信息安全数学基础个人探究报告

姓名：孙蕗　学号: 2112060

一、 素性检测

1. 数学问题

素性检测（Primality Testing）是指判断一个随机给定的数是否为素数的过程。

2. 算法思想及特点

除了暴力求解、素数表、Eratosthenes 筛法、欧拉线性筛法等方法，下面概述下其他常见的求解方法。

（1） Fermat 素性检验

A. 费马素性检测是一种基于费马小定理的素性检测方法。费马小定理指出，p 是素数时，对于任意整数 a，都有 $a^{p-1} \equiv 1(\mathrm{mod}\ p)$。思考费马小定理的逆定理，随机选取一小于 p 的整数，若 $a^{p-1} \equiv 1(\mathrm{mod}\ p)$，则是否可以证明 p 是一素数。但显然，存在一些合数也满足这个等式，例如最小的费马伪素数为 341。但发现，通过多选取几个底数，很大程度上可以降低错误的概率。

引入二次探测定理，对于质数 p，若 $x^2 \equiv 1(\mathrm{mod}\ p)$，小于 p 的解只有两个，$x_1=1$，$x_2=p-1$。二次探测定理可以通过判断平方剩余的性质来辅助判断一个数是否为素数。具体来说，如果对于一个待检测的数 n，选择一个随机数 a，并计算 $a^{\frac{n-1}{2}}(\mathrm{mod}\ n)$ 的结果，根据二次探测定理的性质，如果结果等于 1 或者等于 n-1，则 n 可能是素数；如果结果不等于 1 且不等于 n-1，则 n 一定是合数。

B. 基于费马小定理，我们可以进行如下的素性检测步骤：

a）选择一个待检测的数 p，确保 p 大于 2。

b）选择一个随机整数 a，其中 1 ＜ a ＜ p。

c）计算 $a^{p-1}(\mathrm{mod}\ p)$。

d）如果计算结果等于 1，该随机数无法提供关于 n 是否为素数的确凿证据，继续进行下一步。

e）如果计算结果不等于 1，那么该数 p 一定不是素数，结束算法。

f）重复步骤 B 至步骤 E 多次，选择不同的随机数 a 进行检测。如果对于每个随机数 a，都满足计算结果等于 1，那么该数 n 可能是素数，但不一定确定是素数。

费马素性检测是一个概率性算法，存在一定的错误率。可能存在一些伪素数，即合数但通过费马素性检测误判为素数的数。为了提高判断的准确性，可以选择多次进行费马素性检测，每次选择不同的随机数 a，并且根据需要进行多轮检测。

C. C++代码

a）以 b 为基的 Fermat 拟素性检验

算法 1–1 以 b 为基的 *Fermat* 拟素性检测

输入: 待测奇数 n, 基底 b, 下界 $i \geq 3$, 上界 $j > i$

输出: 待测奇数 n 是否是素数的判断 π

```
1: function FERMAT(n,b,i,j)
2:     for k = i to j do
3:         if b^n ≡ b(mod n) then
4:             π = 1
5:         else
6:             π = 0
7:         end if
8:     end for
9:     return π
10:    if π = 1 then
11:        n 是以 b 为基的概素数
12:    else
13:        n 是合数
14:    end if
15: end function
```


b）

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;
// 计算快速幂，用于计算 a^b mod m 的结果
long long fastModExp(long long a, long long b, long long m)
{
    long long res = 1; // 结果初始化为 1
    a = a % m; // 取 a 对 m 的余数
    while (b > 0)
    {
        if (b & 1) // 如果 b 的最低位为 1
            res = (res * a) % m; // 更新结果
        a = (a * a) % m; // a 自乘，取对 m 的余数
        b = b >> 1; // b 右移一位
    }
    return res;
}

// 费马素性检测函数
bool isPrimeFermat(long long n, int k)
{
    if (n <= 1) // 小于等于 1 的数不是素数
        return false;
    if (n <= 3) // 2 和 3 是素数
```

```
            return true;
        // 进行 k 次检测
        for (int i = 0; i < k; i++)
        {
            // 选择一个随机数 a，其中 1 < a < n-1
            long long a = 2 + rand() % (n - 3);
            // 计算 a^(n-1) mod n
            long long res = fastModExp(a, n - 1, n);
            // 如果计算结果不等于 1，那么 n 一定不是素数
            if (res != 1)
                return false;
        }
        return true; // 如果通过所有检测，n 可能是素数
    }

    int main()
    {
        srand(time(NULL)); // 初始化随机数种子
        long long n;
        int k;
        cout << "Enter a number: ";
        cin >> n;
        cout << "Enter the number of iterations: ";
        cin >> k;
        if (isPrimeFermat(n, k))
            cout << n << " is probably prime." << endl;
        else
            cout << n << " is composite." << endl;
        return 0;
    }
```

上述代码首先定义了一个 fastModExp 函数，用于快速计算幂运算的结果。然后定义了 isPrimeFermat 函数，用于执行费马素性检测。在 isPrimeFermat 函数中，通过循环进行 k 次检测，每次选择一个随机数 a，并计算 a^(n-1) mod n 的结果。如果计算结果不等于 1，那么 n 一定不是素数。如果通过所有检测，n 可能是素数。

在 main 函数中，用户可以输入待检测的数 n 和迭代次数 k，费马素性检测的时间复杂度为 O(k log n)。调用 isPrimeFermat 函数进行费马素性检测，并输出结果。

（2） 米勒-拉宾素性检测 Miller-Rabin primality test

A. 引入 Carmichael 数，是对于合数 n，如果对于所有与 n 互质的正整数 b，都有同余式 $b^{n-1} \equiv 1 (\mathrm{mod}\, n)$ 成立，则称合数 n 为 Carmichael 数。

米勒-拉宾素性检测是在费马素性检测算法基础上的改进，费马检验的问题在于没有把 Carmichael 数排除出去，Miller-Rabin 的做法就是将这个条件变得更加严格。

根据二次探测定理，对于质数 p，若 $x^2 \equiv 1 (\mathrm{mod}\, p)$，小于 p 的解只有两个，$x_1=1$，$x_2=p-1$。

即当 p 为素数时，方程 $x^2 \equiv 1(\mathrm{mod}\, p)$ 只有±1 两个解。

假设 n 为奇素数，令 n-1=2^lm，其中 l 是非负整数，m 是正奇数。如果 a 与 n 互素，则 $a^l \equiv 1(\mathrm{mod}\, n)$ 或 $\exists r, 0 \le r < l,$使得$a^{2^r m} \equiv -1(\mathrm{mod}\, n)$ 。如果 $a^{m2^r} \equiv 1(\mathrm{mod}\, n)$ ，则 $a^{m2^{r-1}} \equiv \pm 1(\mathrm{mod}\, n)$。可知，$a^m, a^{2m}, \ldots, a^{2^r m}$ 模 n 要么全都是 1，要么某个位置开始为±1，后面都是 1。

B. 米勒-拉宾素性检测的具体步骤

输入待检测的数 n 和迭代次数 k。

a）如果 n 为 2 或 3，直接返回素数。

如果 n 为偶数或小于 2，返回合数。

b）将 n-1 分解为 d * 2^s的形式，其中 d 是奇数。

c）随机选择一个整数 a，满足 2 ≤ a ≤ n-2。计算 x = a^d mod n。

d）如果 x 等于 1 或 x 等于 n-1，则进入下一次迭代。

e）对于 r 从 1 到 s-1，计算 x = x^2 mod n。

如果 x 等于 1，则返回合数；如果 x 等于 n-1，则进入下一次迭代。

返回合数。

f）重复 c 到 e，执行 k 次迭代。

C. C++代码

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;
// 快速幂取模，用于计算 a^b mod n
long long modular_exponentiation(long long a, long long b, long long n)
{
    long long res = 1;
    a = a % n;
    while (b > 0)
    {
        if (b & 1)
        {
            res = (res * a) % n;
        }
        a = (a * a) % n;
        b >>= 1;
    }
    return res;
}

// 米勒-拉宾素性检测函数
bool miller_rabin_test(long long n, int k)
```

```
    {
        if (n <= 1 || n == 4)
        {
            return false;
        }
        if (n <= 3)
        {
            return true;
        }
        // 将 n-1 表示为 d * 2^s 的形式
        long long d = n - 1;
        int s = 0;
        while (d % 2 == 0)
        {
            d /= 2;
            s++;
        }
        // 执行 k 次迭代
        for (int i = 0; i < k; i++)
        {
            // 随机选择一个整数 a，满足 2 ≤ a ≤ n-2
            long long a = 2 + rand() % (n - 3);
            // 计算 x = a^d mod n
            long long x = modular_exponentiation(a, d, n);
            // 如果 x 等于 1 或 x 等于 n-1，则进入下一次迭代
            if (x == 1 || x == n - 1)
            {
                continue;
            }
            // 对于 r 从 1 到 s-1，计算 x = x^2 mod n
            bool is_prime = false;
            for (int r = 1; r < s; r++)
            {
                x = modular_exponentiation(x, 2, n);
                // 如果 x 等于 1，则返回合数
                if (x == 1)
                {
                    return false;
                }
                // 如果 x 等于 n-1，则进入下一次迭代
                if (x == n - 1)
                {
                    is_prime = true;
```

```
                    break;
                }
            }
            // 如果在迭代中没有找到 n-1，则返回合数
            if (!is_prime)
            {
                return false;
            }
        }
        // 在所有迭代中都通过了检测，则返回可能为素数
        return true;
    }

    int main()
    {
        // 设置随机种子
        srand(time(0));
        long long n;
        int k;
        cout << "Enter a number to test for primality: ";
        cin >> n;
        cout << "Enter the number of iterations: ";
        cin >> k;
        if (miller_rabin_test(n, k))
        {
            cout << n << " is probably a prime number." << endl;
        }
        else
        {
            cout << n << " is composite." << endl;

        return 0;
    }
```

（3） Solovay-Strassen 素性检验

A. 该算法基于 Solovay-Strassen 定理，通过在模重复平方剩余上引入随机数的方法来进行判断。

设 p>2 是一个素数，则对任意 $(\frac{a}{p}) \equiv a^{\frac{p-1}{2}} (\mathrm{mod}\, p)$，其中 $(\frac{a}{p})$ 为 Jacobi 符号。

如果 n>2 是一个奇合数，则至少 50%的 $1 \le a \le n-1$，使得同余式 $(\frac{a}{n}) \equiv a^{\frac{n-1}{2}} (\mathrm{mod}\, n)$ 不

成立。

B. 检测步骤

a）随机均匀的选取$1 \le a \le n-1$

b）计算 gcd(a,n),如果$\gcd(a,n) \neq 1$，则 n 不是素数

c）计算$(\frac{a}{n}), a^{\frac{n-1}{2}} \pmod n$

如果$(\frac{a}{n}) \equiv a^{\frac{n-1}{2}} \pmod n$，则 n 很可能是素数，否则 n 不是素数。

C. C++代码

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;
// 快速幂取模，用于计算 a^b mod n
long long modular_exponentiation(long long a, long long b, long long n)
{
    long long res = 1;
    a = a % n;
    while (b > 0)
    {
        if (b & 1)
        {
            res = (res * a) % n;
        }
        a = (a * a) % n;
        b >>= 1;
    }
    return res;
}

// 计算雅可比符号
int jacobi_symbol(long long a, long long n)
{
    if (a == 0)
    {
        return (n == 1) ? 1 : 0;
    }
    if (a == 1)
    {
        return 1;
```

```
        }
        int e = 0;
        while (a % 2 == 0)
        {
            a /= 2;
            e++;
        }
        int s;
        if (e % 2 == 0 || n % 8 == 1 || n % 8 == 7)
        {
            s = 1;
        }
        else
        {
            s = -1;
        }
        if (n % 4 == 3 && a % 4 == 3)
        {
            s = -s;
        }
        if (a == 1)
        {
            return s;
        }
        return s * jacobi_symbol(n % a, a);
    }

    // Solovay-Strassen 素性检验函数
    bool solovay_strassen_test(long long n, int k)
    {
        if (n <= 1)
        {
            return false;
        }
        if (n == 2)
        {
            return true;
        }
        if (n % 2 == 0)
        {
            return false;
        }
```

```
        // 执行 k 次迭代
        for (int i = 0; i < k; i++)
        {
            // 随机选择一个整数 a，满足 1 < a < n
            long long a = 2 + rand() % (n - 2);

            // 计算雅可比符号和 a^((n-1)/2) mod n
            int jacobi = jacobi_symbol(a, n);
            long long exp = modular_exponentiation(a, (n - 1) / 2, n);

            // 如果雅可比符号和计算结果不相等，则返回合数
            if (jacobi == 0 || jacobi != exp)
            {
                return false;
            }
        }
        // 在所有迭代中都通过了检测，则返回可能为素数
        return true;
    }

    int main()
    {
        // 设置随机种子
        srand(time(0));
        long long n;
        int k;
        cout << "Enter a number to test for primality: ";
        cin >> n;
        cout << "Enter the number of iterations: ";
        cin >> k;
        if (solovay_strassen_test(n, k))
        {
            cout << n << " is probably a prime number." << endl;
        }
        else
        {
            cout << n << " is composite." << endl;
        }
        return 0;
    }
```

（4） Lucas-Lehmer 素性检验

A. Lucas-Lehmer 素性检验是一种用于验证梅森素数的方法，特别适用于检验大梅森素数的素性，但仅适用于梅森数，对其他形式的数不起作用。梅森数是指形如 2^p-1 的正整数，

其中指数 p 是素数。

引入定理，对于一个梅森数 M_p，有这样的一个数列 $L_{n+1}=L_n^2-2$，$L_n=4$，当且仅当 $L_{p-2}\equiv 0(\mathrm{mod}\ p)$ 时，M_p 为素数。

B. Lucas-Lehmer 素性检验算法步骤

1） 输入潜在的梅森素数 n，计算 $p=\log_2 n+1$。

2） 对 i 从 0 到 p-2，根据 $L_0=4$，计算 $L_{i+1}=L_i^2-2$

3） 如果 $L_{p-2}\equiv 0(\mathrm{mod}\ n)$，则 n 为素数，否则 n 为合数

C. C++代码

```
#include <iostream>
#include <cmath>
// 函数用于判断给定的数是否为素数
bool isPrime(int num)
{
    if (num <= 1)
    {
        return false;
    }
    int sqrtNum = sqrt(num);
    for (int i = 2; i <= sqrtNum; i++)
    {
        if (num % i == 0)
        {
            return false;
        }
    }
    return true;
}
// Lucas-Lehmer 素性检验函数
bool lucasLehmerTest(int p)
{
    if (p <= 2)
    {
        return false;
    }
    int mersenneNum = pow(2, p) - 1; // 计算梅森数
    if (!isPrime(p) || mersenneNum <= 2)
    {
        return false;
    }
    int s = 4; // 初始化序列的初始值
```

```
        for (int i = 1; i < p - 1; i++)
        {
            s = (s * s - 2) % mersenneNum; // 序列元素的递推计算
        }
        return (s == 0); // 如果最终序列的值为0，则梅森数为素数，否则不是素数
    }
    int main()
    {
        int p;
        std::cout << "Enter a prime number (p): ";
        std::cin >> p;
        if (lucasLehmerTest(p))
        {
            std::cout <<"Mersenne number (2^" << p <<" - 1) is prime." << std::endl;
        }
        else
        {
            std::cout << "Mersenne number (2^" << p << " - 1) is not prime." <<
std::endl;
        }
        return 0;
    }
```

3. 密码学的应用

A. 密钥生成和共享

在某些加密算法中，素性检测可以用于生成密钥或加密算法的参数。通过检测随机数的特定素性，如素数性质、离散对数问题等，可以生成强大的密钥，增加密码算法的安全性。

一些密钥交换协议（如 Diffie-Hellman 密钥交换）和密钥生成算法（如 ElGamal 加密算法）需要使用素数来生成和共享密钥。

Diffie-Hellman 密钥交换协议允许两个参与方在公开信道上安全地协商一个共享密钥。该协议基于离散对数问题的难度，其中一个重要的步骤是选择一个素数 p 和一个原根 g 作为参数。参与方使用这些参数进行计算，并交换计算结果，最终得到一个共享的密钥。素性检测用于验证选择的素数 p 是否符合加密协议的要求，以确保生成的密钥对的安全性和有效性。

ElGamal 加密算法是一种公钥加密算法，其中涉及到选择一个素数 p 和一个生成元 g 作为参数。算法中的私钥是一个随机选择的整数，公钥则是通过对私钥进行一系列运算得到的。素性检测用于验证选择的素数 p 是否满足加密算法的要求，以确保生成的密钥对的安全性和有效性。

在这些密钥生成和共享的算法中，素性检测的作用是确保使用的素数满足密码学算法的要求。如果选择的素数不是素数，可能会导致算法的弱点和安全性问题。通过进行素性检测，可以排除那些不满足素数性质的数，从而确保生成的密钥的安全性和有效性。

B. RSA 加密算法

在 RSA 算法中，素性检测用于选择合适的大素数 p 和 q 作为私钥的一部分。在 RSA 算法中，私钥由两个大素数 p 和 q 的乘积组成。这些素数的选择必须满足一定的条件以确保加密的强度和安全性。

选取的素数必须满足以下条件：

难以分解：RSA 算法的安全性基于大数分解的困难性。因此，选取的素数 p 和 q 必须足够大，以使得对它们进行分解变得极其困难，即找到它们的质因数。素性检测用于验证选择的大数是否为素数，从而确保其难以分解。

相对难以预测：选择的素数 p 和 q 应该是相对难以预测的，以防止攻击者通过猜测或推断来获取私钥。素性检测有助于确保选取的素数具有一定的随机性和难以预测性。

差异性：选择的素数 p 和 q 应该是不同的，以增加攻击者对私钥的推断难度。素性检测可以帮助验证选择的素数是否具有足够的差异性，以满足这一要求。

通过素性检测，可以验证选择的素数是否符合这些条件。素性检测算法可以识别不符合素数条件的大数，并确保选取的素数足够强大和安全，以保护加密系统免受质因数分解等攻击。

C. 椭圆曲线密码学

在椭圆曲线密码学中，素性检测被用于验证素数域和有限域中的素数。这些素数用于定义椭圆曲线的参数，从而影响密码算法的安全性和性能。通过素性检测，可以确保使用的素数满足密码学算法的要求，避免潜在的漏洞和攻击。

素性检测的目的是验证选择的素数是否满足密码学算法的要求。它确保使用的素数具有以下特性：

素数性质：验证素数的基本性质，即不能被其他数整除，除了 1 和自身。这样可以确保所选择的素数是独特的，不可约分的。

充分大：素数应该足够大，以保证椭圆曲线上的离散对数问题的困难性。素性检测可以验证所选素数的大小，以确保它们足够强大，使得攻击者难以通过穷举法或其他方法破解密码算法。

安全性要求：椭圆曲线密码学的安全性与选择的素数的大小和特定属性相关。素性检测可以确保所选素数满足密码学算法的安全性要求，如满足特定的素性检测定理或密码算法的参数要求。

一、 RSA 问题

1. 数学问题

RSA 问题是 Rivest、Shamir、Adleman 三位密码学家于 1978 年在注明的 RSA 公钥密码体制中提出的，其反映了 RSA 加密算法（数字签名算法）的安全等级。尽管目前仍然没有人能够证明解决 RSA 问题的困难度与解决整数分解问题的困难度相同，但普遍认为 RSA 问题是一个困难问题。

RSA 是一种非对称加密算法，使用不同的密钥进行加密和解密。公钥用于加密数据，私钥用于解密数据。这种非对称性使得 RSA 算法在数据安全和身份验证方面非常有用。RSA 算法基于数论的难题，通过大数分解问题和模反问题的困难性，提供了安全的加密和解密机制。RSA 算法的主要步骤包括生成公钥、私钥、加密、解密操作。

RSA 算法的安全性基于大数分解问题和模反问题的困难性。大数分解问题指的是将大合数分解为其素因子的问题，而模反问题是在已知公钥和密文的情况下找到对应的明文的问题。目前，最好的已知方法是使用大整数分解算法和模反算法来解决这些问题，但对于足够大的密钥长度，这些问题仍然非常困难，需要大量计算和时间。

RSA 问题：令 n=pq 是一个成熟合数，e 是一个正奇数且满足 $(e,\varphi(n))=1$。给定一个随机整数 $c\in Z_n^*$，将寻找一个整数 m 使其满足 $m^e\equiv c(\mathrm{mod}\, n)$ 的问题。

在 RSA 问题的基础上，提出了强 RSA 问题。

强 RSA 问题：令 n=pq 是 RSA 问题中的模数，G 是 Z_n^* 的一个循环子群。给定 G 中的一个随机元素 z，寻找一组整数 $(u,e)\in G\times Z_n$，使其满足 $z\equiv u^e(\mathrm{mod}\,n)$ 的问题。

RSA 问题和强 RSA 问题形式上的不同使得二者在密码学中用于构造不同的密码学方案，RSA 问题通常用于构造公钥加密方案，强 RSA 问题通常用于构造群签名方案。

2. 算法思想及特点

RSA 算法具体描述如下：

（1） 密钥生成：

a) 随机选择两个大素数 p 和 q，通常选择的素数长度在几百位到几千位之间。

b) 计算两个素数的乘积 n = p×q，并计算欧拉函数 $\varphi(n)=(p-1)\times(q-1)$。

c) 选择一个整数 e，满足 $1<e<\varphi(n)$，且 e 与 Φ(n)互质，$\gcd(e,\varphi(n))=1$。e 作为公钥的一部分，记为(public_key, e)。

d) 计算 e 的模反元素（解密钥）d，满足 $de\equiv 1(\mathrm{mod}\,\varphi(n))$，即 $de=k\varphi(n)+1$，k>=1 是一个任意的整数。d 作为私钥的一部分，记为(private_key, d)。

公钥(public_key, e)和私钥(private_key, d)对是一对密钥。

（2） 加密操作：

将明文消息转换为整数 m，使得 0 ≤ m < n。将明文 m（m<n 是一个整数）加密成密文 c，计算密文 $c=E(m)=m^e(\mathrm{mod}\,n)$，mod 表示取模运算。

密文 c 即为加密后的消息，可传输给接收方。

（3） 解密操作：

接收方使用私钥(private_key, d)来解密密文 c。

将密文 c 解密成明文 m，计算明文消息 $m=D(c)=c^d(\mathrm{mod}\,n)$。

明文消息 m 即为解密后的原始消息。

但是只根据 n 和 e（不是 p 和 q）要计算出 d 是不可能的，任何人都可以对明文进行加密，但只有知道 d 的人才能对密文解密。

（4） 数字签名生成

使用私钥(private_key, d)将要签名的信息进行哈希运算，得到消息摘要。

对消息摘要进行加密，计算签名 $s=\text{摘要}^d(\mathrm{mod}\,n)$，将消息和签名一起传输给接收方。

（5） 数字签名验证

接收消息和签名后，使用公钥(public_key, e)对签名进行解密，得到解密后的摘要。摘要 $m'=\text{签名}^e(\mathrm{mod}\,n)$。

对接受到的消息进行哈希运算，生成消息的摘要。

比较计算得到的摘要 m’和接收到的摘要是否相同。如果两个摘要值相同，说明签名验证通过，消息未被篡改。

尽管 RSA 算法具有许多优点，但在某些特定情况下可能不适合使用。例如，由于计算复杂性较高，RSA 不适合用于加密大量数据。在这种情况下，通常会使用对称加密算法来加密

数据，然后使用 RSA 算法来加密对称密钥。这种组合使用对称和非对称加密的方式可以充分发挥各自的优势。

3. 密码学的应用

RSA 问题和强 RSA 问题形式上的不同使得二者在密码学中用于构造不同的密码学方案，RSA 问题通常用于构造公钥加密方案，强 RSA 问题通常用于构造群签名方案。

RSA 算法可用于对数据进行加密和解密，保护数据的机密性。发送方使用接收方的公钥对数据进行加密，只有接收方持有相应的私钥才能解密数据。这种应用在保护敏感数据的传输和存储过程中非常重要。

RSA 算法可用于安全地进行密钥交换。通过使用对方的公钥对生成的随机密钥进行加密，发送方可以将密钥安全地传输给接收方，而不用担心被中间人攻击窃取密钥。这种应用在建立安全通信通道时非常重要。

RSA 算法可用于对存储在设备或服务器上的数据进行加密和解密，确保数据在存储过程中的安全性。通过使用 RSA 算法加密存储的数据，即使设备或服务器被盗或遭受攻击，数据仍然是加密的，难以被窃取或篡改。

RSA 算法可以用于生成和验证数字签名。发送方可以使用私钥生成数字签名，并将其与消息一起发送。接收方可以使用发送方的公钥验证签名的有效性，从而确保消息的完整性和认证发送方的身份。

RSA 算法常用于生成和验证数字证书。数字证书用于认证和验证实体（如网站、组织、个人）的身份。例如在网站登录、支付等场景中，可以使用 RSA 算法来加密和验证用户的身份信息，确保用户的身份安全。证书包含了实体的公钥和相关信息，并由证书颁发机构签名，接收方可以使用 RSA 算法验证证书的合法性。

RSA 算法在 SSL/TLS 协议中扮演重要角色。SSL/TLS 协议用于加密网站和应用程序的通信，确保通信的保密性和安全性。RSA 算法用于生成和交换会话密钥，用于对通信数据进行对称加密。

二、 安全多方计算

安全多方计算最早于 1982 年被姚期智提出，也就是为人熟知的百万富翁问题：两个富翁在街头相遇，他们如何在不暴露各自财富不借助第三方的前提下比较出谁更富有？姚氏“百万富翁问题”后经发展，成为现代密码学中非常活跃的研究领域，即安全多方计算。

安全多方计算用于解决一组互不信任的参与方各自持有自己的秘密数据，协同计算一个既定函数的问题。多方计算的目标就是对一组每个计算的参与者，能够在不暴露自己的私密数据的情况下进行计算，保证参与方都能获得正确计算结果，但不能获得计算结果之外的任何信息，不知道其他参与者的输入。在整个计算过程中，参与方对其所拥有的数据始终拥有绝对的控制权。其基本思想是通过使用密码学技术和协议，在计算过程中保持数据的加密状态，确保计算结果是正确的，同时隐藏每个参与者的私密输入。

在安全多方计算中，根据参与方的可信程度可以建立几种安全模型。

理想模型 ：在理想模型中，每一个参与方都是可信的，一方将其信息发送给另一方，另一方不会去查看这份信息，只会根据规定计算出结果，并发送给下一方或者所有参与方。

半诚实模型 ：半诚实模型就是参与方会诚实的运行协议，但是他会根据其它方的输入或者计算的中间结果来推导额外的信息。

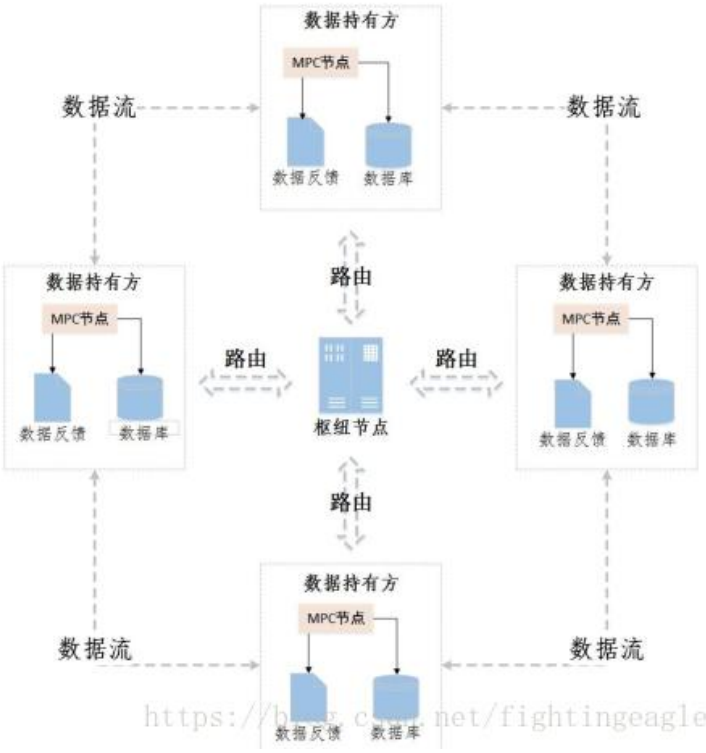
隐蔽的攻击模型：一个隐蔽的对手可能表现出恶意行为，但它有一定的概率被诚实的参与者发现作弊。这一模型代表了许多金融或政治环境，在这些环境中，诚实的行为是不可能假设的，但所涉及的公司和机构不能承受与被发现作弊有关的名誉损失。

恶意攻击模型 ：恶意模型则可能不会诚实的运行协议，甚至会搞破坏。

主流两方安全计算框架的核心用了加密电路和不经意传输这两种密码学技术：一方将计

算逻辑转化为布尔电路，针对电路中每个门进行加密处理。接下来，该参与方将加密电路（即计算逻辑）和加密后的标签输入给下一个参与方。另一方作为接收方，通过不经意传输按照输入选取标签，对加密电路解密进行解密获取计算结果。通用的多方安全计算框架可以让多方安全地计算任意函数或一类函数的结果，自从姚期智提出第一个通用的安全多方框架以来，30 多年已陆续有了 BMR、GMW、BGW、SPDZ 等，这些多方安全计算框架涉及到加密电路、秘密分享、同态加密、不经意传输等相关技术。

安全多方计算技术框架：



1. 密码原语

（1） 同态加密

同态加密（Homomorphic Encryption）是一种特殊的加密技术，允许在加密状态下对密文进行计算，得到与对应的明文计算结果相对应的密文结果。在整个过程中，数据是以密文形式处理的，不需要解密。换句话说，同态加密允许在不解密密文的情况下进行计算，并在密文域中得到加密结果。

传统的加密算法，如对称加密算法和非对称加密算法，不具备同态性质。对密文进行计算通常需要将密文解密为明文，执行计算操作，然后再将计算结果加密为密文。这样的过程会暴露密文的内容，可能会导致数据的机密性丧失。同态加密的出现解决了这一问题，它可以在密文域中进行加法、乘法或其他计算操作，并保持计算结果的加密状态。

（2） 哈希函数

哈希函数可以将任意长度的输入数据映射为固定长度的哈希值。它常用于验证数据的完整性和一致性，以及在协议中生成随机数或者生成公开的承诺值。

从哈希值推导出原始输入应该是计算上不可逆的；对于不同的输入，其哈希值应该是唯一的；输出应该均匀分布在输出空间中，即对于一个随机的输入，哈希值应该看起来是随机的。

（3） 密钥分享

密钥分享的基本思路是将每个数字 x 拆散成多个数 x1,x2,… ,xn，并将这些数分发到 n

个参与方。每个参与方拿到的都是原始数据的一部分，一个或少数几个参与方无法还原出原始数据，只有大家把各自的数据凑在一起时才能还原真实数据。

计算时，各参与方直接用它自己本地的数据进行计算，并且在适当的时候交换一些数据（交换的数据本身看起来也是随机的，不包含关于原始数据的信息），计算结束后的结果仍以 secret sharing 的方式分散在各参与方那里，并在最终需要得到结果的时候将某些数据合起来。只有在满足特定条件的情况下，这些参与者才能合作恢复原始密钥。这样的话，密钥分享便保证了计算过程中各个参与方看到的都是一些随机数，但最后仍然算出了想要的结果。

Shamir's Secret Sharing 是一种经典的密钥分享方案，由 Adi Shamir 于 1979 年提出。它基于多项式插值的原理，将密钥拆分为多个密钥片段，并将这些片段分发给不同的参与者。只有当足够数量的参与者合作时，才能重构出原始密钥。该方案使用了多项式的性质来实现安全性和恢复性。

阈值密码学是一类密钥分享方案的集合，其中包括 Shamir's Secret Sharing 作为其中一种特例。它的特点是将密钥分享成多个部分，并指定一个阈值。只有当达到或超过阈值数量的参与者合作时，才能恢复出原始密钥。阈值密码学可以应用于各种密码算法和安全协议中，如对称加密、非对称加密和数字签名等。

（4） 零知识证明

零知识证明是一种协议，用于让参与者能够向其他参与者证明某个陈述为真，而不需要透露相关的私密信息。零知识证明在安全多方计算中用于确保参与者提供的输入是合法和正确的，同时不泄露其他私密信息。证明者通过与验证者的交互，能够说服验证者该陈述为真，但验证者无法从交互中获得任何有关陈述的额外信息。

零知识证明通常涉及证明者与验证者之间的交互过程。这个过程可以包括多轮的挑战-回应协议，其中验证者提出挑战，证明者根据挑战生成回应，并不断进行交互。通过这种交互，证明者逐步建立了对陈述真实性的信任，而验证者则逐渐确认陈述的真实性。零知识证明的完备性要求，如果陈述为真，则证明者可以成功地说服验证者接受陈述的真实性。而正确性要求，如果陈述为假，则无论证明者如何尝试，验证者不会被欺骗为陈述为真。

（5） 差分隐私

同态加密、混淆电路、密钥分享都属于非噪音方法。这些方法一般是在源头上就把数据加密或编码了，计算操作方看到的都是密文，因此只要特定的假设条件满足，这类方法在计算过程中是不会泄露信息。

在安全多方计算中，多个参与者可能希望合作进行数据分析，但担心泄露自己的私密数据。差分隐私可以通过对数据添加一定的随机噪声来保护个体数据的隐私，同时保持数据分析的有效性。参与者可以在保护隐私的前提下，共享经过差分隐私保护的数据，并进行联合分析。

2. 应用场景

安全多方计算兼具理论研究和实际应用价值，在电子投票、隐私保护的数据挖掘、机器学习、区块链、生物数据比较、云计算等领域有着广泛的应用前景。

（1） 投票场景

在选举或民意调查中，安全多方计算可用于设计匿名投票系统。参与者可以投票并通过安全多方计算确保选票的安全性和隐私性，同时保证选举结果的准确性。

比如在电子投票领域，投票人的投票结果可能被恶意获取篡改，安全性和隐私性会受到一定的影响。因此电子投票系统必须保证投票人被完整正确地提交，并且投票人的投票信息不被除了计票人外的其他人获取。安全多方计算为这种分布式环境下如何进行保护隐私信息和确保结果正确性的问题提供了良好解决思路。

比如,Cramer 等人基于 ElGamal 门限加密技术和零知识证明提出了首个多选一电子投票方案，之后 Damgard 等人基于 Pailier 同态加密技术提出了多选多的电子投票方案。在1992 年，A.Fujioka 等使用盲签名技术提出了著名的 FOO 电子投票协议。

（2） 数据保护与合作分析

在金融领域，安全多方计算可以用于合并数据以进行风险评估、投资决策等任务，同时保护参与者的敏感交易和投资信息。 iCube 团队通过与美国普渡大学区块链人工智能实验室深度合作，依托社区网络和数据网络，通过支持安全多方的分布式计算技术和面向个人的人工智能引擎，形成一种自维护、自发展、自运行的全新数字经济生态，实现了区块链的安全多方计算。iCube 建立了面向信息的终极抽象基础层和基于个人工智能的算法模型层，内置图灵完备编程语言和自主开发的 MPC 算法沙盒，从而实现了区块链的多方安全计算。iCube 完全自主开发了一套可以支持联合计算并保护参与者私密的协议，并将该协议添加到区块链的最底层,从而实现了各个节点在信息隐私保护的前提下实现数据联合共享计算的功能。

在医疗健康领域，参与者可以通过安全多方计算共享病历、基因数据以及其他病人敏感信息，在保护各方数据信息不被泄露的同时多方协作完成数据挖掘，以促进疾病研究、个性化医疗等。

在私密计算时，参与者拥有敏感数据，希望进行某些计算操作，但不希望将数据暴露给其他参与者。多方安全计算可用于联合分析，允许不同组织或个体在保护隐私的前提下共享数据和计算结果，可以确保计算过程在不泄露私密数据的情况下进行，只输出计算结果给参与者，防止其他参与者获知个体的敏感信息，进行跨边界的合作分析。

在云计算环境中，用户可以通过安全多方计算将计算任务委托给云服务提供商，同时保护数据的隐私，确保云服务提供商无法访问用户的敏感数据。

（3） 身份验证

在需要多个参与者进行身份验证的场景中，例如跨机构的身份验证系统，安全多方计算可以用于协作验证参与者的身份，而无需暴露敏感的身份信息。

（4） 机器学习

在机器学习想要取得好的效果，需要大量数据进行模型训练。训练数据的隐私保护同样是问题。多个数据持有方希望合作进行机器学习模型的训练，但不愿意共享原始数据。通过安全多方计算,数据持有方可以在不泄露数据的情况下,共同训练模型并获得联合学习结果。