上海大学
Shanghai University

《密码学》实验报告

网络空间安全专业　计算机工程与科学学院
23121677 范舒舰

RSA 加密算法实现及 RSA 攻击

2025 年 12 月 24 日

一、实验目的

1. 实验目的

理解RSA加密算法的基本原理，并探索RSA在不正确生成模N的情况下的安全性。

2. 实验要求

实现RSA加密算法，要求手工实现随机生成大素数(3072位)、计算逆元、快速模幂运算、生成RSA公钥和私钥对、使用公钥加密文本信息、使用私钥解密密文等阶段，使用测试用例#1验证加解密的正确性。

测试用例#1:

原始明文消息:"Hello,RSA!"

分解挑战 #1

如下的整数 N 是两个素数 p, q 的乘积，且满足 $|p-q| < 2N^{\frac{1}{4}}$，请分解整数 N，并给出其十进制结果。

N = 17976931348623159077293051907890247336179769789423065727343008115773267580550562068698537944921298295958550138753716401571013985864783377860692558349754108519659161512805757594075263500747593528871082364994994077189561705436114947486504671101510156394068052754007158456087857766374304008634074285527854909258 1

分解挑战 #2

如下的整数 N 是两个素数 p, q 的乘积，且满足 $|p-q| < 2^{11}N^{\frac{1}{4}}$，请分解整数 N，并给出其十进制结果。（提示：在此条件下有 $A - \sqrt{N} < 2^{20}$，因此需要从 $\sqrt{N}$ 开始搜索 A 的值）。

N = 648455842808071669662824265346772278726343720706976263060439070378797308618081116462714015276061417569195587321840254520655424906719892428844841839353281972988531310511738648965962582821502504990264452100885281673303711142296421027840289307657458645233683357077834689715838646088239640236866252211790085787877

二、实验环境

1. 操作系统：Windows 11
2. 编程语言：Python 3.11（gmpy2）
3. 随机数源：os.urandom
4. 开发工具：Spyder

三、实验内容

1. 算法原理与流程

（1）RSA 密钥生成原理

RSA 密钥生成的数学过程如下：

1. 随机生成两个大素数 p, q；
2. 计算模数 $n = pq$；
3. 计算欧拉函数 $\varphi(n) = (p-1)(q-1)$；
4. 选取公钥指数 e，并要求 $\gcd(e, \varphi(n)) = 1$；
5. 计算私钥指数 d，满足

$$ed \equiv 1 \pmod{\varphi(n)}.$$

最终得到公钥 (n, e) 与私钥 (n, d)。本实验固定使用 $e = 65537$，并在生成后输出 $\gcd(e, \varphi(n))$ 以验证逆元存在。

（2）RSA 加密与解密原理

RSA 运算在整数域上进行。将明文消息编码成字节串后，按大端解释为整数 m，并要求 $0 \le m < n$。加密与解密分别为：

$$c \equiv m^e \pmod{n}, \qquad m \equiv c^d \pmod{n}.$$

解密得到的整数 m 再转换回字节串并还原为字符串，从而完成“文本信息”的加密与解密验证。

（3）随机生成大素数的基本思路

随机生成大素数采用“随机候选 + 素性检测”的方式：

- 使用 `os.urandom` 生成随机字节并转为整数，强制最高位为 1 以保证位长；
- 将候选数置为奇数以减少无效尝试；
- 先进行小素数试除快速过滤；
- 对剩余候选执行 Miller–Rabin 概率素性测试，通过则视为高概率素数。

其中 Miller–Rabin 的核心运算是幂模计算，因此快速模幂是关键性能点。

（4）快速模幂与模逆元

快速模幂：RSA 与 Miller–Rabin 都大量使用 $a^b \bmod n$。直接计算 a^b 再取模不可行，因此使用平方–乘算法将复杂度降低到 $O(\log b)$。

模逆元：私钥 d 是 e 在模 $\varphi(n)$ 下的逆元。使用扩展欧几里得算法在求 $\gcd(e, \varphi(n))$ 的同时得到系数，从而求出逆元。

（5）RSA 攻击：Fermat 分解（挑战 #1 与 #2）

当 p, q 很接近时，可写：

$$N = pq = (A - x)(A + x) = A^2 - x^2,$$

其中 $A = \frac{p+q}{2}$，$x = \frac{p-q}{2}$。因此

$$x = \sqrt{A^2 - N}.$$

若 $A^2 - N$ 为完全平方，则可恢复：

$$p = A + x, \qquad q = A - x.$$

挑战 #1 的条件保证用 $A = \lceil \sqrt{N} \rceil$ 即可求得分解；挑战 #2 需要从 $\sqrt{N}$ 起对 A 进行搜索，并且题目给出 $A - \sqrt{N} < 2^{20}$ 的上界提示。

（6）解密挑战：恢复私钥并按 0x00 分隔拆包

分解挑战 #1 得到 p, q 后，计算 $\varphi(N) = (p - 1)(q - 1)$，求逆得到 d，再解密：

$$m \equiv C^d \pmod{N}.$$

题目说明采用 PKCS v1.5 填充，指出用 `0x00` 分隔“随机填充”和“消息”。因此实现上将解密结果恢复为与模数等长的字节块后，从第 2 字节开始寻找第一个 `0x00`，其后部分即为明文消息。

2. 关键函数与代码实现

（1）随机数生成：randbits / randbelow（拒绝采样）

```python
    def randbits(bits: int) -> int:
    # Generate an integer with exactly `bits` bits using os.urandom.
    nbytes = (bits + 7) // 8
    b = os.urandom(nbytes)
    x = int.from_bytes(b, "big")
    excess = nbytes * 8 - bits
    if excess:
    x &= (1 << bits) - 1
    x |= (1 << (bits - 1))  # ensure top bit is 1
    return x

    def randbelow(n: int) -> int:
    # Rejection sampling to get uniform integer in [0, n).
    k = n.bit_length()
    while True:
    x = randbits(k)
    if x < n:
    return x
```

实现说明：

- randbits(bits)：用 os.urandom 获取随机字节并转整数，掩码截断到指定位数，并强制最高位为 1，保证候选数位长满足要求；
- randbelow(n)：采用拒绝采样而不是 x % n，避免产生分布偏差，使 Miller–Rabin 选取底数更合理。

（2）快速模幂：mod_pow（平方–乘）

```python
    def mod_pow(base: int, exp: int, mod: int) -> int:
    # Square-and-multiply modular exponentiation.
    base, exp, mod = mpz(base), mpz(exp), mpz(mod)
    base %= mod
    r = mpz(1)
    while exp > 0:
    if exp & 1:
    r = (r * base) % mod
    base = (base * base) % mod
    exp >>= 1
    return int(r)
```

实现说明：

- 指数 exp 每次右移一位，相当于按二进制位从低到高处理；
- 若当前位为 1，则将当前底数乘入结果；每轮底数平方一次并取模；

（3）扩展欧几里得与模逆元：egcd / modinv

```
    def egcd(a: int, b: int):
    # Extended Euclid: returns (g, x, y) such that a*x + b*y = g.
    a, b = mpz(a), mpz(b)
    x0, y0, x1, y1 = mpz(1), mpz(0), mpz(0), mpz(1)
    while b:
    q = a // b
    a, b = b, a - q * b
    x0, x1 = x1, x0 - q * x1
    y0, y1 = y1, y0 - q * y1
    return int(a), int(x0), int(y0)

    def modinv(a: int, m: int) -> int:
    g, x, _ = egcd(a % m, m)
    if g != 1:
    raise ValueError("modular inverse does not exist")
    return x % m
```

实现说明：扩展欧几里得在求 $\gcd(a,m)$ 的同时给出系数 x,y 使得 $ax+my=\gcd(a,m)$。当 $\gcd(a,m)=1$ 时，$ax\equiv 1 \pmod m$，因此 $x \bmod m$ 即为 $a^{-1} \pmod m$，用于计算 RSA 私钥指数 d。

（4）Miller–Rabin 素性测试

```
    def is_probable_prime(n: int, rounds: int = 64) -> bool:
    if n % 2 == 0:
    return False
    # small prime trial division omitted ...
    d = n - 1
    s = 0
    while d % 2 == 0:
    d //= 2
    s += 1

    nn = mpz(n)
    for _ in range(rounds):
    a = randbelow(n - 3) + 2  # a in [2, n-2]
    x = mod_pow(a, d, n)
    if x == 1 or x == n - 1:
    continue
```

```
17
18      x_mp = mpz(x)
19      composite = True
20      for _ in range(s - 1):
21      x_mp = (x_mp * x_mp) % nn
22      if int(x_mp) == n - 1:
23      composite = False
24      break
25      if composite:
26      return False
27      return True
```

实现说明：

- `_SMALL_PRIMES`：
  小素数表用于试除过滤。对随机候选数，先检查是否能被小素数整除，可以在 Miller–Rabin 前快速剔除大量合数。

- `is_probable_prime(n, rounds)`：
  Miller–Rabin 概率素性测试实现。关键步骤：

  1. 排除偶数与小素数整除情况；
  2. 写成 $n-1=d\cdot 2^s$，其中 d 为奇数；
  3. 每轮随机取底数 $a\in[2,n-2]$，计算 $x=a^d \bmod n$；
  4. 若 $x=1$ 或 $x=n-1$ 则该轮通过；
  5. 否则重复平方 $s-1$ 次：$x\leftarrow x^2 \bmod n$，若出现 $n-1$ 则通过；若始终不出现则判定为合数。

  本实验 rounds=64：即使存在极少数强伪素数，连续 64 轮随机见证仍将误判概率降到极低。

- `generate_prime(bits, rounds)`：
  循环生成候选数并调用 `is_probable_prime` 检测，直到得到素数。实现中将候选数强制为奇数（`| 1`）减少一半无效尝试。该函数同时返回 `tries` 可用于观察随机生成素数的尝试次数波动（实验中不同运行的 tries 会不同，属于正常现象）。

（5）RSA 密钥生成与加解密

```
1       def rsa_keygen(bits: int = 3072, e: int = 65537, rounds: int = 64):
2       while True:
3       p, _ = generate_prime(bits, rounds)
4       q, _ = generate_prime(bits, rounds)
```

```
    if p == q:
    continue
    phi = (p - 1) * (q - 1)
    if gcd(e, phi) == 1:
    n = p * q
    d = modinv(e, phi)
    return (n, e), (n, d), p, q

    def rsa_encrypt_text(pub, text: str) -> int:
    n, e = pub
    m = int.from_bytes(text.encode("ascii"), "big")
    return mod_pow(m, e, n)

    def rsa_decrypt_text(priv, c: int) -> str:
    n, d = priv
    m = mod_pow(c, d, n)
    k = (n.bit_length() + 7) // 8
    raw = int(m).to_bytes(k, "big").lstrip(b"\x00")
    return raw.decode("ascii")
```

实现说明：

- `rsa_keygen(bits=3072, e=65537, rounds=64)`：
  完整实现 RSA 密钥生成流程：
  1. 调用 `generate_prime` 得到 p, q；
  2. 计算 $\varphi(n) = (p-1)(q-1)$，并检查 $\gcd(e, \varphi(n)) = 1$；
  3. 若互素则计算 $d = \texttt{modinv}(e, \varphi(n))$；
  4. 返回公钥 (n, e)、私钥 (n, d) 以及 p, q。
- `rsa_encrypt_text(pub, text)`：
  将文本转为整数 m，再计算 $c = m^e \bmod n$：
  - 文本按题意编码为字节串，再用大端转换为整数；
  - 只要保证 $m < n$，RSA 运算是合法的；
  - 运算核心调用 `mod_pow` 完成幂模。
- `rsa_decrypt_text(priv, c)`：
  计算 $m = c^d \bmod n$ 后将整数转为字节串并还原文本：
  - 将 m 转为“与模数长度一致”的字节块（字节数 $k = \lceil \text{bitlen}(n)/8 \rceil$）；
  - 去掉前导 `0x00`（用于测试用例的短消息还原）；
  - 将剩余字节解码回字符串，从而验证 RSA 过程正确。

（6）Fermat 分解：挑战 #1 与挑战 #2

```
    def fermat_factor_challenge1(N: int):
    Nn = mpz(N)
    A = gmpy2.isqrt(Nn)
    if A * A < Nn:
    A += 1
    x2 = A * A - Nn
    x = gmpy2.isqrt(x2)
    p = int(A + x)
    q = int(A - x)
    return (q, p) if q <= p else (p, q)

    def fermat_factor_challenge2(N: int, max_steps: int = (1 << 20)):
    Nn = mpz(N)
    A = gmpy2.isqrt(Nn)
    if A * A < Nn:
    A += 1

    x2 = A * A - Nn
    steps = 0
    while steps <= max_steps:
    if gmpy2.is_square(x2):
    x = gmpy2.isqrt(x2)
    p = int(A + x)
    q = int(A - x)
    if p * q == N:
    return ((q, p) if q <= p else (p, q)), steps

    x2 = x2 + 2 * A + 1
    A += 1
    steps += 1

    raise ValueError("Challenge #2 search exceeded max_steps")
```

实现说明：

- 挑战 #1：题设保证 $A = \lceil\sqrt{N}\rceil$ 时 $A^2 - N$ 为完全平方，因此可直接求 $x = \sqrt{A^2 - N}$ 得到 p, q；
- 挑战 #2：需要从 $\sqrt{N}$ 开始搜索 A。为提高效率，使用递推

$$(A+1)^2 - N = (A^2 - N) + 2A + 1$$

增量更新 `x2`，避免每步重新计算平方；并输出 `steps` 作为搜索步数，便于与题目提示的上界对照。

3. 结果输出与正确性验证

实验实现中：

- RSA 加密相关的大整数数据（$p, q, n, \varphi(n), e, d, m, c$）会完整保存到 `rsa_encrypt_data.txt`；
- 终端仅展示大整数前 30 位与后 20 位以便查看流程；
- 分解挑战与解密挑战的长整数按每行 60 位输出，便于人工核对。

（1）程序运行原始输出

```
========== 实验五：RSA 加密算法实现及 RSA 攻击 ==========
[1] 生成 RSA 密钥（p,q 各 3072 位）
p = 39512348942403895218407522797 4...20451201157409284387
q = 47499857120578724690390563257 2...41275505372811221143
n = 18768309292626349041141051206 3...96930709563234194341
e = 65537
d = 57999995133063986317687674150 6...99110222827221664701
RSA 加密相关数据已保存到：rsa_encrypt_data.txt

[2] 测试用例#1: "Hello, RSA!" 加密/解密验证
明文 = Hello, RSA!
m = 87521618088895491219865889
c = 16109440317883166406276423654 1...35433146539355860969
解密得到 = Hello, RSA!
验证结果 = 通过

========== RSA 攻击：分解挑战 #1 ==========
N =
179769313486231590772930519078902473361797697894230657273430
081157732675805505620686985379449212982959585501387537164015
710139858647833778606925583497541085196591615128057575940752
635007475935288710823649949940771895617054361149474865046711
015101563940680527540071584560878577663743040086340742855278
549092581
p =
134078079299425970995740249982058461274793658205923933777235
614437217640300736627688911116143623269986750405460943393208
384195233759860275304415621357243 01
```

```
q =
134078079299425970995740249982058461274793658205923933777235
614437217640300737785609803489305577505696600492340021925908
230851639400254851144494752653642 81
校验: p*q==N -> True

========== RSA 攻击: 分解挑战 #2 ==========
N =
648455842808071669662824265346772278726343720706976263060439
070378797308618081116462714015276061417569195587321840254520
655424906719892428844841839353281972988531310511738648965962
582821502504990264452100885281673303711142296421027840289307
657458645233683357077834689715838646088239640236866252211790
085787877
搜索步数 = 72077
p =
254647961469961834380088165639739422293414542685241578463285
819278857779699852228351438510732495734541073844615571931733
044972448140715057905665932064197 59
q =
254647961469961834380088165639739422293414542685241578463285
819278857779701063980544912465269708141676325635095417847347
418713798566823547477183464713754 03
校验: p*q==N -> True

========== 解密挑战（使用挑战#1 的 N, e=65537） ==========
密文 C =
220964518674103817763065611348834180174100697878928310717318
391436761356001205380042823296504735094243439462197515122564
658399679428894607645420405815647489880137348641204523252293
201764879166664029975091887299716905260832220677716000193292
608700095799937240774589677736978175712672299511486629596279
34791540
解密明文 = Factoring lets us break RSA.
```

（2）输出解析与正确性说明

- **RSA 加/解密验证：**输出中“解密得到 = Hello, RSA!”且“验证结果 = 通过”，说明对测试用例完成了正确的加密与解密流程。
- **分解挑战 #1：**输出给出十进制 p, q，并校验 `p*q==N -> True`，说明 Fermat 分解在题设条件下正确恢复因子。
- **分解挑战 #2：**输出显示“搜索步数 = 72077”，且同样校验 `p*q==N -> True`。步数小于题目提示的搜索上界 2^{20}。
- **解密挑战：**使用挑战 #1 的 p, q 恢复私钥后，对密文解密并按 `0x00` 分隔拆包得到明文：

  `Factoring lets us break RSA.`

  说明“分解 → 恢复私钥 → 解密”完整成立。

4. 遇到的问题与解决方法

1. **大素数生成方式不明确，不知道如何进行素性判断**
   解决：在实现中尝试试除过滤，查阅文献采用多轮 Miller–Rabin 概率素性测试，通过则视为高概率素数；同时在输出/文件中保留关键信息，便于对生成过程进行检查与复现。

2. **随机候选位长不满足要求导致密钥规模偏小**
   解决：在 `randbits(bits)` 中强制设置最高位为1，确保候选整数始终为指定位长，从源头保证密钥规模。

3. **Miller–Rabin 底数选择的分布问题**
   原因：若通过 `x%n` 生成随机底数，会带来轻微偏差。
   解决：实现 `randbelow(n)` 的拒绝采样，使底数在区间内近似均匀，测试更符合理论假设。

4. **挑战 #2 Fermat 搜索效率问题**
   原因：若每步都显式计算 $A^2 - N$，会引入大量重复的大整数平方运算。
   解决：使用递推更新 $(A+1)^2 - N = (A^2 - N) + 2A + 1$，并配合 gmpy2 的平方判定函数，显著降低每步开销，从而在可接受时间内完成分解。

四、体会与收获

通过本次实验我对 RSA 的实现细节与安全性依赖条件有了更深入的认识：

- 学会了把 RSA 的完整流程落地实现：生成 p, q，计算 $n, \varphi(n)$，验证 $\gcd(e, \varphi(n)) = 1$，再求逆得到 d，最终用幂模完成加密与解密，并通过测试用例 `"Hello, RSA!"` 验证正确性。
- 理解了随机数与素数生成对 RSA 的基础性作用，学习了 Miller–Rabin 多轮检测稳定生成大素数。
- 通过 Fermat 分解挑战直观认识到“密钥生成不当会破坏安全性”：当 p, q 过于接近时，N 可在很小搜索范围内被分解。