密码学实验报告 实验七

2019 年 5 月 12 日

1 大数运算实现

1.1 算法原理

在 C++环境中，整数类型的数据结构所支持的最大范围是 unsigned long long 或 long long 类型，占 8 个字节，能够表示$0 \sim 2^{64}-1$或$-2^{63} \sim 2^{63}-1$的整数，然而在密码学算法中常常需要大整数（超过 1024bit）来保证算法的安全性，原有的整数型数据结构不支持如此大的数，所以我们需要另辟蹊径编写大整数类来支持相关大数的运算。在本次实验中我们使用 C++标准库中的字符串 string 类型来表示大整数，并且实现了多种运算函数，基本覆盖了原本 int 类型的基础操作。

首先定义大整数的数据结构，数字部分由变长字符数组 string 表示，符号位由一个 char 型变量表示。

对于大整数之间的加减法，我们可以直接对整数的加减运算进行模拟，加法涉及到进位，减法涉及到借位，从低位到高位进行遍历即可得到结果。

对于大整数之间的乘法，起初打算模拟整数相乘的操作，即令当前数字执行乘数次与自身的加法，然而这种实现会远慢于普通整数型的乘法，原因在于原生的乘法在内部做了很多关于硬件的优化，使计算复杂度大大降低，所以为了利用原来的乘法，我们更换了一种新的大整数相乘方法，使用递归的方式不断地将当前整数二分，用前半段和后半段分别与另一个数相乘，直到切分至乘积小于 long long 的范围，这时便可以使用原生的乘法算法，经过实验测试，新的方法的速度明显要比原方法有较大提升。

对于大整数之间的整除，同样为了提高运算效率，使用了一种优化后的整除算法，首先实现普通的带余除法运算，这个算法使用的是普通的被除数不断减去除数的方法，如果直接使用这个算法来实现整除，会带来与上述乘法相同的计算复杂度，所以我们需要减少带余除法中两个数字相减的次数，在这里采用了分块的方法，与乘法的二分法不同，分块指每次计算部分被除数与整个除数

的商，使每次计算后的商都小于 10，然后不断累加，直至被除数分块结束，最后得到整个被除数与除数的商。经过实验验证，这种整除方法的速度相较原来的除法也有了较大的提升。

对于大整数之间的取余，由于我们已经实现了高性能的乘法与整除，所以利用$r = a - q * b$的方法即可得到余数，其中a为被除数，b为除数，q为商。

最后作为附加功能，我们还实现了大整数的随机数产生、最大公约数、最小公倍数、快速模幂、素性测试等算法，均可以产生正确输出，也为之后的 RSA 算法实现提供了便利。

1.2 算法实现的伪代码

大数加法 计算两大整数相加的和

输入： 大整数a, b
输出： 大整数$a + b$

```
function add(a, b)
    if a < b then
        swap(a, b)
    end if
    for i ← length(a) to 1 do
        sum ← a[i] + b[i] + carry
        result ← sum%10 ⊕ result
        carry ← sum/10
    end for
    if carry ≠ 0 then
        result ← carry ⊕ result
    end if
    return result
end function
```

（⊕代表字符串拼接操作）

大数减法 计算两大整数相减的差

输入： 大整数a, b
输出： 大整数$a - b$

```
function sub(a, b)
    if a < b then
        swap(a, b)
    end if
    for i ← length(a) to 1 do
        diff ← a[i] − b[i]
        if diff < 0 then
            for j ← i − 1 to 0 do
```

```
                    if a[j] ≠ 0 then
                        a[j] ← a[j] − 1
                        break
                    end if
                end for
                j ← j + 1
                while j ≠ i do
                    a[j] ← 9
                    j ← j + 1
                end while
                diff ← diff + 10
            end if
            result ← diff ⊕ result
        end for
        return result
    end function
```

大数乘法 计算两大整数相乘的积

输入： 大整数a, b

输出： 大整数$a * b$

```
    function mul(a, b)
        LMAX ← FLOOR_SQRT_LLONG_MAX
        if a ≤ LMAX and b ≤ LMAX then
            result ← a * b
        else
            if a < b then
                swap(a, b)
            end if
            half_length ← length(a)/2
            a_high ← a[0: half_length]
            a_low ← a[half_length: ]
            b_high ← b[0: half_length]
            b_low ← b[half_length: ]
            s_high ← a_high * b_high
            s_low ← a_low * b_low
            s_mid ← (a_high + a_low) * (b_high + b_low) − s_high − s_low
            result ← s_high ⊕ s_mid ⊕ s_low
        end if
        return result
    end function
```

大数整除 计算两大整数相整除的商

输入： 大整数a, b
输出： 大整数a/b

function $floordiv(a, b)$
  if $a \leq LLONG\_MAX$ and $b \leq LLONG\_MAX$ **then**
    $result \leftarrow a/b$
  else
    $chunk\_remainder \leftarrow a[0: length(b) - 1]$
    $chunk\_index \leftarrow length(b) - 1$
    while $chunk\_index < length(a)$ **do**
      $chunk \leftarrow chunk\_remainder \oplus a[chunk\_index]$
      $chunk\_index \leftarrow chunk\_index + 1$
      while $chunk < b$ **do**
        $result \leftarrow result \oplus 0$
        if $chunk\_index < length(a)$ **then**
          $chunk \leftarrow chunk\_remainder \oplus a[chunk\_index]$
        else
          break
        end if
      end while
      if $chunk = b$ **then**
        $result \leftarrow result \oplus 1$
        $chunk\_remainder \leftarrow 0$
      else if $chunk > b$ **then**
        $chunk\_q, chunk\_remainder \leftarrow divide(chunk, b)$
        $result \leftarrow result \oplus chunk\_q$
      end if
    end while
  end if
  return $result$
end function

1.3 测试样例及运行结果

我们首先随机生成了 50 位数的十进制大整数a, b和 25 位的十进制大整数c，对加法、减法、乘法、整除、取余、最大公约数、快速模幂、素性检测进行实验验证，结果如表 1 所示。

表 1 大数运算的实验验证

数据	运算结果
a	64249429690157419770113014735437668553950668173322
b	31281199468928063225846197447068861614470950501130
c	60846595452621469924394 65
$a + b$	95530629159085482995959212182506530168421618674452
$a - b$	32968230221229356544266817288368806939479717672192

$a*c$	390935905641865531516015496056707948511561657195551694047578427336212952730
a/c	10559248091404815754655190
$a\%c$	1251262833142642645099972
$a+b-b$	642494296901574197701130147354376685539506681733322
$a*c/c$	642494296901574197701130147354376685539506681733322
$gcd(a,c)$	1
$a^b(\text{mod } c)$	3020298047425937562266239
$is\_prime(a)$	False

2 RSA 算法实现

2.1 算法原理

RSA 是一种公钥密码体制，属于分组密码，基于大整数因子分解的困难问题，其明文和密文都是 0 至某 n-1 之间的整数，通常 n 的大小为 1024 位二进制数或 309 位十进制数，也就是说 n 小于2^{1024}。RSA 算法使用乘方运算，明文以分组为单位进行加密，每个分组的二进制值均小于 n，也就是说，分组的大小必须小于或等于$\log_2(n)+1$位，在实际应用中，分组的大小是i位，其中$2^i < n \leq 2^{i+1}$。对明文分组 M 和密文分组 C，加密和解密过程如下：

$$C = M^e \text{mod } n$$

$$M = C^d \text{mod } n = (M^e)^d \text{mod } n = M^{ed} \text{mod } n$$

其中收发双方均已知n，发送方已知e，只有接收方已知d，因此公钥加密算法的公钥为$PU=\{e,n\}$，私钥为$PR=\{d,n\}$。其中e和d需满足：

$$ed \text{ mod } \phi(n) \equiv 1$$

也就是说e和d是模$\phi(n)$的乘法逆元，如此我们就可以保证下式成立：

$$M^{ed} \text{mod } n = M$$

除此之外，运用中国剩余定理可以加快解密时的运算速度，首先定义

$$V_p = C^d \text{mod } p \quad V_q = C^d \text{mod } q$$

运用 CRT，定义

$$X_p = q \times (q^{-1} \text{mod } p) \quad X_q = p \times (p^{-1} \text{mod } q)$$

由 CRT 可得

$$M = (V_p X_p + V_q X_q) \text{mod } n$$

进一步还可以使用费马定理来简化V_p和V_q的运算

$$V_p = C^{d \bmod(p-1)} \bmod p \quad V_q = C^{d \bmod(q-1)} \bmod q$$

根据资料，使用了中国剩余定理的解密速度大约比原来的方法快 4 倍。

2.2 算法实现的伪代码

RSA 密钥生成算法 产生 RSA 算法的公私钥对

输入： 无
输出： 公钥PU，私钥PR

```
function genKey()
    选择两个素数p, q且p ≠ q
    n ← p × q
    φ(n) ← (p − 1) × (q − 1)
    选择整数e使gcd(φ(n), e) = 1且1 < e < φ(n)
    d ← e^{-1} (mod φ(n))
    PU ← {e, n}
    PR ← {d, n}
    return PU, PR
end function
```

RSA 加密算法 利用 RSA 加密

输入： 明文$M < n$，公钥$PU = \{e, n\}$
输出： 密文C

```
function rsa_encrypt(M, PU)
    C ← M^e (mod n)
    return C
end function
```

RSA 解密算法 利用 RSA 解密

输入： 密文C，私钥$PR = \{d, n\}$
输出： 明文M

```
function rsa_decrypt(C, PR)
    M ← C^d (mod n)
    return M
end function
```

2.3 测试样例与运行结果

首先我们随机产生了 1024bit 的公私钥对，预先分别保存至 pk.txt 和 sk.txt 文件中，具体的密钥如表 2 所示。

表 2 RSA 公私钥对

e	14889033130493204963085754737883244287273661546523443759166405483747313126775475365510381155774027807224041590637666097280050903112898847463430115332560642969134284128144000687995839801273388011393027035589692898514114912836304969802731364163282106410674925094591491282904458407975403692103893154402701512196 7	309 位
d	15943149207169858375594153401307552029081711518354240625963270227319142532549340354872373828615277032445862340361534951533239218652870582847605769228572010195732249219149364723576356109111125875970912117461659449635777828269992179837955435362036091638761307368210166830832084782141413957390931998273721561230 3	309 位
n	16681031293259953737849231641443281450169386118056712602066730479486533865037947104423875199842575457281232277448640510674723209027033219925758857832944730550232126678787549131550725552748824973919343625495846967736210715158921738265117975391587666228714561181867715704592520175953334985773786831949084995961 1	309 位

我们对明文消息$M = 1145141919810$进行加密，得到的密文C为

954026581459277863853751272907853543555401011013538739198198056576752028404053389438404509756324896844707901034016622557194592006105178757883128925047246279306930129704259248403493688170035330413210034591940963340050952229062447550257653610544155365495662222932882771621616950642800623661582758425669482656 84

对密文C用私钥解密后得到相同的明文，证明算法正确。

我们还对文件做了加解密测试，具体结果如图 1 所示。其中明文消息为 text.txt，加密后内容为 output_text.txt，解密后内容为 output_output_text.txt。

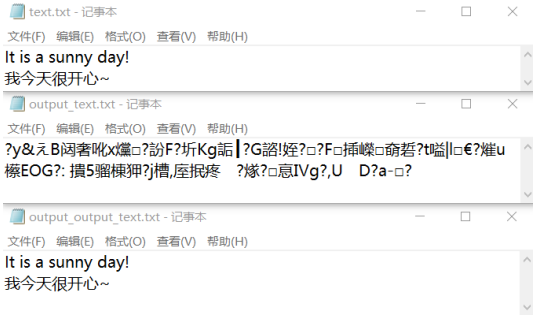

图 1 RSA 文件加解密测试

3 RSA-OAEP 算法实现

3.1 算法原理

如果使用传统的 RSA 算法对需要加密的明文消息直接进行加密，则会很容易地受到选择密文攻击（CCA），为了防止这种简单攻击，RSA 专利推荐使用最优非对称加密填充（OAEP）的程序对明文进行修改。

OAEP 是常常与 RSA 结合使用的一种填充机制，其结构形式属于 Feistel 网络，使用随机预言 G 和 H 来处理非对称加密前的明文。当与任何安全陷门函数 f 复合使用时，对于选择明文攻击是语义安全的，OAEP 在选择密文攻击同样是安全的。

OAEP 满足以下两个特性：

(1) 添加了具有随机性的元素，能够将原有的确定性加密方案（例如传统 RSA）转化成为一种概率性的体制。

(2) 防止对于密文的部分解密，可以确保对方在不能反转陷门单向转换的情况下，无法复原明文的任何部分。

3.2 算法实现的伪代码

MGF1 掩模函数 产生与输入长度相同的散列值

输入： 掩模种子$mgfSeed$，掩模输出长度$maskLen$

输出： 掩模$mask$

function $MGF1(mgfSeed, maskLen)$

令T为空比特串

for $i \leftarrow 0$ to $\lceil maskLen/hLen \rceil - 1$ **do**

$C \leftarrow I2OSP(i, 4)$

$T = T||Hash(mgfSeed||C)$

end for

return $mask[0: maskLen]$

end function

RSA-OAEP 加密算法 利用 RSA-OAEP 加密

输入： 明文M，公钥$PU = \{e, n\}$

输出： 密文C

function $rsa\_oaep\_encrypt(M, PU)$

$lHash \leftarrow Hash(L)$

$PS \leftarrow 0$

$DB \leftarrow lHash||PS||0x01||M$
生成一个长度为的$hLen$随机比特串$seed$
$dbMask \leftarrow MGF1(seed, k - hLen - 1)$
$maskedDB \leftarrow DB \oplus dbMask$
$seedMask \leftarrow MGF1(maskedDB, hLen)$
$maskedSeed \leftarrow seed \oplus seedMask$
$EM \leftarrow 0x00||maskedSeed||maskedDB$
$m \leftarrow OS2IP(EM)$
$c \leftarrow rsa\_encrypt(m, PU)$
$C \leftarrow I2OSP(c, k)$
return C
end function

RSA-OAEP 解密算法 利用 RSA-OAEP 解密
输入：密文C，私钥$PR = \{d, n\}$
输出：明文M
function $rsa\_oaep\_decrypt(C, PR)$
$c \leftarrow OS2IP(C)$
$m \leftarrow rsa\_decrypt(c, PR)$
$EM \leftarrow I2OSP(m, k)$
$lHash \leftarrow Hash(L)$
分解$EM = Y||maskedSeed||maskedDB$
$seedMask \leftarrow MGF1(maskedDB, hLen)$
$seed \leftarrow maskedSeed \oplus seedMask$
$dbMask \leftarrow MGF1(seed, k - hLen - 1)$
$DB \leftarrow maskedDB \oplus dbMask$
分解$DB = lHash'||PS||0x01||M$
return M
end function

3.3 并行化加解密实现

经过上次实验的尝试，我们发现并行化地处理数据在多核处理器上有着更佳的表现，经过对算法的分析，由于 RSA 也属于分组密码体制，每次使用相同的密钥，所以可以将不同组的加解密分配到不同的处理器核心上运行，具体实现使用了 Python 的多进程库 multiprocessing，将需要加密的数据库分为 n 组，其中 n 为处理器核心数目，这样做是为了减少进程创建的开销，然后并行地计算这些数据分组，将输出合并即可得到理想的结果。而且并行化的加解密与利用中国剩余定理加快解密算法并不冲突，我们可以在利用 CRT 进行快速解密的基础上再利用并行化进一步提升程序的速度。

3.4 测试样例及运行结果

为了充分对算法的性能进行验证，我们对于优化前、优化后、优化后且使用了并行处理的 RSA 加解密算法进行了实验验证和对比分析，实验环境如表 3 所示。

表 3 实验环境

处理器	RAM	填充模式	编程语言
Intel i7-6700HQ	8GB	OAEP	Python 3.6

接下来输入不同大小的文件并记录算法的执行时间，结果如表 4 所示。

表 4 算法执行时间（单位：秒）

	加密		解密		
文件大小	普通加密	并行加密	普通解密	快速解密	快速+并行
10k	1.122	0.986	1.109	0.389	0.779
20k	2.325	1.287	2.259	0.757	0.876
50k	5.791	1.957	5.658	1.896	1.249
100k	11.916	3.059	11.856	3.763	1.495
200k	22.966	5.372	22.569	7.542	2.292
500k	-	11.915	-	19.595	4.698
1000k	-	23.694	-	-	8.518
无缓存初始化		1.811	有缓存初始化		0.001

利用 Python 的 matplotlib 模块绘制数据曲线，结果如图 2 所示。

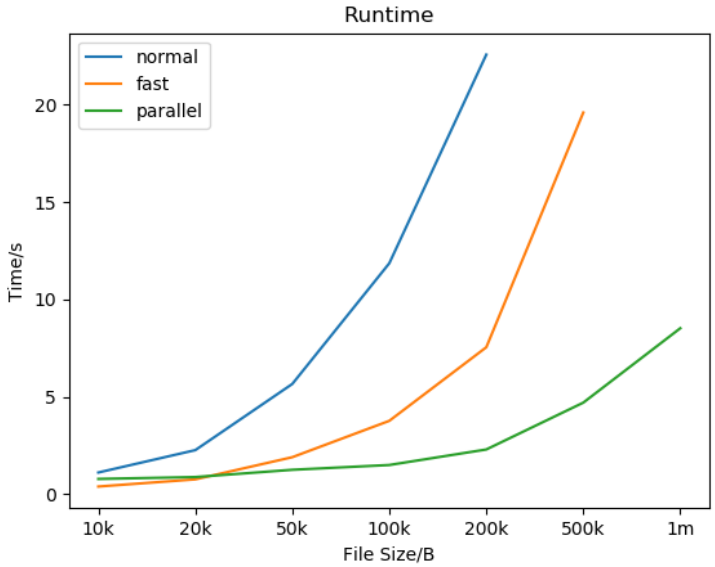

图 2 算法运行时间对比

从表中数据和图中曲线可以看出，快速解密平均比普通解密提高了 3 倍的速度，而对于较大的文件使用并行化处理后又可以提升近 10 倍的速度，最终我们在当前的实验环境下达到了 14KB/s 的加密速度和 37KB/s 的解密速度。

除此之外，为了验证并行化处理时的加解密算法是否可以正确地进行运算，我们挑选了一张图片对其进行了加解密操作，原文件为 1m.png，加密后的文件为 en_1m.png，解密后的文件为 de_1m.png，结果如图 3 所示，说明算法运行正常。

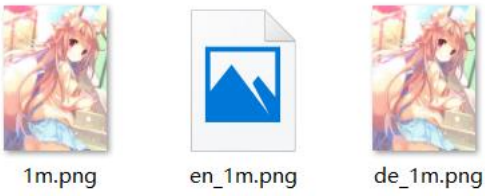

图 3 对文件的加解密验证

4 背包密码体制

4.1 背包密码算法原理

Merkle–Hellman 背包密码算法是一种早期的公钥密码体制，目前已被破解。该算法基于背包问题的复杂性，给定一个集合A和数字b，找到一个A的子集使其和为b是 NP-complete 的。然而如果背包向量是超递增的，即每个元素都大于集合中比它小的元素之和，则这个问题很容易利用贪心算法在多项式时间内求解得到。

背包公钥密码体制的基本思想为将一根易于求解的问题伪装成一个难解问题，其公私钥对为两个背包向量，公钥为难解的背包向量A，私钥为易解的超递增背包向量B，以及两个额外的乘数和模数。乘数和模数可以将超递增背包向量转化为难解的普通背包向量，同样也可以将难解的背包向量的结果转化为易解的背包向量子集，并且能在多项式时间内完成。

算法在加密时使用难解的背包向量A，得到A的一个子集和，解密时先将密文转化为超递增背包向量的子集和，然后使用贪心算法即可在$O(n)$的时间复杂度内得到明文。

目前 Merkle–Hellma 背包密码算法已经有多种破译方法，Shamir 在 1983 年最先提出了一种破译方法，可以在多项式时间内破解。由于时间原因，在这里我们没有对破解方法进行进一步的探索。

4.2 背包密码算法实现的伪代码

MH 密钥生成算法 产生 MH 背包密码算法的公私钥对

输入： 密钥长度n

输出： 公钥PU，私钥PR

function $genKey(n)$

选择一个 n 比特的超递增背包向量$w = (w_1, \cdots, w_n)$

随机选择整数q使$q > \sum_{i=1}^{n} w_i$

随机选择整数r使$\gcd(r, q) = 1$

for $i \leftarrow 1$ to n **do**

$\beta_i = r w_i \mod q$

end for

$PU \leftarrow \{\beta\}$

$PR \leftarrow \{w, q, r\}$

return PU, PR

end function

MH 加密算法 利用 MH 背包密码算法加密

输入： 明文$M < 2^n$，公钥$PU = \{\beta\}$

输出： 密文C

function $mh\_encrypt(M, PU)$

令 n 比特消息$M = (m_1, \cdots, m_n)$，其中$m_i \in \{0,1\}$

$C \leftarrow \sum_{i=1}^{n} m_i \beta_i$

return C

end function

MH 解密算法 利用 MH 背包密码算法解密

输入： 密文C，私钥$PR = \{w, q, r\}$

输出： 明文M

function $mh\_decrypt(C, PR)$

$r_{inv} \leftarrow r^{-1} \mod q$

$C \leftarrow (C * r_{inv}) \mod q$

for $i \leftarrow 1$ to n **do**

if $C \geq w_i$ **then**

$M[i] \leftarrow 1$

$C \leftarrow C - w_i$

else

$M[i] \leftarrow 0$

end if

end for

return M

end function

4.3 测试样例及运行结果

我们利用 MH 背包密码算法对测试数据进行加密，首先随机生成一组公私钥对，然后尝试用公钥对明文$M = (b19ce)_{16}$进行加密，得到的密文为

$$C = (b33b4f6549500872af9bae)_{16}$$

使用私钥对密文进行解密得到相同明文，证明算法正确。

除此之外，我们也对文件的加解密进行了验证，效果如图 4 所示。

图 4 背包密码算法对文件的加解密

5 总结

在学习 RSA 公钥密码体制后，我本以为这次将会是一次较为容易的实验，但是实际做起来发现这次实验的难度与我想象中的相差甚远，虽然 RSA 的原理十分简单，但是在实现的过程中我仍然学到了许多新的知识。

在大数运算的实现中，利用 C++面向对象的特性，我实现的大数类基本覆盖了原本整数类型的操作，并且通过思考，改进了原有的算法，加速了大整数时间相乘和整除的运算，使程序的效率得到了提高。

虽然在 RSA 加解密算法的实现上并没有遇到困难，但是在实现最优非对称加密填充（OAEP）时，我遇到了一些知识盲区，例如对随机预言机模型不太了解，不知道掩模生成函数该如何工作等等，经过文献查询，我使用了较为常见的 MGF1 掩模生成函数，也发现 RSA 标准中的八位组串可以看作比特串，最终成功实现了最优非对称加密填充下的 RSA 加解密算法，并且对算法速度进行了深入探究，首先使用中国剩余定理（CRT）来加快解密速度，然后还使用了并行化处理进一步加速，最终达到了较为理想的速度提升效果，经过编程实现，我也对最优非对称加密填充以及随机预言机的原理有了更深的了解。

最后我们还实现了一种早期的公钥密码体制——背包密码，虽然这种密码早

已被破译，但是我们可以从中看到一些早期的公钥密码思想，由于时间原因，本次实验无法对背包密码体制的攻击进行更进一步的探索，作为未来的工作，可以在此实验上作进一步的扩展，分析背包密码体制的攻击方法。