第 18 章　RSA算法——加密与签名

RSA算法（Rivest-Shamir-Adleman）是非对称公钥加密体系的开山鼻祖，经过几十年的发展，RSA算法在银行、军事、通信等领域得到了广泛的应用。RSA算法不仅用于数据加密，还可用于数字签名和身份验证。虽然现在椭圆加密算法（Elliptic Curves Cryptography，ECC）的应用也是如日中天，但是RSA算法仍然在非对称公钥加密体系中占有一席之地。

RSA算法是一种非常简洁的加密算法，远没有人们想象的那么复杂和神秘。RSA算法背后的数学理论就是大素数分解难题，其算法实现的核心是大整数的模幂运算。有了第17章介绍的大整数运算基础，实现RSA算法就易如反掌。

18.1 RSA算法的开胃菜

RSA算法的核心是大整数的模幂运算（Modular Power），模幂运算又称为模乘方运算。用数学表达式表示模幂运算就是：

$$C = A \wedge B \pmod{n}$$

我们已经实现了大数的乘方运算和取模运算，只需要先计算A的B次方，然后再对这个中间结果用除法求余数就可以得到结果。但是这个方案存在一个很大的问题，就是乘方和除法取余数的计算量都非常大，效率不高。除此之外，乘方计算的中间结果将是一个非常大的数，因为操作数不确定，所以也无法估计这个结果会多大，大数必须支持非常多的位才能保存这个中间结果。

根据RSA算法的性质可以看出，模幂运算的性能决定了RSA算法的性能。为了解决模幂运算效率的问题，现代数学界提出了很多解决方案。这些解决方案的基本思想都是先将模幂运算转换成模乘运算（Modular Multiplication），然后再用高效的算法处理模乘运算。本节要介绍的快速模幂和模乘算法都是实现高效RSA算法必不可少的组件，可以称为RSA算法的开胃菜。

18.1.1 将模幂运算转化为模乘运算

前面介绍过，对于模幂运算，如果用先求幂再取模的方式直接计算结果，在很多情况下是不能接受的。即便不考虑乘方计算的低效率，中间结果的存储也是个棘手的问题。以1024比特的大整数的乘方计算为例，其二次方的结果最大可能需要2048比特的存储空间，其1024次方的结果最大可能需要1兆比特（约128千字节）的存储空间。对于大数计算来说，1024作为指数简直就是个微不足道的值，考虑到指数也可

能是1024比特的大整数，存储中间结果最终需要的内存将超出计算机的能力。

模幂计算的解决思路是将其转化为模乘计算，避免直接求幂带来的存储和效率问题。模乘的数学表达式是：

$$C = A \times B \pmod{n}$$

那么，如何将模幂计算转化成模乘计算呢？在第17章介绍大整数计算时，提到过一种优化乘方运算的“平方-乘降幂法”，在计算大数乘方时可以有效地减少乘法计算的次数。在处理模幂运算时，同样可以利用这种思想将模幂运算转化成一些列模乘运算。将模幂运算转化成模乘运算，需要利用模运算的两个特性，即：

$$(a \times b)\%n = (a\%n \times b\%n)\%n$$
$$(a + b)\%n = (a\%n + b\%n)\%n$$

以计算$a^9 \% n$ 为例，可以分解为$(a^8 \% n \times a \% n)\%n$，$a^8 \% n$ 又可以分解为$(a^4 \% n \times a^4 \% n)\%n$，$a^4 \% n$ 又可以继续分解为$(a^2 \% n \times a^2 \% n)\%n$，$a^2 \% n$ 最终分解为$(a \% n \times a \% n)\%n$。利用这种思想，$a^9 \% n$ 的模幂运算就转换成5次模乘运算。这种转换的算法实现类似于CBigInt::Power()函数的实现，非常简单：

```
CBigInt ModularPower(const CBigInt& M, const CBigInt& E, const CBigInt& N)
{
    CBigInt k = 1;
    CBigInt n = M % N;

    for(__int64 i = 0; i < E.GetTotalBits(); i++)
    {
        if(E.TestBit(i))
```

```
        {
            k = (k * n) % N;
        }
        n = (n * n) % N;
    }

    return k;
}
```

CBigInt ModularPower()函数可以将$M^E \% N$ 的计算转化成平均3log(E)/2次模乘运算。这只是对模幂运算优化的第一步，接下来还要利用蒙哥马利模乘再对模乘运算进行优化，化解不必要的除法计算，进一步提高模幂计算的速度。

18.1.2 模乘运算与蒙哥马利算法

影响模乘运算速度的关键在于费时的取模运算（除法计算），如果在模乘运算中不用除法或尽量少用除法，将大大提高模幂运算的速度。数学家们研究了很多快速计算模乘的算法，蒙哥马利算法（Montgomery Reduction）就是其中的一种。大家可能对二战时期著名的英国陆军元帅蒙哥马利比较熟悉，但是此蒙哥马利非彼蒙哥马利。蒙哥马利算法是由彼得·蒙哥马利（Peter L. Montgomery）在1985年提出的一种大数模乘快速计算方法，又称为蒙哥马利约分算法。

蒙哥马利约分的基本思想就是选择一个适当的$R = 2^k$，k 满足条件：$n < 2^k$，将对 n 的取模运算转化为对R 的完全剩余系计算，对R 的除法计算可以转换为移位操作，从而避免了除法计算。因为$R > n$，且R 是2的整数幂，n 是素数，因此R 和n 互素，根据欧拉方程有解的条件可知，一定存在整数$0 < R^{-1} < n$ 和$0 < n' < R$，满足$RR^{-1} - nn' = 1$。此时可以将$A \times B \pmod n$的计算转化为计算$A' \times B' \times R^{-1}$

(mod n)，其中A' 和B' 分别是A 和B 对R 的剩余系表达，即：

$$A' = A \times R \pmod{n}$$

$$B' = B \times R \pmod{n}$$

$A' \times B' \times R^{-1} \pmod{n}$称为蒙哥马利模乘，它可以利用蒙哥马利约分算法高效地计算出来，蒙哥马利约分算法的计算方法如下：

```
function REDC(A', B', n', R, N)
    S = A' × B'
    m = (S mod R) × n' mod R
    t = (S + mN) / R
    if(t >= N)
        then return t - N
        else return t
```

根据以上描述的方法可以很容易写出蒙哥马利约分算法的实现代码：

```
CBigInt MontgomeryReduction(const CBigInt& X, const CBigInt& Y, const CBigInt& Np, const
CBigInt& N, const CBigInt& R)
{
    CBigInt S = X * Y;
    CBigInt m = (S * Np) % R;
    S = (S + m * N) / R;
    if(S >= N)
        return S - N;
    else
        return S;
}
```

因为R 是2的整数幂，只需要将对R 的取模和除法运算转化成移位运算，就可以得到真正高效的模乘算法。蒙哥马利约分算法需要为计算R 的剩余系而付出一些额外的开销，因此对于单次模乘计算，蒙哥马利约分算法并没有优势，但是对于像模幂运算这样需要多次反复计算模乘的情况，使用蒙哥马利约分算法可以极大地提高模幂计算的速度。

18.1.3 模幂算法

现在可以使用蒙哥马利约分算法改造18.1.1节给出的模幂算法。首先要利用同余方程计算出n'，然后再将模幂运算的底数M 转换到R 的剩余系，并用“平方-乘降幂法”逐次计算蒙哥马利模乘，最后将蒙哥马利模乘运算的结果转出R 的剩余系，得到最终的结果。在选择R 的时候，我们取k 值为32的整数倍，这样在计算蒙哥马利约分算法的移位处理的时候，对于我们的大整数CBigInt来说，一次移动一个unsigned int大数位，速度更快。最后给出使用蒙哥马利算法优化后的模幂算法实现：

```
CBigInt ModularPower(const CBigInt& M, const CBigInt& E, const CBigInt& N)
{
    CBigInt R = 1;
    R <<= N.m_nLength * 32;

    CBigInt Np = CongruenceEquation(R - N, R);
    //转换到R的剩余系
    CBigInt Mp = (M * R) % N;
    CBigInt D = R % N;

    for(__int64 i = 0; i < E.GetTotalBits(); i++)
    {
```

```
        if(E.TestBit(i))
        {
            D = MontgomeryReduction(D, Mp, Np, N, R);
        }
        Mp = MontgomeryReduction(Mp, Mp, Np, N, R);
    }
    //转出R的剩余系
    D = MontgomeryReduction(D, 1, Np, N, R);

    return D;
}
```

18.1.4 素数检验与米勒-拉宾算法

素数在数论中是一个很大的分支，很多数学定理都和素数有关，有人甚至将其独立出来称为素论，可见素数对于数学的重要性。RSA算法在生成密钥对时需要两个随机大素数，并将它们的乘积作为公共模数n，这就需要有对应的素数生成算法。素数生成没有什么特殊方法，就是生成随机大数作为疑似素数，然后用素数检验方法检验是否是“真”的素数，如果是就返回结果，如果不是就继续上述过程。由此可见，要生成一个素数，必须要有一套判断素数的方法。1000以内的小素数可以用素数的定义直接判断，大素数则要采用特定的算法进行素性测试。要进行素性测试，先来了解一下费马小定理，定义如下：

设p 是素数，a 是任意整数，且a !≡ 0(mod p)，则$a^{(p-1)} \equiv 1 \pmod{p}$

一般来说，可以利用费马小定理直接进行素数测试，这就是费马测试（Fermat）。费马测试实际上是利用费马小定理的逆定理进行反向证明，不幸的是，费马小定理只是素数检验的必要条件，其充分条件，也就是费马小定理的逆定理并不成立，因

为存在卡米歇尔数[1]（Carmichael）。费马测试是个概率测试，并没有得到广泛的应用，目前判断大素数（特别是超过512位的大素数）普遍采用的方法是米勒-拉宾（Miller-Rabin）算法。

> [1]卡米歇尔数：能满足费马小定理，但是又不是素数的数。

1975年，卡内基梅隆大学计算机系的米勒（Gary Lee Miller）教授首先提出了基于广义黎曼猜想[2]的确定性算法，由于广义黎曼猜想并没有被证明，直接引用广义黎曼猜想就存在理论上的缺陷。所以以色列耶路撒冷希伯来大学的拉宾（Michael O. Rabin）教授对其进行了改进，提出了不依赖于该假设的随机化算法，这就是米勒-拉宾算法的由来。米勒-拉宾素性检验算法利用随机化算法判断一个数是合数还是可能是素数，请注意，这里用词是“可能”，因为米勒-拉宾算法也是一种判断素性的概率算法。虽然是概率算法，如果加上限制条件，米勒-拉宾算法也可以作为一种确定性算法。

> [2]德国数学家黎曼在1858年写了一篇只有8页长的关于素数分布的论文，提出了著名的广义黎曼猜想（Riemanns Hypothesis）。这个猜想是指黎曼ζ函数（ζ音：齐塔）：ζ(s) = Σ1/n^s (n从1到无穷大)的非平凡零点都在Re(s) = 1/2的直线上（也就是说所有非平凡零点的实部都是1/2）。看似简单的问题实际上并不容易，求多项式的零点，特别是求代数方程的复根都不是简单的问题。数学家把复平面上Re(s) = 1/2的直线称为临界线（Critical Line）。1914年，英国数学家哈代（G.H. Hardy）首先证明这条临界线上有无穷个零点。三位荷兰数学家利用计算机对最初的两亿个ζ函数的零点进行检验，目前已经证明了2/5的复零点都在这条直线上，并且在这条直线之外至今还没有发现其他复零点。这初步证明黎曼的假设是对的，但是这个检验的过程还在继续，因此，广义黎曼猜想是对还是错还没有定论。

用米勒-拉宾素性检验法检验算法判断n 是否是素数，首先将n-1分解为$m*2^k$，然后在[1, n-1]区间上随机选一个整数a，对于[0, k-1]区间中的每一个值r，检测：

$$a^m \pmod n \neq 1 \text{ 和 } a^{m\times 2^r} \pmod n \neq -1$$

两个条件是否同时成立，如果两个条件同时成立，则n 是一个合数，否则，n 有75%的概率是一个素数。有此可知，做一次检验，即便n 不满足两个成为合数的条件，仍然有1/4的可能性是合数。但是，如果用足够多的随机数a 对其进行多次检验，则可以降低n 是合数的可能性。假设检验次数是t，则n 是合数的可能性是$P(c) = 1/4^t$。如果进行5次检验都符合上述情况，n 是合数的可能性就降到0.098%，即n有99.9%的可能是素数。如果进行10次检验，则n 是素数的可能性就达到99.9999%。用米勒-拉宾算法检验素数，一般至少需要检验5次，严格的场合可能需要检验更多的次数，比如50次。

MillerRabinHelper()函数是一次米勒-拉宾检验的算法实现，其中m 和k 两个参数需要实现计算出来，计算的方法将在MillerRabin()函数中给出。根据检验规则的定义，需要计算a 与m 关于n 的模幂，以及a 与$m \times 2^r$关于n 的模幂，根据幂运算关系的特点，a 的$m \times 2^r$次方与a 的m 次方存在平方的关系。如果令$b = a^m$ ，则a 的m×2次方就是b^2，则a 的m×4次方就是b^4，以此类推。如果采用b^2的累积，可以减少很多计算量，因此，一般算法实现都会采用b^2的累积代替计算a 的$m \times 2^r$次方，MillerRabinHelper()函数也不例外。

```
bool MillerRabinHelper(const CBigInt& a, const CBigInt& m, int k, const CBigInt& n)
{
    CBigInt b = ModularPower(a, m, n);

    if(b != 1)
    {
        for(int r = 0; r < k; r++)
        {
            if(b != (n - 1)) //b != 1 && b != n-1，满足合数条件
            {
```

```
                return false;
            }
            b = ModularPower(b, 2, n);
        }
    }

    return true;
}
```

MillerRabinHelper()函数返回false表示n 不是素数，返回true表示n 有75%的可能是一个素数。当MillerRabinHelper()函数返回true时，需要使用新的随机数a 对n 继续检验，直到满足检验次数条件。MillerRabin()函数首先根据n 计算出m 和k，然后多次调用MillerRabinHelper()函数进行检验。产生随机数a 的时候，总是选一个32位以内的小随机数，目的是减少检验的计算量。a 的二进制位数总是比n 少一位，且最大不超过32位，保证a 总是小于或等于n-1。

```
int MillerRabin(const CBigInt& n)
{
    CBigInt m = n - 1;
    int k = 0;

    //根据n-1 = m*2^k，计算m和k
    while(!m.TestBit(0))
    {
        m >>= 1; //m = m / 2;
        k++;
    }

    CBigInt a,b;
```

```
    for(int i = 0; i< M_R_TEST_COUNT; i++)
    {
        __int64 nbits = n.GetTotalBits();
        // 1 <= a <= n - 1
        a = CBigInt::GenRandomInteger((nbits > 32) ? 32 : nbits - 1) + 1;
        if(!MillerRabinHelper(a, m, k, n))
        {
            return 0;//测试失败，明确是合数
        }
    }
    return 1;
}
```

前面介绍过，米勒—拉宾检验算法是一个概率方法，但是如果加上限制条件，米勒-拉宾算法也可以作为一种确定性算法使用。限制条件就是数的范围。根据数学家的证明，只要用2和3作为随机数进行两次检验，就可以100%正确地检验小于1373653的所有素数。再比如，只要用2、3、5、7、11作为随机数进行5次检验，就可以100%正确地检验小于2152302898747的所有素数。这些经过证明的经验值都是100%正确的，但是不在上述范围中的其他大素数的判断，目前还只能是概率结果，也就是说，即使能通过检验，也要打上“伪素数”的标签。

18.2 RSA算法原理

传统的加密模式一般是信息发送者用特定的密钥对信息加密（加密和解密算法都是公开的），然后将密文传递给接收者，同时还要将密钥告诉接收者，这样接收者就可以用密钥对密文进行解密。这种方式的隐患在于密钥的传递，密钥在传递过程中有可能被截取，此外，密钥分发出去以后就很难控制分发的范围，一旦失控，发出去的加密信息就等于是明文了。

1976年，维特菲尔德·迪菲（Whitfield Diffie）和马丁·赫尔曼（Martin Hellman）在一篇革命性文章“密码学的新方向”（New Directions in Cryptography）一文中提出了一种使用非对称密钥的密码学新方法，可以在不用传递密钥的情况下完成信息的加密和解密。这就是现代非对称公钥体系的基础，在这种体系中，发送者要给接收者传递密文，首先要得到接收者对外发布的公钥，然后用该公钥加密信息，并将加密的信息发送给接受者。接收者受到密文后用自己的私钥对信息解密，在这个过程中，不需要密钥传递，接受者的私钥自己保管，不对外公开。

这种思想也对密码学家提出了新的挑战，也鼓舞了很多数学家寻找一种满足非对称公钥体系的加密算法。第二年，美国麻省理工学院的罗纳德·李维斯特（Ron Rivest）、阿迪·萨莫尔（Adi Shamir）和伦纳德·阿德曼（Leonard Adleman）三位研究员在“实现数字签名和公钥密码体制的一种方法”一文中首次提出了一种非对称公钥加密算法，因为三个人的姓氏首字符分别是R、S、A，这种算法就被命名为RSA算法。经过四十多年的发展，RSA算法已经成为现代非对称公钥体系中最基本也是目前应用最广泛最有影响力的公钥加密算法。

18.2.1 RSA算法的数学理论

在研究RSA算法的数学原理之前，先来介绍几个数学概念。首先是“同余”，假定三个整数a、b和n(n ≠0)，如果a 和b 的差是n 的整数倍，则称a 在模n 时与b 同余，记做a≡b(mod n)。可以将同余简单理解为等式：a-b=kn，k 为任意整数。然后是“欧拉函数”，欧拉函数φ(n)定义为所有小于或等于n，且与n 互素的正整数的个数，φ(n)的值又被称为n的欧拉数。以8为例，1、3、5、7都与8互素，所以就有φ(8)=4。当n 是素数时，φ(n)=n-1，因为所有比n 小的数都与它互素。欧拉函数还有一个特性，当n 可以分解为两个互素的数的乘积时n 的欧拉函数就是两个因子的欧拉函数的乘积，即φ(n)= φ(pq)=φ(p)φ(q)=(p-1)(q-1)。最后是“乘法逆元”，若ab≡1(mod n)，则称b 为a 在模n 的乘法逆元，b 可以表示为a^{-1}。第17章已经介绍过，可以使用欧拉算法求解乘法逆元，相关算法的实现在第17章已经给出。

RSA算法基于一个十分简单的数论事实：将两个大素数相乘十分容易，但那时想要对其乘积进行因式分解却极其困难，因此可以将乘积公开作为加密密钥的一部分。由此可知，密钥的生成是RSA算法的核心，先来看看RSA密钥对的生成过程。

1．任意选择两个大素数，p 和q，计算出n=p×q，n 又被称为RSA算法的公共模数。

2．计算n 的欧拉数φ(n)=(p-1)(q-1)。

3．随机选择加密密钥指数，从[0,φ(n) -1]中选择一个与φ(n)互质的数e 作为公开的加密指数。

4．求解与e 对应的解密指数d，d 和e 满足条件：(d × e)≡1 mod φ(n)。因为e

和φ(n)互素，因此可以利用扩展欧几里得算法求解同余方程，得到唯一整数解d。

5. 销毁p 和q，妥善保存私有密钥SK=(d，n)，将公开密钥PK=(e，n)分发给希望给你发送信息的人。

由以上过程可知，在p 和q 不可知的情况下，要得到私有密钥的解密指数d，必须知道φ(n)，而φ(n)必须通过第2步给出的方法计算，也就是说，必须要分解公共模数n，重新得到p 和q。对n 的分解是个数学难题，目前没有有效的方法可以快速分解n，n 越大越难分解，这就是RSA算法的数学原理。以目前计算机的处理能力，当n 大到一定的程度，可以认为是不可分解的，这也是RSA算法安全性的基本保证。

18.2.2 加密和解密算法

RSA算法的加密和解密其实就是模幂运算，这也是我为什么说RSA算法简单的原因，因为18.1节已经给出了加密和解密的算法实现代码，就是ModularPower()函数。假设M 是明文，C 是密文，加密的过程就是：

$$C = M^e \pmod{n}$$

解密的过程就是：

$$M = C^d \pmod{n}$$

现在举个经典的密码学示例来解释一下RSA加密和解密的过程。爱丽丝希望鲍勃给她发送的信息进行加密，她首先选择两个素数p=11和q=13，计算它们的乘积，得到公共模数n=143，同时计算出n 的欧拉数φ(n)=120。接下来爱丽丝需要选择一个小于119（φ(n)-1=119），且与φ(n)互素的数作为加密指数e，爱丽丝选择e=7。然后爱丽丝需要求解同余方程7d≡1 mod 120，得到d=103。最后，爱丽丝销毁p 和q，

将(7,143)作为公开密钥发送给鲍勃，将(103,143)作为私钥自己保存。鲍勃需要发送信息M=85给爱丽丝，他先用爱丽丝的公钥对M 进行加密，得到密文$C = 85^7 \pmod{143} = 123$，然后将密文C=123发送给爱丽丝，爱丽丝得到C 后，用私钥对C 进行解密，得到明文$M=123^{103} \pmod{143}=85$。

以上就是整个RSA加密和解密的过程，其核心就是模幂运算，使用的过程很简单，但是简单并不意味着不安全，接下来要介绍一下RSA算法的安全性。

18.2.3 RSA算法的安全性

人们在提到RSA加密的时候，都会附带一个很重要的参数，就是RSA密钥长度，比如1024比特的RSA密钥，2048比特的RSA密钥等。这里的1024和2048实际上是RSA密钥中公共模数n 的二进制位长度，n 越大就越难分解，这就是通常人们会认为1024比特的RSA密钥要比512比特的RSA密钥更安全的原因。但是从数学上看这个问题，对n 的分解并没有被证明是NP问题，也就是说，增加n 的长度就能提高安全性还没有被用数学的方法证明（当然，也没有被证伪）。当然，RSA加密的安全性是由很多因素决定的，比如组成n 的两个随机素数p 和q 的选择就很有讲究。p 和q 必须是随机生成的强素数，绝对不能用别人用过的素数，或者从某个素数表中选择p 和q。p 和q 的差值应该尽量大，增加分解n 计算的难度。

加密指数e 的选择也很重要，e 越大计算量就越大，为了加快加密计算的速度，RSA算法对公钥e 选择通常是3、5、17、257或65537。X.509证书体系建议使用65537，PEM建议使用3，PKCS#1建议使用3或65537。但是使用太小的e 会引入小指数攻击问题，在原始数据中填充随机数值，使得$m^e \pmod{n} \neq m^e$，可以有效地抵抗小指数攻击。因此使用RSA加密数据通常都会指定随机值填充模式，单纯的直接用模幂算法加密数据是不安全的。

除此之外，还有针对明文破解的选择密文攻击方式。攻击者知道了A的公开密钥(e，n)，同时截获了用A的公钥加密的信息$Y = X^e \pmod n$。攻击者首先选择一个r（r < n)，计算$Y_1=r^e \pmod n$，这意味着用A的私钥对Y_1解密可得到r，即$r = Y_1^d \pmod n$。接下来，攻击者计算$Y_2 = Y \times Y_1 \pmod n$，求解r的模n乘法逆元$t = r^{-1} \pmod n$。因为$r = Y_1^d \pmod n$，所以$r = Y_1^{-d} \pmod n$。现在，攻击者以验证身份的名义将$Y_2$发给A，请A对消息$Y_2$签名，于是得到$S = Y_2^d \pmod n$。最后，攻击者做以下计算：

$$t \times S = (Y_1^{-d} Y_2^d) \pmod n$$

将$Y_2 = Y \times Y_2 \pmod n$代入上式得到：

$$t \times S = (Y_1^{-d} Y_2^d) \pmod n = (Y_1^{-d} Y^d Y_1^d) \pmod n = Y^d \pmod n = X$$

最终攻击者在不知道私有密钥d的情况下得到了明文X。

以上选择密文攻击过程关键的一步就是攻击者需要骗取A对Y_2进行签名，只要用户A不对来历不明的数据直接签名，就可以阻断这种攻击。实际上，RSA的签名是不对数据直接计算的，而是像加密一样要填充一些随机数值，具体的签名算法请看18.4节的介绍。

18.3 数据块分组加密

前面我们讨论了RSA加密的数学原理和加密解密过程的算法实现，但是所给出的算法实现还都是在数学领域的大数计算，怎么将其应用到现实生活领域呢？加密和解密领域最典型的问题就是对数据分组加密，RSA同样支持对数据分组加密。和DES、AES这样的分组加密算法不同，RSA算法所支持的每个数据分组的大小不仅与RSA密钥长度有关，还和数据分组的填充模式（padding scheme）有关。填充模式是和RSA算法安全性相关的内容，不同的填充方式需要插入原始数据中的padding数据量不一样，因此会影响数据分组中有效载荷的大小。

RSA算法有多种填充模式，常用的填充模式有PKCS #1 1.5和OAEP两种，我们将在后面介绍这两种填充模式，这里只给出使用两种填充模式对原始数据载荷大小的影响。RSA加密算法每个数据分组的有效载荷大小与密钥长度和填充模式的关系如表18-1所示。

表18-1 RSA填充模式与数据分组大小关系表

填充模式	RSA密钥长度（位）	输入分组有效载荷（字节）	输出分组长度（字节）
PKCS #1 1.5	768	85	96
PKCS #1 1.5	1024	117	128
PKCS #1 1.5	2048	245	256
OAEP	768	54	96

OAEP	1024	86	128
OAEP	2048	214	256

18.3.1 字节流与大整数的转换

被加密的数据可以理解为字节流，对数据加密时，除了按照表18-1给出的输入分组有效载荷大小对字节流进行分组外，还要将分组后的数据转换成大整数才能进行RSA加密运算。完成加密运算后，还要将计算得到的大整数转换成字节流数据，这样才能进行存入文件或通过网络传送。解密的过程与之类似，都需要一套大整数与字节流互相转换的方法，这样RSA算法才具有实用性。

其实将加密数据转换成大数对象的方法非常朴实无华，如果将大数也看成按照顺序在内存中存放的字节流，这种转换就一目了然。只要逐字节将加密数据转换成CBigInt类的大数位数组（m_ulValue），并正确设置大数位的位数（m_nLength），即可完成加密数据转换成大数对象的过程。反之亦然，只要将CBigInt类的大数位数组中的数据逐字节转换到指定的缓冲区，即完成大数对象转换成字节流数据的过程。转换过程唯一需要注意的是对数据中0的特殊处理，对于CBigInt大数对象来说，最高的大数位不能是0，如果是0，则要调整m_nLength。

18.3.2 PCKS与OAEP加密填充模式

18.2.3节介绍RSA算法的安全性时，提到为了对抗小指数攻击，需要在原始数据中添加随机填充数据以提高RSA的安全性。常用的填充模式有PKCS #1 1.5和OAEP两种，PKCS的全称是公钥加密标准（Public-Key Cryptography Standards），是

RSA实验室发布的一个标准。OAEP的全称是最优非对称加密填充（Optimal Asymmetric Encryption Padding），是一个比PKCS #1 1.5新的填充标准，被PKCS #1 2.0接受为新的填充标准。理论上OAEP有比PKCS #1 1.5更好的安全性，但是从兼容性来说，PKCS #1 1.5具有更好的兼容性。

从表18-1可以看出，PKCS #1 1.5需要额外的11字节进行随机填充，而OAEP需要额外的42字节用于随机数据填充，而且OAEP的填充方式更具随机化特性，由此可知，OAEP填充方式对原始数据造成的混乱程度（熵值）比较大，具有更好的安全性。这里的更好是相对的，并不是说PKCS #1 1.5填充方式不安全。当你在两个不同的系统之间传递RSA加密的信息时，填充模式的兼容性就是需要特别注意的地方，PKCS #1 1.5因为发布得早，应用更广泛一些，兼容性就更好一些。

前面提到PKCS #1 1.5需要额外的11字节进行随机数据填充，实际上是不完全正确的。只有当数据的有效载荷足够多的时候，PKCS #1 1.5填充的长度才被压缩为11字节，当有效载荷不足的时候，实际需要填充的数据会超过11字节。总的来说，PKCS #1 1.5填充可以分为四部分，分别是一字节前导0，一字节标志T 和k-|M|-3字节的随机数据和一字节的截断符号0。k 是RSA密钥公共模数的字节长度，|M|表示实际载荷数据长度，|M|通常要满足|M|≤k-11，当|M|=k-11的时候，PKCS #1 1.5要求的最小填充长度是11字节时，其结构如图18-1所示：

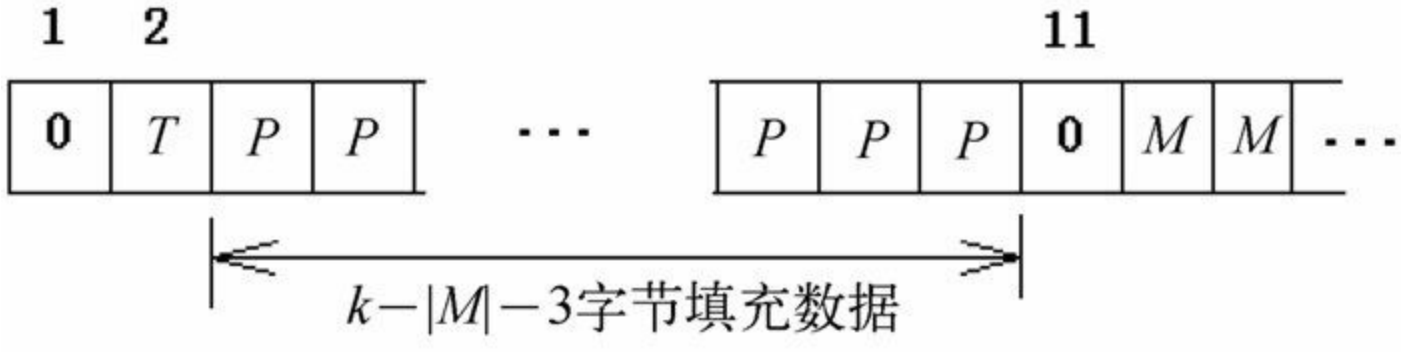

图 18-1 PKCS #1 1.5填充模式

其中标志字节T 表示拼凑填充数据的方法，如果T 是0，表示填充数据P 全部是0；如果T 是1，表示填充数据P 全部是0xFF；如果T 是2，表示填充数据是k-|M|-3个1-0xFF之间的随机数据。前导位设为0是为了保证转换后得到的大数是正数，这k-|M|字节的填充数据放在实际的数据载荷之前，构成完整的加密数据分组，然后转换成大整数进行加密计算。对加密过的数据解密时，解密计算后得到的数据也是包含填充数据的，要从中提取出有效数据载荷，其方法就是从前导0和标志T 开始向后搜索，直到找到截断0标志为止（随机填充数据不会是0），在这之间的都是填充数据，剩下的就是有效数据载荷。

相对于PKCS #1 1.5来说，OAEP稍显复杂一点。OAEP是Mihir Bellare和Philip Rogaway两位密码学家提出的一种加密方案，后来被PKCS #1 2.0接受为标准，因此也被称为PKCS #1 2.0填充方案。OAEP模式中明文载荷的长度|M|要满足|M|≤k-2hLen-2，其中hLen是OAEP模式所选择的哈希函数的输出长度。OAEP模式的哈希函数和掩码生成函数都不是固定的，可以通过参数化配置和选择，如果选择SHA（Secure Hash Algorithm），输出长度是20字节，则明文载荷不能超过k-42字节。OAEP模式需要指定一个与明文有关联的标签L，如果没有指定，则默认L 是空。OAEP的加密过程如下。

1. 计算标签L 的哈希输出，得到一个长度为hLen的字节串IHASH=HASH(L)；

2. 生成一字节串PS，内容是0，长度为k-|M|-2hLen-2字节。当|M|=k-2hLen-2时，PS长度有可能是0；

3. 连接IHASH，PS，一字节的0x01和明文M，得到一个k-hLen-1字节长度的字节流

DB= IHash|PS|0x01|M；

4．生成一个长度为hLen的随机字节串seed，并使用掩码生成函数将其转换为长度k-hLen-1字节的掩码DBmask，DBmask= MGF(seed, k-hLen-1)；

5．用DBmask与DB做掩码计算得到mDB=DB⊕DBmask；

6．用掩码生成函数将mDB转换为长度为hLen字节的掩码seedMask，将其与随机字节串seed做掩码计算，得到Mseed。即seedMask=MGF(mDB, hLen)，mSEED= seed⊕seedMask；

7．将一字节的0x00、mSEED和mDB拼在一起，组成长度为k的加密数据分组EM，即EM=0x00|mSEED|mDB，然后将EM转换为大整数即可进行RSA加密计算。

OAEP解密的过程与加密过程相反，首先要将加密数据转换成大整数，进行RSA解密计算，然后将解密后的大整数转换成字节流，此时就得到加密过程第7步拼接的数据分组EM，然后再按照以下过程分解出原始加密数据M。

1．计算标签L 的哈希输出，得到一个长度为hLen的字节串IHASH=HASH(L)；

2．从EM中分解出mSEED和mDB，用掩码生成函数将mDB转换为长度为hLen字节的掩码seedMask，即seedMask=MGF(mDB, hLen)；

3．计算seed=mSEED⊕seedMask，然后用掩码生成函数将其转换成长度为k-hLen-1字节的掩码DBmask，DBmask= MGF(seed, k-hLen-1)；

4．根据DBmask和mDB计算得到DB，DB=mDB⊕DBmask，如果解密计算过程没有错误，DB的内容应该和加密过程第3步得到的DB是一样的；

5．分解DB，首先匹配一下IHash与第1步算出来的是否一致，如果不一致说明解密错误。如果一致，则匹配一连串0和一个0x01，如果能匹配，则剩下的就是原始明文M，如果不能匹配，说明解密错误。

18.3.3　数据加密算法实现

分组数据的加密过程，实际上是一个和填充模式捆绑在一起对数据进行处理的过程。现在就以简单的PKCS #1 1.5填充模式为例，说明一下RSA分组数据加密过程。这个过程正如Rsa_Pkcs15_Encrypt_Block()函数所展示的那样，首先是填充前导0，然后是T 标识，我们选择T=2，这也是PKCS #1 1.5推荐的模式。接下来是随机字节串，长度由k-|M|-3计算得到，GeneratePkcsPad()函数产生长度为pad_len的随机字节串。在拼接原始数据之前，还要再添加一个截断标识0。完成填充之后，将数据转化成大整数，用模幂运算进行加密，得到密文大数c，将c 转换成字节串，即可作为加密后的数据进行存储或分发。

```
int Rsa_Pkcs15_Encrypt_Block(CBigInt& e, CBigInt& n, int kbits,
    void *pSrcBlock, int srcSize, CBigInt& c)
{
    int k = kbits / 8;
    int pad_len = k - srcSize - 3;

    char *padBlock = new char[k];
    if(padBlock == NULL)
        return -1;

    padBlock[0] = 0x00;
    padBlock[1] = 0x02; //填充随机数
    GeneratePkcsPad(2, padBlock + 2, pad_len);
```

```
    padBlock[pad_len + 2] = 0x00;
    memcpy(padBlock + k - srcSize, pSrcBlock, srcSize);
    CBigInt em;
    em.GetFromData(padBlock, k);//OS2IP
    c = ModularPower(em, e, n);

    delete[] padBlock;

    return k;
}
```

18.3.4 数据解密算法实现

分组数据的解密过程也是和填充模式捆绑在一起的处理过程。首先将得到的密文转换成大整数c，然后对c 进行模幂运算解密，得到密文em，最后将em 转换成字节串，并分离出随机填充信息，得到原始明文信息。Rsa_Pkcs15_Decrypt_Block()函数展示的就是分组解密的实现过程，其中分离填充信息需要用到截断标识，就是插入到随机填充字节串和原始数据之间的那个0。PKCS #1 1.5要求填充的随机字节都是1~0xFF的数值，因此，从填充标识T 开始搜索，遇到的第一个0肯定就是截断标识，其后跟的就是原始数据。

```
int Rsa_Pkcs15_Decrypt_Block(CBigInt& d, CBigInt& n, int kbits,
    CBigInt& c, void *pDecBlock, int blockSize)
{
    char *padBlock = new char[kbits / 8];
    if(padBlock == NULL)
        return -1;

    CBigInt em = ModularPower(c, d, n);
```

```
    int dataSize = em.PutToData(padBlock, kbits / 8);
    int pad_len = 2;
    for(int i = 2; i < dataSize; i++)
    {
        pad_len++;
        if(padBlock[i] == 0)
            break;
    }
    memcpy(pDecBlock, padBlock + pad_len, dataSize - pad_len);
    delete[] padBlock;

    return dataSize - pad_len;
}
```

18.4 RSA签名与身份验证

RSA密钥中的加密指数e 和解密指数d 是关于模φ(n)的乘法逆元，这个关系使得RSA的加密和解密过程具有一个很有意思的特点。这个特点就是用e 加密的数据可以用d 解密，反过来，用d 加密的数据也可以用e 解密。利用这个特点，RSA算法还可以用来做数字签名和身份验证。数字签名是只有信息发送者才能产生、别人无法伪造的一段信息，这段信息还可以用来验证发送者所发送信息的真实性。可以这样理解，数字签名就是信息发送者用自己的私钥对一段公开信息加密后得到密文信息，因此它具有两个特征：

- 任何人都可以利用发送者的公钥验证签名的有效性（对其解密并比较）
- 签名具有不可伪造性和不可否认性（因为只有发送者有与公钥对应的私钥）

签名的验证就是利用发送者的公钥对加密信息解密，然后比较解密后的信息是否是原始信息。数字签名的一般过程是：用户A用选择的哈希算法计算出文件M的HASH值，然后用自己的私钥对这个HASH值进行加密，这个过程就是“签名”。现在A把文件M（不加密）和加密后的HASH值发给B，B于是用A的公钥对HASH值解密，然后再计算文件M的HASH值，比较文件的HASH值是否和A发来的HASH值一样，如果一样则说明文件确实是A发来的，并且文件没有被修改。否则就说明文件不是A发出的，或者文件在传递过程中被篡改了。由此可知，数字签名除了证明文件M是A发出的，还可以验证文件M是否被其他人（包括接受者）篡改或伪造。

身份验证的过程和签名的过程类似，当B需要验证A的身份时，就将一段信息发给A，请A对其进行签名（加密）。得到A的签名后，B使用A的公钥对签名解密，验证是否

和自己发给A的信息一致，以此验证A身份。其他人没有A的私钥，无法伪造A的签名。

和RSA的加密一样，使用RSA签名一样需要面对各种攻击，因此，不要随便对某一个人发过来的东西进行签名（有潜在危险）。如果必须要这么做（比如为了验证身份），最好先用哈希算法计算出信息的HASH值，然后对HASH值进行签名。

18.4.1 RSASSA-PKCS与RSASSA-PSS签名填充模式

RSA的签名和身份验证过程，面临着和RSA加密过程一样的攻击方法，因此RSA实验室对签名算法也制定了和加密过程一样的随机填充模式标准——带填充的签名算法。PKCS制定了两种带填充的签名算法，分别是RSASSA-PSS和RSSSA-PKCS #1 1.5。尽管从理论上讲RSASSA-PSS比RSSSA-PKCS #1 1.5有更好的健壮性，但是目前还没有发现针对RSSSA-PKCS #1 1.5的有效的攻击手段。RSSSA-PKCS #1 1.5的签名算法已经在很多系统上得到了广泛的应用，具有很好的兼容性，不过PKCS标准建议新的应用系统应该能够平滑地过度到RSASSA-PSS算法。

RSSSA-PKCS #1 1.5签名算法的填充方式和PKCS #1 1.5加密算法的填充方式类似，主要由以下六部分组成：

$$EM = 0x00 \mid 0x01 \mid PS \mid 0x00 \mid T \mid H$$

首先是前导0，T 标识固定是1，也就是说，PS使用随机长度的0xFF填充，跟在截断标识0后的T是一个与哈希算法相关的字节串，其内容与哈希算法有关，但是每种哈希算法对应的T 的内容和长度是固定的。哈希算法与T 的关系如表18-2所示。

表18-2 PKCS #1 1.5签名算法中哈希算法与T填充内容的关系

Hash	T
MD5	30 20 30 0c 06 08 2a 86 48 86 f7 0d 02 05 05 00 04 10
SHA-1	30 21 30 09 06 05 2b 0e 03 02 1a 05 00 04 14
SHA-224	30 2d 30 0d 06 09 60 86 48 01 65 03 04 02 04 05 00 04 1c
SHA-256	30 31 30 0d 06 09 60 86 48 01 65 03 04 02 01 05 00 04 20
SHA-384	30 41 30 0d 06 09 60 86 48 01 65 03 04 02 02 05 00 04 30
SHA-512	30 51 30 0d 06 09 60 86 48 01 65 03 04 02 03 05 00 04 40

H 是明文M 的哈希值，填充数据PS的长度等于k-|T |-|H |-3，其中k 是RSA密钥公共模数n 的字节长度，|T |是填充T 的长度，|H |是明文M 的哈希值的长度。完成填充后得到EM，将其转换成大整数，用私钥进行加密，就得到数字签名。

RSSSA-PKCS #1 1.5的签名验证过程与解密过程类似，首先将签名数据转换成大整数，然后用公钥解密，就得到签名之前的填充状态EM。分解EM，得到签名中的原始信息的哈希值H，然后将这个H 与原始明文计算出来的哈希值比较，如果一致则签名验证成功，如果不一致，说明这是一个无效的签名，或者是伪造的签名。

RSSSA-PSS签名填充模式是一种新的签名填充，在RSSSA-PKCS #1 v2.1中被接受为PKCS的标准。RSSSA-PSS源于Mihir Bellare和Philip rogaway发明的概率填充方案（Probabilistic Signature Scheme），将其应用于RSA加密体系，就是RSSSA-PSS。RSSSA-PSS签名算法的输入参数是明文M和签名体最大比特长度

emBits，emBits的最小值不能小于8hLen+8sLen+9，其签名过程如下。

1．计算明文M 的哈希值mHash=HASH(M)，长度为hLen，生成一个随机字节串salt，长度为sLen；

2．拼接MP=0 0 0 0 0 0 0 0 |mHash|salt，计算MP的哈希值H=HASH(MP)，H长度是hLen；

3．生成由0字节组成的字节串PS，长度为emLen-hLen-sLen-2，$\mathrm{emLen} = \lceil emBits/8 \rceil$。PS的长度也可以是0；

4．令DB = PS|0x01|salt，DB是一个长度为emLen-hLen-1的字节串；

5．用掩码生成函数将MP的哈希值H转换成长度为emLen-hLen-1的掩码，DBmask=MGF(H, emLen-hLen-1)；

6．计算mDB = DB⊕DBMask，将mDB的左边最高有效位的8emLen-emBits个比特置为0；

7．拼接mDB、MP的哈希值H以及一个字节的固定值0xBC，得到EM=mDB|H |0xBC。将EM转换成大整数，用私钥加密即可得到RSSSA-PSS填充的数字签名。

RSSSA-PSS签名的验证过程首先要将签名体转换成大整数，用公钥解密得到EM，然后按照以下步骤验证EM。

1．从左到右对EM进行分解，前emLen-hLen-1个字节是mDB，接下来是hLen字节的H，最右边是一个字节的固定值0xBC，如果最右边一个字节不是0xBC，则输出“验证失败”，并停止；

2．如果mDB左边最高8emLen-emBits个比特不是0，则输出“验证失败”，并停止；

3．用掩码生成函数将H 转化成emLen-hLen-1字节的掩码DBmask= MGF(H, emLen-hLen-1)，并计算出DB=mDB⊕DBMask；

4．将DB的左边最高有效位的8emLen-emBits个比特置为0，并判断emLen-hLen-sLen-1位置的一个字节是否是0x01，如果不满足这两个条件，则输出“验证失败”，并停止；

5．DB的最后sLen个字节是salt，计算出明文M 的哈希值mHask，并拼接出MP=0 0 0 0 0 0 0 0 |mHash|salt；

6．计算MP的哈希值，并与H 比较，如果相等则输出“验证成功”，否则输出“验证失败”，并停止。

由此可见，RSSSA-PSS和其他签名填充模式一样，也是遵循“先哈希再签名”的原则，不直接使用明文M 签名。

18.4.2 签名算法实现

有了上一节分析的签名算法的原理和实现步骤，写出签名算法的实现代码就易如反掌。Rsa_Pkcs15_Sign()函数就是RSSSA-PKCS算法的实现，采用了MD5作为哈希值计算函数，可以看到其填充方式和PKCS加密填充方式类似，最终pSignBuf得到k bits（k 字节）的签名数据：

```
int Rsa_Pkcs15_Sign(CBigInt& d, CBigInt& n, int kbits,
    void *pSrcData, int dataSize, void *pSignBuf, int bufSize)
```

```
{
    int k = kbits / 8;

    char *padBlock = new char[k];
    if(padBlock == NULL)
        return -1;

    unsigned char md5Hash[MD5_DIGEST_SIZE] = { 0 };
    CalcMD5Hash(pSrcData, dataSize, md5Hash);

    int pad_len = k - MD5_DIGEST_SIZE - Md5SignPadSize - 3;
    padBlock[0] = 0x00;
    padBlock[1] = 0x01; //填充全xFF
    GeneratePkcsPad(1, padBlock + 2, pad_len);
    padBlock[pad_len + 2] = 0x00;
    memcpy(padBlock + pad_len + 3, Md5SignPadding, Md5SignPadSize);
    memcpy(padBlock + pad_len + 3 + Md5SignPadSize, md5Hash, MD5_DIGEST_SIZE);
    CBigInt em;
    em.GetFromData(padBlock, k);//OS2IP
    CBigInt  c = ModularPower(em, d, n);
    c.PutToData((char *)pSignBuf, k);

    delete[] padBlock;
    return k;
}
```

18.4.3 验证签名算法实现

RSSSA-PKCS算法的签名验证实现也不复杂，出于篇幅考虑，Rsa_Pkcs15_Verify()函数给出了概念性实现代码，这个函数只处理了具体哈

希值的解析和判断（这已经是签名验证算法的主体），没有对填充数据的格式校验，有兴趣的读者可自行加上校验，使之成为更有实用性的签名验证算法。

```
bool Rsa_Pkcs15_Verify(CBigInt& e, CBigInt& n, int kbits,
    void *pSignData, int dataSize, void *pSrcData, int srcSize)
{
    char *padBlock = new char[kbits / 8];
    if(padBlock == NULL)
        return false;

    CBigInt c;
    c.GetFromData((const char *)pSignData, dataSize);
    CBigInt em = ModularPower(c, e, n);
    int emSize = em.PutToData(padBlock, kbits / 8);
    int pad_len = 2;
    for(int i = 2; i < emSize; i++)
    {
        pad_len++;
        if(padBlock[i] == 0)
            break;
    }

    unsigned char md5Hash[MD5_DIGEST_SIZE] = { 0 };
    CalcMD5Hash(pSrcData, srcSize, md5Hash);

    int result = memcmp(padBlock + pad_len + Md5SignPadSize, md5Hash, MD5_DIGEST_SIZE);

    delete[] padBlock;

    return (result == 0);
}
```

18.5 总结

这一章我们介绍了RSA算法的原理，把看似神秘的RSA算法在放大镜下看了个底朝天，原来如此嘛，你应该有这种感觉吧？蒙哥马利算法、米勒-拉宾算法、欧几里得算法，一个个看似高高在上的名词，原来有如此平易近人的实现，举重若轻，算法的乐趣尽在于此吧。现在，你可以用本章的算法给自己弄个签名什么的，也可以把自己的公钥散发出去，让别人也给你发送加密邮件，最重要的，这些都是你自己实现的。

还记得介绍广义黎曼猜想时提到的英国数学家哈代吗？有一次，哈代要乘船渡北海回英国，那天天气恶劣，浪涛汹涌，而船又很小，因此他在船开之前就写了一张明信片寄给丹麦物理学家波尔（Harald Bohr），上面只写了一句话：“我已经证明了黎曼猜想。哈代。”哈代寄这张明信片的用意是：万一这船沉入大海，哈代死了，世人就会认为哈代真的解决了这个世界数学难题，而为这个解法及哈代一起沉入海底而惋惜。但是上帝如此不喜欢哈代，一定不会让哈代享有解决这个著名难题的声誉，肯定不会让这艘船沉入大海，于是哈代就可以平安回到英国，这样这张明信片就是他的护身符了。本章有些内容还是比较严肃的，大家看看这个乐一下吧，顺便说一下，这不是笑话，是真事儿。

18.6 参考资料

[1] Montgomery P. Modular Multiplication Without Trial Division. Math. Computation, Vol. 44:519-521, 1985

[2] Schneier B. 应用密码学：协议、算法与C源程序（中文版）. 吴世忠，祝世雄，张文政，等译. 北京：机械工业出版社，2004

[3] 王小云，王明强，孟宪萌. 公钥密码学的数学基础. 北京：科学技术出版社，2013

[4] Stinson D R. 密码学原理（第3版）. 北京：电子工业出版社，2009

[5] 乔纳森·卡茨，耶胡达·林德尔. 现代密码学:原理与协议. 北京：国防工业出版社，2011

[6] Bajard J-C, Didier L-S, Komerup P. An RNS Montgomery Modular Multiplication Algorithm. IEEE Transaction on Computers, 47(7):766-776, July 1998

[7] Koc C K, Acar T, Burton S, et al. Analyzing and Comparing Montgomery Multiplication Algorithms. IEEE Micro, 16(3): 26-33, June 1996

[8] Quisquater J-J, Couvreur C. Fast Decipherment Algorithm for RSA Public-Key Cryptosystem. Electronics Letters, Vol. 18, pp. 905-907, October 1982

[9] Wu C-H, Hong J-H, Wu C-W. VLSI Design of RSA Cryptosystem Based on the Chinese Remainder Theorem. Journal of Information Science and Engineering 17, 967-980, 2001

[10] Rivest RL, Shamir A, Adleman L. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. Communications of the ACM, 21(2): 120-126, February 1978

[11] RSA Laboratories. PKCS #1 v2.1: RSA Encryption Standard, June 2002