长度大于 n。若密钥长度大于 b，则将密钥作为 Hash 函数的输入以产生一个 n 位的密钥。
- K^+：为使 K 为 b 位长而在 K 左边填充 0 后所得的结果。
- ipad：00110110(十六进制数 36)重复 $b/8$ 次的结果。
- opad：01011100(十六进制数 5C)重复 $b/8$ 次的结果。

HMAC 可描述为：

$$\mathrm{HMAC}_K = H[(K^+ \oplus \mathrm{opad}) \| H[(K^+ \oplus (\mathrm{ipad}) \| M]]$$

算法的处理流程如下所示。

(1) 在密钥 K 后面填充 0，得到 b 位的 K^+(例如，若 K 是 160 位，$b=512$，则在 K 中加入 44 个 0 字节 0×00)。

(2) K^+ 与 ipad 执行异或运算(位异或)产生 b 位的分组 S_i。

(3) 将 M 附于 S_i 后。

(4) 将 H 作用于第(3)步所得的结果。

(5) K^+ 与 opad 执行异或运算(位异或)产生 b 位的分组 S_o。

(6) 将第(4)步中的散列值附于 S_o 后。

(7) 将 H 作用于第(6)步所得出的结果，输出最终结果。

在上述操作中，K 与 ipad 异或后，其信息位有一半发生了变化；同样，K 与 opad 异或后，其信息位的另一半也发生了变化，这样，通过将 S_i 与 S_o 传给 Hash 算法中的压缩函数，可以从 K 伪随机地产生出两个密钥。

如图 5.14 所示，为了有效地实现 HMAC，HMAC 中多执行的 3 次 Hash 运算(对 S_i、S_o 和 $H(S_i \| M)$)可以采用预计算的方式先求出下面的值。

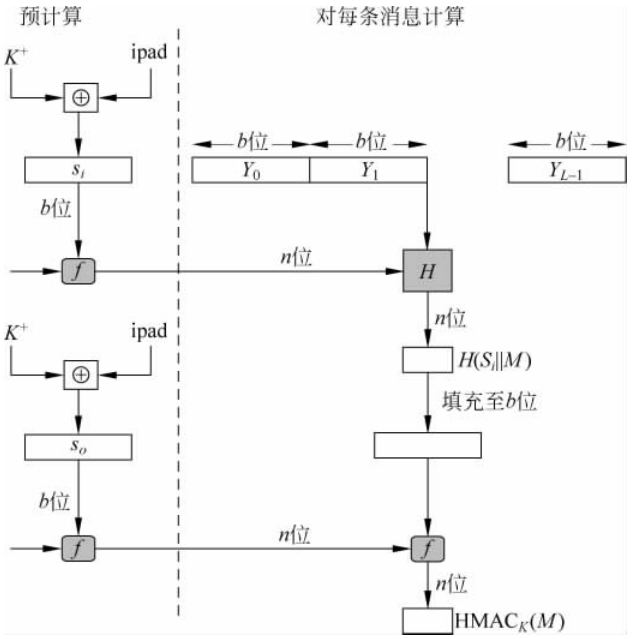

图 5.14 HMAC 的实现方案

$$f(\mathrm{IV},(K^{+}\oplus \mathrm{ipad}))$$
$$f(\mathrm{IV},(K^{+}\oplus \mathrm{opad}))$$

其中 $f(\mathrm{cv},\mathrm{block})$是 Hash 函数中的压缩函数，其输入是 n 位的链接值和 b 位的消息分组，输出是 n 位的链接变量。上述这些值只在初始化或密钥改变时才须计算。实际上，这些预先计算的值取代了 Hash 函数中的初始值 IV。这样的处理方式使得 HMAC 只须多执行一次压缩函数，但这种处理方式只有在对较长消息处理的时候可以显示出效果，对于较短的消息处理，则没有太大意义。

5.4　数 字 签 名

5.4.1　数字签名的基本概念

前面章节讨论的消息认证方法主要是保护通信双方之间的消息不被第三方篡改，但却无法防止通信双方互相欺骗。例如在下面的情形中，通信双方会产生某些纠纷。

(1) 在通信中，通信方 A 和 B 是通过共享的秘密钥对传输的消息计算 MAC 以提供认证。这样 B 可以伪造一个消息，并使用共享的密钥对其生成 MAC，然后声称这个消息是来自于 A 的。

(2) A 否认曾经发送过某条消息给 B，但事后他可以辩称 B 收到的这条消息是 B 伪造的，即否认自己的行为。

(3) B 收到 A 发送的某条消息后，出于某种原因，他否认收到过这条消息。

在上述情形中，由于通信方存在互不信任的情况，单纯地使用签名讨论的消息认证方法无法解决这些问题。数字签名是解决这些问题的最好选择，数字签名主要的功能是保证信息传输的完整性、发送者的身份认证、防止交易中发生否认现象。简单地说，数字签名技术可以解决如下问题。

- 否认：发送方否认发送过或签名过某条消息。
- 伪造：用户 A 伪造一份消息，并声称该消息来自 B。
- 冒充：用户 A 冒充其他用户接收或发送报文。
- 篡改：消息接收方对收到的消息进行篡改。

数字签名也是一种认证机制，它是公钥密码学发展过程中的一个重要组成部分，是公钥密码算法的典型应用。数字签名的应用过程是，数据源发送方使用自己的私钥对数据校验和(或)其他与数据内容有关的信息进行处理，完成对数据的合法“签名”，数据接收方则利用发送方的公钥来验证收到的消息上的“数字签名”，以确认签名的合法性。

数字签名需要满足以下条件。

- 签名的结果必须是与被签名的消息相关的二进制位串。
- 签名必须使用发送方某些独有的信息(发送者的私钥)，以防伪造和否认。
- 产生数字签名比较容易。
- 识别和验证签名比较容易。
- 给定数字签名和被签名的消息，伪造数字签名在计算上是不可行的。

• 保存数字签名的副本,并由第三方进行仲裁是可行的。

数字签名的典型使用方法如下(见图5.6(a))。

(1) 发送方计算消息的哈希值。

(2) 发送方使用自己的私钥对消息散列值进行计算,得到一个较短的数字签名串。

(3) 这个数字签名将和消息一起发送给接收方。

(4) 接收方首先从接收到的消息中用同样的散列函数计算出一个消息摘要,然后使用这个消息摘要、发送者的公钥以及收到的数字签名,进行数字签名合法性的验证。

数字签名技术是在网络虚拟环境中确认身份、提供消息完整性和保证消息来源的重要技术,可以提供和现实中的"亲笔签字"类似的效果,在技术和法律上都有保证。数字签名通常和Hash函数结合使用,用来向用户提供安全高效的数字签名的方法,被广泛应用在各种认证协议中和网络应用中,如电子商务中安全、方便的电子支付、数据传输的完整性、身份验证机制以及交易的不可否认性的实现。

5.4.2 数字签名方案

本节给出一些常见的数字签名方案及数字签名的应用。

1. Schnorr数字签名

1989年,Schnorr提出一签名算法,其算法安全性基于求解离散对数难题。

算法描述如下所示。

(1) 系统参数的选择。

p 和 q:满足 $q \mid p-1$,$q \geqslant 2^{160}$,$q \geqslant 2^{512}$。

g:$g \in Z_p$,满足 $g^q = 1 \bmod p$,$g \neq 1$。

H:为散列函数。

x:为用户的私钥,$1 < x < q$。

y:为用户的公钥,$y = g^x \bmod p$。

(2) 签名。

设要签名的消息为 M,$0 < M < p$。签名者随机选择一整数 k,$1 < k < q$,并计算:

$$e = H(r, M)$$

$$s = k - xe \bmod q$$

(e,s)即为 M 的签名。签名者将 M 连同(r,s)一起存放,或发送给验证者。

(3) 验证。

验证者获得 M 和(e,s),需要验证(e,s)是否是 M 的签名。

计算:

$$r' = g^s r^e \bmod p$$

检查 $H(r', M) = e$ 是否成立,若成立,则(e,s)为 M 的合法签名。

2. 数字签名标准(DSS)

1991年,美国国家标准局(NIST)发布了数字签名标准(FIPS PUB186),简称DSS(Digital Signature Standard)。DSS采用了SHA散列算法,给出了一种新的数字签名方法,即数字签名算法(DSA)。DSS被提出后,1996年又被稍做修改,2000年发布了该标准的扩

充版，即 FIP 186-2。DSA 的安全性是建立在求解离散对数难题之上的，算法基于 ElGamal 和 Schnorr 签名算法，其后面发布的最新版本还包括基于 RSA 和椭圆曲线密码的数字签名算法。这里给出的算法是最初的 DSA 算法。

DSA 只提供数字签名功能的算法，虽然它是一种公钥密码机制，但是不能像 RSA 和 ECC 算法那样还可以用于加密或密钥分配。

DSS 方法使用 Hash 函数产生消息的散列值，和随机生成的 k 作为签名函数的输入，签名函数依赖于发送方的私钥(PR_A)和一组参数，这些参数为一组通信伙伴所共有，我们可以认为这组参数构成全局公钥 PU_G。签名由两部分组成，标记为 r 和 s。

接收方对收到的消息计算散列值，和收到的签名(r,s)一起作为验证函数的输入，验证函数依赖于全局公钥和发送方公钥，若验证函数的输出等于签名中的 r，则签名合法。

DSA 算法的具体描述如下所示(见图 5.15)。

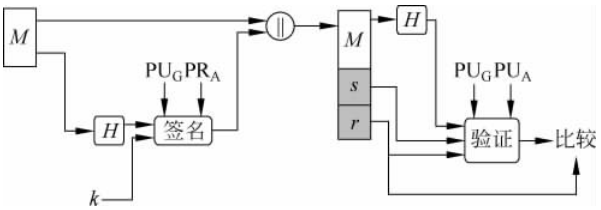

图 5.15　DSS 签名算法

(1) DSA 的系统参数的选择如下所示。

p：512 的素数，其中 $2^{L-1}<p<2^L$，$512\leqslant L\leqslant 1024$，且 L 是 64 的倍数，即 L 的位长在 512～1024 位之间并且其增量为 64 位。

q：160 位的素数且 $q|p-1$。

g：满足 $g=h^{(p-1)/q} \bmod p$。

H：为散列函数。

x：为用户的私钥，$0<x<q$。

y：为用户的公钥，$y=g^x \bmod p$。

p、q、g 为系统发布的公共参数，与公钥 y 公开；私钥 x 保密。

(2) 签名。

设要签名的消息为 M，$0<M<p$。签名者随机选择一整数 k，$0<k<q$，并计算：

$$r=(g^k \bmod p) \bmod q$$

$$s=[k^{-1}(H(M)+xr)] \bmod q$$

(r,s)即为 M 的签名。签名者将 M 连同(r,s)一起存放，或发送给验证者。

(3) 验证。

验证者获得 M 和(r,s)，需要验证(r,s)是否是 M 的签名。

首先检查 r 和 s 是否属于$[0,q]$，若不属于，则(r,s)不是签名值。

否则，计算：

$$w=s^{-1} \bmod q$$

$$u_1=(H(M)w) \bmod q$$

$$u_2=rw \bmod q$$

$$v=((g^{u1}y^{u2}) \bmod p) \bmod q$$

如果 $v=r$,则所获得的(r,s)是 M 的合法签名。

在 DSA 中,签名者和验证者都需要进行一次模 q 的求逆运算,这个运算是比较耗时的。Yen 和 Laih 提出了两种改进的方法,可以免去签名者或验证者的求逆运算,其方法如下所示。

(1) DSA 改进方法 1。

签名:$r=(g^k \bmod p) \bmod q$

$s=(rk-H(M))x^{-1} \bmod q$

验证:$t=r^{-1} \bmod q$

$v=(g^{h(M)t}y^{st} \bmod p) \bmod q$

判断 v 是否和 r 相等。

(2) DSA 改进方法 2。

签名:$r=(g^k \bmod p) \bmod q$

$s=(k(H(M)+xr)^{-1}) \bmod q$

验证:$t=sH(M) \bmod q$

$v=(g^t y^{sr} \bmod p) \bmod q$

判断 v 是否和 r 相等。

在上述方法中,有些计算可以预先完成。在改进方法 1 中,签名时会用到的 x^{-1},如果 x 不是经常更换,则 x^{-1}可以预先计算并保存以便多次使用,这样就可以省掉一次求逆运算。在改进方法 2 中,验证者无须计算逆元。即便对于初始 DSA,也可以采用预计算的方法提高效率:签名时所计算的 $g^k \bmod p$ 并不依赖于消息,因此可以预先计算出。用户还可以根据需要预先计算出多个可用于签名的 r,以及相应的 r^{-1},这样可以大大提高效率。

以上给出的签名方案是直接数字签名,或称为普通数字签名,包括 RSA、Schnorr、DSA、ECC、Fiat-Shamir、Guillou-Quisquarter、Schnorr、Ong-Schnorr-Shamir 等数字签名算法。这类数字签名只涉及通信双方,即签名方使用自己的私钥对整个消息或者对于消息的散列值进行签名,验证者使用签名者的公钥进行验证。即便发生纠纷,仲裁法也是根据密钥及签名值进行仲裁。该方案的有效性完全依赖于签名方的私钥。如果签名者的私钥丢失或者被攻击者获取,则有可能被他人伪造签名,这时产生纠纷后,仲裁者无法给出实时的判断。因此,在实际应用中,除了普通数字签名外,还有些特殊的签名方案,更多的可以说是一种安全协议,如仲裁数字签名、盲签名、代理签名、多重签名、不可否认签名、公平盲签名、门限签名、具有消息恢复功能的签名等,它们与具体应用环境密切相关。

3. 仲裁数字签名

仲裁签名中除了通信双方外,还有一个仲裁方。发送方 A 发送给 B 的每条签名的消息都先发送给仲裁者 T,T 对消息及其签名进行检查以验证消息源及其内容,检查无误后给消息加上日期再发送给 B,同时指明该消息已通过仲裁者的检验。因此,仲裁数字签名实际上涉及多余一步的处理,仲裁者的加入使得对于消息的验证具有实时性。

下面是使用对称密码的仲裁签名的例子。

(1) A→T:$M \| E_{K_{AT}}[ID_A \| H(M)]$。

(2) T→B：$E_{K_{TB}}[ID_A \| M \| E_{K_{AT}}[ID_A \| H(M)] \| T]$。

在这个例子中，签名采用的是对消息的加密处理，即整个密文就是消息的签名：发送方 A 和仲裁者 T 共享密钥 K_{AT}，A 和 B 共享密钥 K_{AB}。A 产生消息 M 并计算出其散列值 $H(M)$，然后将消息 M 及其签名发送给 T，其签名由 A 的标识 ID_A 和消息散列值组成，并且用 K_{AT} 加密。A 对签名解密后，通过检查散列值来验证该消息的有效性，然后 T 用 K_{TB} 对 ID_A、来自 A 的原始消息 M、来自 A 的签名和时间戳加密后传给 B。B 对 T 发来的消息解密即可恢复消息 M 和签名。B 检查时间戳以确定该消息是实时的、而不是重放的消息。B 可以存储 M 及其签名，如果和 A 发生争执，则 B 可将下列消息发给 T 以证明曾收到过来自 A 的消息：$E_{K_{TB}}[ID_A \| M \| E_{K_{AT}}[ID_A \| H(M)] \| T]$。

下面是使用公钥密码体制的签名，并且仲裁者不能阅读消息，只能仲裁发送者的行为。

(1) A→T：$ID_A \| E_{PR_A}[ID_A \| E_{PU_B}(E_{PR_A}[M])]$。

(2) T→B：$E_{PR_T}[ID_A \| E_{PU_B}[E_{PR_A}[M]] \| T]$。

发送者 A 首先使用自己的私钥对消息进行签名，然后再使用接收者 B 的公钥对消息及签名进行加密，A 再次使用自己的私钥对连同标识和密文的内容进行签名。仲裁者收到消息后，使用 A 的公钥验证外层签名的合法性，但是无法获得原始消息 M 的内容。如果签名合法，仲裁方对密文消息加上时间戳，并使用自己的私钥进行签名，并发送给 B。B 收到后可以验证仲裁方的签名以及消息的实时性，并使用自己的私钥对密文进行解密，再使用 A 的公钥验证解密后的消息中的签名，以判断消息的合法性。B 可以保存第(2)步中的消息，如果和 A 产生纠纷，则可以作为证据提供给仲裁方。

和前面一个方案相比，采用公钥密码的方案可以更有效地保护发送者和接收者的利益，因为在前面的方案中，仲裁者可以看到消息的内容，可能和接收者勾结欺骗发送者，也可能和发送者勾结欺骗接收者，而这个方案中，仲裁者看不到消息的内容，并且也无法进行联合欺骗。

4. 盲签名

盲签名是 Chaum 在 1982 年首次提出的，Chaum 利用盲签名技术提出了第一个电子现金方案。盲签名因为具有盲性这一特点，可以有效地保护所签名的消息的具体内容，所以在电子商务等领域有着广泛的应用。

盲签名允许消息发送者先将消息盲化，而后让签名者对盲化的消息进行签名，最后消息拥有者对签名除去盲因子，得到签名者关于原消息的签名。消息发送者可以使用盲签名让签名者对给定的消息进行签名，但不泄露关于消息和消息签名的任何信息。它除了满足一般的数字签名条件外，还必须拥有如下所示的两条性质。

(1) 签名者不知道其所签名的消息的具体内容。

(2) 签名消息不可追踪，即当签名消息被公布后，签名者无法知道这是他哪次签署的。

关于盲签名，我们曾经给出了一个非常直观的说明：所谓盲签名，就是先将隐蔽的文件放进信封里，而除去盲因子的过程就是打开这个信封，当文件在一个信封中时，任何人都不能阅读它。文件签名就是通过在信封里放一张复写纸，签名者在信封上签名时，他的签名便透过复写纸签到文件上。

A 期望获得对消息 m 的签名，B 对消息 m 的盲签名实现的描述如下所示。

(1) 盲化：A 对于消息进行处理，使用盲因子合成新的消息 M 并发生给 B。

(2) 签名：B对消息M签名后，将签名(M,sign(M))返回给A。

(3) 去盲：A去掉盲因子，从对M的签名中得到B对m的签名。

一般来说，一个好的盲签名应该具有以下的性质。

(1) 不可伪造性。除了签名者本人外，任何人都不能以他的名义生成有效的盲签名。

(2) 不可否认性。签名者一旦签署了某个消息，他无法否认自己对消息的签名。

(3) 盲性。签名者虽然对某个消息进行了签名，但他不可能得到消息的具体内容。

(4) 不可跟踪性。一旦消息的签名被公开后，签名者不能确定自己何时签署的这条消息。也就是说，即使签名者存储了盲消息M和相应的签名sign(M)，等到m和其签名sign(m)公布后，他也无法找出(m,sign(m))和(M,sign(M))之间的联系。

不满足上述第(4)条的盲签名方案成为弱盲签名方案，否则称为强盲签名方案。这4条性质既是我们设计盲签名所应遵循的安全标准，又是我们判断盲签名性能优劣的根据。

自Chaum提出首个基于大整数分解难题上的盲签名方案后，研究者陆续提出了基于离散对数的盲签名方案、基于二项剩余的盲签名方案、基于ElGamal且具有匿名性的盲签名方案、群盲签名方案、基于比特承诺的盲签名等。

利用盲签名技术可以保护用户的隐私权，因此，盲签名技术在诸多电子支付、电子现金方案中被广泛使用。盲数字签名技术在充分保护用户隐私的同时，也为不法分子提供了可乘之机，他们利用电子现金的完全匿名性特点进行欺骗等违法犯罪活动。1995年M. Stadler、J. M. Piveteau和J. Camenischt提出了公平盲签名的概念，可用于条件匿名支付系统，在一定程度上消除了电子现金极端匿名性带来的负面影响。

5. 代理签名

代理签名的目的是当某签名人因某种原因不能行使签名权力时，将签名权委派给其他人替自己行使签名权。由原始签名者(部分)授权代理签名者，使代理签名者产生代替原始签名的签名就是代理数字签名。这个概念是由Mambo、Usada和Okamoto于1996年首先提出的，并且给出了一个代理签名方案(下面简称为MUO方案)。

MUO代理数字签名方案描述如下所示。

系统参数：p是一个大素数，q为$p-1$的大素因子，$g\in Z_p^*$，且$g^q\equiv 1 \bmod p$。原始签名者A、代理签名者B的私钥为PR_A、$PR_B\in\{1,2,\cdots,q-1\}$；公钥分别为$PU_A=g^{PU_A} \bmod p$、$PU_A=g^{PU_B} \bmod p$。代理签名步骤如下所示。

(1) 产生代理密钥：A随机选择一个数$k\in Z_p^*$，计算$r=g^k \bmod p$，然后计算代理签名密钥$s=(PR_A+kr) \bmod q$。

(2) 代理密钥的传递：A将(s,r)以安全的方式发送给B。

(3) 代理密钥的验证：B检查等式$g^s=PU_A r^r \bmod p$是否成立，如果成立则接受，否则拒绝。

(4) 代理签名者对消息签名：对于消息m，B将s作为新的私钥(替代PR_A)使用签名算法产生对m的签名$s_p=\text{sig}(s,m)$，然后将(s_p,r)作为他代表A对于消息m的数字签名(即代理签名)。

(5) 对代理签名的验证：接收方收到消息m和代理签名(s_p,r)，验证ver(PU_A,(s_p,r),m)=true，是否成立，如果成立则认为代理签名成立，否则拒绝。

也就是说，在上述签名方案中，A 并没有泄露自己的私钥，但是通过计算为 B 生成了一对代理公私钥对，设为(PR_B，PU_B)，其中 $PR_B=s$，$PR_B=g^s \bmod q$。在 B 代理签名中，将利用这对密钥对对消息进行签名。

现在已经出现了一些基于离散对数和素因子分解问题的代理签名方案。在这些代理签名方案中，假设 A 委托 B 进行代理签名，则签名必须满足以下 3 个最基本的条件。

(1) 签名接收方能够像验证 A 的签名那样验证 B 的签名。

(2) A 的签名和 B 的签名应当完全不同，并且容易区分。

(3) A 和 B 对签名事实不可否认。

5.5 关 键 术 语

消息认证码(Message Authentication Code，MAC)

散列函数(也称杂凑函数)(Hash Function)

散列消息认证码(Hashed Message Authentication Code，HMAC)

安全散列函数(Secure Hash Function，SHF)

数字签名标准(Digital Signature Standard，DSS)

数字签名(Digital Signature)

盲签名(Blind Signature)

5.6 习　题　5

5.1　为什么需要消息认证？

5.2　SHA 中使用的基本算术和逻辑函数是什么？

5.3　一个安全的散列函数需要满足的特性有哪些？

5.4　什么是生日攻击？

5.5　散列函数和消息认证码有什么区别？各自可以提供什么功能？

5.6　数字签名和散列函数的应用有什么不同？

5.7　数字签名需要满足哪些条件？

5.8　说出几种数字签名技术，并分析其优缺点。