

Github（或者 **Coding**）账号：<https://github.com/jiaxinw-se>

个人博客关于密码学大作业的连接：<https://zhuanlan.zhihu.com/p/611006663>

题目（中文）：**RSA 大礼包**

摘要（中文）：

本文介绍了 RSA 公钥加密体系以及复现常见的攻击手段。RSA 公钥加密算法是基于大整数分解的困难问题建立的。本次实验的数据来源为首届全国高校密码数学挑战赛赛题三。采用 Python 编写程序。通过预处理截获数据，最终实现多种由于参数选取不当造成的攻击手段：因素碰撞、共模攻击、低加密指数广播攻击、费马分解攻击、pollard p-1 分解法。

关键词：RSA 算法，加密破解，因素碰撞、共模攻击、低加密指数广播攻击、费马分解攻击。

题目描述（清楚描述题目中文，写出自己的理解，请勿复制原题目）

Alice 正在使用一个使用 RSA 加密的通信程序。她发了 21 条信息，但其中有些是相同的，还有些参数不是很合适。现在已经拦截了所有 21 条信息，告诉你密文的格式是：

$$1024\text{bit } n \mid 1024\text{bit } e \mid 1024\text{bit } c = m^e \bmod n$$

即前 256 个数字为 n ，后 256 数字为 e ，其余为密文。要求你尽可能多地破解 Alice 的信息。

过程（包括背景，原理：必要的公式，图表；步骤，如有必要画出流程图，给出主要实现步骤代码）

1. RSA 加密体系介绍

1.1 背景

RSA 公开密钥密码体制是一种使用不同的加密密钥与解密密钥，“由已知加密密钥推导出解密密钥在计算上是不可行的”密码体制。

在公开密钥密码体制中，加密密钥（即公开密钥）PK 是公开信息，而解密密钥（即秘

密密钥) SK 是需要保密的。加密算法 E 和解密算法 D 也都是公开的。虽然解密密钥 SK 是由公开密钥 PK 决定的, 但却不能根据 PK 计算出 SK。

1.2 RSA 算法

RSA 算法的具体如下:

- (1) 任意选取两个不同的大素数 p 和 q 计算乘积:

$$n = pq$$

$$\varphi(n) = (p - 1)(q - 1)$$

- (2) 任意选取一个大整数 e , 满足:

$$\gcd(e, \varphi(n)) = 1$$

整数 e 用做加密钥 (注意: e 的选取是很容易的, 所有大于 p 和 q 的素数都可用);

- (3) 确定的解密密钥 d , 满足

$$(de) \bmod \varphi(n) = 1$$

是一个任意的整数; 所以, 若知道 e 和 $\varphi(n)$, 则很容易计算出 d ;

- (4) 公开整数 n 和 e , 秘密保存 d ;

- (5) 将明文 m ($m < n$ 是一个整数) 加密成密文 c , 加密算法为:

$$c = E(m) = m^e \bmod n$$

- (6) 将密文 c 解密为明文 m , 解密算法为:

$$m = D(c) = c^d \bmod n$$

然而只根据 n 和 e 要计算出 d 是不可能的。因此, 任何人都可对明文进行加密, 但只有知道私钥 d 才可对密文解密。

公钥: e

私钥: d

1.3 安全性

RSA 的安全性依赖于大数分解, 但是否等同于大数分解一直未能得到理论上的证明, 也没有从理论上证明破译。RSA 的难度与大数分解难度等价。因为没有证明破解 RSA 就一定需要做大数分解。假设存在一种无须分解大数的算法, 那它肯定可以修改成为大数分解算法, 即 RSA 的重大缺陷是无法从理论上把握它的保密性能如何。

目前，RSA 的一些变种算法已被证明等价于大数分解。不管怎样，分解 n 是最显然的攻击方法。现在，人们已能分解 140 多个十进制位的大素数。因此，模数 n 必须选大些，视具体情况而定。

RSA 算法的保密强度随其密钥的长度增加而增强。但是，密钥越长，其加解密所耗用的时间也越长。因此，要根据所保护信息的敏感程度与攻击者破解所要花费的代价值不值得以及系统所要求的反应时间来综合考虑，尤其对于商业信息领域更是如此。

2.RSA 攻击复现

2.1 思路

结合 pdf 说明文档，分析 Frame 数据。得出解密方法总结为：

第 1、18 因数碰撞攻击。

第 0、4 共模攻击

第 3、8、12、16、20 低加密指数广播攻击

第 10 费马分解攻击

第 2、6、19 $p-1$ 方法分解 N

2.2 数据预处理与分拆

因为文件给出的数据前 256 个数字为 n ，后 256 数字为 e ，其余为密文，我们先进行 n, e, c 的读取与分离。代码如下：

```
1.  ni = []
2.  ei = []
3.  ci = []
4.  n_e_c = []
5.  for i in range(21):
6.      with open("Frame" + str(i), 'r') as cipher_file:
7.          temp = cipher_file.read()
8.          ni.append(int(temp[:256], 16))
9.          ei.append(int(temp[256:512], 16))
10.         ci.append(int(temp[512:], 16))
```

2.3 因素碰撞

因数碰撞法指的是，如果参数选取不当， p 或者 q 在多个 rsa 加密中使用多次，那么生成不同

的 n 可能会有相同的因子，我们假设 p 相同, q 不同，那么有：

$$\begin{cases} n_1 = p \times q_1 \\ n_2 = p \times q_2 \end{cases}$$

很容易知道

$$\gcd(n_1, n_2) = p$$

这样很快就能将他们各自的私钥求解出来了。（寻找两个 n 的 $\gcd \neq 1$ ）

代码如下：

```
1.  #两两验证 gcd
2.  print("因素碰撞攻击")
3.  s_f = []
4.  for i in range(frames_num):
5.      for j in range(i + 1, frames_num):
6.          if (ni[i] != ni[j]) and (gmpy2.gcd(ni[i], ni[j]) != 1):
7.              s_f.append((i, j))
8.  #print("Same Factor", s_f)
9.
10. def same_factor_attack(s_f):
11.     for j in range(len(s_f)):
12.         p = gmpy2.gcd(ni[s_f[j][0]], ni[s_f[j][1]])
13.         for i in range(2):
14.             flag = s_f[j][i]
15.             q = ni[flag] // p                #n=p*q
16.             fai_n = gmpy2.mpz((p - 1) * (q - 1))    #y(n)
17.             d = gmpy2.invert(ei[flag], fai_n)    #解密密钥
18.             int_m = pow(ci[flag], d, ni[flag])
19.             m = long_to_bytes(int_m)[-8:].decode()
20.             crack_results[flag] = m
21.             print(f"—frame {j} -> m = {m}")
22.
23. same_factor_attack(s_f)
```

攻击结果为：

```
因素碰撞攻击:
--frame 1 : m = . Imagin
--frame 18 : m = m A to B
```

2.4 共模攻击

共模攻击指的是如果一条消息发送给不同的接收方，接收方的模数 n 相同，公钥 e 不同，那么可以进行如下的攻击：

$$\begin{cases} C_1 = m^{e_1} \bmod n \\ C_2 = m^{e_2} \bmod n \end{cases}$$

我们求解：

$$e_1x + e_2y = 1$$

由扩展欧几里得算法我们可以很轻松的求出 x 和 y

所以有：

$$C_1^xC_2^y = m^{e_1x+e_2y} = m \bmod n$$

所以将明文给求解出来了。

结果如下：

```
共模攻击:
Same Module [[0, 4]]
--Frame [0, 4] : m = My secre
```

代码如下：

```
1. print("共模攻击:")
2. same_module= []
3. for i in range(frames_num):
4.     will_ack = [i]
5.     for j in range(i + 1, frames_num):
6.         if (ni[i] == ni[j]) and (ei[i] != ei[j]):
7.             will_ack.append(j)
8.         if len(will_ack) > 1:
9.             same_module.append(will_ack)
10. #print("Same Module", same_module)
11.
12. # 广义欧几里得
13. def egcd(a, b):
14.     if a == 0:
15.         return (b, 0, 1)
```

```

16.     else:
17.         g, y, x = egcd(b % a, a)
18.         return (g, x - (b // a) * y, y)
19.
20. def same_module_attacker(same_module):
21.     for round in range(len(same_module)):
22.         i = same_module[round][0]
23.         j = same_module[round][1]
24.         g, r, s = egcd(ei[i], ei[j])
25.         if r < 0:
26.             r = -r
27.             ci[i] = gmpy2.invert(ci[i], ni[i])
28.         int_m = pow(ci[i], r, ni[i]) * pow(ci[j], s, ni[i]) % ni[i]
29.         str_m = long_to_bytes(int_m)[-8:].decode()
30.         crack_results[i] = str_m
31.         crack_results[j] = str_m
32.         print("--Frame {} : m = {}".format(same_module[round], str_m))
33.     return
34.
35. same_module_attacker(same_module)

```

2.5 低加密指数广播攻击

低加密指数广播攻击适合于 n 很大, e 很小的情况, 其适用的情况如下, 当一条消息 m 发送给不同的接收者时, 每个接收者的 n 都不同, 但是加密的公钥都是相同的, 我们假设公钥为 3, 那么有:

$$\begin{cases} C_1 = m^3 \bmod n_1 \\ C_2 = m^3 \bmod n_2 \\ C_3 = m^3 \bmod n_3 \end{cases}$$

由中国剩余定理知道, 我们是可以将 m^3 算出来的, 那么对其开立方就将明文给求出来了。结果如下:

```

低加密指数广播攻击
Same E and E < 20: {5: [3, 8, 12, 16, 20], 3: [7, 11, 15]}
--frame [3, 8, 12, 16, 20] : m = t is a f

```

代码如下:

```

1. print("低加密指数广播攻击")
2. low_power = {}
3. for i in range(frames_num):

```

```
4.         if ei[i] <= 20:                #遍历所有指数e, 如果存在e 小于20 可能存在漏洞
5.             if low_power.get(ei[i]) == None: #判断所有公钥相等的情况
6.                 low_power[ei[i]] = [i]
7.             else:
8.                 low_power[ei[i]].append(i)
9. print("Same E and E < 20: ", low_power)
10. #中国剩余定理求解
11. def chinese_remainder_theorem(a_list, m_list):
12.
13.     m = 1
14.     result = 0
15.     for mi in m_list:
16.         m *= mi
17.     for i in range(len(m_list)):
18.         Mi = m // m_list[i]
19.         Mi_re = egcd(Mi, m_list[i])[1]
20.         result += Mi * Mi_re * a_list[i]
21.     return result % m
22. #主函数
23. def low_power_ack(low_power_objs):
24.     for power in low_power_objs.keys():
25.         seqs = low_power_objs[power]
26.         c_list = [ci[i] for i in seqs]
27.         n_list = [ni[i] for i in seqs]
28.         m_pow = chinese_remainder_theorem(c_list, n_list)
29.         int_m = gmpy2.iroot(gmpy2.mpz(m_pow), power)
30.         str_m = long_to_bytes(int_m[0])[-8:].decode()
31.         for i in seqs:
32.             crack_results[i] = str_m
33.         print("—frame {} :  m = {}".format(seqs, str_m))
34.     return
35.
36. low_power_ack(low_power)
```

2.6 费马分解法

如果 p 和 q 相差不大的话我们可以通过费马分解把 p 和 q 求出来, 原理如下:

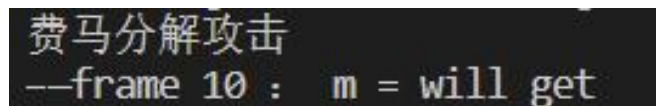
$$n = p \times q = \frac{1}{4}(p+q)^2 - \frac{1}{4}(p-q)^2$$

由于 p 与 q 相差不大, 所以 $p-q$ 相对于 n 和 $(p+q)^2$ 来说可以忽略不计, 所以有:

$$2\sqrt{n} \approx p + q$$

通过不断尝试就可以把 p 和 q 给计算出来了。

结果如下:



代码如下:

```

1.  print("费马分解攻击")
2.  def fermat_ack():
3.      for i in range(frames_num):
4.          pi = gmpy2.isqrt(ni[i]) + 1
5.          qi = gmpy2.isqrt(pi*pi-ni[i])
6.          flag = 0
7.          while (pi*pi-ni[i]!=qi*qi) and (flag<200000):
8.              pi += 1
9.              qi = gmpy2.isqrt(pi*pi-ni[i])
10.             flag += 1
11.         if (pi*pi-ni[i]) == qi*qi:
12.             p = (pi + qi) % ni[i]
13.             q = (pi - qi) % ni[i]
14.             fai_n = (p - 1) * (q - 1)
15.             d = gmpy2.invert(ei[i], fai_n)
16.             _m = pow(ci[i], d, ni[i])
17.             m = long_to_bytes(_m)[-8:].decode()
18.             crack_results[i] = m
19.             print("---frame %d : m = %s"%(i, m))
20.  fermat_ack()

```

2.7 pollard p-1 分解法

如果 p 与 q 都不超过 1020 次方, 若其中一个 $(p-1)$ 或 $(q-1)$ 的因子都很小的时候(在这里为了方便说明我们假设为 $(p-1)$), 可以如下操作:

选取一个整数 k , 使其满足 $(p-1) \mid k!$, 由费马小定理知道, a 与 p 互素的时候有:

$$a^{p-1} \equiv 1 \pmod{p}$$

所以有

$$a^{k!} \equiv 1 \pmod{p}$$

即

$$p | (a^{k!} - 1)$$

那么对于 n 与 $(a^{k!} - 1)$ 必有公因数为 p ，这样就可以把 n 分解出来了。

我们直接计算 $a^{k!} \equiv 1 \pmod{n}$ 和 $\gcd(a-1, n)$ ($n=pq$)

但是对于 k 的选取还是有要求的，太小了 $(p-1) | k!$ 不会成立，太大了花费时间会很多。

结果如下：

```
pollard p-1分解法
--frame 2 : m = That is
--frame 6 : m = "Logic
--frame 19 : m = instein.
```

代码如下：

```
1. print("pollard p-1 分解法")
2. def pollard_p_1(n):
3.     a = 2 # a 选取 2
4.     max = 200000
5.     for k in range(1, max + 1):
6.         a = pow(a, k, n)
7.         if gmpy2.gcd(n, a - 1) != 1:
8.             return gmpy2.gcd(a - 1, n)
9.     return
10.
11. def pollard_1_ack():
12.     for i in range(frames_num):
13.         p = pollard_p_1(ni[i])
14.         if p != None:
15.             q = ni[i] // p
16.             fai_n = (p - 1) * (q - 1)
17.             d = gmpy2.invert(ei[i], fai_n)
18.             _m = pow(ci[i], d, ni[i])
19.             m = long_to_bytes(_m)[-8:].decode()
20.             crack_results[i] = m
21.             print("--frame {} : m = {}".format(i, m))
22.
23. pollard_1_ack()
```

2.8 其他方法

LLL 算法

LLL 算法又叫做格基约简，是 Gauss 算法向高维格的推广，在后面应用到密码设计与分析中，成为密码学非常有力的工具。其可以在部分已知 p 、 q ，并且 p 的未知部分的上界 X 和 q 的未知部分的上界 Y 满足 $X \times Y < \sqrt{n}$ 的情况下把 n 分解出来。并且这种方法在之后广泛应用于 RSA 体制的小指数攻击和部分私钥泄露攻击等领域，具体可见 LLL 算法在 RSA 安全性分析中的应用。

暴力分解 N

推荐常用分解网站 factordb.com。

总结（完成心得与其它，主要自己碰到的问题和解决问题的方法）

结果汇总：

Frame 0 : My secre
Frame 1 : . Imagin
Frame 2 : That is
Frame 3 : t is a f
Frame 4 : My secre
Frame 5 :
Frame 6 : "Logic
Frame 7 :
Frame 8 : t is a f
Frame 9 :

Frame 10 : will get
Frame 11 :
Frame 12 : t is a f
Frame 13 :
Frame 14 :
Frame 15 :
Frame 16 : t is a f
Frame 17 :
Frame 18 : m A to B
Frame 19 : instein.
Frame 20 : t is a f

这次尝试第一届全国高校密码学挑战赛，目的为练习 RSA 算法常见漏洞的利用。学习了多种针对 RSA 的攻击方式，比如 RSA 共模攻击、pollard、低加密指数法、因数碰撞法、Fermat 攻击等，了解了他们的原理，对 RSA 的结构和缺点有了更深入的了解。但是仍然存在一些尚未解决的问题，比如 pollard p-q 方法未能实现，有些赛题未解出等等。

参考文献（包括参考的书籍，论文，URL 等，很重要）

参考链接：

https://blog.csdn.net/weixin_44145452/article/details/109924843

<https://www.cnblogs.com/jcchan/p/8426731.html>

<https://www.tr0y.wang/2017/10/31/RSA2016/#%E8%A7%A3%E9%A2%98%E8%BF%87%E7%A8%8B>

https://blog.csdn.net/weixin_45859850/article/details/109785669