RSA 实验报告

目录

1. 程序使用说明

1.1 开发和运行环境

开发工具：Visual Studio 2017 4.8.03761
开发环境：C++11，MFC，pthread 库
实验硬件配置：

操作系统：Windows 10 1803 专业教育版
CPU：Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz
GPU：Intel(R) UHD Graphics 620；NVIDIA GeForce MX150
RAM：DDR4 2400MHz 16G

运行注意：

推荐在 Windows10 .Net3.5 及以上，1G 内存以上环境运行程序。
操作系统需带有 pthread 库，且能够运行 MFC 程序。

1.2 界面说明

如下图 1 所示，界面由左右两部分构成，左上选择需要的密钥长度，点击生成密钥后，程序会在左侧文本框显示生成结果(16 进制)，并在下方显示花费的时间。右侧上面的文本框输入待加密内容，点击加密，右下角输入框获得加密文本(16 进制)，点击解密，程序将右下 16 进制码解密并输出明文到右上角输入框。

图 1.RSA 程序界面

1.3 具体操作演示

密钥长度选取和生成操作如下 2 所示：

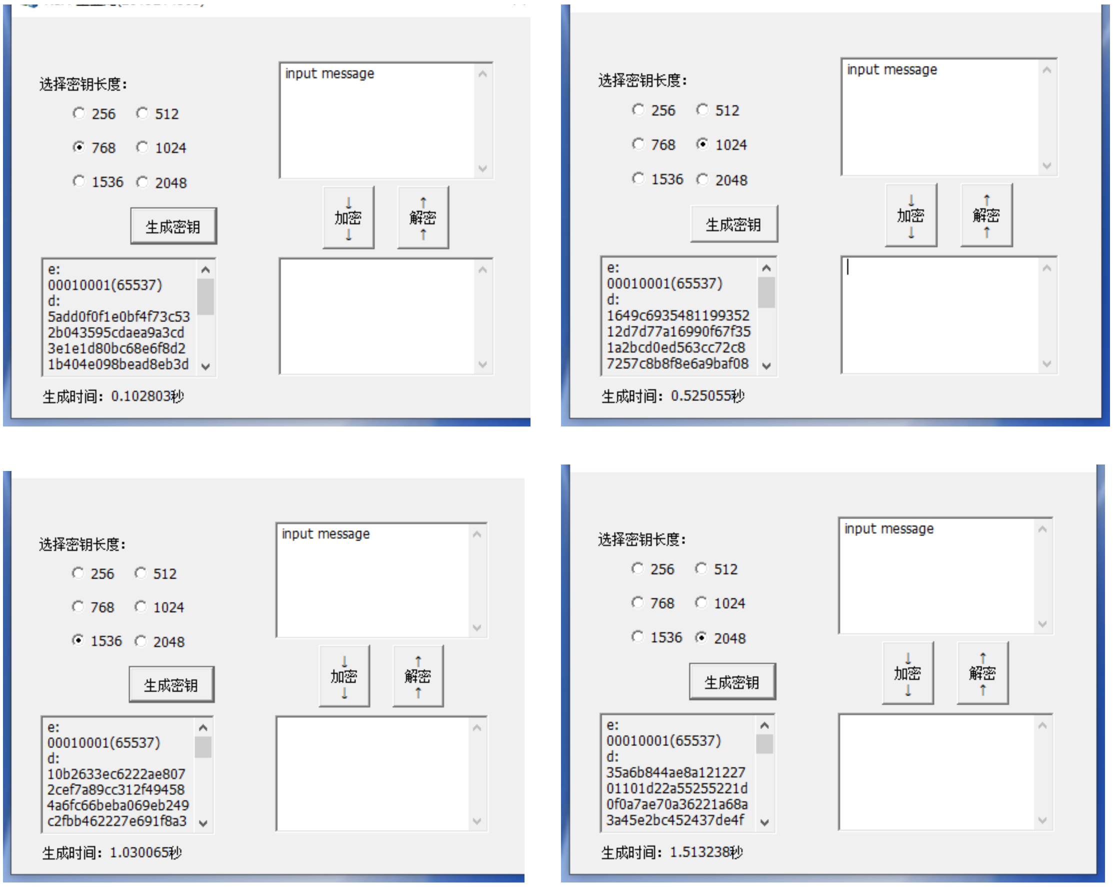

图 2. 选取不同的密钥长度，点击生成密钥，在左下角展示计算结果（可复制）。

加密解密如下图 3 所示，加密后若手动将右上部分文本清空，点击解密：

图 3. 使用生成好的密钥对”input message”加密和解密的过程

更加具体的操作可参考“操作视频”。

2.实现亮点

(1) 使用中国剩余定理优化解密过程，将 E^d % n 改为对（E^d % p, E^d % q）求 CRT，解密性能优化大约 2 倍。

(2) 利用平凡平方根的性质预处理模数后缀零的个数，在前缀数字快速幂后检查每次平方前是否只能是 1 或-1，不是则重新生成素数。这一操作相比朴素检查快速幂结果是否为 1 更加可靠，提升了 Miller Rabin 素性检测正确率。

(3) 生成素数时使用多线程计算，对于 p 和 q 分别开 4 个线程寻找合法的素数（共计 8 线程）。性能优化大约 2~8 倍。

(4) 使用 65536 进制（即 2^{16} 进制）存储高精度数字，相比 10 进制，加减优化 log(65536)/log(10)≈5 倍，乘除优化 25 倍。65536 进制恰好能满足运算不超过 unsigned int 范围，故代码实现中使用 unsigned int 记录数字。

(5) 程序初始化时，使用 O(n)的欧拉筛计算 65536 以内所有素数。对于这些素数做乘法合并，将乘积不超过 65536 的素数合并记录到一个 vector<unsigned int>中（以 2,3,5,7,11,13 为例，将它们的乘积 30030 记录到 vector 中，然后扔掉 2,3,5,7,11,13）。

在进行 Miller Rabin 素性测试前，使用上述记录的数字进行初筛：

对于待测试数字 P 和 vector 中的一个数字 a，计算 r=gcd(P,a)，若 r=1 则允许进入 Miller Rabin。由于 a 是低精度数字，P 是高精度数字，所以 P % a 是低精度数字，只需要 O(N+log(a))即可完成一次 r 的计算，其中 N 为压位后的 P 的数组长度。该过程相比朴素地判断 P 模素数不等于 0 要快大约 2 倍，并且能大幅减小不合法数字进入 Miller Rabin 测试的频次，整体性能优化 2~5 倍。

(6) 使用牛顿迭代优化除法/取模运算：

在快速幂运算时（包括 Miller Rabin、加解密），使用牛顿迭代预处理模数（除数）的倒数并记录小数点位数，当需要进行取模（除法）运算时，转换为对倒数的乘法运算。

优化性能分析：由于朴素取模/除法复杂度是 O(bits*N)，N=bits/16，乘法复杂度是 $O(N^2)$，故从取模/除法转变为乘法，理论上要快 16 倍。而牛顿迭代的收敛速度极快，对于 2048bit 数字，牛顿迭代能在大约 17 次迭代后算出倒数结果，这个初始化过程的复杂度是 $O(kN^2)$的，且只需要算一次，将快速幂整体的复杂度优化到 $O(bits*N^2)$。

(7*) 使用 NTT 优化大数乘法：

由于 65536 进制下 FFT 会丢失精度，故采用 NTT 优化大数乘法，由于 long long 范围内，int 平方范围外不存在合适的模数，故在计算时每位 65536 进制数字转化为两位 256 进制数字，使用原根 G=3，模数 NTTP=$119*2^{23}+1$=998244353 的 NTT，将乘法优化到 O(NlogN)，除法在牛顿迭代基础上复杂度也为 O(NlogN)。

*但由于压位后，存储的位数已经变得很少（768 bit RSA 的素数 p 只需要 24 个 unsigned int 存储），其运算已经相当快速，应用 NTT 后并没有得到优化，反而由于其常数较大，性能会慢于朴素求法，故在程序中特判当数组大小超过 200 时才使用 NTT 优化的乘法。

3. 感想

g++的代码写好后嵌入 Visual C++会出现很多兼容问题，需要各种调整。

O2/OX 优化要谨慎使用，本次作业开发过程中不使用 O2/OX 优化时程序一切正常，使用 O2/OX 优化后，加密的牛顿迭代莫名其妙的出 bug，而且 Visual Studio 的报错还很模糊。

由于 NTT 的模数问题，本程序不得不将 65536 进制数字先拆成 256 进制，事实上 NTT 更适合对 32768 进制进行计算，使用模数 2281701377 和原根 3 即可。

虽然 NTT 由于常数过大本次几乎没派上用场，但就地的 Karatsuba 算法（非递归版本，若使用递归仍然慢）也是可以对乘法进行优化。

本程序的大数都初始化了一个数组大小，事实上如果使用变长数组性能会更快，因为频繁运算时初始化反而成为了计算瓶颈。但变长数组的开发更加麻烦，开发起来实在是心有余力不足。

烧了好多时间啊。