加密解密系统

通常一个密钥加密解密系统应该包括以下部分

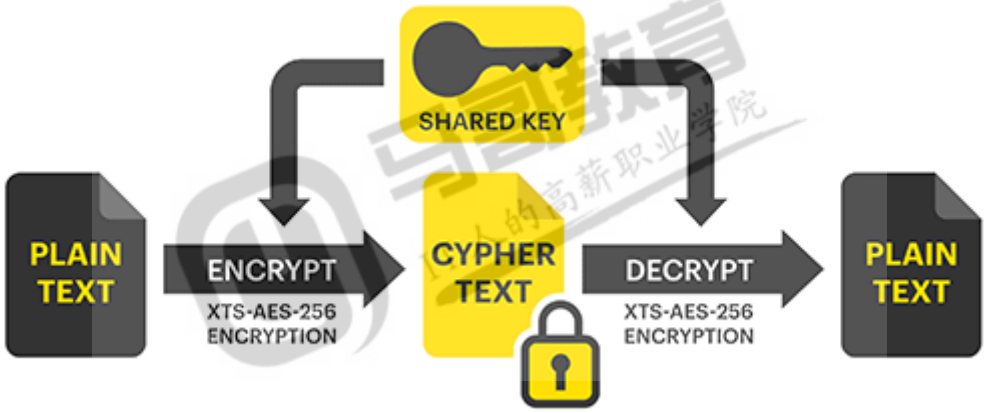
- 消息空间M（Message），就是未加密的数据空间
- 密文空间C（Ciphertext），加密后得到的数据空间
- 密钥空间K（Key），可以使用的各种密码
- 加密算法E（Encryption Algorithm）
- 解密算法D（Decryption Algorithm）

加密过程：消息空间中的某一个消息M_1（明文）使用密钥空间中的摸一个密钥K_1，通过加密算法E加密后得到密文C_1，记作$E_{k1}(M_1) = C_1$。

解密过程：将密文C_1使用解密密钥K_2通过解密算法D解密后得到原始的明文，记作$D_{k2}（C_1）= M_1$。

对称加密

对称加密：加密密钥和解密密钥相同密钥的加密解密系统，称为对称加密，也称单密钥加密。



- 加密、解密使用同一组密钥
- 算法公开、计算量小、加密速度快、加密效率高，适合大量数据的场合
- 加密、解密双方，如果一方密钥泄露，数据就不安全了
- 常见加密算法：DES、AES

对称加密算法有两种类型：分组密钥（Block Cipher）和流密码（Stream Cipher）。

流密码：对输入元素进行连续处理，同时产生连续单个输出元素。

分组密码：将明文分成固定长度的分组，各个分组在密钥控制下使用算法变换成等长的密文分组。常见的算法有DES、3DES、AES、Blowfish等。

DES

DES（Data Encryption Standard）数据加密标准，是目前最流行的加密算法之一。

DES分组密码算法

- 用户提供64位初始密钥，经过一系列处理获得k1、k2、...、k16总共16个48位子密钥，分别用在1~16轮运算中

- 将明文分组（Block），每组64位，每组使用16个子密钥通过加密函数处理得到64位密文
- 加密解密过程相同，但密钥应用次序和加密时相反

DES工作模式

1. ECB（Electornic Codebook电子密码本）模式：64位的明文使用加密得到64位密文，将各组密文按顺序连接得到最终密文
2. CBC（Cipher Block Chaining分组链接）模式：每一组明文先和前一组密文计算后，再加密后得到密文。也就是说前一组的结果需代入下一组的计算，这看起来就是一个链。需提供一个初始向量IV和第一组明文计算。
3. CFB（Cipher Feed Back密文反馈）模式：增加移位寄存器。假设n为32，每组明文、密文都是n位。将上一组的n位密文分组，加入到移位寄存器中，使用移位寄存器中最右端64位加密得到64位结果，取结果中n位和n位明文计算得到n位密文
4. OFB（Output Feed Back输出反馈）模式：和CFB不同，前一组加密后的64位结果的前n位加入到下一组的移位寄存器中

AES

AES（Advanced Encryption Standard）高级加密标准，主要是为了取代DES算法。

- 分组大小为128位
- 密钥长度也为128位
- 采用Rijndael加密算法，允许128、192、256位分组加密

相较于DES

- 计算对内存要求极低，适用于资源受限的硬件环境
- 计算更加快速，在各种处理器平台都能实现更快速的计算
- 安全性更高，具有很好的抵抗差分密码分析及线性密码分析的能力
- 分组长度和密钥长度可变，都可以是32位的任意倍数，最小为128位，最大为256位

块填充

分组加密时，要求分组（块）大小必须是固定大小，例如为16字节，如果明文长度正好不是16字节的倍数，就需要补齐为16字节。

- 待填充字节数n = 块大小 - 数据长度 % 块大小
    - r的范围是[1, n]
    - n等于块大小，明文正好对齐
    - r的值在[1, n-1]，说明要补齐
- Zero：待填充字节的每个字节都补零0x00
- PKCS7是当下各大加密算法都遵循的填充算法
    - 如果需要补齐，待填充字节的每个字节都填充为待填充字节数n
    - 如果不需要补齐，需要追加一个块大小的数据，每个字节填充为块大小

AES128，数据块为16Byte（数据长度需要为16Byte的倍数）。

假设数据为0x11223344556677889900AA，共11个Byte,缺了5个Byte,采用PKCS7Padding方式填充之后的数据为0x11223344556677889900AA0505050505。

如果是数据刚好满足数据块长度，也需要在数据后面按PKCS7规则填充一个数据块数据，这样做的目的是为了区分有效数据和补齐数据。
假设数据为0x11223344556677889900AABBCCDDEEFF,共16个Byte，采用PKCS7Padding方式，仍需要在数据后面补充一个16个Byte的块，填充之后的数据为
0x11223344556677889900AABBCCDDEEFF10101010101010101010101010101010

库

Go语言提供了标准库 https://pkg.go.dev/crypto，包含了DES、AES、MD5、SHA1等常用加密解密算法库。

CBC例子参考 https://pkg.go.dev/crypto/cipher#example-NewCBCEncrypter

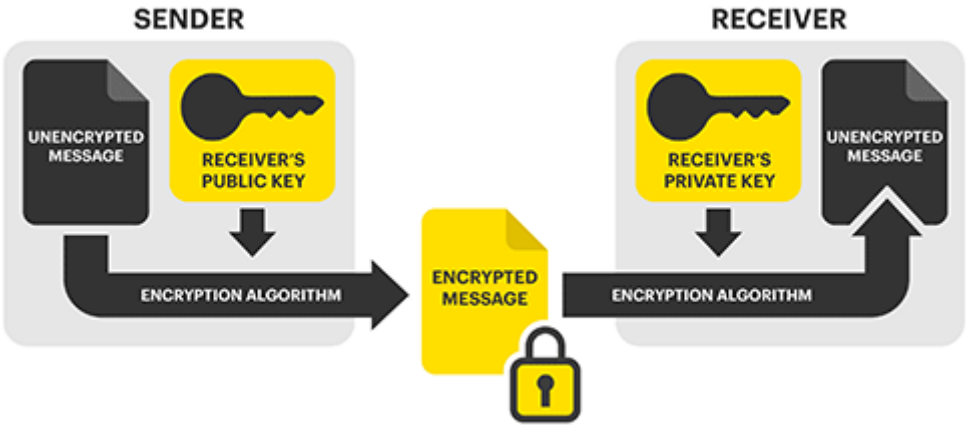
```go
package main

import (
    "bytes"
    "crypto/aes"
    "crypto/cipher"

    "encoding/hex"
    "fmt"
)

func main() {
    // 为了让大家理解算法，就没有封装成函数
    password := "6368616e6765207468697320070617373" // 密码,32个字符
    plainText := []byte("exampleplaintext")        // 明文，16

    fmt.Println(password, len(password))
    // key密钥，自定义，可以是16、24、32字节
    key, _ := hex.DecodeString(password) // 按照16进制理解password
    fmt.Println(key, len(key))
    fmt.Printf("明文长度=%d字节，密钥长度%d字节\n", len(plainText), len(key)) //
16 16

    // 块必须为指定大小，不够就补齐
    // 本次采用PKCS7Padding方式
    blockSize := aes.BlockSize
    fmt.Printf("默认分组大小为 %d字节\n", blockSize)
    r := len(plainText) % blockSize // 余数
    n := blockSize - r              // 待填充字节数
    if n == blockSize {
        // 正好满足分组字节要求，追加一个块，每个字节填充块大小
        fmt.Printf("正好满足分组块大小要求，追加一个块（%d个字节）\n", n)
    } else {
        fmt.Printf("不满足块大小要求，需要补充%d字节\n", n)
    }
```

```go
    padding := bytes.Repeat([]byte{byte(n)}, n)
    fmt.Printf("%d, %v\n", len(padding), padding)
    paddingText := append(plainText, padding...) // 补完后去加密
    fmt.Printf("%d, %v\n", len(paddingText), paddingText)

    block, err := aes.NewCipher(key)
    if err != nil {
        panic(err)
    }

    // 加密
    // CBC模式需要提供一个与第一分组计算的初始向量iv, iv字节数为块大小，这里取16字节
    iv := []byte("abcdef0123456789")
    enMode := cipher.NewCBCEncrypter(block, iv)
    cipherText := make([]byte, len(paddingText))
    enMode.CryptBlocks(cipherText, paddingText)
    fmt.Printf("密文：%x\n", cipherText)

    // 解密，使用密文cipherText解密
    deMode := cipher.NewCBCDecrypter(block, iv)
    text := make([]byte, len(cipherText))
    deMode.CryptBlocks(text, cipherText)
    padding1 := text[len(text)-1] // text中最后一个字节一定是补充的字节数
    fmt.Printf("明文：%x, %[1]s\n", text[:len(text)-int(padding1)])
}
```

国人写的库 https://github.com/golang-module/dongle/blob/main/README.cn.md

非对称加密

非对称加密：加密和解密密钥不同且其中一个密钥不能通过另一个密钥推算出来。这种算法需要2个密钥，公开密钥和私有密钥。如果公钥对数据加密，就只能使用私钥才能解密；如果使用私钥对数据加密，也只能使用公钥解密。这就是非对称加密方式。



RSA密码系统

1977年提出了RSA加密算法，名字是三位作者姓氏首字母的组合。

RAS密钥系统的安全性基于大数分解的困难性。计算一对大素数的乘积很容易，但是要对这个乘积进行因式分解则非常困难。因此，把一对大素数的乘积公开作为公钥，把素数作为私钥，那么从公钥和密文中恢复出（暴力破解）明文的难度等价于大素数乘积的质因数分解。

> 对极大整数做因数分解的难度决定了RSA算法的可靠性。换言之，对一极大整数做因数分解愈困难，RSA算法愈可靠。假如有人找到一种快速因数分解的算法，那么RSA的可靠性就会极度下降。但找到这样的算法的可能性是非常小的。今天只有短的RSA密钥才可能被暴力破解。到2008年为止，世界上还没有任何可靠的攻击RSA算法的方式。 **只要密钥长度足够长，用RSA加密的信息实际上是不能被破解的。**
>
> 维基百科

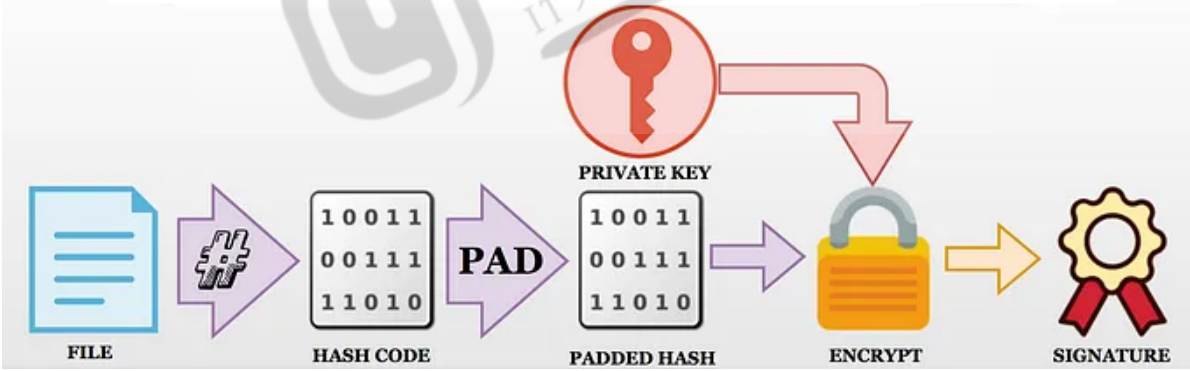
RSA目前推荐长度至少2048位，相对安全，据称目前使用超级计算机破解也需要80年。但是，有研究表示使用了量子计算机破解2048位只需要数小时。

数字签名

在计算机系统中，当接收者接收到一个消息时，往往需要验证消息在传输过程中是否有被篡改过，也要确定消息发送者的身份，这就需要数字签名技术来实现。

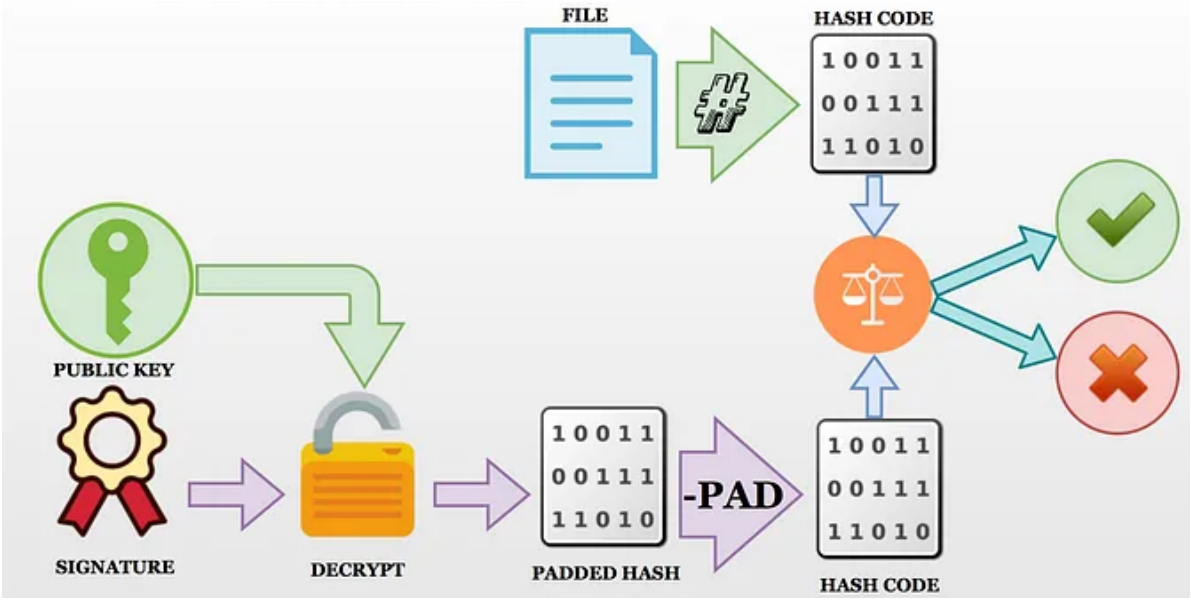
RSA签名方案

1、签名



消息是公开的，使用哈希函数对消息哈希后得到哈希值，再使用私钥对哈希值进行加密得到签名。

2、校验



对消息再次哈希得到哈希值h1。使用公钥对签名解密得到哈希值h2。比对h1、h2是否相同，如果不同，则被篡改过，否则没被篡改过，是可信的。

可以看出，整个方案都是以RSA算法为基础，是以RSA算法难以破解为前提的。

单向散列函数

参考数据结构哈希表章节。