山东大学
网络空间安全学院

密码工程第二次实验
实验报告

学生姓名　陈嘉蘅-202200460178

胡承旭-202200460148

冯奕楠-202200460037

指导教师　王伟嘉

学　　院　网络空间安全学院

专业班级　22 级密码一班

完成时间　2024 年 10 月 14 日

一、实验目标

使用任意两个 2048 位数字来实现蒙哥马利乘法和 Barrett 归约。

二、实验背景与原理

蒙哥马利算法和 Barrett 归约原理都是用于大数模运算的优化算法，尤其在公钥密码学中有着广泛的应用。以下是对这两种算法原理的详细解释：

蒙哥马利算法（Montgomery Algorithm）是一种用于快速模运算的算法，其主要思想就是简化除法运算，转化成位运算。

它主要分为蒙哥马利模乘、蒙哥马利约简和蒙哥马利模幂三种形式。

1. 蒙哥马利形式：

为了计算$ab \pmod N$，找一个R，然后使得$a' \equiv aR \pmod N, b' \equiv bR \pmod N$,

其中R需要满足：$R = 2^k > N$（其中k是满足条件的最小正整数）,$\gcd(R, N) = 1$

2. 蒙哥马利约简 Montgomery reduction(a,b,N)：

计算$XR^{-1} \pmod N$,由此前的R的定义可知，$R = 2^k$,所以$XR^{-1} = X >> k$（X右移k位）

但是右移 k 位可能会抹掉 X 的低位中的一些 1，这不是精确计算，而是想下取整的除法，当且仅当 X 是 R 的整数倍时，X/R 严格等于 X>>k,所以找到一个 m，使得 X+mR 是 R 的倍数，计算这个数 mod N 即可。

根据R的定义，$\gcd(R, N) = 1$，根据扩展的欧几里得算法，有$RR' - NN' = 1$并且有$0 < N' < R, 0 < R' < N < R$。（注意这里是$-NN'$，所以有$N' = -N^{-1} \pmod R$）

$$
\begin{gathered}
X + mN \equiv 0 \pmod R \\
XN' + mNN' \equiv 0 \pmod R \\
XN' + m(RR' - 1) \equiv 0 \pmod R \\
XN' \equiv m \pmod R
\end{gathered}
$$

约简流程如下：

计算$N' = -N^{-1} \pmod R$,计算$m = XN' \pmod R$;

计算$y = \dfrac{X + mN}{R}$：将$X + mN$右移k位；

若$y > N$，则$y = y - N$,这时的y满足:$0 < y < 2N$。

因为$X < N^2, m < R, N < R$,所以$\dfrac{X + mN}{R} < \dfrac{N^2 + RN}{R} < \dfrac{RN + RN}{R} = 2N$;

返回y。

3. 蒙哥马利模乘 Montgomery Multiply(a,b,N)：

计算 $a' \equiv aR \pmod N, b' \equiv bR \pmod N, X = a'b'$；

调用蒙哥马利约简，$X_1 = Montgomeryreduction(X, R, N) = X/R = abR \pmod R$；

再调用蒙哥马利约简，$y = Montgomeryreduction(X_1, R, N) = X_1 / R = ab(\mathrm{mod}\, N)$；

返回 y。

3.蒙哥马利模幂：

是基于蒙哥马利模乘和蒙哥马利约简实现的幂运算。

Barrett 归约原理是一种用于大数模运算的优化方法，它利用预计算和简单的算术运算来避免大整数除法。

1.预计算：

在进行模运算之前，预计算出一些常数，以便在后续的归约过程中使用。

2.归约过程：

通过乘法和加法运算，将大整数转化为一个较小的范围，然后利用预计算的常数进行调整，最终得到模运算的结果。归约过程中避免了除法运算，从而提高了计算效率。

三、实验环境与设置

（一） 硬件配置详情

处理器 12th Gen Intel(R) Core(TM) i7-12700H 2.30 GHz

机带 RAM 16.0 GB (15.7 GB 可用)

系统类型 64 位操作系统, 基于 x64 的处理器

（二） 软件配置详情

操作系统版本：windows 11 家庭中文版 22631.4169

VirtualBox 版本 7.1.0 r164728 (Qt6.5.3)

编程语言：python 3.12

四、实验步骤与结果展示

实验代码：

```
import random
from Crypto.Util.number import getPrime, inverse


# 蒙哥马利约简
def montgomery_reduce(a, n, n_inv, r):
    # 计算中间变量 q
    q = ((a & (r - 1)) * n_inv) & (r - 1)
    # 计算约简后的 a
    a = (a + q * n) >> (r.bit_length() - 1)
    # 如果 a 大于等于模数 n，则减去 n
    if a >= n:
```

```
        a -= n
    return a

# 蒙哥马利乘法
def montgomery_multiply(a_bar, b_bar, n, n_inv, r):
    # 计算乘积
    t = a_bar * b_bar
    # 对乘积进行蒙哥马利约简
    return montgomery_reduce(t, n, n_inv, r)

# 巴雷特约简
def barrett_reduction(x, n):
    # 计算模数 n 的位数
    k = n.bit_length()
    # 计算 2 的 2k 次幂
    r = 1 << (k * 2)
    # 计算 n 的逆元（使用 Crypto 库的 inverse 函数）
    n_inv = inverse(n, r)
    # 计算中间变量 q2
    q2 = ((x * n_inv) >> k) >> k
    # 计算约简后的结果
    r = x - q2 * n
    # 如果结果大于等于 n，则减去 n
    if r >= n:
        r -= n
    return r

# 生成两个随机的 2048 位数和一个 2048 位的素数模数
a = random.randint(2**2047, 2**2048 - 1)
b = random.randint(2**2047, 2**2048 - 1)
n = getPrime(2048)

# 初始化蒙哥马利乘法的参数
r = 1 << n.bit_length()  # r 是大于 n 的最小 2 的幂
n_inv = inverse(n, r)  # n 的模逆
```

```
# 将 a 和 b 转换为蒙哥马利形式
a_bar = (a * r) % n
b_bar = (b * r) % n

# 执行蒙哥马利乘法
montgomery_result = montgomery_multiply(a_bar, b_bar, n, n_inv, r)
# 将结果转换回普通形式
# 但为了与直接计算的结果对比，我们可以再次约简以确保一致性（实际上这一步是多余的）
montgomery_result_normal = montgomery_reduce(montgomery_result * n_inv, n, n_inv, r) % n

# 执行巴雷特约简
# 为了对比，我们直接对 a*b 的结果进行巴雷特约简
barrett_result = barrett_reduction(a * b, n)

# 输出结果
print("Answer")
print((a * b) % n)
print("Montgomery Multiplication (Normal Form)")
print(montgomery_result_normal)
print("Barrett Reduction ")
print(barrett_result)
```

实验结果：