|  |
| --- |
| 华中科技大学 电信系 2011级 |
| 应用密码学课程报告 |
| Caesar DES MR-test RSA SHA1 |

|  |
| --- |
| 姓名：邹天宇  学号：U200913745  班级：电信1105  日期：2014-10-31 |

## 凯撒密码

凯撒密码是古典密码学中最简单的加密技术，又称移位密码。它是一种替换加密的技术，明文中的所有字母都在字母表上向后（或向前）按照一个固定数目进行偏移后被替换成密文。凯撒密码的加解密流程见下：

 

### 核心加密代码

// 加密单个字符  
Caesar.prototype.\_encryptChar = **function** (ch) {  
 **var** x = **this**.map[ch]  
 **var** y = (x + **this**.k) % **this**.n  
 **return this**.pool[y]  
}  
  
  
*/\*\*  
 \* 依次加密每个字符  
 \** ***@param*** *{String} text 明文字符串  
 \** ***@returns*** *{String} 密文字符串  
 \*/*Caesar.prototype.encrypt = **function** (text) {  
 **return** \_.map(text, **function** (ch) {  
 **return this**.\_encryptChar(ch)  
 }, **this**).join('')  
}

### 核心解密代码

// 解密单个字符  
Caesar.prototype.\_decryptChar = **function** (ch) {  
 **var** y = **this**.map[ch]  
 **var** x = (y - **this**.k + **this**.n) % **this**.n  
 **return this**.pool[x]  
}  
  
*/\*\*  
 \* 依此解密每个字符  
 \** ***@param*** *{String} cipherText 密文字符串  
 \** ***@returns*** *{String} 明文字符串  
 \*/*Caesar.prototype.decrypt = **function** (cipherText) {  
 **return** \_.map(cipherText, **function** (ch) {  
 **return this**.\_decryptChar(ch)  
 }, **this**).join('')  
}

### 加解密实例

**var** caesar = **new** Caesar(15)  
assert.equal(  
 caesar.encrypt('Caesar is a title of imperial character.'),  
 'RptFpEoxFopoGxG,toBuox.CtExp,orwpEprGtEO')  
assert.equal(  
 caesar.decrypt('RptFpEoxFopoGxG,toBuox.CtExp,orwpEprGtEO'),  
 'Caesar is a title of imperial character.')

### 讨论与分析

凯撒密码是一种非常容易破解的加密方式，攻击者可以通过使用频率分析法，即在密文长度足够大的情况下，先分析出密文中每个字母出现的频率，然后将这一频率与正常情况下的该语言字母表中的所有字母的出现频率做比较。例如，在英文中字母e、t和a的出现频率都比较高。凯撒密码无法保证通信安全。

## DES加密

DES是一种是用56位密钥对64位分组进行加密的对称加密算法，DES的主要流程有：

* 密钥编排：从原始的56位密钥 k 中得到加密过程中用到的16个子密钥 ki
* 明文加密：加密过程用到了Feistel网络
* 密文解密：需要将16个子密钥逆向输入，初次之外与加密过程完全一致

### 密钥编排

密钥编排从原始的56位密钥中得到16个子密钥ki，每个子密钥都是48位的

DES的输入密钥通常是64位，其中每个第8位都作为前面7位的一个奇校验位，所以实际有效位只有56位

**PC-1** 首先通过PC-1置换表（具体查阅关手册）将k置换为56位并分成左右两部分C0和D0

**轮变换** 轮变换一共有16轮，以C0和D0作为初始输入，每轮中分别对Ci和Di做循环移位操作（具体操作查阅手册）

**PC-2** 每轮变换中，将Ci和Di合并并通过PC-2置换表（具体查阅手册）输出ki，即是该轮的输出子密钥

### 加密

加密结构如图，该结构又称Feistel网络，其中ki来自于上一阶段的密钥编排，IP和IP-1分别是两张64位的置换表（具体查阅手册），f函数将在下面介绍。

循环流程一共要执行16轮

### f函数

如图所示，E盒将32位扩充为48位（扩充表查阅手册）并与ki异或。然后将异或结果分成8份，分别送入S盒做置换位的操作，每个S盒的置换都是精心设计的并且各不相同（具体查阅手册）。通过S盒的位组装成一个32位的数，最终他通过P置换输出。

f函数在DES的安全性中扮演着非常重要的角色

### 解密

解密结构与加密结构完全一致，只不过密钥ki是逆向送入，在次不在累述。

### 密钥编排源码

// 密钥编排  
Des.prototype.\_generateKeys = **function** () {  
 **var** cd = **this**.key.getPermute(PC1)  
 **var** subKeys = **new** Array(16)  
 **for** (**var** i = 1; i <= 16; i++) {  
 **if** (i == 1 || i == 2 || i == 9 || i == 16) {  
 cd.shiftCD1()  
 }  
 **else** {  
 cd.shiftCD2()  
 }  
 subKeys[i] = cd.getPermute(PC2)  
 }  
 **return** subKeys  
}

### feistel网络源码

// feistel网络  
Des.prototype.\_feistel = **function** (subKeys, data64, isEncrypt) {  
 **var** tempData = data64.getPermute(IP)  
 **var** l = tempData.slice(0, 33)  
 **var** r = tempData.slice(32, 65)  
 **if** (isEncrypt) {  
 **for** (**var** i = 1; i <= 16; i++) {  
 desRound(l, r, subKeys[i])  
 }  
 }  
 **else** {  
 **for** (**var** i = 16; i >= 1; i--) {  
 desRound(l, r, subKeys[i]);  
 }  
 }  
  
 tempData.fill(r, 1, 1, 32)  
 tempData.fill(l, 33, 1, 32)  
 **return** tempData.getPermute(FP)  
}

### 加密实例

**var** des = **new** Des('abcdefgh')  
**var** des2 = **new** Des('ijklmnop')  
**var** text = '1234567890123456'  
**var** cipherText = des.encrypt(text)  
  
assert.ok(des.decrypt(cipherText) == text)  
assert.ok(des2.decrypt(cipherText) != text)

### 讨论与分析

如上所示，使用相同的key对明文进行加密及加密后的解密将输出相同的字符串，最后的\0是编码补齐之后多余的字符。如果key不同，则解密出来的字符串将不一样。

DES的密钥空间太小，即该算法很脆弱，易受蛮力攻击，目前已经有成本低廉的破解器专门用来破解DES。总之，56位的密钥大小已经不足以保证当今机密数据的安全。因此，单重DES只能用于要求短期安全性（比如几个小时）的应用或被加密数据价值较低的情况。然而，DES变体仍然很安全，尤其是3DES。

## MillerRabin素性测试

MillerRabin素性测试是针对费马素性测试可能得出“伪素数”也是素数的错误结论，此测试基于以下定理：

给定一个奇素数候选者p的分解 p – 1 = 2u × r，其中r是奇数。如果可以找到一个整数a，使得ar not ≡ 1 mod p 且 (ar)2j not ≡ p - 1 mod p，对所有的j ∈ [0, u-1] 都成立，则p是一个合数；否则，它可能是一个素数

### 源代码

// if true, p is a prime possibly  
**function** primeTest(p, u, r) {  
 **var** a = random.integer({  
 min: 2,  
 max: p - 2  
 })  
  
 **var** result = crt(a, r, p)  
 **if** (result != 1 && result != p - 1) {  
 **for** (**var** j = 1; j < u; j++) {  
 result = result \* result % p  
 **if** (result == 1) {  
 **return false** }  
 }  
  
 **if** (result != p - 1) {  
 **return false** }  
 }  
  
 **return true**}

*/\*\*  
 \* Miller-Rabin prime test  
 \** ***@returns*** *{Boolean} true if it's prime  
 \*/***var** mr = **function** (p) {  
 **if** (p == 2 || p == 3) {  
 **return true** }  
  
 **var** s = 100 // 执行100次测试  
 **var** result = decompose(p)  
 **for** (**var** i = 0; i < s; i++) {  
 **if** (!primeTest(p, result[0], result[1])) {  
 **return false** }  
 }  
  
 **return true**}

### 实例

assert.equal(mr(2), **true**)  
assert.equal(mr(7), **true**)  
assert.equal(mr(44), **false**)  
assert.equal(mr(341), **false**)

### 分析和讨论

合数p得到判断为素数的结论还是可能发生的。然而，如果使用若干个不同的随机基元素a进行测试，则误判的概率将大大降低。测试的次数由Miller-Rabin测试中的安全参数s给出。为了使合数被误检为素数的概率小于2-80，下表给出了所需要选择的不同a值的个数：

|  |  |
| --- | --- |
| p的位长度 | 安全参数s |
| 250 | 11 |
| 300 | 9 |
| 400 | 6 |
| 500 | 5 |
| 600 | 3 |

## RSA加密

RSA是目前使用最广泛的一种非对称加密方案，RSA底层的单向函数就是整数因式分解问题：两个大素数相乘在计算上是非常简单的，但是对其乘积结果进行因式分解却是非常困难的，也正因于此，RSA具有相当高的安全性。RSA技术有3个主要组成部分：密钥生成、加密、解密。

### 密钥生成

RSA技术始于密钥生成，流程见左图。通信甲方生成密钥对，公布公钥并保留私钥。通信乙方通过公钥加密，并将加密后的消息传送，甲方使用私钥对消息进行解密。攻击者因为无法通过公钥推断出私钥，因而消息的安全性得到保证。

RSA.prototype.generateKeys = **function** () {  
 **this**.\_pickPrimes() // 计算素数  
 **this**.n = **this**.p \* **this**.q  
 **this**.r = (**this**.p - 1) \* (**this**.q - 1)  
 **this**.e = **this**.\_pickE() // 选择公钥  
 **this**.d = **this**.\_calcD() // 计算私钥  
}

RSA.prototype.\_pickPrimes = **function** () {  
 **this**.p = **this**.\_pickPrime()  
 **this**.q = **this**.\_pickPrime(**this**.p)  
}

RSA.prototype.\_pickE = **function** () {  
 **while** (**true**) {  
 **var** e = **this**.random.integer({  
 min: 1,  
 max: **this**.r - 1  
 })  
 **if** (gcd(e, **this**.r) == 1) {  
 **break** }  
 }  
 **return** e  
}  
  
RSA.prototype.\_calcD = **function** () {  
 **var** result = eea(**this**.e, **this**.r)  
 **var** s = result[1]  
 **if** (s < 0) {  
 **var** n = Math.*ceil*(-s / **this**.r)  
 s += n \* **this**.r  
 } **else** {  
 **var** n = Math.*floor*(s / **this**.r)  
 s -= n \* **this**.r  
 }  
 **return** s  
}

### RSA加密

由于x只是唯一地取决于模数n的大小，所以一次RSA加密的位数不能超过n位的长度。另外，需要一种相对快速的方法计算xe mod n

RSA.prototype.encrypt = **function** (x) {  
 **return** crt(x, **this**.e, **this**.n)  
}

### RSA解密

解密和加密完全一样。

RSA.prototype.decrypt = **function** (y) {  
 **return** crt(y, **this**.d, **this**.n)  
}

### 实例

**var** rsa = **new** RSA  
**var** rsa2 = **new** RSA  
rsa.generateKeys()  
rsa2.generateKeys()  
**var** x = 9  
**var** y = rsa.encrypt(x)  
assert.equal(rsa.decrypt(y), x)  
assert.notEqual(rsa2.decrypt(y), x)

### 一些分析和讨论

如上所示，对明文进行加密后还原与原文完全一致，但若私钥不对，则解密后结果就不一样。出于实现简单起见这里的代码涉及的整数只在4位之内，但在实际的RSA算法中使用的都是上千位的整数，所以这势必涉及到大整数的加减乘的问题，但这不是RSA的核心因此不再累述。

## SHA1算法

SHA1是MD4家族中使用最广泛的消息摘要函数。它允许的最大消息长度是264位，产生的输出长度为160位。在哈希计算之前，此算法需要先对消息进行预处理。在实际计算期间，压缩函数将消息分成512位的分组进行处理。SHA1的主要流程有：填充、消息调度、哈希计算。

### 填充

在哈希计算之前，消息x必须先进行填充，直到其大小为512位的倍数

**function** pad(binary) {  
 **var** k = zeroPadSize(binary)  
 **var** bitArray = createArray(binary.length + 1 + k + 64, 0)  
 **var** lenBinary = bhd.numberToBinary(binary.length)  
  
 fillBitsToArray(binary, bitArray, 0) // 填充数组  
 bitArray[binary.length] = 1  
 fillBitsToArray(lenBinary, bitArray, binary.length + 1 + k + 64 - lenBinary.length)  
  
 **return** bitArray.join('')  
}

### 消息调度

对每个长度为512消息进行分组xi，共32组，每组大小为16，利用xi计算wi，计算w的意义在于接下来的哈希计算。

**function** calculateW(segment) {  
 **var** w = []  
 **for** (**var** i = 0; i < 16; i++) {  
 w.push(bhd.binaryToNumber(segment.slice(i \* 32, i \* 32 + 16)))  
 }  
 **for** (**var** i = 16; i < 80; i++) {  
 **var** next = (w[i - 16] ^ w[i - 14] ^ w[i - 8] ^ w[i - 3]).rotateLeft(1)  
 w.push(next)  
 }  
  
 **return** w  
}

### 哈希计算

哈希计算是一个80轮的迭代函数，即计算Hi = h(Hi-1)，其中H0是一个160位的常数，h函数每20轮都不一样，最终输出的H80就是最终的哈希值。如果还有下一个512位消息，则作为下一组消息的H0值。

**function** divide(binary, exec) {  
 **var** pack = [  
 0x67452301, // a  
 0xefcdab89, // b  
 0x98badcfe, // c  
 0x10325476, // d  
 0xc3d2e1f0 // e  
 ]  
  
 **for** (**var** stage = 0; stage < binary.length / 512; stage++) {  
 **var** segment = binary.slice(stage \* 512, stage \* 512 + 512)  
 pack = exec(segment, pack[0], pack[1], pack[2], pack[3], pack[4])  
 }  
  
 **return** pack  
}

### 实例

QUnit.test('sha1', **function** (assert) {  
 assert.equal(sha1('abc'), 'a9993e364706816aba3e25717850c26c9cd0d89d')  
 assert.equal(sha1('中文'), '7be2d2d20c106eee0836c9bc2b939890a78e8fb3')  
})

### 分析和讨论

实际中，字符串都按unicode进行编码，因此每个字符对应2个字节及16位长。SHA-1是专门为了良好的软件实现而设计的，它每轮中只需要使用32位的寄存器进行按位布尔操作。SHA1和其他MD4家族算法的一个缺点就是，他们都很难并行化实现，即在一轮中并行地执行多个布尔操作是非常困难的。