RSA 工业级素数

Industrial-Grade Primes in RSA

刘卓

1 为何使用素数

为了保证 RSA 使用过程中的安全性，p, q 的选择必须为素数。如果采用两个合数相乘，那么乘积依然为合数，两把密匙并不唯一，则加密-解密的逆运算将不成立。即解密可能出现多解。

2 因式分解方法

2.1 试除法

给定一个合数 n（这里 n 是一个待分解的正整数），试除法是用小于等于 $\sqrt{n}$ 的每个素数去试除待分解的正整数 n。如果找到一个数能够将 n 整除除尽，则这个数就是待分解整数 n 的因子。时间复杂度为 $O(2^{logN/2})$。

```
def is_prime(a):
    if a == 1:
        return False
    for i in range(2, int(a**0.5)+1):
        if a % i == 0:

            return False
    return True
a = '67280421310721 '

b = is_prime(int(a))
print(b)
```

2.2 普通数域筛选法

普通数域筛选法 (General Number Field Sieve) 是已知效率最高的分解整数的算法。分解一个整数 n，需要

$$\exp\left(\left(\sqrt[3]{\frac{64}{9}}+o(1)\right)(\ln n)^{\frac{1}{3}}(\ln\ln n)^{\frac{2}{3}}\right)=L_n\left[\frac{1}{3},\sqrt[3]{\frac{64}{9}}\right]$$

步才能实现。所以时间负责度是 $\mathrm{O}\left(\exp\left(\left(\frac{64}{9}n\right)^{\frac{1}{3}}(\log n)^{\frac{2}{3}}\right)\right)$.

有关 Python 代码实现，请参考这里primefac

2.3 秀尔算法

在量子领域，秀尔算法 (Shor’ s algorithm) 用于找出给定整数 n 的质因数。在量子计算机上，要分解整数 n，秀尔算法的运作需要多项式时间。该算法的时间复杂度是 O $\left((logN)^3\right)$。速度大大高于 GNFS。

此运算需要量子计算机的支持。

3 因式分解难度

$MIPS-Year$ 是密码学中计算工作量的“标准”度量: 它指的是一台计算机以每秒一百万次操作的速度在一年内所执行的工作量。破解 RSA 挑战的难度以及涉及数据加密标准的挑战通常用 $MIPS-Year$ 计量。概念类似于光年。

行业规定一个 $MIPS-Year$ 大约为 245 次操作。

用 GNFS 算法来破解 RSA-512 长度的测试大约需要 $8000MIPS-Year$。

而破解 RSA-1024 长度的 N 则需要 980 亿 $MIPS-Year$。

即使使用人类目前顶尖的超级计算机，比如美国橡树林国家实验室研究的顶点 (Summit) 超级计算机，其每秒浮点计算能力达到 200 PFLOPS(200×10^{15}), 破解 RSA-1024 也需要数十亿年。

假如用另一种方法破解 RSA，即把已知的所有素数和乘积列成表进行查询，难度也并不会因此出现较大下降。

根据素数定理，素数计数函数 (Prime-counting function) 用来表示小于或等于某个实数 x 的素数的个数，记为 $\pi(x)$, 其中 x 为值。

$$\pi(x)=Li(x)+O\left(xe^{-\frac{1}{15}\sqrt{\ln x}}\right)$$

其中 $Li(x)=\int_2^x\frac{dt}{lnt}$, 而 $O\left(xe^{-\frac{1}{15}\sqrt{\ln x}}\right)$ 是误差估计。

这意味着，如果使用 RSA-1024 作为加密，转化为十进制则有 309 位，意味着 10^{309} 以内的数，也有大约 10^{306} 个素数。因此很难穷举。

现在主流加密手段使用 RSA-2048。而人类已经分解的最大整数是 250 位（十进制）。

(注：生成的第一个 RSA 数字（从 RSA-100 到 RSA-500）根据其十进制数字标记。从 RSA-576 开始，将改为计算二进制数字。RSA-617 是一个例外，它是在更改编号方案之前创建的。)

RSA 安全性建立在大整数的因数分解问题和 RSA 问题困难性的基础上。为了增强大整数的分解难度，其素因子通常为强素数或大的安全素数。值得注意的是如果素数 p 和 q 离得越近则那么 n 越好好分解。一旦素数被找到或者 N 被成功拆分，那么 RSA 就会被非常容易的被破解出来。

4 素数检验

如果分解一个整数太过困难，那么判定一个数整数是否为素数就简单的多了。下面介绍几种主流的算法。算法分为确定性算法和随机算法。确定性算法是指没有使用随机数的算法，速度慢，但保证正确。随机算法是使用了随机数的算法，速度快，但有可能出错。

4.1 试除法

试除法是确定性算法。给定一个合数 n（这里 n 是一个待分解的正整数），试除法是用小于等于 $\sqrt{n}$ 的每个素数去试除待分解的正整数 n。如果找到一个数能够将 n 整除除尽，则这个数就是待分解整数 n 的因子。时间复杂度为 $O(2^{logN/2})$。对于大整数的素性检测，并非完全实际有效。

```
def is_prime(a):
    if a == 1:
        return False
    for i in range(2, int(a**0.5)+1):
        if a % i == 0:

            return False
    return True
a = '67280421310721 '

b = is_prime(int(a))
print(b)
```

4.2 费马素性检验

费马素性检验是一种素数判定法则，利用随机化算法判断一个数是合数还是有可能为素数。参考这里

参考这里根据费马小定理：如果 p 是素数，并且 $1 \leq a \leq p-1$，那么

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

如果我们想验证 n 是否为素数，则可在中间选取 a，看上面等式是否成立。如果对于数值 a 等式不成立，那么 n 为合数；如果存在很多 a 能够使等式成立，那么我们可以说 n 有可能为素数，或者伪素数。

在检验过程中，有可能我们选取的 a 都能让等式成立，然而 n 却是合数。这时等式

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

若奇整数 n，若任取一整数 $2 \leq a \leq n-2$，$gcd(a,n)=1$，使得 $a^{n-1} \equiv 1 \pmod{n}$，则 n 至少有 1/2 的概率为素数。

判断过程：

1. 随机选取整数 a, $2 \leq a \leq n-2$
2. 计算 $gcd=(a,n)$，如果 gcd =1，转（3）；否则 n 为合数
3. 计算 $r=a^{n-1} \equiv 1 \pmod{n}$，如果 r=1,$n$ 可能是素数，转（1）；否则 n 为合数
4. 重复上述过程 k 次，如果每次得到 n 可能为素数，则 n 为素数的概率为 $1-\frac{1}{2^k}$

如果 $k=7$，那么 n 为素数的概率为 0.9921875。

```
#求两个数的最大公约数
def gcd(a,b):
    if b==0: return a
    else: return gcd(b, a%b)
#素性检验
def is_prime(num, k=7):
    for _ in range(k):
        a=random.randrange(2,num-2)
        if gcd(a, num)!=1:
            return False
        if pow(a,num-1,num)!=1:
            return False
    return True
```

该算法时间复杂度为 $O(k\cdot(log(n))^3)$，比试除法提高了很多。但该算法不能保证数字 n 百分之百为素数。

4.3 米勒-拉宾素性检验

米勒-拉宾素性检验 (Miller–Rabin Primality Test) 也是一种随机性检验。它是根据费马小定理

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

的逆命题修改而来的。

现在假设 n 是一个素数，那么 $n-1$ 就是一个偶数，可以写成 $2^s\cdot d$。其中 s 是正整数，d 为正整奇数。为了确定 n 是否为素数，选择一个整数 a，使得 $2\le a\le n-1$，并且对于任意的 $0<r<s-1$，若满足

- $a^d\not\equiv 1 \pmod{n}$
- $a^{2^r d}\not\equiv -1 \pmod{n}$

那么 n 就不是一个素数。

反过来说，如果以下式子满足其中一个:

- $a^d\equiv 1 \pmod{n}$
- $a^{2^r d}\equiv -1 \pmod{n}$

那么 n 很有可能是素数。

例 1

判断 $n=221$ 是否为素数。

$$n-1=2^s\cdot d=220=2^2\cdot 55$$

所以 $s=2,d=55,r=[0,1]$
随机从 $[2,n-1]$ 选择 a。假设 $a=125$, 那么：

$$a^{2^0d}\mod n=125^{55}\pmod{221}1=31\pmod{221}\neq\pm1\pmod{221}$$
$$a^{2^1d}\mod n=125^{2\cdot 55}\pmod{221}=77\pmod{221}\neq\pm1\pmod{221}$$

因为两次计算都不满足 $=\pm1\pmod{221}$，因此 $n=221$ 是一个合数。

□

例 2
判断 $n=131071$ 是否为素数。

$$n-1=2^s\cdot d=131070=2^1\cdot 65535$$

所以 $s=1,d=65535,r=[0]$
随机从 $[2,n-1]$ 选择 a。假设 $a=98$, 那么：

$$a^{2^0d}\mod n=98^{65535}\mod 131071=1\pmod{131071}$$

证明 n 很有可能是是一个素数。
我们进行再次随机选取 a。假设 $a=10003$, 那么：

$$a^{2^0d}\mod n=10003^{65535}\mod 131071=131070\pmod{131071}=-1\pmod{131071}$$

再次证明 n 很有可能是是一个素数。
重复选取 a 数次，提高准确率。

□

例 3
判断 $n=104513$ 是否为素数。

$$n-1=2^6\cdot 1633$$

所以 $s=6,d=1633,r=[0,1,2,3,4,5]$

$$\begin{aligned}
3^{1633} &\equiv 88958 \pmod n\\
3^{2\cdot 1633} &\equiv 88958^2\equiv 10430 \pmod n\\
3^{2^2\cdot 1633} &\equiv 10430^2\equiv 91380 \pmod n\\
3^{2^3\cdot 1633} &\equiv 91380^2\equiv 29239 \pmod n\\
3^{2^4\cdot 1633} &\equiv 29239^2\equiv 2781 \pmod n\\
3^{2^5\cdot 1633} &\equiv 2781^2\equiv -1 \pmod n
\end{aligned}$$

因此 $n = 104513$ 很有可能为素数。

□

没有证据表明米勒-拉宾素性检验对于素数不会做出错误判断。但一般来说，单次错误概率不超过 $\frac{1}{4}$。其单次检验比费马素性检验准确率高。并且当重复选取 a 达到 $k(k = 13)$ 次时，其错误概率能达到 $< 10^{-40}$ 的数量级。而工业级一般采取 64 次检验。

如果在 $n < 2^{64}$ ，选取 a = 2,325,9375,28178,450775,9780504,1795265022 共七个数就可以判定 n 是否为素数。

他在时间复杂度为 $O(klog^3n)$, 其中 k 是次数，n 是判定值的数值。如果使用快速傅里叶变换，能够达 $O(klog^2n)$。目前来说最为接近线性素数检验。

更多资料参考资料 1和 资料 2

```
# 判断s的最大值
for s in range(1000):
    r = 2**s
    a = 104513
    even = a - 1

    if even%r == 0 and s != 0:
        print('n-1 = ',even, ', s=',s , ', r的最大值是: ', s-1 ,'d= ', even/r )
```

```
#米勒-拉宾素性检验
import random

def power(x, y, p):
    """
    模幂运算。

    Parameters
    ----------
    x : int

    y : int

    p : int
        mod .

    Returns
    -------
    int
        (x^y) % p.

    """

    # 初始化结果
    res = 1
```

```
    # 如果x大于或等于p，更新x
    x = x % p
    while (y > 0):

        # 如果y是奇数，乘以x
        if (y & 1):
            res = (res * x) % p

        # 如果是y是偶数(也是必须的)
        y = y>>1 # y = y/2
        x = (x * x) % p

    return res

def miillerTest(d, n):
    """
    单次米勒-拉宾素性检验

    Parameters
    ----------
    d : int
        偶数，n-1拆分而来.
    n : int
        判定值.

    Returns
    -------
    bool
        可能是素数返回True,否则False.
    """

    # 随机在区间[2..n-2]内选择a
    a = 2 + random.randint(1, n - 4)# 极端情况下，确保n > 4


    x = power(a, d, n) # 计算 a^d % n

    if (x == 1 or x == n - 1):
        return True;

    # 继续求x的平方，而下面的情况不会发生
    # (i) d没有达到n-1
    # (ii) (x^2) % n ! = 1
    # (iii) (x^2) % n ! = n-1
    while (d != n - 1):
        x = (x * x) % n
        d *= 2
```

```
        if (x == 1):
            return False
        if (x == n - 1):
            return True

    return False

def isPrime( n, k):
    """
    判断是否是素数

    Parameters
    ----------
    n : int
        判定值
    k : int
        是一个确定精度等级的输入参数。k值越大，精度越高

    Returns
    -------
    bool
        n是合数则返回false，如果n可能是素数则返回true。

    """

    # 极端情况
    if (n <= 1 or n == 4):
        return False
    if (n <= 3):
        return True

    # 计算r和d
    d = n - 1;
    while (d % 2 == 0):
        d //= 2

    # 迭代k次
    for i in range(k):
        if (miillerTest(d, n) == False):
            return False

    return True

k = 64

print(isPrime(int(a), k))
print(isPrime(1000000000061, k))
```

```
print(isPrime(1000000000063, k))
print(isPrime(798263, k))
```

□

4.4 APRCL 算法

APRCL 算法 (Adleman–Pomerance–Rumely Primality Test) 由 Adleman、Pomerance、Rumely、Cohen 和 Lenstra 发明的时间复杂度为 $O\left(logN^{clogloglogN}\right)$。该算法有可能在相对合理的时间内检测出 1000 位整数的素性。

5 工业级素数

计算机领域中，如果把大素数生成算法想象成一个个生产素数的工厂的话，那么生产出来的素数就可以称为“工业级素数”。但目前的“工业级素数”仍存在为合数的概率，即有可能生产出“残次品”，虽然这样的残次品目前还未被发现。定义是没有经过严格数学证明、但是通过了“可能素数判定法”（米勒-拉宾素性检验）的整数。用这种方法判定大整数是否是素数并不完全准确，但速度非常快，而且不会漏掉任何素数。

下面简单给一种生成素数的 Python 方法：

```
#生成大素数
def largePrime_Generate(bit=1024):
    print("Generating large prime......")
    i=1
    while(True):
        num=random.randrange(2**(bit-1),2**(bit))
        print("第{}次随机生成大整数:{}.".format(i,num))
        if(isPrime(num,k)):
            print("大整数:{}通过Miller—Rabin素性检验说明很有可能为素数.".format(num))
            return num
        else:
            i+=1

def generateLargePrime_basedOnMR():
    while(True):
        try:
            bit=eval(input("请输入需要产生的大整数比特位数: "))
            largePrime_Generate(bit)
            break
        except:
            print("！！！请输入整数.")
if __name__=="__main__":
    generateLargePrime_basedOnMR()
```

□

这份代码只适用于研究目的，工业级素数可以使用OpenSSL生成密钥。

在通过了 64 轮米勒-拉宾测试后，该数不是素数的概率已经下降到了 2^{-128}，这对加密应用来说已非常足够，没有必要使用绝对正确但速度慢的 AKS 素数测试。

关于其他 RSA 的注意事项，可以参考RSA 有多安全，有多不安全?