计算机网络安全

第一部分 密码学

第3章 公钥密码和消息认证

主要内容

- 3.1 消息认证方法
  - 3.2 安全散列函数
  - 3.3 消息认证码
- 3.4 公钥密码原理
- 3.5 公钥密码算法
- 3.6 数字签名

回顾

- 1976年W．Diffie和Hellman 在《密码学的新方向》中首次提出了非对称密码算法的思想。
- 1978年后Rivest，Shamir和Adleman提出的RSA算法体现了公钥算法的思想。
- 公钥密码体制是现代密码学的一个标志，到目前为止，是密码学史上最大也是唯一真正的革命。
- 引起密码界高度关注，并得到迅速的发展，尤其在信息安全的应用中涉及公钥密码技术。

01

Part

消息认证

目录/消息认证

- 加密认证
- 非加密消息认证
  - 单向散列函数
    - 简单散列函数
    - SHA
    - MD5
  - 消息认证码
    - 基于分组密码CMAC
    - 散列MAC（HMAC）

3.1 消息认证方法

- 被动攻击（窃听）---机密性
  - 加密
- 主动攻击（伪造数据等）---完整性
  - 消息认证

消息认证

- **完整性**
  - **消息未被篡改、来源可信**
  - **时效性（人为延迟、重放）、顺序未乱**
- **消息认证方法**
  - **常规加密的消息认证**
  - **非加密的消息认证**

3.1.1 利用常规加密的消息认证

- **对称加密适合消息认证？**
  - 假设只有发送方和接收方共享密钥，因此只有真正的发送方才能成功加密消息。
  - 若消息中带有错误检测码和序列号，则接收者能够确认消息是否被篡改过和序列号是否正常。
  - 如果消息包括时间戳，则接收方者能够确认消息未超出网络传输的正常延时。
- **问题？**
  - 分组重排

3.1.2 非加密的消息认证

- 非加密消息认证特点
  - 生成认证标签并将其附加到每个消息以进行传输
  - 消息本身未加密，可以在目的地读取，与目的地的身份认证功能无关
  - 由于消息未加密，因此不提供消息机密性
- 消息认证（无保密要求）
  - 广播
  - 解密工作量大，随机抽取消息进行认证
  - 节省技术资源

3.1.2 非加密的消息认证

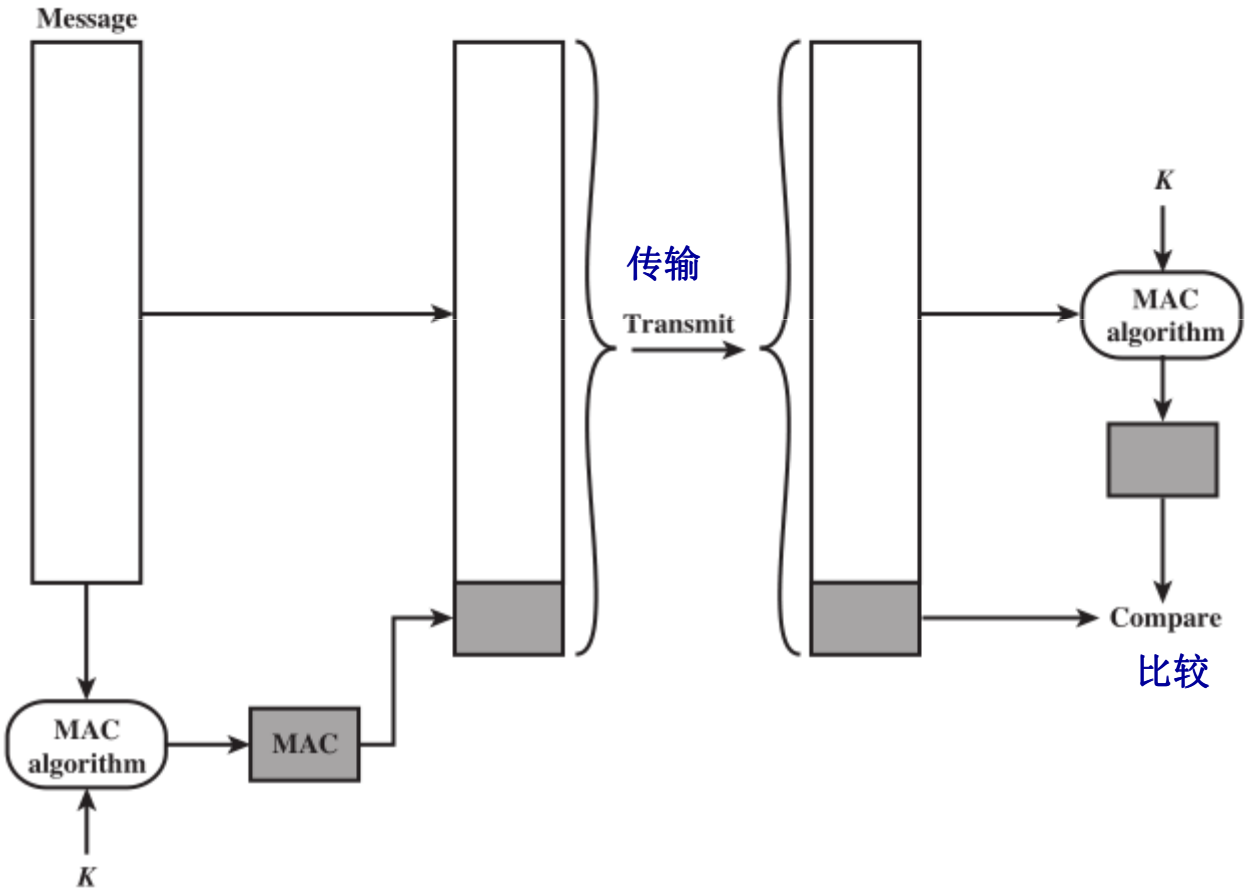
- 非加密消息认证方法
  - 消息认证码
  - 单向散列函数

1. 消息认证码（MAC）

消息认证码是一种认证技术，利用私钥产生一小块数据，称之为消息认证码MAC。将其附到消息上。

设通信实体A和B共享一个公共密钥K_{AB}。当A发消息给B时，A计算消息认证码（MAC），是消息和密钥的一个函数：$MAC_M=F(K_{AB}，M)$。消息与MAC一起传送到接收者。接收者用相同的密钥做相同运算，生成新的MAC。比较收到的MAC和计算新生成MAC。

图3.1 使用消息认证码（MAC)的消息认证



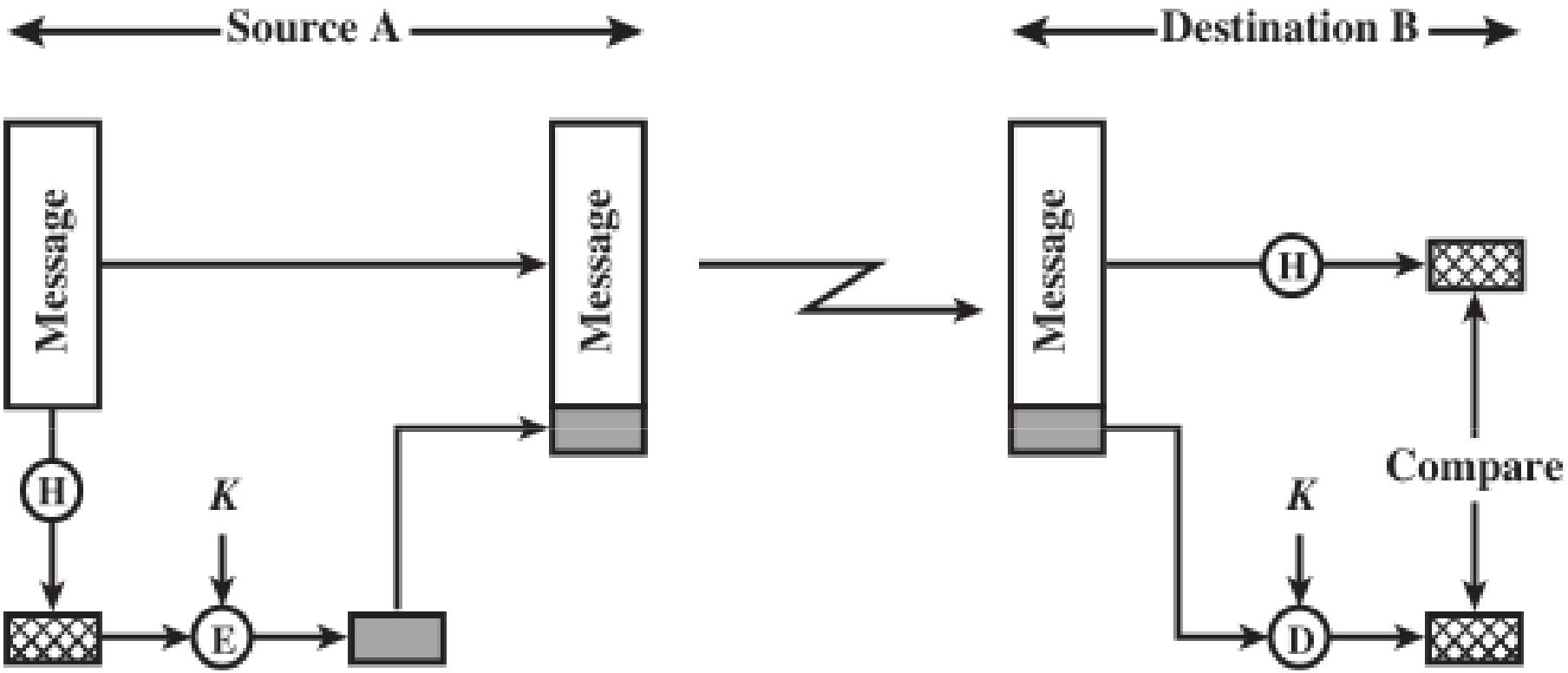
2. 单向散列函数

单向散列函数也称Hash函数或哈希函数

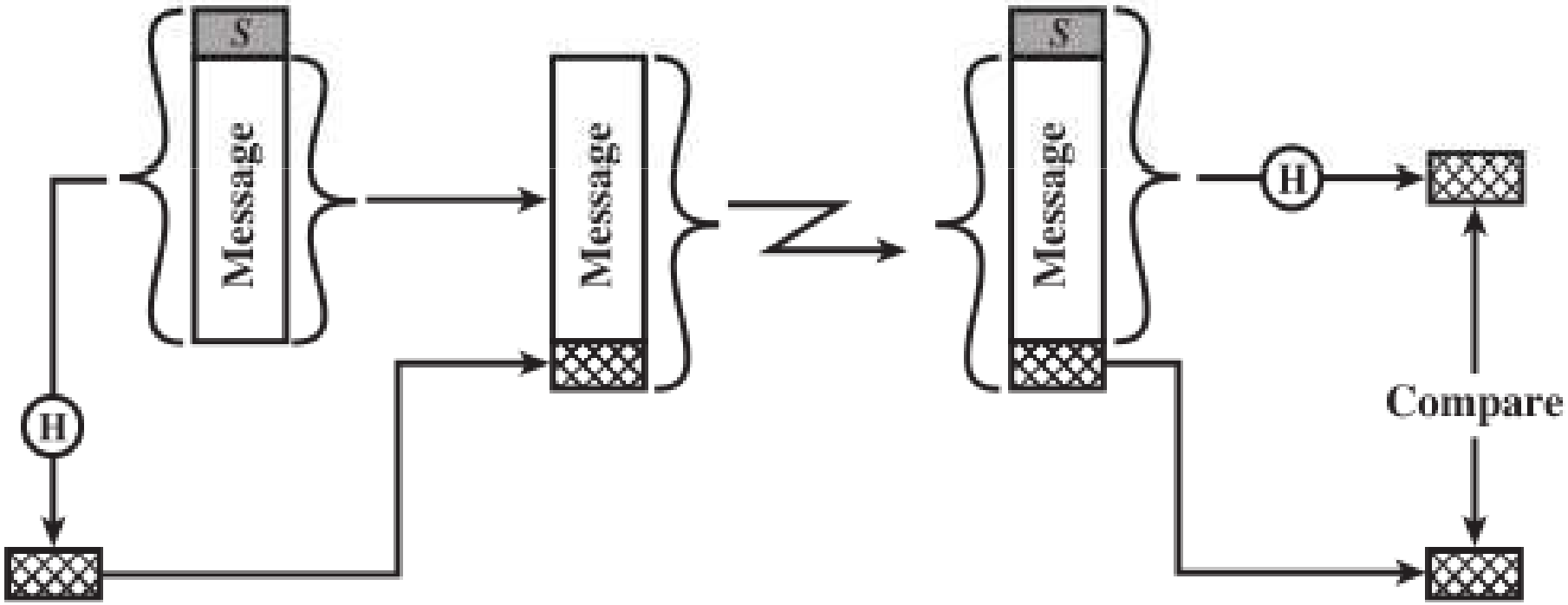
- 接受可变大小的消息M作为输入，并产生固定大小的消息摘要H（M）作为输出
- 不接受密钥作为输入
- 要对消息进行身份认证，消息摘要将与消息一起以可信的形式传送

○ **哈希函数三种用法**

- 传统加密
- 公钥加密
- 共享秘密值


Source A
Destination B
Message
Message
Message
H
K
E
H
K
D
Compare

(a) 使用传统加密



(C) 使用秘密值

图**3.2** 使用单向散列函数的消息认证

3.2 安全散列函数

3.2.1 散列函数的要求

- 不仅在消息验证中很重要，而且在数字签名中也很重要
- 目的是产生文件，消息或其他数据块的“指纹”
- 要对消息身份认证有用，哈希函数H必须具有以下属性：

1. H可以应用于任何大小的数据块。
2. H产生固定长度的输出。
3. 对于任何给定的x，H（x）相对容易计算，使得硬件和软件实现都是实用的。
4. 对于任何给定的值h，找到满足H（x）= h的x在计算上是不可行的。 满足这一特性的散列函数为具有单向性或抗原像攻击性。
5. 对于任何给定的数据块x，找到满足H（y）= H（x）的y≠ x在计算上是不可行的。 满足这一特性的散列函数被称为具有抗第二原像攻击性， 有时也被称为抗弱碰撞攻击性。防伪造
6. 找到任何一对（x，y），使得H（x）= H（y）在计算上是不可行的。
   满足这一特性的散列函数称为抗碰撞性，有时也被称为抗强碰撞性。

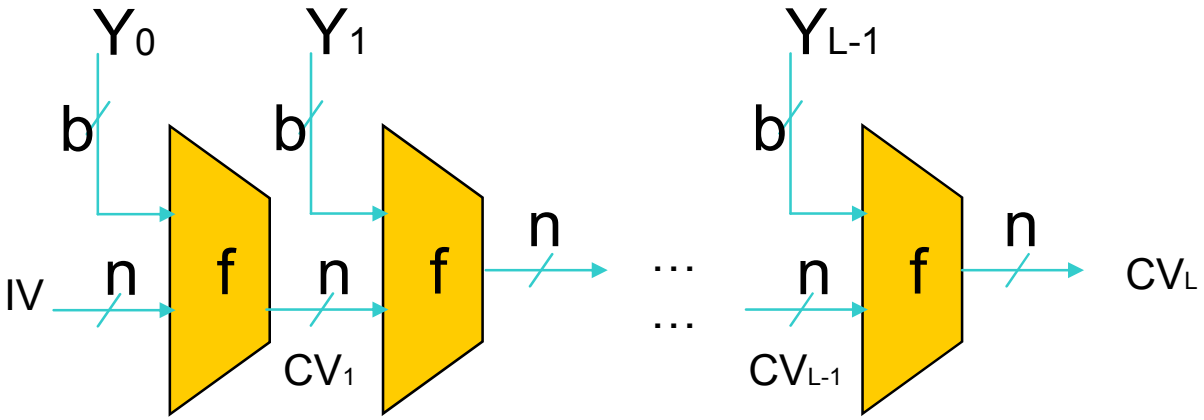
3.2.2 散列函数的安全性

- 思考：H(m)抗碰撞，H(x)=H(x’)?
- 攻击安全散列函数有两种方法
  - 密码分析

    利用了算法在逻辑上的缺陷(王小云）
  - 蛮力攻击

    针对此攻击的散列函数的强度仅取决于算法生成的散列码的长度。
- MD5和SHA-1碰撞示例
  - MD5

安全Hash函数的一般结构



IV = 初始值

CV = 链接值

Y_i = 第i 个输入数据块

f = 压缩算法

n = 散列码的长

b = 输入块的长

IV= initial n-bit value

$CV_i=f(CV_{i-1}, Y_{i-1})$ $(1 \le i \le L)$

$H(M) = CV_L$

3.2.3 简单散列函数

	bit 1	bit 2	• • •	bit n
block 1	b_{11}	b_{21}		b_{n1}
block 2	b_{12}	b_{22}		b_{n2}
	•	•	•	•
	•	•	•	•
	•	•	•	•
block m	b_{1m}	b_{2m}		b_{nm}
hash code	C_1	C_2		C_n

图**3.3** 使用按比特异或的简单散列函数

3.2.4 SHA安全散列函数

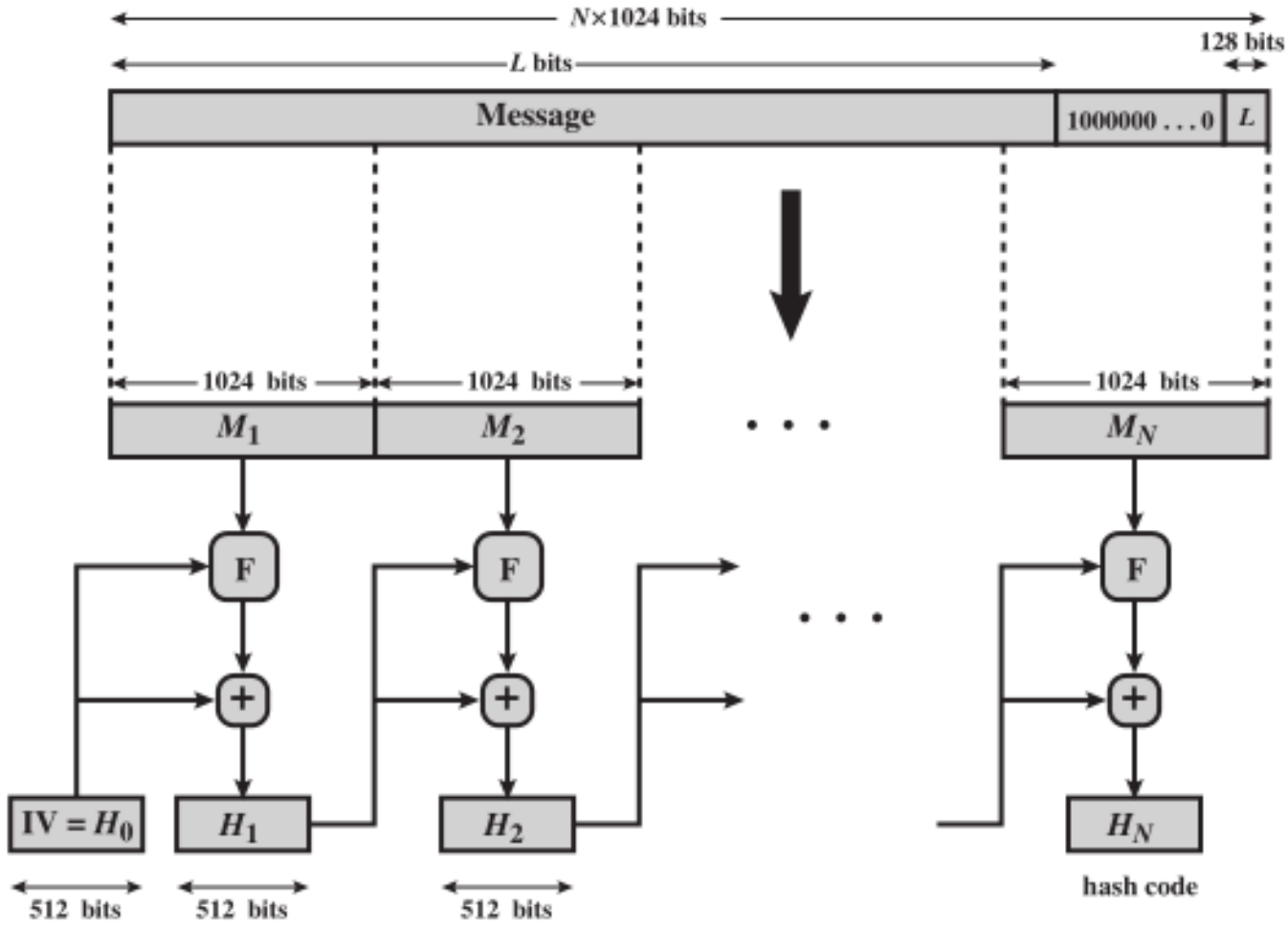
- SHA由NIST开发，并于1993年公布成为联邦信息处理标准（FIPS 180）。
- 于1995年修订为SHA-1并作为FIPS 180-1发布。实际标准文件名为“安全散列标准”。
- SHA是基于散列函数MD4，并且其架构跟MD4高度相仿。
- SHA-1生成160比特的散列值。
- 2005年，NIST宣布有意逐步取消SHA-1的批准，并在2010年之前转向依赖SHA-2。

表3.1 SHA参数的比较

	SHA-1	SHA-224	SHA-256	SHA-384	SHA-512
Message Digest Size	160	224	256	384	512
Message Size	$< 2^{64}$	$< 2^{64}$	$< 2^{64}$	$< 2^{128}$	$< 2^{128}$
Block Size	512	512	512	1024	1024
Word Size	32	32	32	64	64
Number of Steps	80	64	64	80	80

注：所有大小以比特为单位。

图3.4 用SHA-512生成消息摘要

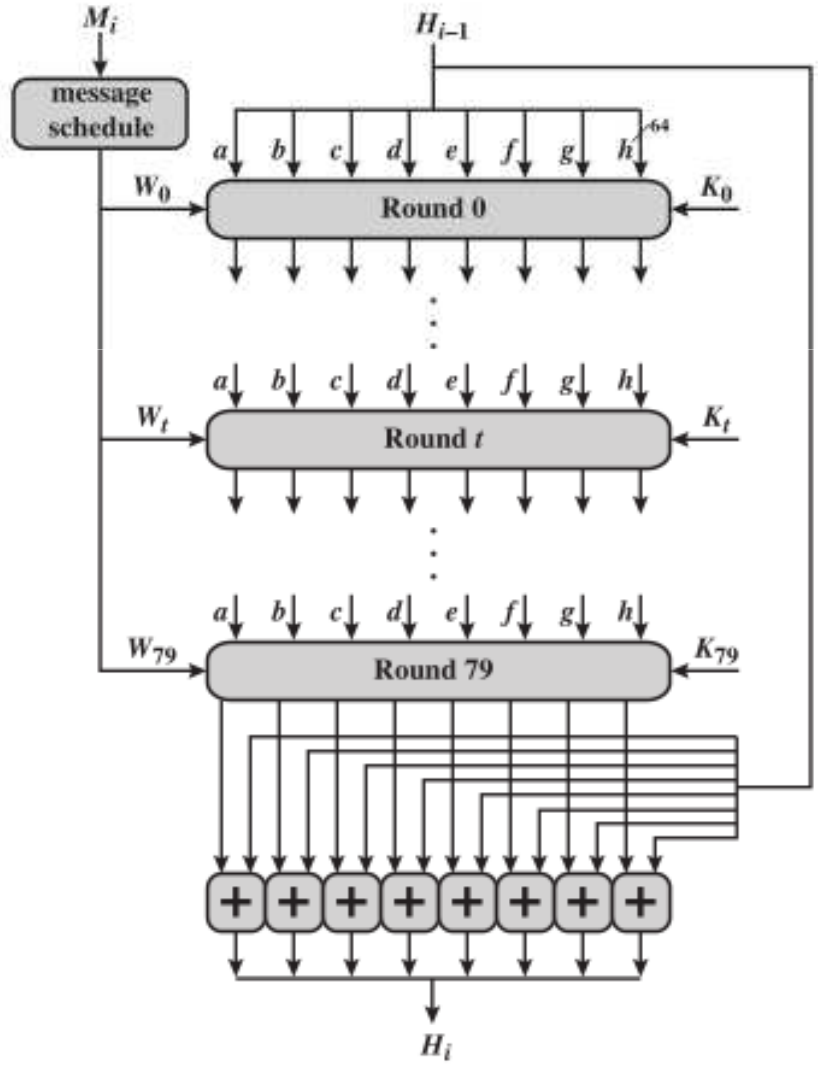


+=模2^{64}逐字相加

SHA-512步骤

- **Step 1：追加填充比特（mod1024=896)**
- **Step 2：追加长度（128比特）**
- **Step 3：初始化缓冲区**
  - **8个64比特（a-g寄存器）**
  - **取前8个质数平方根小数位前64位**
- **Step 4：处理1024比特数据（F函数）**

图3.5 SHA-512处理单个1024比特数据块



SHA-512步骤

- **Step 4：处理1024比特数据（F函数）**
  - **80轮运算 Round($H_{i-1}W_j M_i$,a-g)**
  - **H_i=F(H_{i-1}, M_i) XOR H_{i-1}**
  - **循环处理下一1024比特数据**
- **Step 5：输出**

SHA使用方法

- **Python3**
  - **import hashlib**
  - **hashlib.sha512('This is a sha512 test!'.encode('utf-8')).hexdigest()**
- **HashCalc**

哈希函数练习

小明入侵网站后获得了管理员的密文，由于太高兴了手一抖把密文删除了一部分，只剩下前10位a74be8e20b，小明根据社工知道管理员的密码习惯是key{4位的数字或字母}，所以管理员的密码是？（MD5）

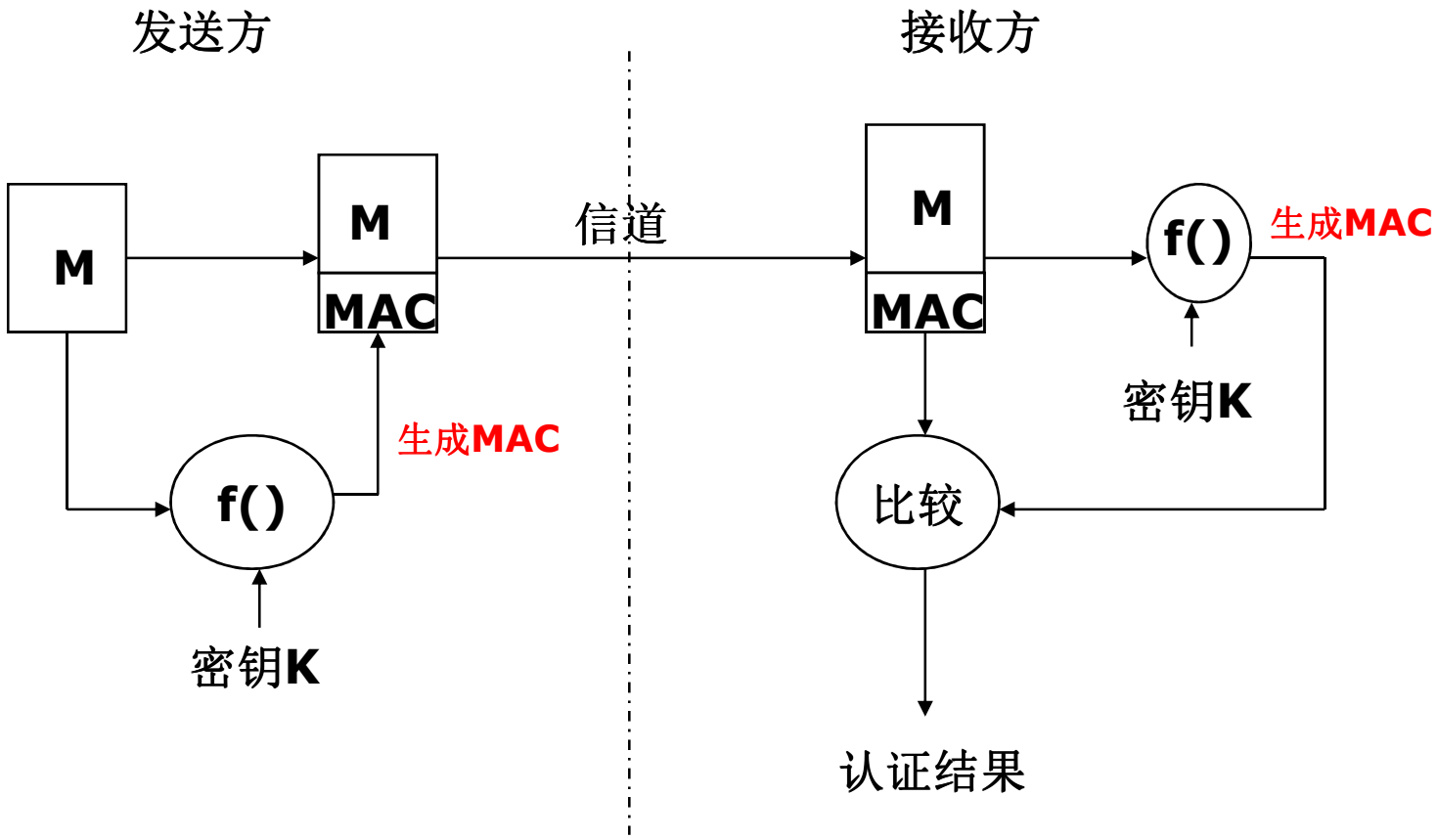
3.3 消息认证码（MAC）

- 回顾：认证
  - 1. 验证消息来源是否可信（确定消息发送者）
  - 2. 验证消息是否被篡改
  - 3. 验证消息时效性（消息有没有被延迟或重放）
- 认证方法
  - 加密
  - Hash
  - MAC

3.3 消息认证码（MAC）

- **消息认证码（Message Authentication Code, MAC）**
  - 发送方采用一种类似于加密的算法和一个密钥，根据消息内容计算生成一个**固定大小的小数据块**，并加入到消息中，称为MAC。
    - $MAC = f_k(M)$
    - 需要**密钥**
  - **核心**是类似于加密算法的**认证码生成算法**。
  - MAC被附加在消息中，用于**消息的合法性认证**。
- **功能**
  - 接收者可以确信消息M未被改变
  - 接收者可以确信消息来自所声称的发送者

1. MAC的基本原理

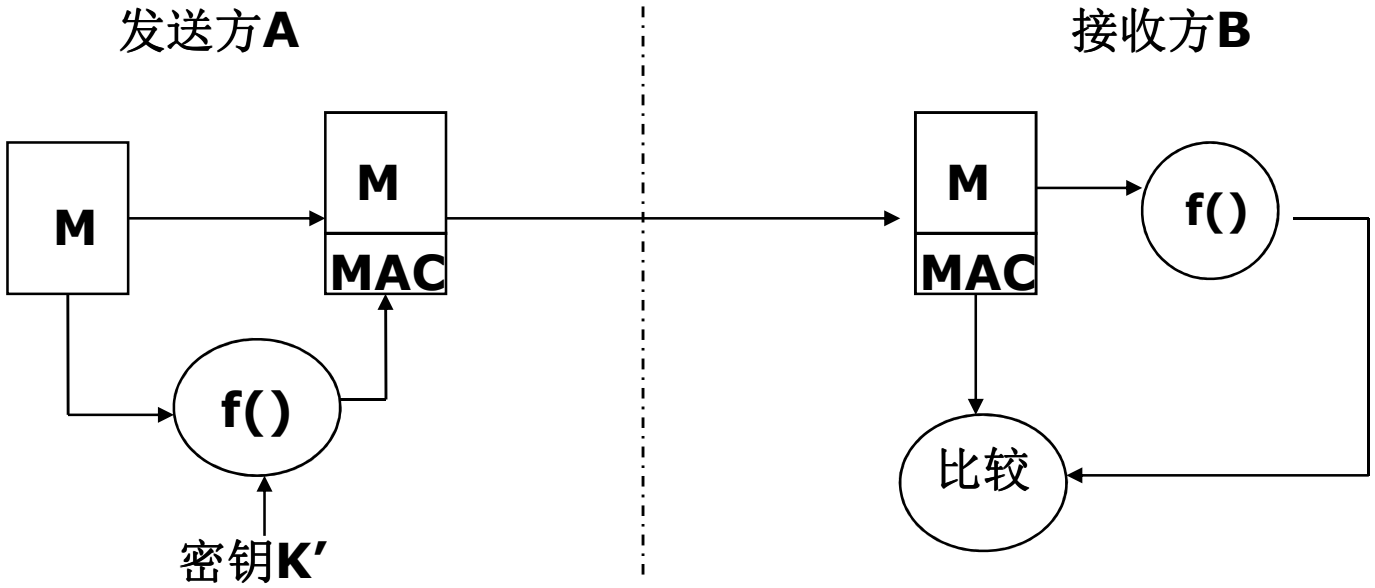


*双方共享密钥 **K***

2. MAC函数的特点

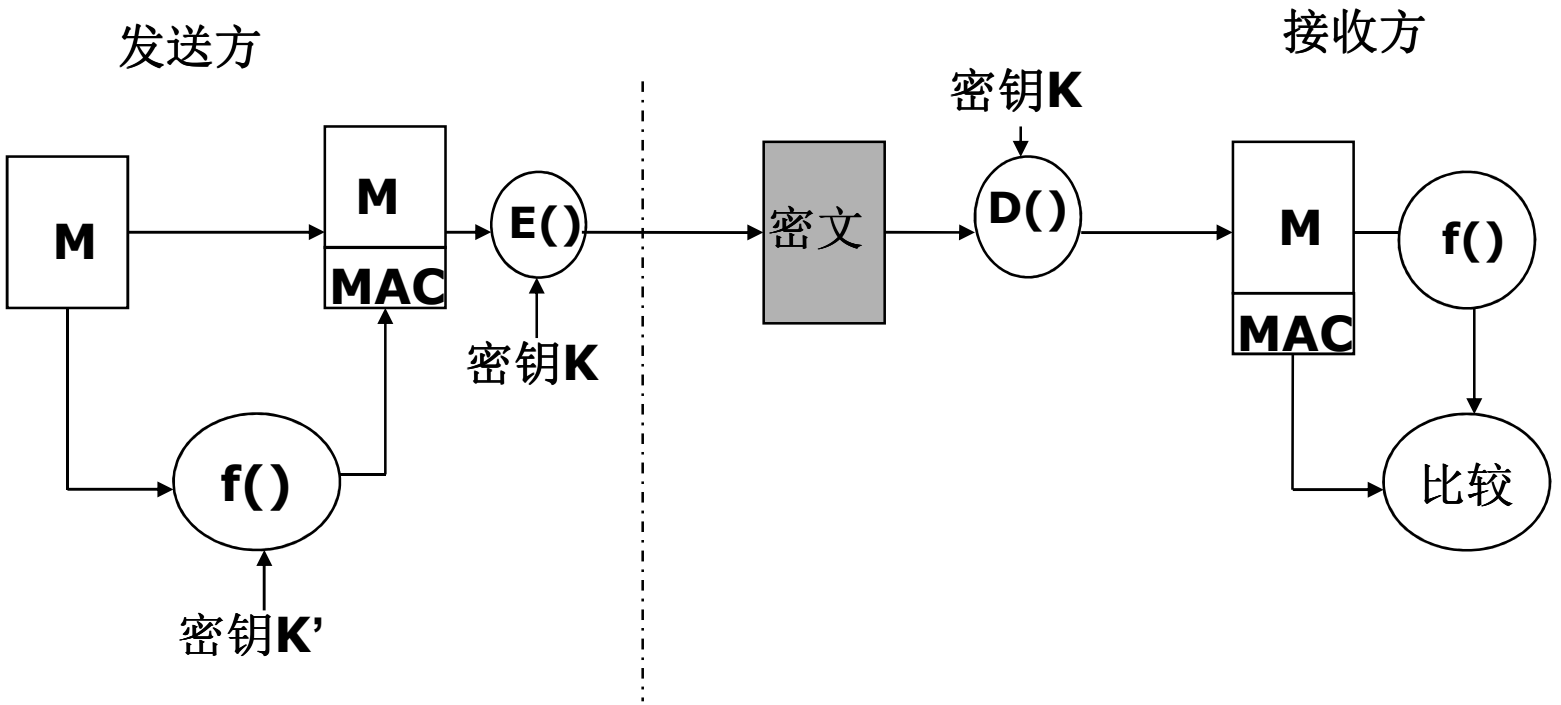
- MAC函数类似于加密函数，需要密钥
- MAC函数无需可逆，可以是单向函数
  - 使得MAC函数更不易被破解
- MAC函数不能提供对消息的保密
  - 保密性通过对消息加密而获得
    - 需要两个独立密钥
  - 两种方式：
    - 先计算MAC再加密
    - 先加密再计算MAC
- 基于MAC的认证过程独立于加、解密过程
- 可以用于不需加密保护的数据的认证
- MAC方法不能提供数字签名功能
  - 无法防止对方的抵赖

3. MAC的基本应用 1



- **提供消息认证**
  - **仅A和B共享密钥K'**
- $M||f_{k'}(M)$

4. MAC的基本应用 2



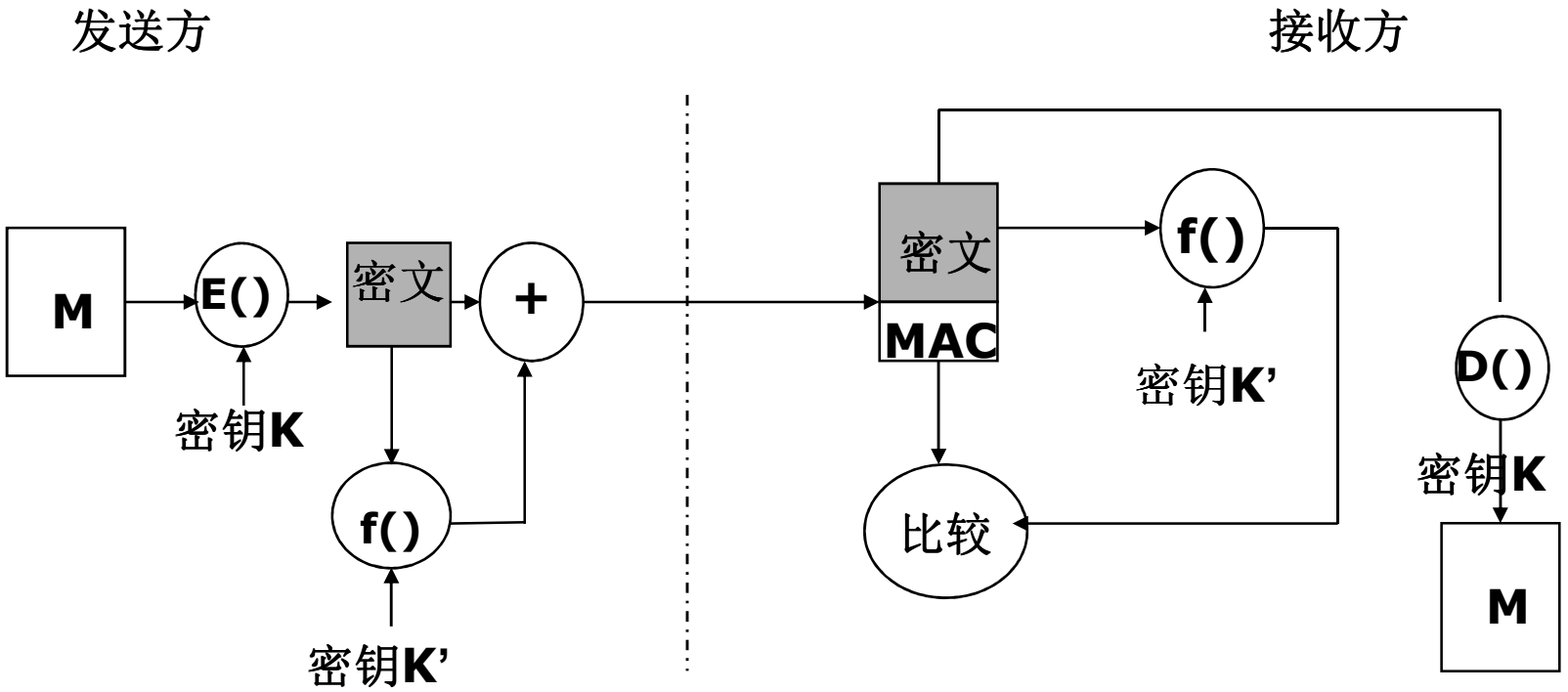
1、先计算**MAC**再加密

2、**K**是加密密钥

3、**K'**是认证密钥

- 提供保密
  - 只有**A**和**B**共享**k**
- 提供认证
  - 只有**A**和**B**共享**k'**

5. MAC的应用 3



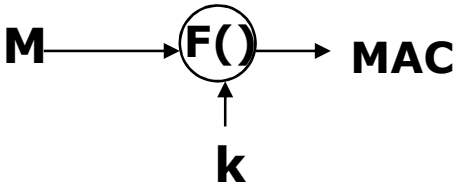
1、先加密再计算MAC

2、K是加密密钥

3、K'是认证密钥

- 提供保密
  - 只有A和B共享k
- 提供认证
  - 只有A和B共享k'

6. MAC函数的安全性分析



- **MAC函数的特点**
  - 多对一映射
    - 对同一个M，存在多个key产生相同的MAC
    - Key相同，存在多个M产生相同的MAC
  - n-bit MAC: 2^n个可能的MAC
  - K-bit密钥： 2^k个可能的密钥
  - N个可能的消息：$N >> 2^n$
  - 一般情况下，$n < k$
- **MAC函数的安全性**
  - 主要是对强行攻击的抵抗能力
  - 方式：穷举法，根据已知消息M和对应的MAC值，来推断密钥k
  - 前提：MAC算法已知

7. 对MAC函数强行攻击的方式

- 已知消息M_1和对应的MAC_1值
- 对所有可能的密钥k_i，计算消息M_1的MAC，其中至少存在一个k，使得$f_k(M_1)=MAC_1$
- 由于密钥空间为2^k，以上计算将产生2^k个MAC结果；而MAC空间为2^n，且$2^n < 2^k$，故存在多个k产生相同的MAC：一般是个2^{k-n}个key对应一个MAC
- 为了确定哪一个是正确的key，分析者需要选择另一组M_2和MAC_2，对上面的个结果进行验证，以缩小搜索范围
- 因此，可能需要多轮验证：大约需要k/n轮
- 计算量为$2^k+2^{k-n}+2^{k-2n}+\cdots\cdots$
- 可见强行攻击难度很大

8. MAC函数的安全性总结

- 需要大量的（M，MAC)对，对MAC的强行攻击通常不能离线进行
- MAC算法抵抗强行攻击的有效级为（2^k, 2^n）中的最小值
- 为了足够安全，MAC函数应该具有以下的性质：
  - 给定一个或多个(M, MAC)对而不知道密钥的情况下，对于一个新的消息，要计算出对应的MAC在计算上不可行
  - 攻击者得到一个消息M及对应的MAC，则构造消息M’使得MAC’ = MAC在计算上是不可行的

3.3.1 基于散列函数的认证

- 回顾--散列函数（哈希函数）
  - 一种单向函数
  - 输入：任意长度的消息M
  - 输出：固定长度的消息摘要
    - 是一个固定长度的哈希值H(M)
  - 哈希值是消息中所有比特的函数值
    - 消息中任意内容的变化将导致哈希值的变化
  - 具有完整性检测功能
  - 可用于数字签名

HMAC

- 人们越来越关注从密码散列码（如SHA-1）中开发MAC
  - 密码散列函数通常在软件中比传统加密算法（如DES）执行得更快。
  - 有许多共享的密码学散列函数代码库。
  - 诸如SHA-1之类的散列函数并不是专为MAC设计的，因为散列函数不依赖于密钥。所以它不能直接用于此目的。
- 已经有许多关于将秘密密钥结合到现有散列算法中的提议
  - 获得最多支持的方法是HMAC

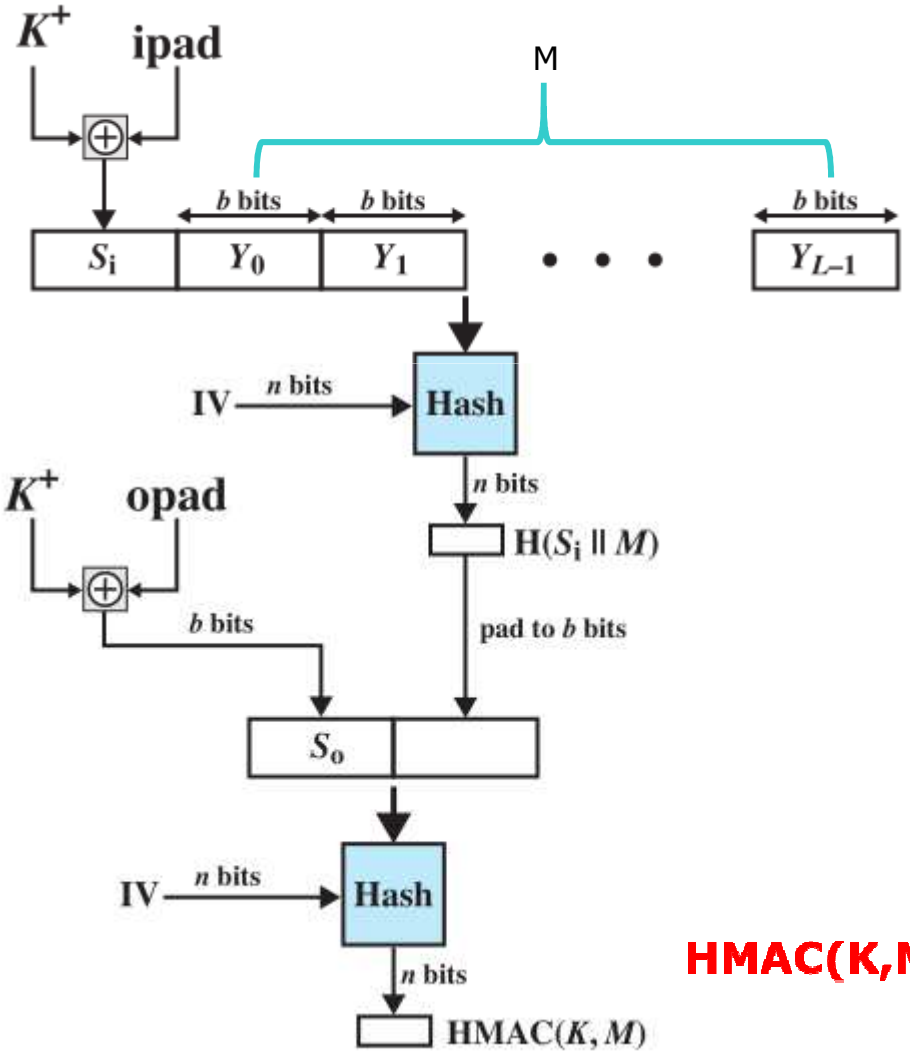
HMAC（续）

- 已发布为RFC 2104
- 已被选为IP安全的强制实施MAC
- 用于其他Internet协议，例如传输层安全性（TLS）和安全电子事务（SET）

HMAC设计目标

- 不必修改而直接使用现有的散列函数。 特别是很容易免费得到软件上执行速度较快表现良好的散列函数及代码。
- 嵌入式散列函数要有很好的可移植性，以便开发更快或更安全的散列函数。
- 保持散列函数的原始性能，不会导致显着退化。
- 以简单的方式使用和处理密钥。
- 如果已知嵌入式散列函数的强度，则完全可以知道认证机制抗密码分析的强度。

图3.6 HMAC结构



M=输入**HMAC**的消息

L=**M**中的分组数

b=分组中的比特数

n=嵌入式散列函数产生的散列码长度。

K=密钥；如果密钥长度大于**b**，则将其输入给散列函数生成**n**比特的密钥； 建议长度≥**n**

K+=为使**K**为**b**位长而在**K**左边填充**0**后所得的结果。

ipad=00110110

(十六进制数为**36**）重复**b/8**次

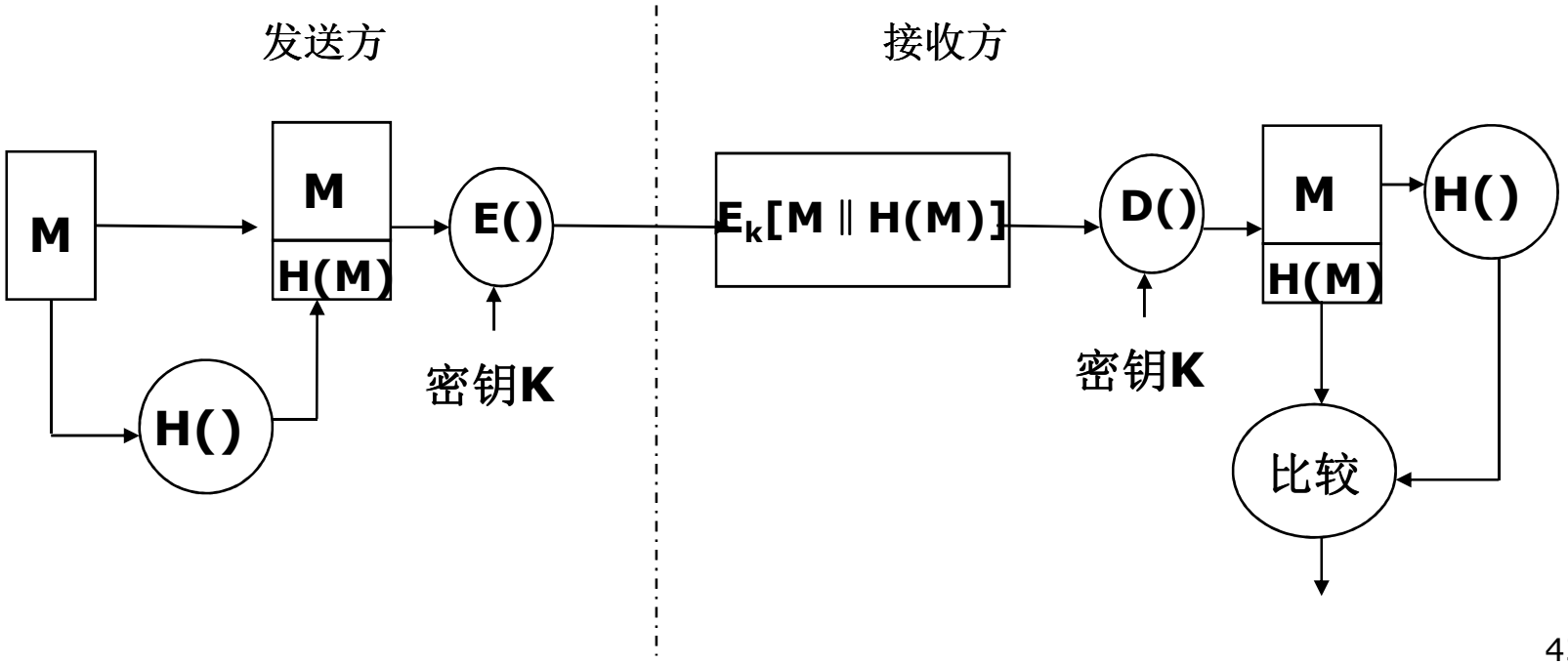
opad=01011100

(十六进制数为**5C**）重复**b/8**次

$$HMAC(K,M)=H[(K^+ \oplus opad)||H[(K^+ \oplus ipad)||M]]$$

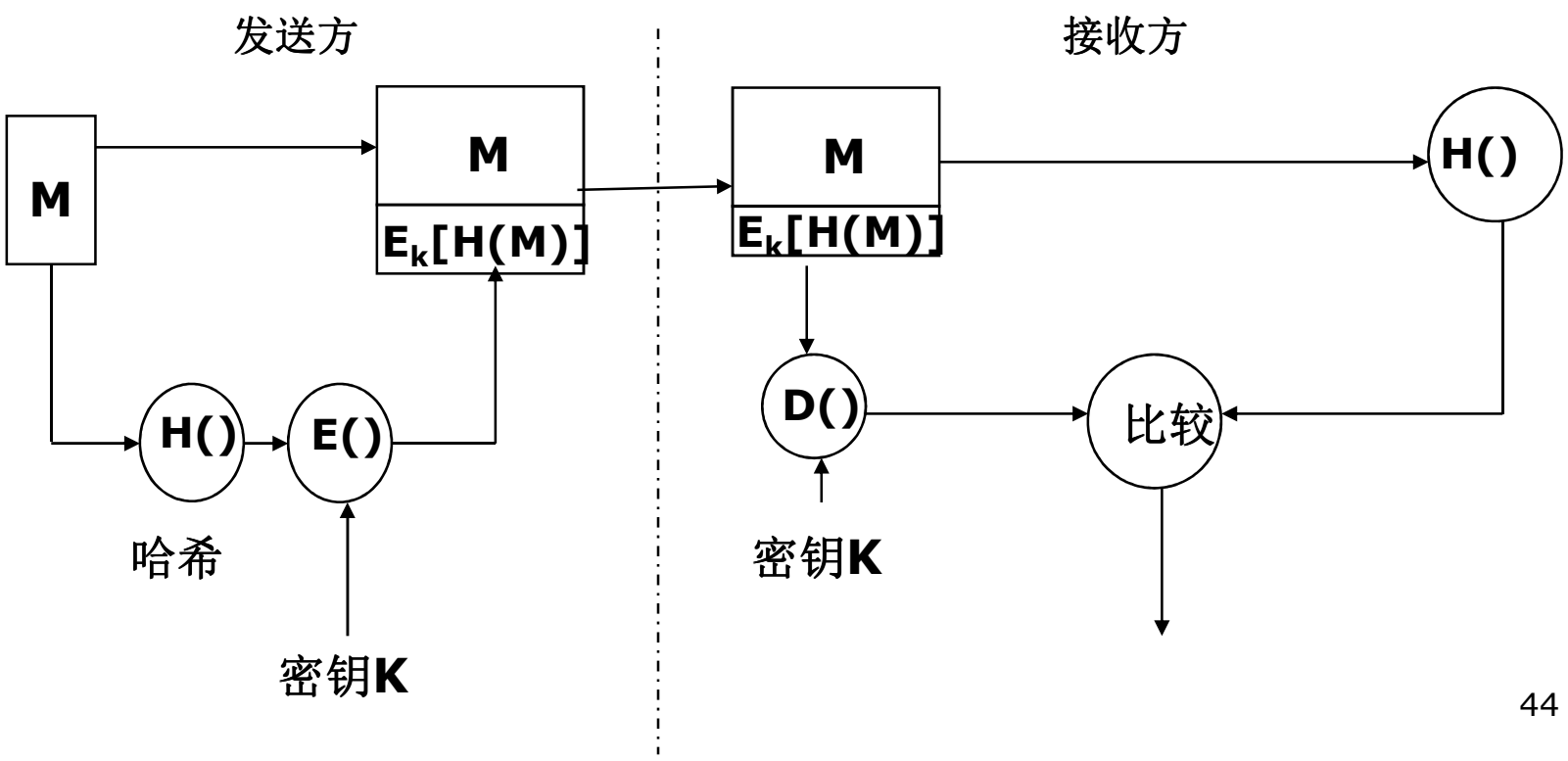
散列（哈希）函数消息认证(1)

- 基本的哈希函数消息认证
  - 对称加密：发端和收端共享加密密钥k
  - 哈希值提供了消息认证需要的结构和冗余
  - 提供保密和认证双重功能：消息和H(M)被加密保护



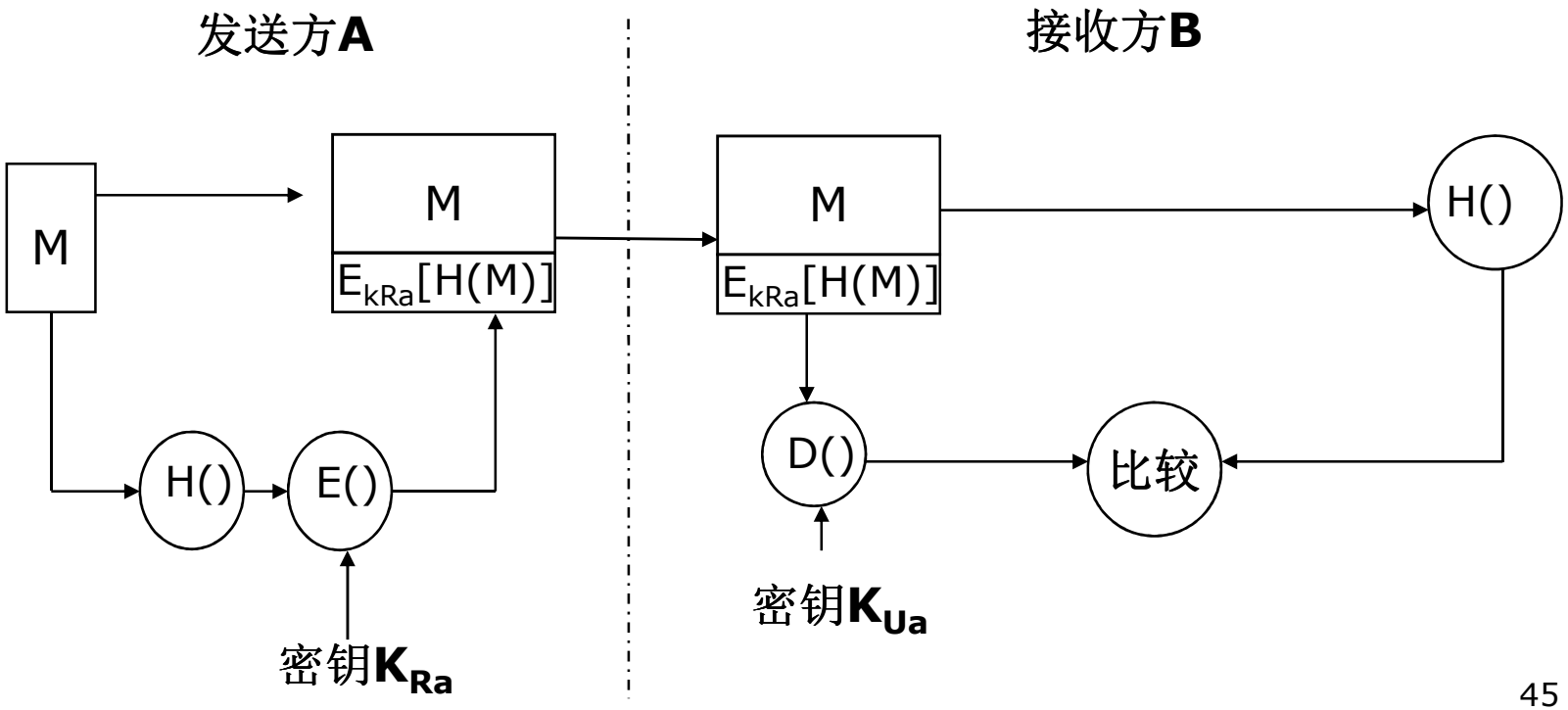
散列（哈希）函数消息认证(2)

- 仅对哈希值进行加密的认证方案
  - 用于不需对消息加密的场合
  - 基于对称密钥机制
  - 提供认证：H(M)被加密保护

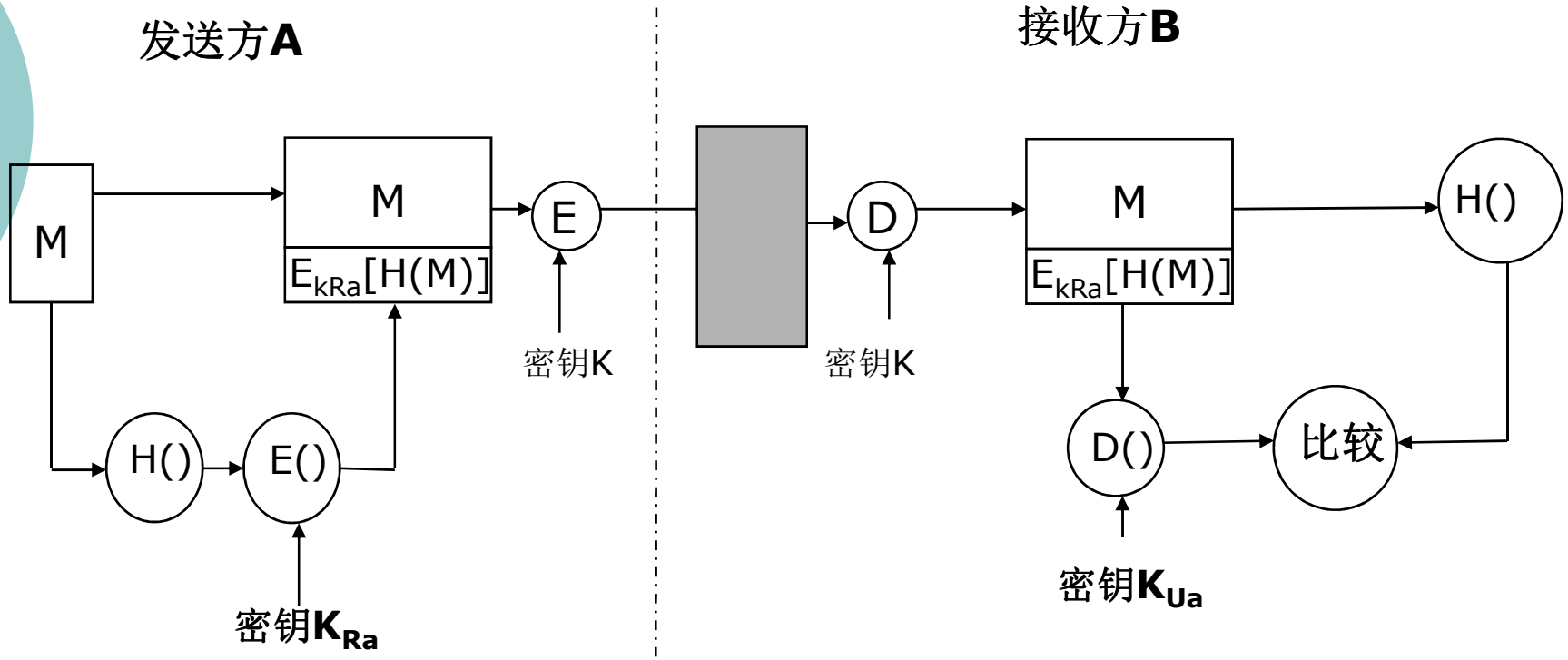


散列（哈希）函数消息认证(3)

- 仅对哈希值进行加密的认证方案
  - 用于不需对消息加密的场合
  - 基于公开密钥机制
  - 提供认证和数字签名：发送方用自己的私钥加密H(M)



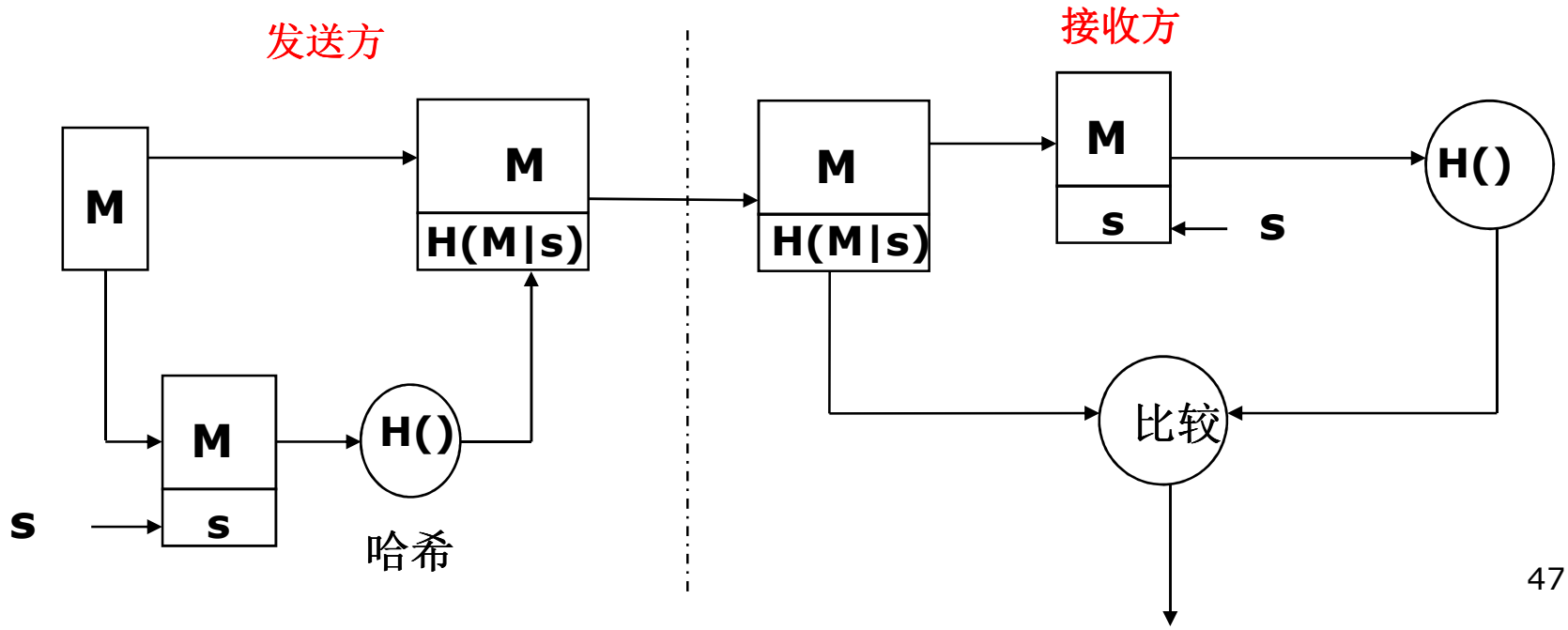
散列（哈希）函数消息认证(4)



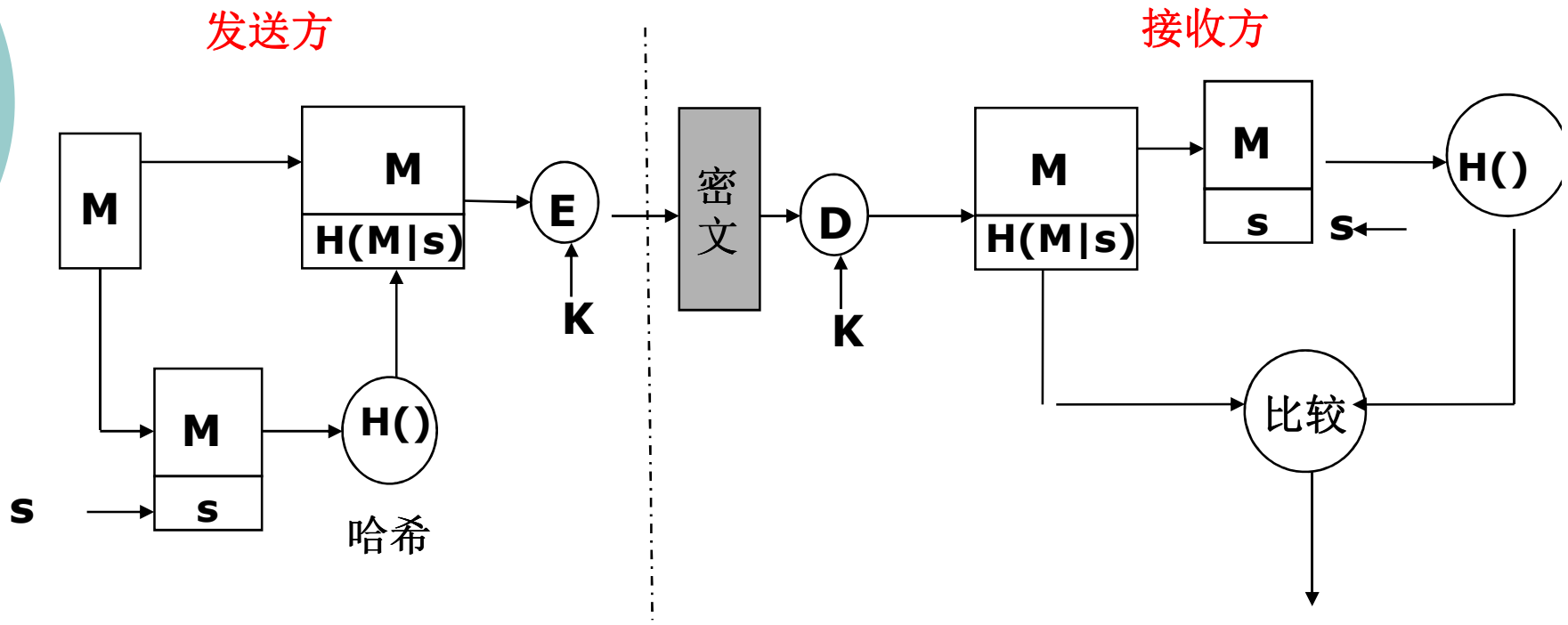
- 提供**保密**：对称密钥算法、密钥k进行**外层加密**
- 提供**认证和数字签名**：A用私钥对消息明文的哈希值进行加密

散列（哈希）函数消息认证(5)

- 采用秘密数值的哈希认证
  - 通信双方共享一个秘密数值s
  - 计算哈希值时，s附加到消息M上一起计算得到H(M|s)
  - 传输过程中，数据不加密，但不传送s
  - 提供认证：只有发送方和接收方共享秘密S
  - 适用于加密算法不可用的场合



散列（哈希）函数消息认证(6)



- 提供**认证**：只有发送方和接收方才共享S
- 提供**保密**：只有发送方和接收方才共享k

HMAC实例

- 简单HMAC
  - import hmac
  - h=hmac.new(b'jxust') #key
  - h.update(b'hash is interesting')
  - h_str=h.hexdigest()
  - print(h_str)
  - print(h.name)

3.3.2 基于密文的消息认证码（CMAC）

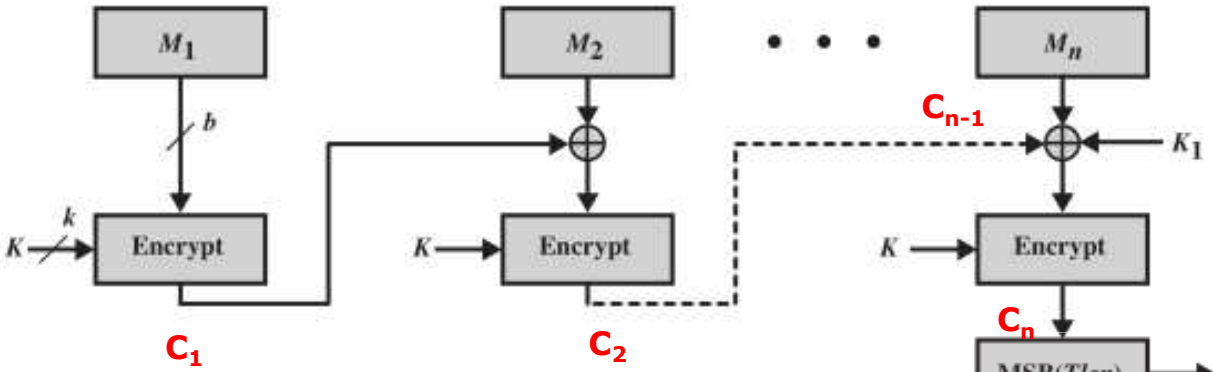
- CMAC的操作模式适用于AES和3DES。
- 在AES中，b=128，在3DES， b=64，并且这个信息被分成为n块（M_1,M_2,…, M_n）。
- 基于密文的消息认证码CMAC的计算如图3.7所示。

  T是消息认证码（即MAC），或者称为标签。

  Tlen=T的位长度

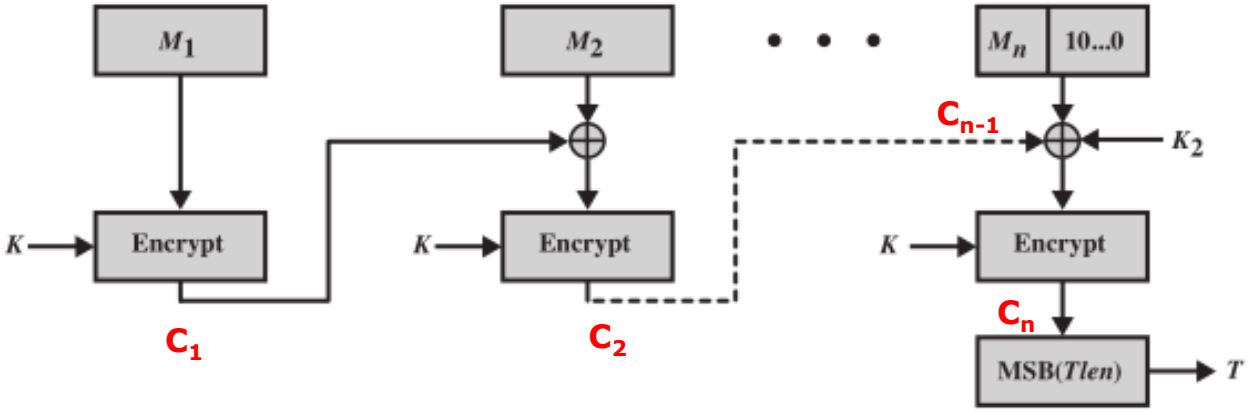
  MSBs(X)=位串X的最左边的s比特二进制数

图3.7 基于密文的消息认证码（CMAC）



(a) 信息长度是块大小的整数倍

信息认证码MAC



(b) 信息长度不是块大小的整数倍

CMAC函数实例

- Python
  - from Crypto.Hash import CMAC
  - from Crypto.Cipher import AES
  - secret = b'Sixteen byte key'
  - cobj = CMAC.new(secret, ciphermod=AES)
  - cobj.update(b'jxust')
  - print (cobj.hexdigest())

小结/消息认证

- 加密认证
- 非加密消息认证
  - 单向散列函数
    - 简单散列函数
    - SHA
    - MD5
  - 消息认证码
    - 散列MAC（HMAC）
    - 基于分组密码CMAC

02

Part

公钥加密

3.4 公钥密码原理

- 3.4.1 公钥密码思想
- 3.4.2 公钥密码系统的应用
- 3.4.3 公钥密码的要求
- 3.4.4 对称密码和公钥密码对比

3.4 公钥密码原理

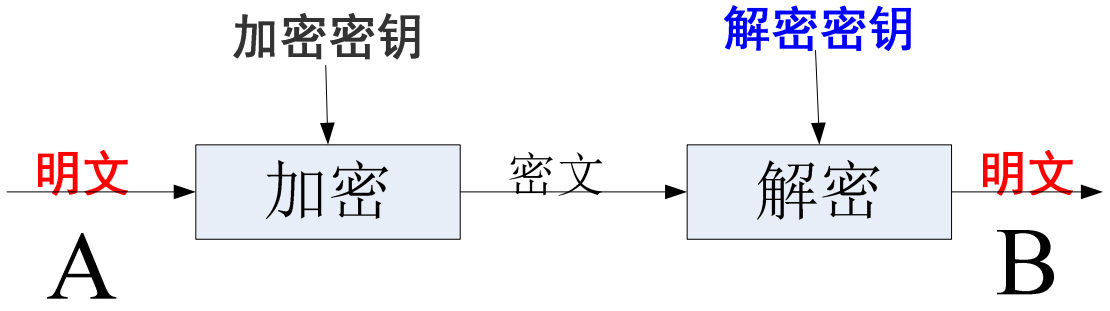
- 不同于以往的加密技术，公钥密码体制是建立在**数学函数**基础上的，而不是建立替换和置换这些初等方法的
- **加/解密时**，分别使用了两个不同的密钥：一个可对外界公开，称为“**公钥**”；一个只有所有者知道，称为“**私钥**”
- 公钥和私钥之间具有紧密联系，用公钥加密的信息只能用相应的私钥解密，反之亦然。**同时，要想由一个密钥推知另一个密钥，在计算上是不可能的**

3.4.1 公钥密码思想

- 公钥密码与传统密码同等重要，它还可以进行消息认证和密钥分发
- 由Diffie和Hellman于1976年首次公开提出
- 基于数学函数而不是基于位模式的简单操。
- 是不对称的，涉及使用两个单独的密钥。传统密码仅用一个密钥。
- 公钥密码方案由6个部分组成：

明文、加密算法、公钥和私钥（这对密钥如果一个密钥用于加密，则另一个密钥就用于解密）、密文、解密算法。

公开密钥算法加/解密模型

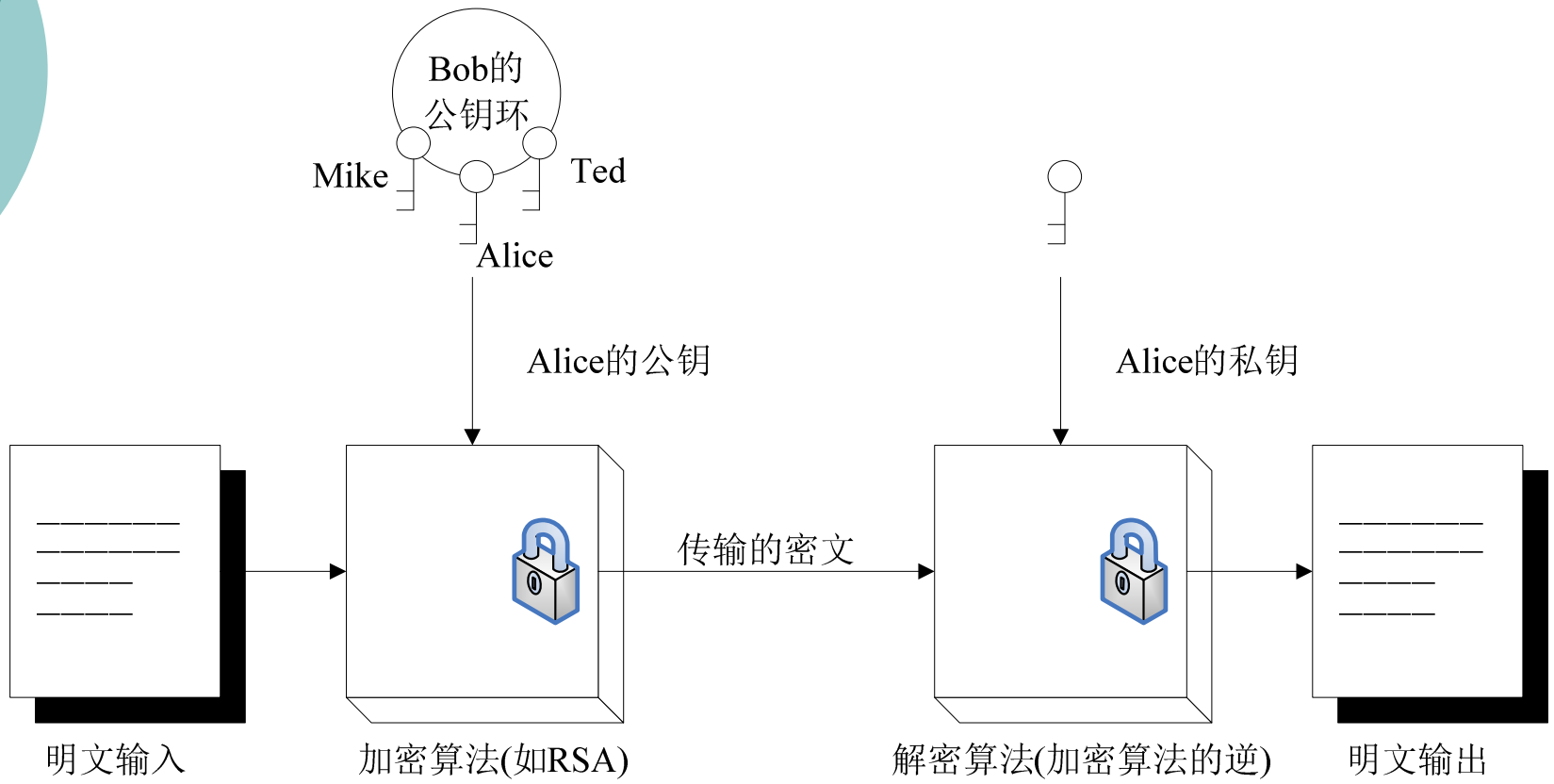


- **A**查找**B**的公钥。
- **A**采用公钥加密算法以**B**的公钥作为加密密钥对明文加密。
- **A**通过非安全信道将密文发送给**B**。
- **B**收到密文后使用自己的私钥对密文解密还原出明文。

公开密钥算法的基本工作过程

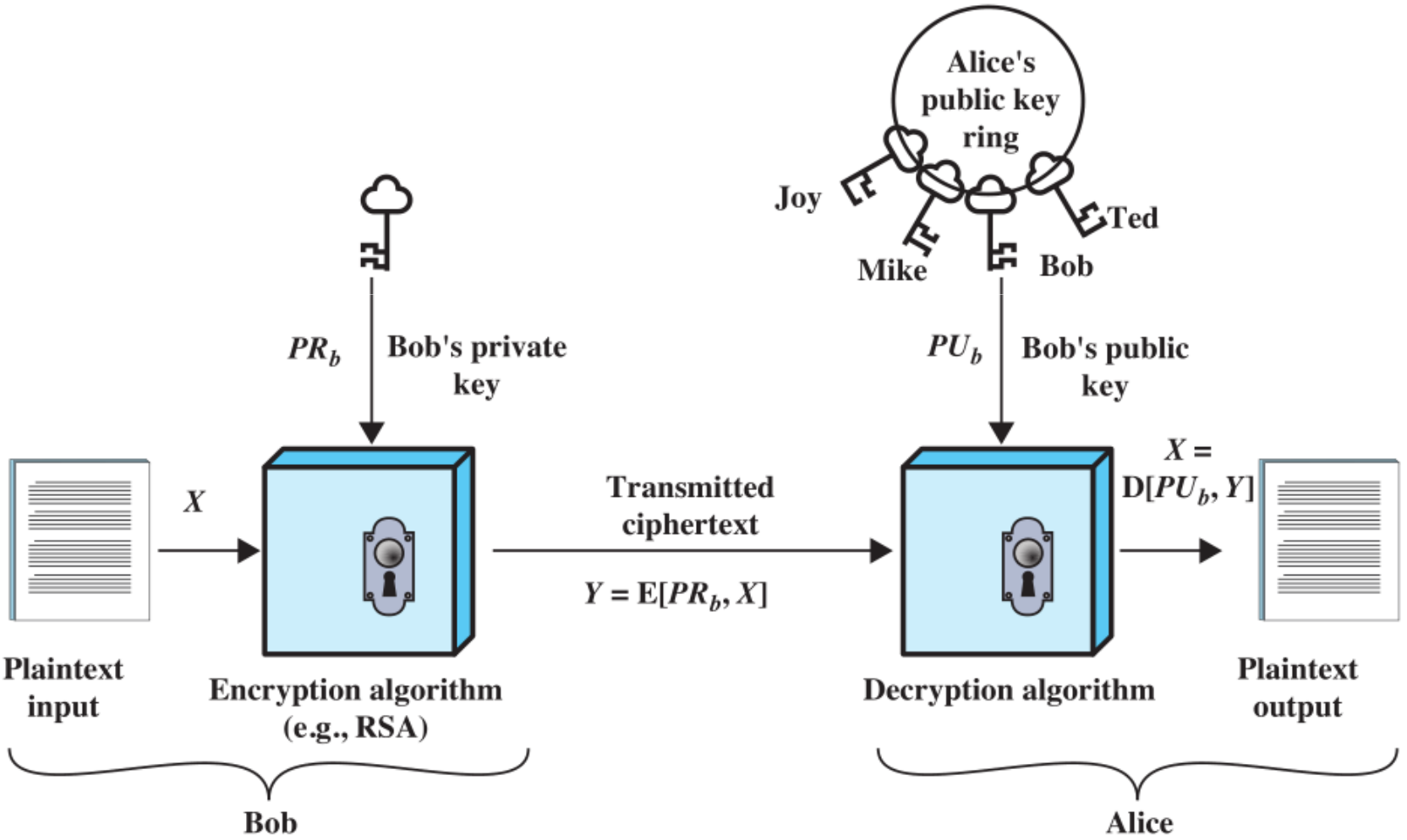
- 每一个终端产生其自身用于通信的加密和解密密钥
- 各个终端将其自身的加密密钥通过某种方式向外界公布
  - 目录服务、公共服务器、公共文件等
  - 被公布的加密密钥称为公开密钥
  - 另一个被终端自己保存的解密密钥称为私有密钥
- 通常，公开密钥和私有密钥应可以互为加密/解密密钥
- 若A 需向B 发送消息，可用B 的公开密钥对消息进行加密
  - B 收到消息后，可用其私有密钥对消息进行解密
  - 由于公钥密码算法的特性，可以保证只有B 才能正确地解读消息
- 任何人都可以使用B 的公开密钥来向B 发送加密消息，但只有B 能够解密

图3.9 公钥密码



（a）用公钥加密

图3.9 公钥密码



（b）用私钥加密（认证）

3.4.2 公钥密码系统的应用

- **公钥系统的特征**在于使用具有两个密钥的加密类型的算法，一个是私有的，一个是公开的。
- 根据应用，发送者要么使用发送者的私钥，要么使用接收者的公钥，或者两个都使用。从而实现某种类型的加密功能。

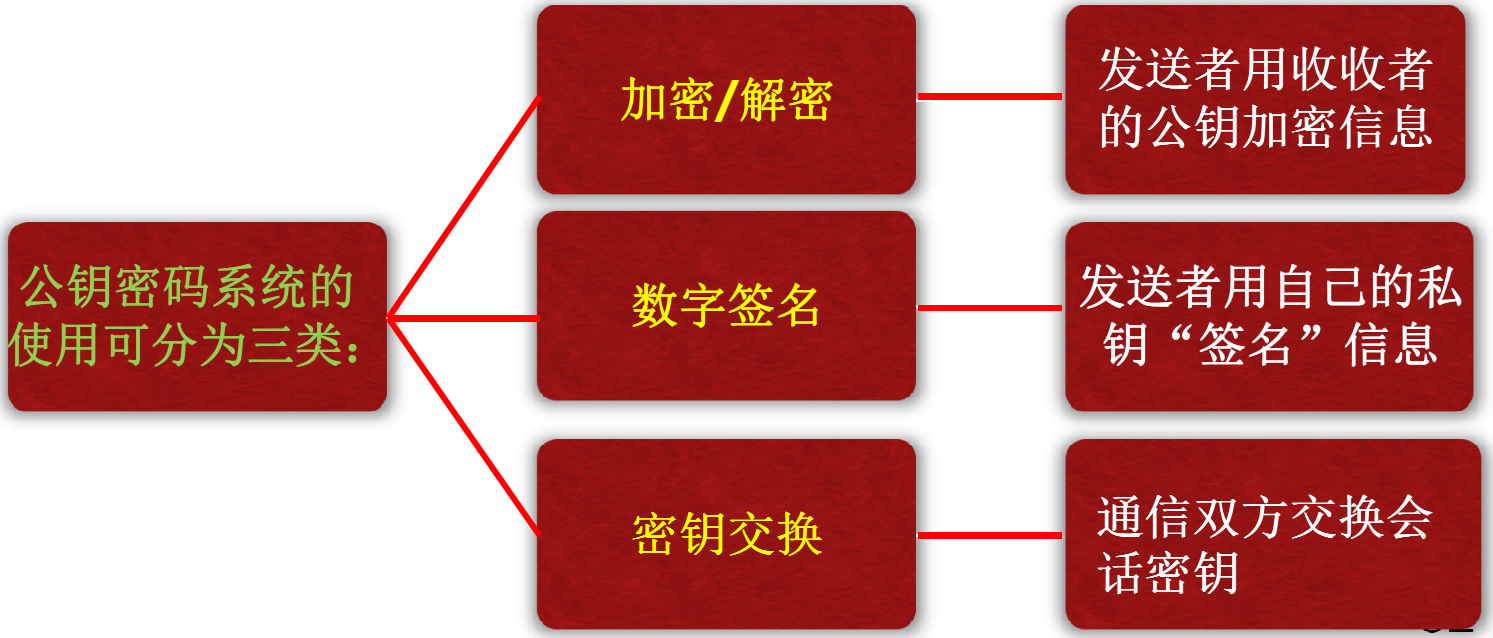


表3.2 公钥密码系统的应用

Algorithm	Encryption/Decryption	Digital Signature	Key Exchange
RSA	Yes	Yes	Yes
Diffie-Hellman	No	No	Yes
DSS	No	Yes	No
Elliptic Curve	Yes	Yes	Yes

算法

加密/解密

数字签名

密钥交换

椭圆曲线

3.4.3 公钥密码的要求

- 产生一对密钥(公钥*PU*，私钥*PR*)在计算上是容易的。
- 已知接收方*B*的公钥PU_b和要加密的消息*M*，消息发送方*A*产生相应的密文在计算上是容易的。

$$C=E(PU_b,M)$$

- 消息接收方*B*使用其私钥对接收的密文解密以恢复明文在计算上是容易的.

$$M=D(PR_b,C) = D[PR_b , E(PU_b,M)]$$

- 已知公钥PU_b时，攻击者要确定对应的私钥PR_b在计算上是不可行的。

3.4.3 公钥密码的要求（续）

- 已知公钥PU_b和密文C，攻击者要恢复明文M在计算上是不可行的。
- 两个相关的密钥中的任何一个都可以用于加密，另一个用于解密：

$$M = D[PU_b, E(PR_b, M)] = D[PR_b, E(PU_b, M)]$$

3.4.4 对称密码和公钥密码对比

对称密码	公钥密码
一般要求	一般要求
1．加密和解密使用相同的密钥	1．同一算法用于加密和解密，但加密和解密使用不同密钥
2．收发双方必须共享密钥	2．发送方拥有加密或解密密钥，而接收方拥有另一密钥
安全性要求	安全性要求
1．密钥必须是保密的	1．两个密钥之一必须是保密的
2．若没有其他信息，则解密消息是不可能或至少是不可行的	2．若没有其他信息，则解密消息是不可能或至少是不可行的
3．知道算法和若干密文不足以确定密钥	3．知道算法和其中一个密钥以及若干密文不足以确定另一密钥

3.5 公钥密码算法

- 3.5.1 RSA公钥密码算法
- 3.5.2 Diffie-Hellman密钥交换
- 3.3.3 其它公钥密码算法

3.5.1 RSA公钥密码算法

- Rivest、Shamir、Adleman联合提出的简称为RSA公钥密码系统。
- 理论基础是数论中“大整数的素因子分解是困难问题”的结论
  - 即求两个大素数的乘积在计算机上时容易实现的，但要将一个大整数分解成两个大素数之积则是困难的。

1. 欧拉函数

- **给定一个正整数** n **，用** $\phi(n)$ **表示比** n **小且与** n **互为素数的正整数的个数，称** $\phi(n)$ **为欧拉函数**
- $\phi(n) = r_1^{a1-1}(r_1-1)r_2^{a2-1}(r_2-1)\cdots r_n^{an-1}(r_n-1)$

  其中 $n=r_1^{a1}r_2^{a2}\cdots r_n^{an}$。
- **例如：**

  $24 = 2^3*3^1$

  $\phi(24)= 2^{3-1}(2-1)*3^{1-1}(3-1)=8$

  $\{1, 5, 7, 11, 13, 17, 19, 23\}$

2. 欧拉定理

- 若整数 a 和 n 互素，则$a^{\phi(n)} = 1 \pmod{n}$

  （该式的含义：$a^{\phi(n)}$ ÷ n的余数为1）

- **举例说明：**

$\phi(24)=8$

{1, 5, 7, 11, 13, 17, 19, 23}**是小于**24**并与**24**互素的数**

$1^8=1 \mod 24$　　**即：**$1^{\phi(24)}=1 \mod 24$

$5^8=1 \mod 24$　　**即：**$5^{\phi(24)}=1 \mod 24$

$7^8=1 \mod 24$　　**即：**$7^{\phi(24)}=1 \mod 24$

……

3. RSA公钥密码算法

- **RSA**算法的加密对象是数字化信息，待加密的明文，不管是语言、文字，还是数据或是**ASCII**码，总可用一个**0**～**(n-1)**之间的一个整数来表示。也就是先把一条长的明文消息，按定长划分为一串分组，分别用一个整数来代替每一个分组。
- 对于某一明文块**M**和密文块**C**，加密和解密有如下形式：

$$C = M^e \bmod n$$

$$M = C^d \bmod n$$

RSA的密钥对生成算法

- 选取两个大素数p和q，且p≠q，两个数长度接近，一般在256比特长
- 计算n=p*q，$\phi(n)=(p-1)(q-1)$
- 随机选取整数e，满足$\gcd(e, \phi(n))=1$

  （表示e和$\phi(n)$互为素数，两者最大公约数为1）
- 计算d，满足$d*e=1 \pmod{\phi(n)}$。

  （表示：$de \bmod \varphi(n) = 1$，即$de \div \varphi(n)$余数为1）
- e为公钥，n和e公开，公钥PU={e，n}
- d为私钥，d、p和q保密，私钥PR={d，n}

RSA的加解密过程

- **加密：明文**$M < n$

  加密算法：$C = \mathrm{E}(M) = M^e \bmod n$

  C**为密文**

- **解密算法：**$M = \mathrm{D}(C) = C^d \bmod n$

  发送方和接收方都要知道n**和**e**的值，并且只有接收者知道**d**的值。**

RSA的加解密过程（续）

- RSA算法必需满足下列要求：

（1）可以找到e、d、n的值，使得对所有的$M<n$，

$M^{ed} \bmod n = M$ 成立。

（ 因为 $C = M^e \bmod n$

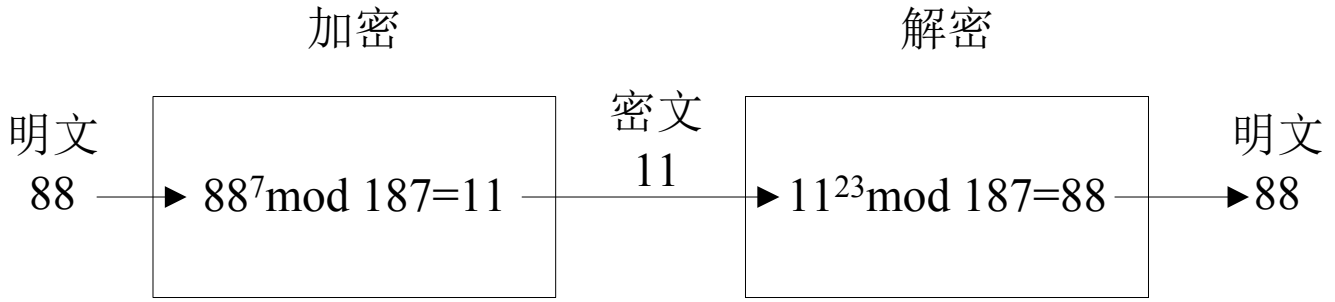
所以 $M = \mathrm{D}(C) = C^d \bmod n = (M^e)^d \bmod n$

$= M^{ed} \bmod n$ ）

（2）对所有满足$M<n$ 的值， 计算M^e和M^d相对容易。

（3）给出e和n，不可能推出d。

RSA的加解密举例



设明文为88，用RSA算法进行加解密。

（1）选择两个素数：p=17和q=11

（2) 计算n=pq=17×11=187

（3）计算φ(n)=(p-1)(q-1)=16×10=160

（4）选择e，使得e与φ(n)=160互素且小于φ(n)；选择e=7

（5）计算d，使得de mod 160=1 且d<160。正确的值是d=23。

RSA的加解密举例（续）

因为 $23\times7=161=10\times16+1$。

得到公钥PU={7，187}，私钥PR= {23，187}。

输入明文88时，加密：

$C=88^7 \bmod 187$，利用模运算的性质，计算：

$$88^7 \bmod 187=[(88^4 \bmod 187)\times(88^2 \bmod 187)\times(88^1 \bmod 187)]\bmod 187$$

$$=11$$

即密文C=11

RSA的加解密举例（续）

解密：

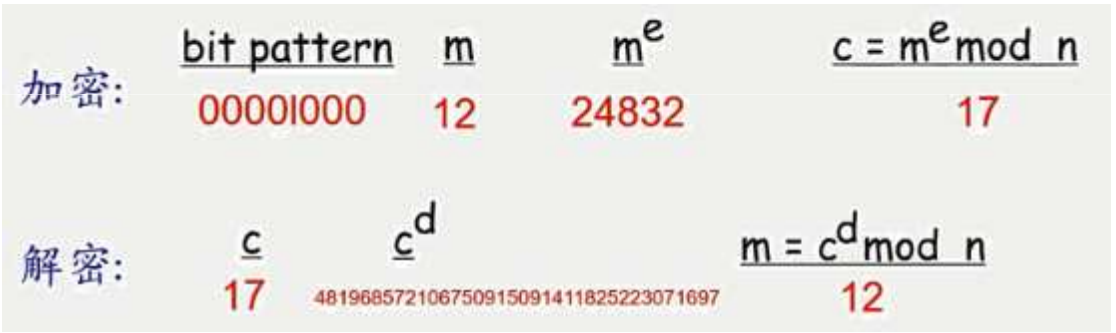
计算：$M=11^{23} \bmod 187$

$M=11^{23} \bmod 187 = 88$

即明文M=88

RSA的加解密举例（2）

- p=5,q=7,n=35,phi=24,e=5,d=29，当m=12时，求c



幂指数模快速算法

- **推导**
- **Python代码**

```
def quickpow(a,b,c):
    A=1
    T=a%c
    while(b!=0):
        if(b&1):
            A=(A*T)%c
        b>>=1
        T=(T*T)%c
    return A
```

Python RSA加解密

- **步骤**
  - **生成私钥**
  - **生成公钥**
  - **用公钥加密文本**
  - **用私钥解密文本**

非对称密码

题目练习：

RSA密钥生成

在一次RSA秘钥生成过程中，假设p=473398607161,q=4511491,e=17,求d

4. RSA算法的安全性

- 对RSA算法的数学攻击实际上等效于对模n乘积因子的分解。
- 随着计算机计算能力的不断提高，原来被认为是不可能分解的大数已被成功分解。**目前密钥长度介于1024比特和2048比特之间的RSA算法是安全的。**
- **对素数p和q的选取的一些限制：**
  - p和q的长度相差不能太大
  - p-1和q-1都应有大的素数因子
  - gcd(p-1,q-1)应该偏小(gcd()是求最大公约数函数)

5. RSA公钥密码的小结

- 第一个较完善的公开密钥算法。
- 目前使用最多的一种公钥密码算法。
- RSA的基础是数论的欧拉定理。
- RSA的安全性依赖于大对数的因数分解的困难性。
- 密码分析者既不能证明也不能否定RSA的安全性。
- 既能用于加密也能用于数字签名。
- RSA算法在美国申请了专利（2000年9月30日到期），但在其他国家无专利。

03

Part

密钥交换

3.5.2 Diffie-Hellman密钥交换

- Diffie-Hellman是第一个发布的公钥算法。
- 许多商业产品采用这种密钥交换技术。
- 该算法的目的是使两个用户能够安全地交换密钥，然后可以用于后续的消息加密。
  - 算法本身仅限于密钥的交换。
- Diffie-Hellman的有效性是建立在计算离散对数是很难这一基础上的。

3.5.2 Diffie-Hellman密钥交换（续）

- **定义素数*p*的本原根，它是一个整数，且它的幂能够生成1～*p*-1的所有整数的数。即如果*a*是素数*p*的一个本原根，则下列各数：**

  $a \bmod p,\ a^2 \bmod p,\ \ldots,\ a^{p-1} \bmod p$

各不相同，但它们组成了1～*p*-1之间整数的一个置换。

对于任意小于*p*的整数*b*和*p*的本原根*a*，能够找到唯一的指数*i*，使得：

$b = a^i \bmod p$　其中 $0 \le i \le (p-1)$

例如：2不是7的本原根，3是7的本原根

3.5.2 Diffie-Hellman密钥交换（续）

- **算法：如图3.12。这个方案有两个公开的值：素数*q*及它的本原根*a*。假设用户A和B希望交换密钥。**

  用户A选择一个随机整数（即A的私钥X_A）$X_A<q$。计算A的公钥Y_A：$Y_A=a^{X_A} \bmod q$。

  用户B选择一个随机整数（即B的私钥X_B）$X_B<q$。计算B的公钥Y_B：$Y_B=a^{X_B} \bmod q$。

  双方保持私钥*X*值，向对方公开公钥*Y*值。

  用户A计算密钥$K=(Y_B)^{X_A} \bmod q$。

  用户B计算密钥$K=(Y_A)^{X_B} \bmod q$。

  用户A与用户B的密钥K计算结果相同。

3.5.2 Diffie-Hellman密钥交换（续）

密钥K=$(Y_B)^{X_A} \bmod q$

$=(a^{X_B} \bmod q)^{X_A} \bmod q$

$=(a^{X_B})^{X_A} \bmod q$

$=a^{X_B X_A} \bmod q$

$=(a^{X_A})^{X_B} \bmod q$

$=(a^{X_A} \bmod q)^{X_B} \bmod q$

$=(Y_A)^{X_B} \bmod q$

这样，双方就交换了密钥K。

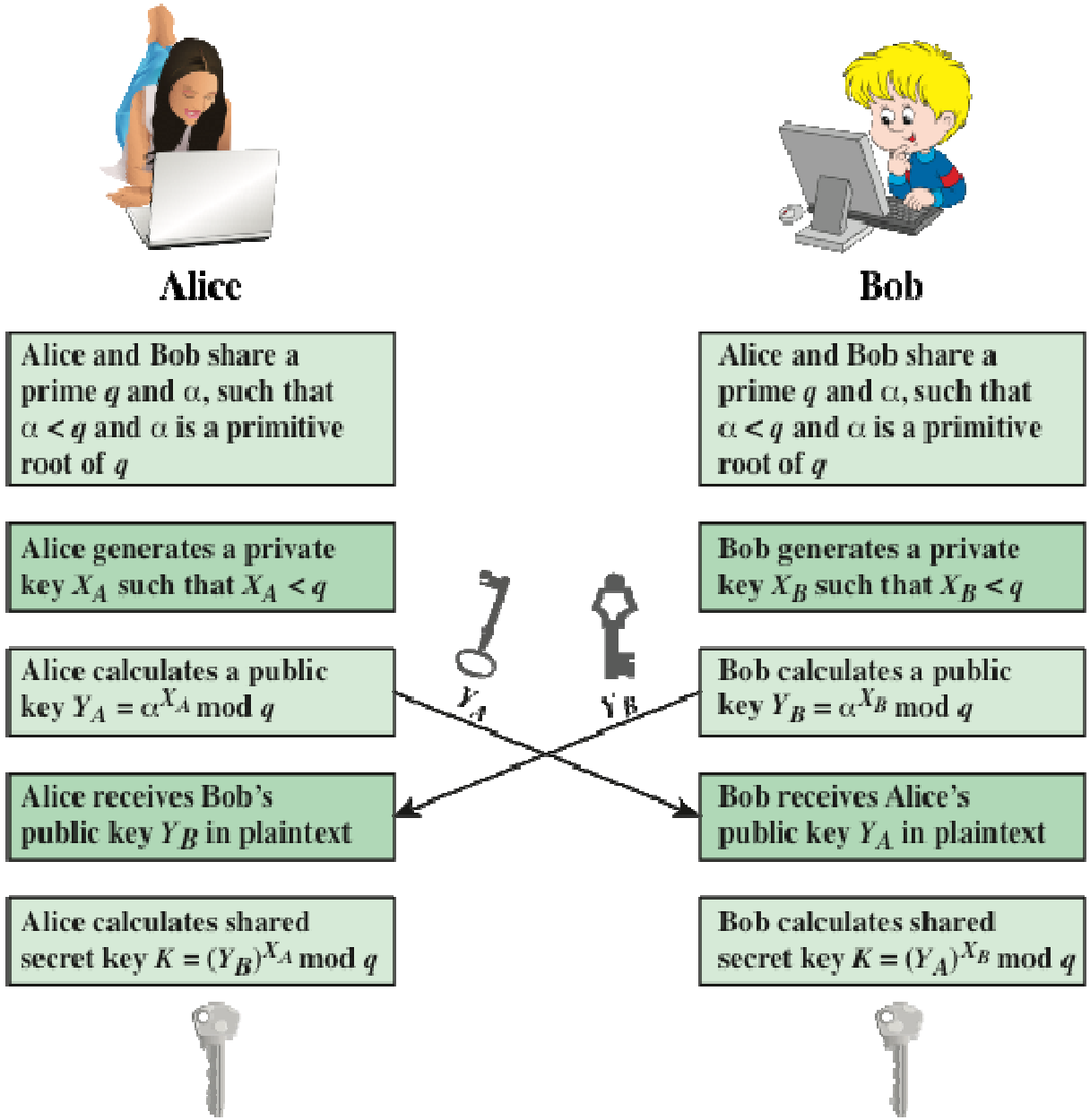


Figure 3.12 Diffie-Hellman Key Exchange

3.5.2 Diffie-Hellman密钥交换（续）

例：取素数q=353和本原根a=3。

用户A的私钥X_A=97，用户B的私钥X_B=233，

用户A计算公钥$Y_A=3^{97} \bmod 353=40$

用户B计算公钥$Y_B=3^{233} \bmod 353=248$

它们交换公钥后，分别计算公共密钥：

用户A计算K $=(Y_B)^{X_A} \bmod 353 =(248)^{97} \bmod 353=160$

用户B计算K $=(Y_A)^{X_B} \bmod 353 =(40)^{233} \bmod 353=160$

蛮力破解

中间人攻击

- **Diffie-Hellman中间人攻击**
  - **不能抵御中间人攻击**
  - **没有对通信的参与方进行认证**
- **使用数字签名和公钥证书**

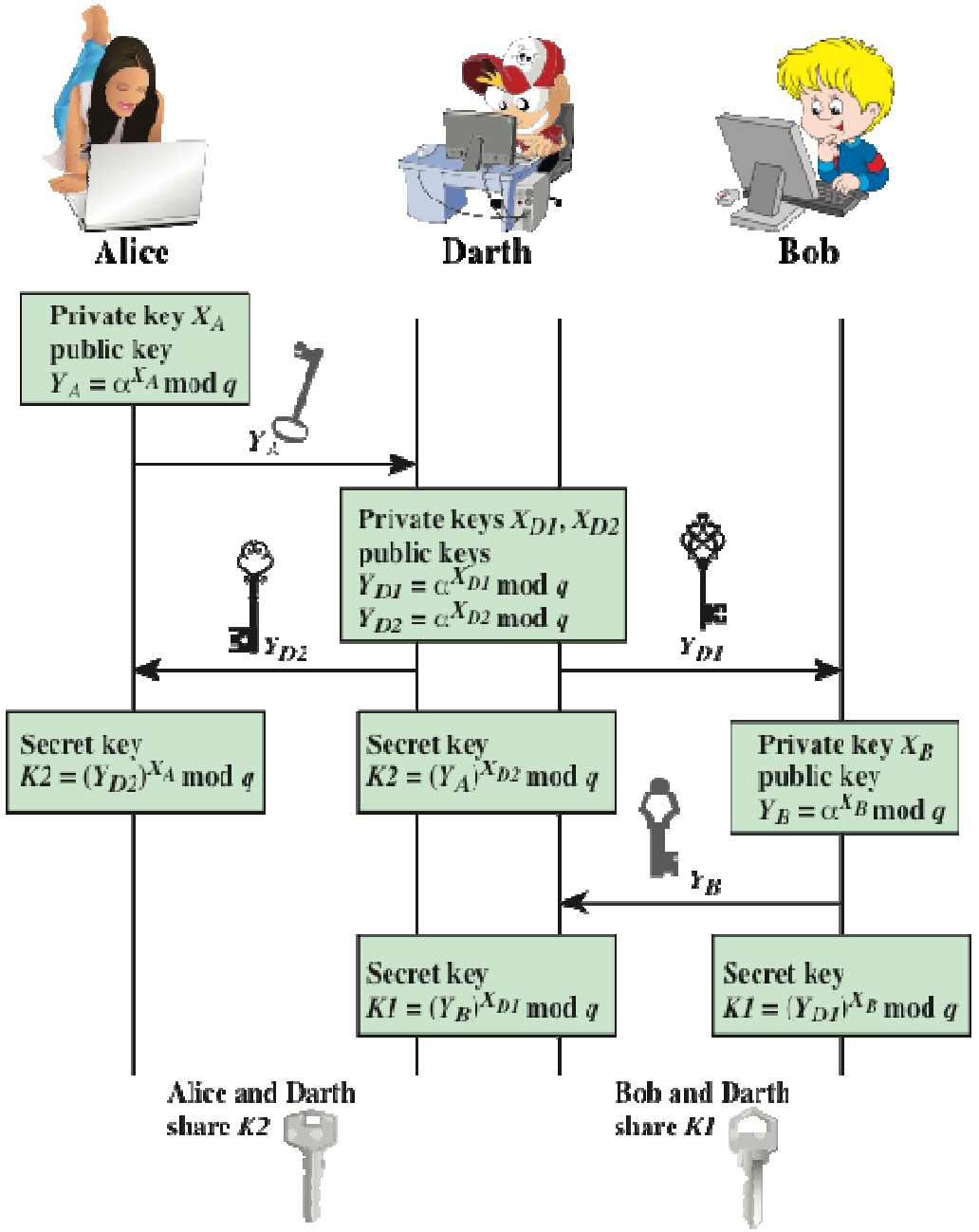


Figure 3.13 Man-in-the-Middle Attack

3.5.3 其它公钥密码算法

已被商业接受的两种其它的公钥算法是DSS和椭圆曲线密码。

1. 数字签名标准(DSS)

- DSS是NIST发布的联邦信息处理标准FIPS PUB 186。
- DSS使用了SHA-1，并提出一种新的数字签名技术，即数字签名算法（DSA）。
- 最初于1991年提出并于1993年和1996年再次修订。
- 使用了一种专为数字签名功能而设计的算法。
- 与RSA不同，它不能用于加密或密钥交换。

2. 椭圆曲线密码(ECC)

- ECC基于使用称为椭圆曲线的数学结构。
- 与RSA相比，ECC的主要吸引力在于它只需要更少的比特数就可以提供相同强度的安全性，从而减少了处理开销。
- 人们对ECC的信赖水平还没有RSA高。

ElGamal Cryptography

- public-key cryptosystem related to D-H
- so uses exponentiation in a finite (Galois)
- with security based difficulty of computing discrete logarithms, as in D-H
- each user (eg. Alice) generates their key
  - chooses a secret key (number): $1 < x_A < q-1$
  - compute their public key: $y_A = a^{x_A} \bmod q$

ElGamal Message Exchange

- **Bob encrypt a message to send to Alice computing**
  - **represent message M in range $0 <= M <= q-1$**
    - **longer messages must be sent as blocks**
  - **chose random integer k with $1 <= k <= q-1$**
  - **compute one-time key $K = y_A^k \bmod q$**
  - **encrypt M as a pair of integers (C_1, C_2) where**
    - $C_1 = a^k \bmod q \; ; \; C_2 = KM \bmod q$
- **Alice then recovers message by**
  - **recovering key K as $K = C_1^{x_A} \bmod q$**
  - **computing M as $M = C_2 \; K^{-1} \bmod q$**
- **a unique k must be used each time**
  - **otherwise result is insecure**

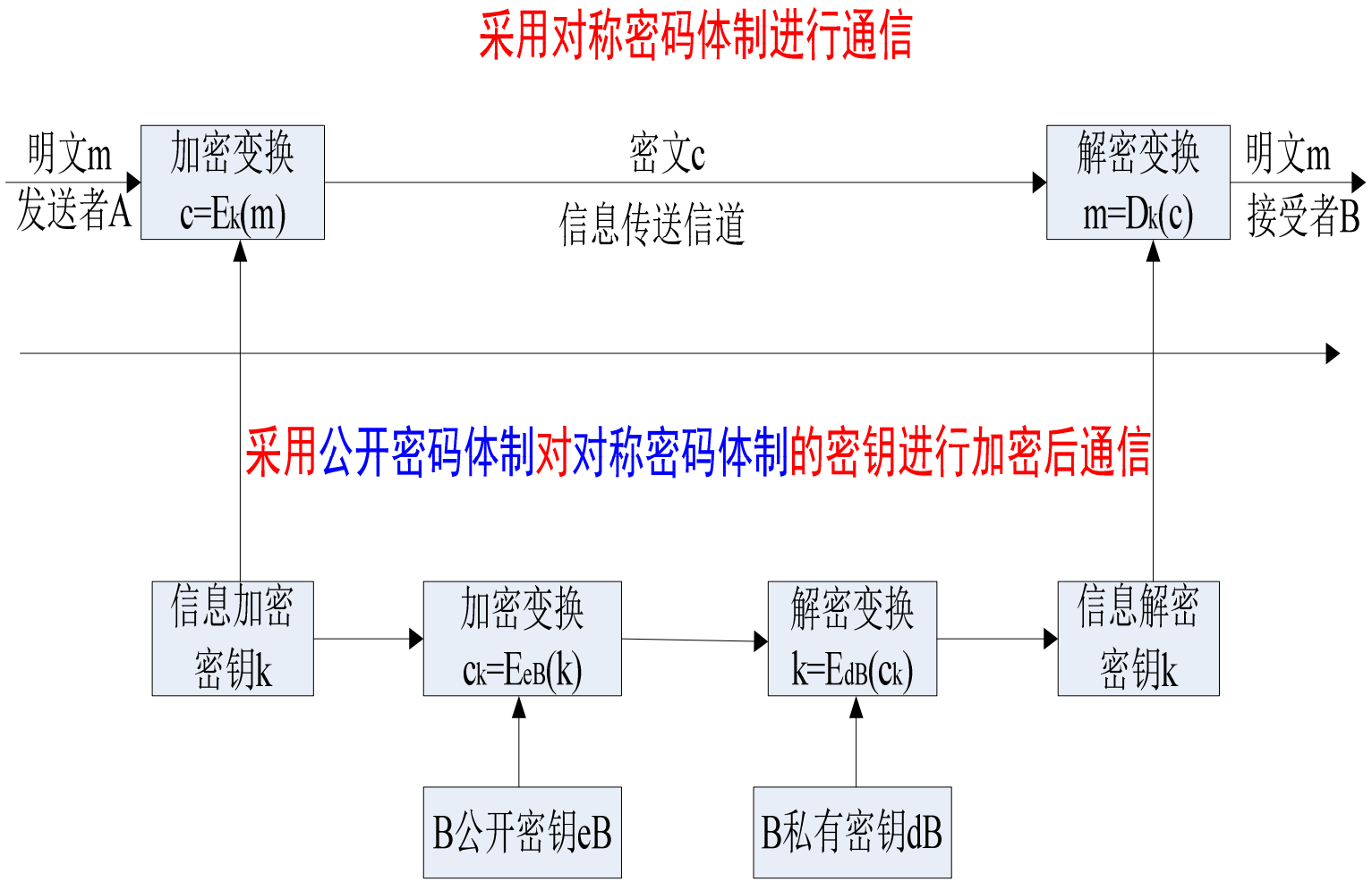
ElGamal Example

- use field GF(19) $q=19$ and $a=10$
- Alice computes her key:
  - A chooses $x_A=5$ & computes $y_A=10^5 \bmod 19 = 3$
- Bob send message $m=17$ as $(11,5)$ by
  - chosing random $k=6$
  - computing $K = y_A^k \bmod q = 3^6 \bmod 19 = 7$
  - computing $C_1 = a^k \bmod q = 10^6 \bmod 19 = 11;$
    $C_2 = KM \bmod q = 7.17 \bmod 19 = 5$
- Alice recovers original message by computing:
  - recover $K = C_1^{x_A} \bmod q = 11^5 \bmod 19 = 7$
  - compute inverse $K^{-1} = 7^{-1} = 11$
  - recover $M = C_2\ K^{-1} \bmod q = 5.11 \bmod 19 = 17$

公钥密码体制的优缺点

- 优点：
  - 解决密钥传递的问题
  - 大大减少密钥持有量
  - 提供了对称密码技术无法或很难提供的服务（数字签名）
- 缺点：
  - 计算复杂、耗用资源大
  - 非对称会导致得到的密文变长

混合加密体制


采用对称密码体制进行通信
明文m
发送者A
加密变换
c=Ek(m)
密文c
信息传送信道
解密变换
m=Dk(c)
明文m
接受者B
采用公开密码体制对对称密码体制的密钥进行加密后通信
信息加密
密钥k
加密变换
ck=EeB(k)
解密变换
k=EdB(ck)
信息解密
密钥k
B公开密钥eB
B私有密钥dB

04

Part

数字签名

为什么需要数字签名

- 消息认证
  - 保护信息交换双方免受第三方攻击
- 在某些场合下，可能存在通信双方自身相互欺骗。假定A发送一个消息息给B,双方之间的争议可能有多种
  - B伪造一个不同B的消息，但声称是从A收到
  - A可以否认发过该消息，B无法证明A确实发了该消息
- •典体实例
  - 电子金融交易中篡改金额
  - 股票交易亏损后抵赖

数字签名

- **数字签名是笔迹签名的模拟，其特征：**
  - **必须能够验证作者及其签名的日期时间**
  - **必须能够验证认证签名时的内容**
  - **签名必须能够由第三方验证，以解决争议**
- **因此，数字签名功能包含了认证的功能**

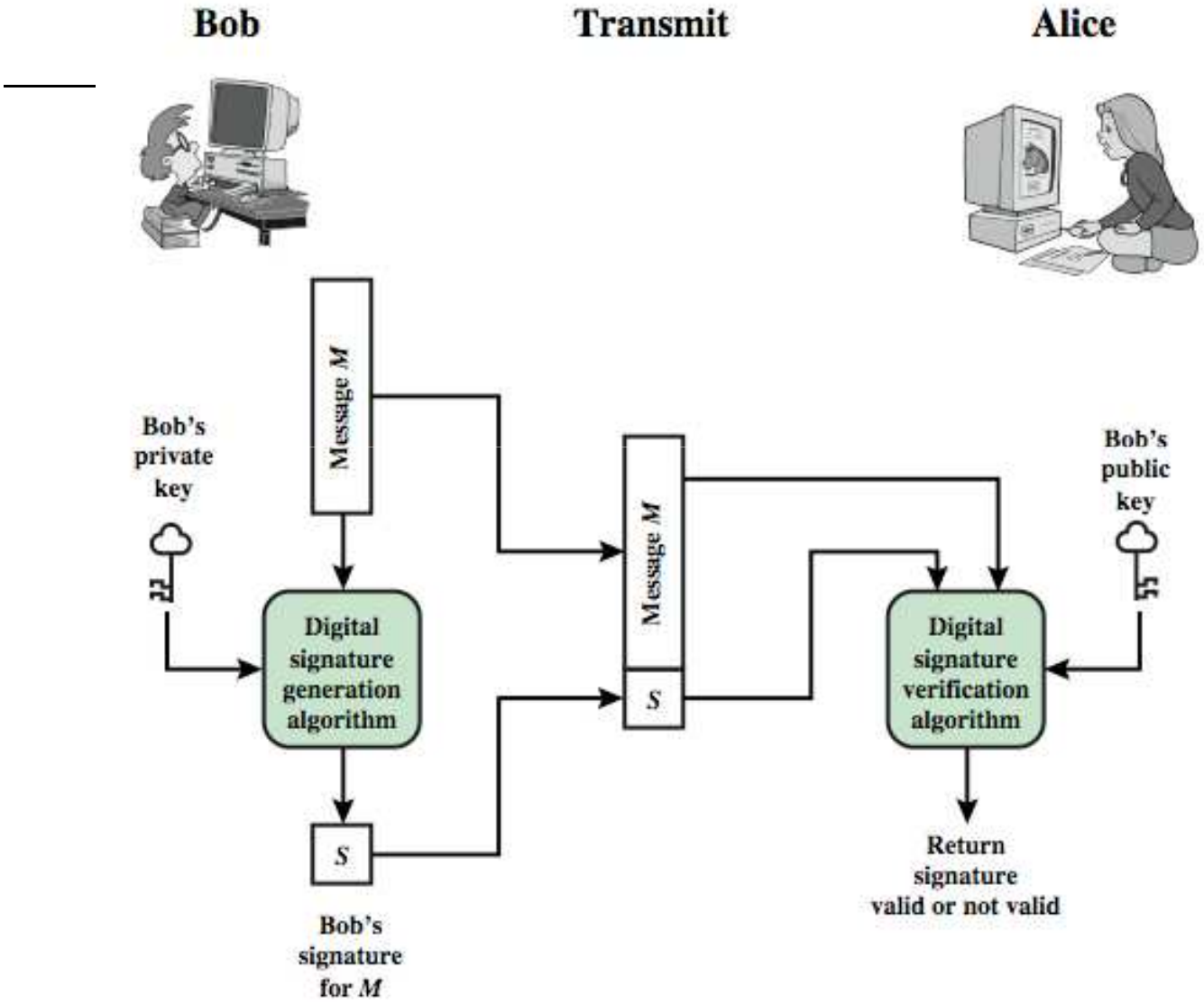
数字签名的设计需求

- 签名必须是依赖于被签名信息的一个位串模式；签名必须使用某些对发送者是唯一的信息，以防止双方的伪造与否认
- 生成该数字签名必须比较容易；识别和验证该数字签名必须比较容易
- 伪造该数字签名在计算上不可行，既包括对一个已有的数字签名构造新的消息，也包括对一个给定消息伪造一个数字签名
- 保存一个数字签名副本是可行的

数字签名

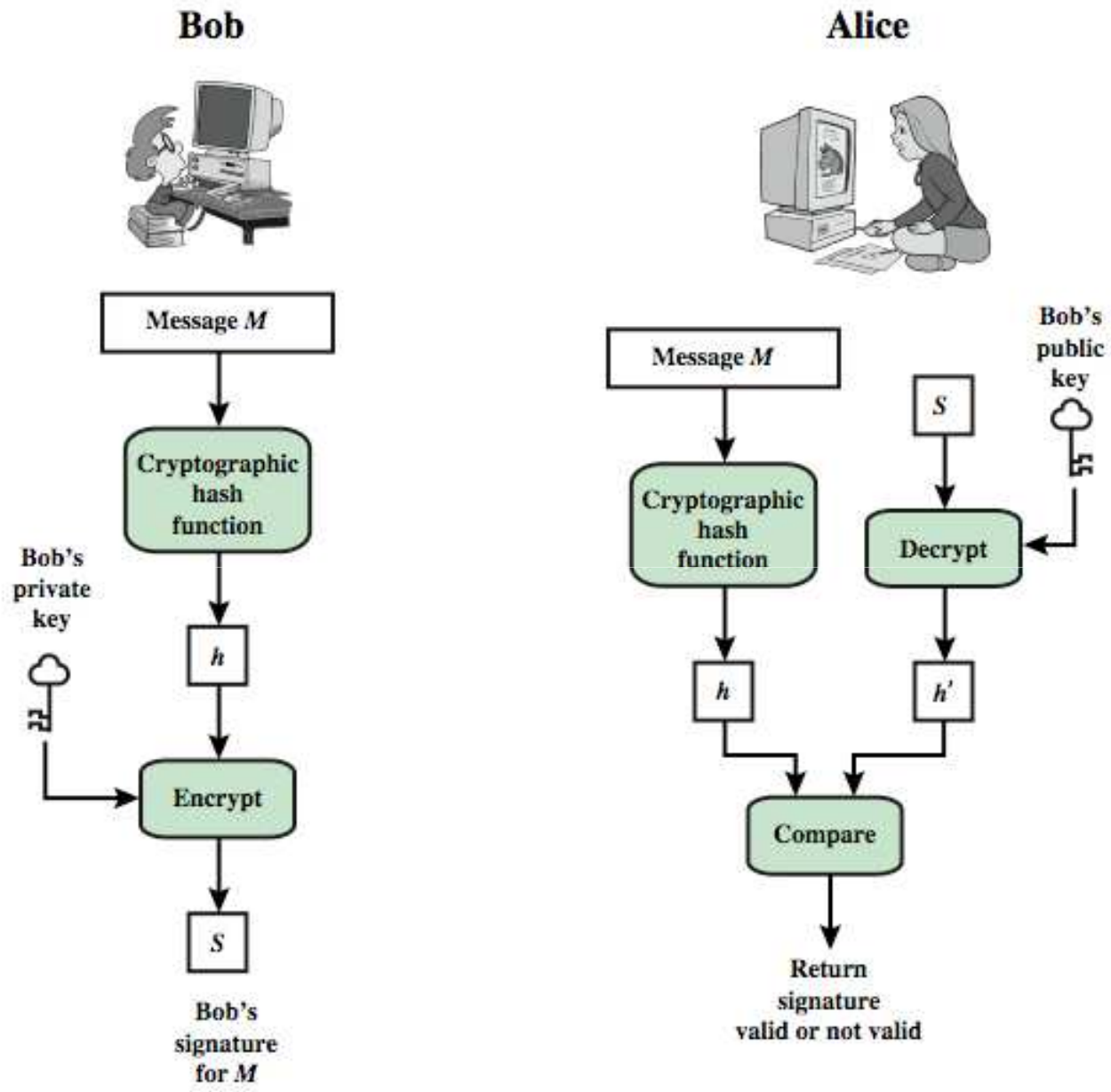
- **只有信息的发送者才能产生的别人无法伪造的一段数字串**
- **这段数字串同时也是对信息的发送者发送信息真实性的一个有效证明**

数字签名一般模型


Bob
Transmit
Alice
Message M
Bob's private key
22
Digital signature generation algorithm
S
Bob's signature for M
Message M
S
Bob's public key
55
Digital signature verification algorithm
Return signature valid or not valid

数字签名技术

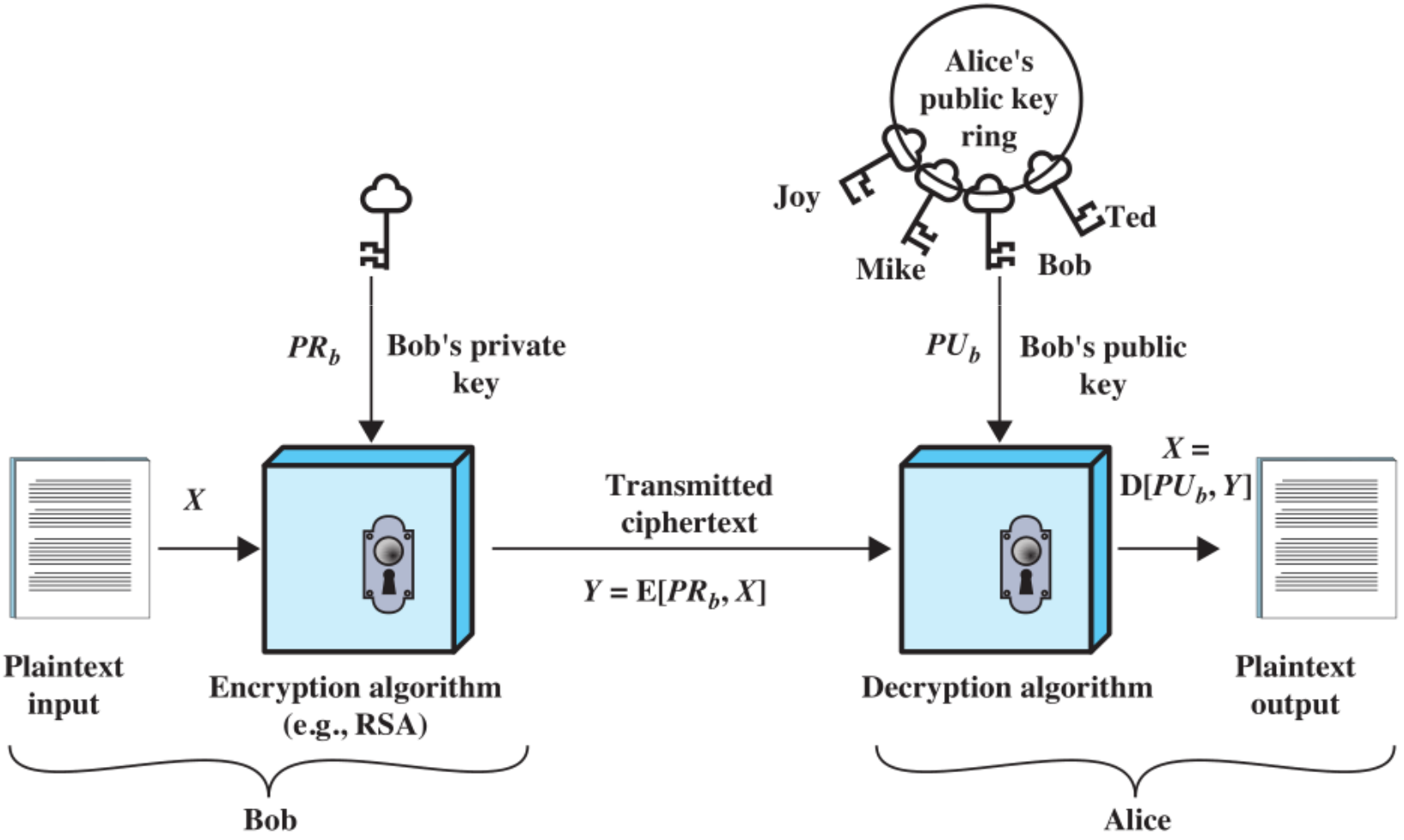
- 将摘要信息用发送者的私钥加密
- 与原文一起传送给接收者
- 接收者只有用发送者的公钥才能解密被加密的摘要信息
- 接收者用HASH函数对收到的原文产生一个摘要信息，与解密的摘要信息对比
  - 如果相同，则说明收到的信息是完整的，在传输过程中没有被修改
  - 否则说明信息被修改过，因此数字签名能够验证信息的完整性


Bob
Alice
Message M
Cryptographic hash function
Bob's private key
22
h
Encrypt
S
Bob's signature for M
Message M
Cryptographic hash function
h
S
Bob's public key
55
Decrypt
h'
Compare
Return signature valid or not valid

数字签名（例）

- 假设Bob想给Alice发送消息。 Alice要确定这个消息是Bob发送的。
- Alice收到密文时，能够用Bob的公钥进行解密，从而证明这个消息确实是Bob加密的
  - 因为其他人没有Bob的私钥，所以其他人无法创建由Bob的公钥能解密的密文
  - 因此，整个加密消息就为一个数字签名。
- 由于没有Bob的私钥就不能篡改消息，所以，数字签名不仅认证了消息源，还保证了数据的完整性。

图3.9 公钥密码



（b）用私钥签名（认证）

数字签名

- 如图3.2（b）所示，加密一个小的数据块（称为**认证符**），这个数据块是整个文档的函数
- 认证符必须具备这样的性质：只改变文档而不改变认证符是做不到的。这样发送者用私钥加密认证符，它就可以作为认证源、内容和顺序的**签名**
- **安全散列码**就可以完成这种功能，如SHA-1

数字签名

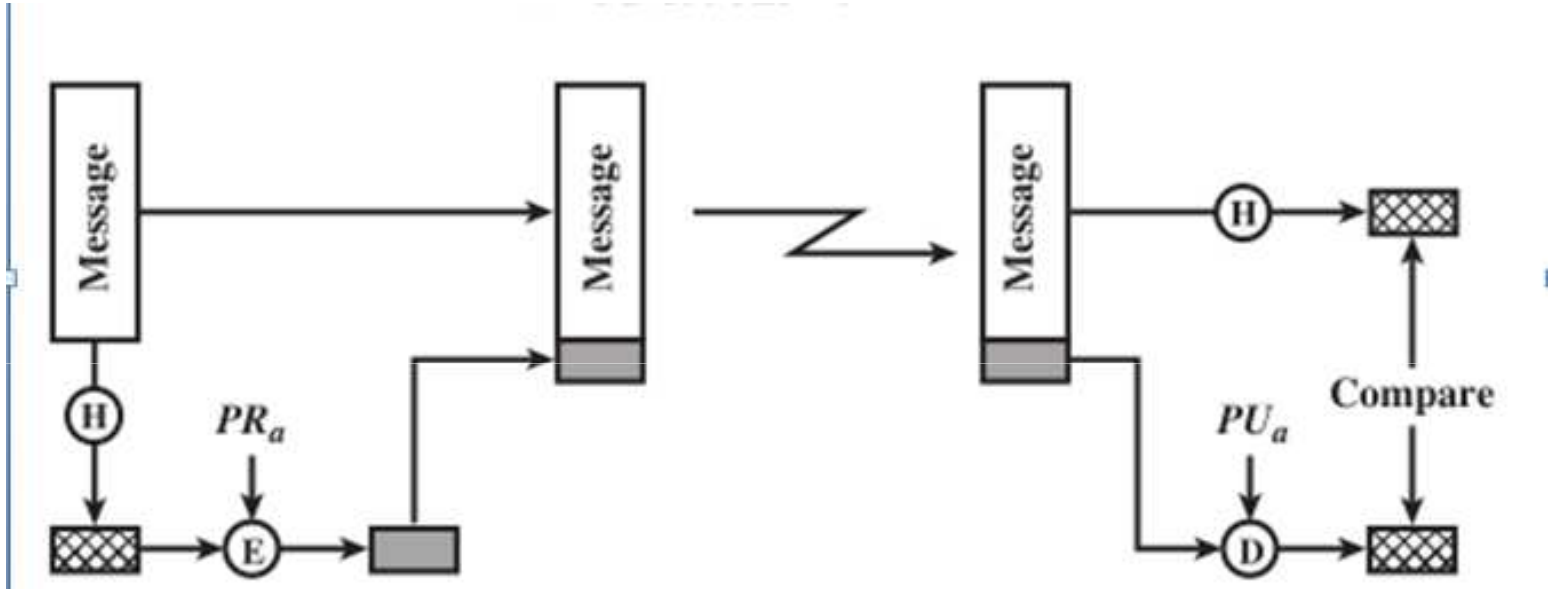
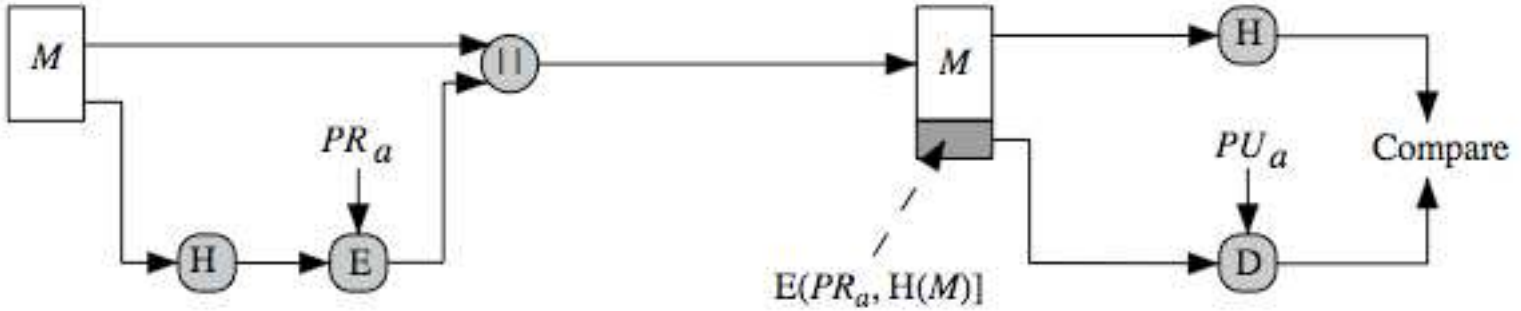


图3.2（b) 使用单向散列函数的消息认证

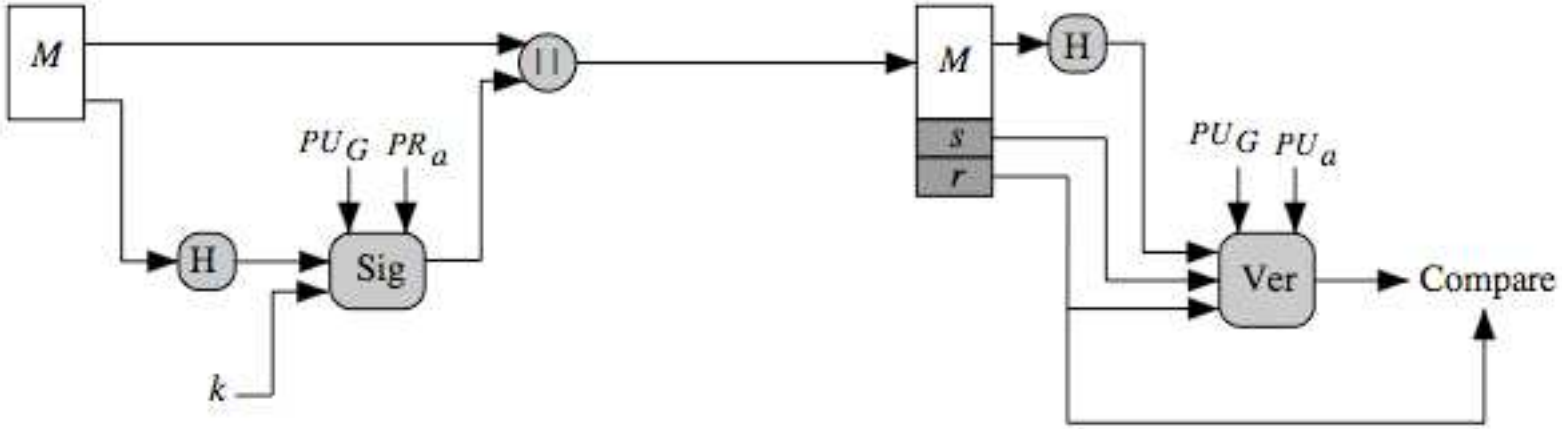
Digital Signature Standard (DSS)

- US Govt approved signature scheme
- designed by NIST & NSA in early 90's
- published as FIPS-186 in 1991
- revised in 1993, 1996 & then 2000
- uses the SHA hash algorithm
- DSS is the standard, DSA is the algorithm
- FIPS 186-2 (2000) includes alternative RSA & elliptic curve signature variants
- DSA is digital signature only unlike RSA
- Is a public-key technique

DSS vs RSA Signatures



(a) RSA Approach



(b) DSS Approach

Digital Signature Algorithm (DSA)

- creates a 320 bit signature
- with 512-1024 bit security
- smaller and faster than RSA
- a digital signature scheme only
- security depends on difficulty of computing discrete logarithms
- variant of ElGamal & Schnorr schemes

DSA Key Generation

- have shared global public key values (p,q,g):
  - choose 160-bit prime number q
  - choose a large prime p with $2^{L-1} < p < 2^{L}$
    - where L= 512 to 1024 bits and is a multiple of 64
    - such that q is a 160 bit prime divisor of (p-1)
  - choose $g = h^{(p-1)/q}$
    - where $1<h<p-1$ and $h^{(p-1)/q} \bmod p > 1$
- users choose private & compute public key:
  - choose random private key: $x<q$
  - compute public key: $y = g^{x} \bmod p$

DSA Signature Creation

- **to sign a message `M` the sender:**
  - **generates a random signature key `k, k<q`**
  - **`k` must be random, be destroyed after use, and never be reused**
- **then computes signature pair:**

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

  $s = [k^{-1}(H(M) + xr)] \bmod q$
- **sends signature `(r,s)` with message `M`**

DSA Signature Verification

- **having received M & signature `(r,s)`**
- **to verify a signature, recipient computes:**

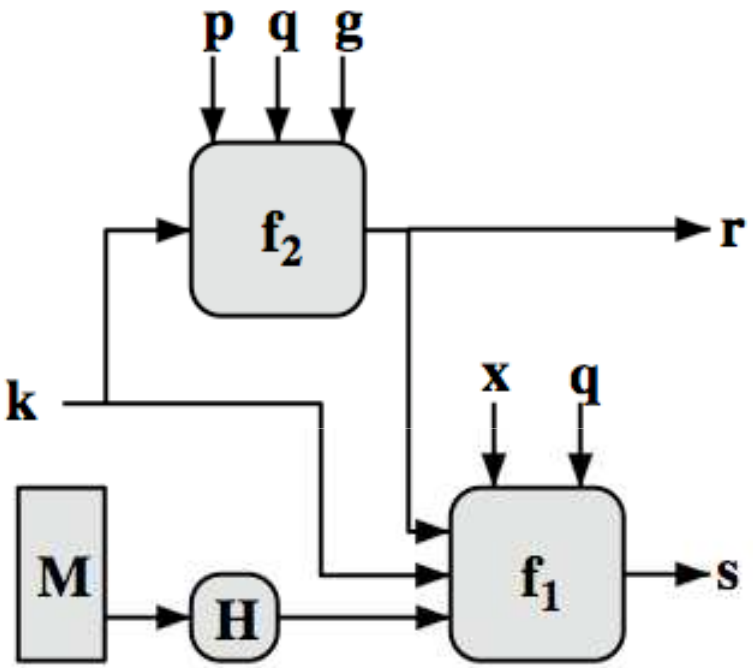
  $w = s^{-1} \bmod q$

  $u1 = [H(M)w] \bmod q$

  $u2 = (rw) \bmod q$

  $v = [(g^{u1}\ y^{u2}) \bmod p] \bmod q$
- **if `v=r` then signature is verified**

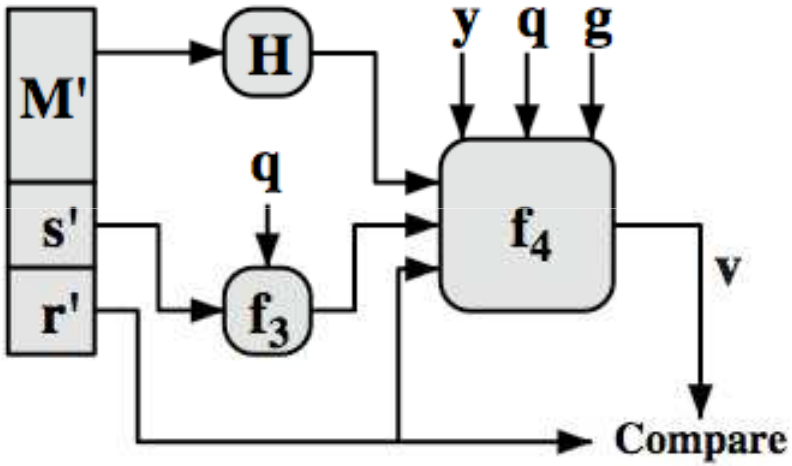
DSS Overview



$s = f_1(H(M), k, x, r, q) = (k^{-1} (H(M) + xr)) \bmod q$

$r = f_2(k, p, q, g) = (g^k \bmod p) \bmod q$

(a) Signing



$w = f_3(s', q) = (s')^{-1} \bmod q$

$v = f_4(y, q, g, H(M'), w, r')$

$= ((g^{(H(M')w) \bmod q} y^{r'w \bmod q}) \bmod p) \bmod q$

(b) Verifying

Python RSA签名

- **步骤**
  - **生成私钥**
  - **生成公钥**
  - **用私钥签名文本摘要**
  - **用公钥验证文本摘要**

04

Part

小结

小结/Summary

01. 消息认证

02. 公钥密码

03. 数字签名