信息安全数学基础个人探究报告

姓名：龚子超 学号：2312005 专业：信息安全 班级：信息安全 1 班

一、素性检测

（一）背景与定义

很多公钥密码算法（例如 RSA 算法、ElGamal 算法和 ECC 算法）都需要用到大素数，因此，快速生成指定位数的大素数十分重要，其一般步骤如下：

1. 随机生成一个指定位数的大奇数 p，作为候选素数；
2. 判断候选素数 p 是素数还是合数。

显而易见，最重要的一步就是判断候选素数 p 是素数还是合数。由此引出素性检测的定义：给定一个随机整数，判断其是否是一个素数的过程。

（二）素性检测算法

1. 分类

素性检测算法可分为两类：

一是确定性素性检测算法：可以明确判断候选素数 p 是否为素数，但其计算复杂度较高。

二是概率素性检测算法：若判断候选素数 p 是合数，则判 p 必为合数。反之则两者皆有可能。

本文只讲解概率素性检测算法的实现。

2. 基础数学知识

(1) 引理：n 为素数，对于任意 $b\in\mathbb{Z}^*_n$，有 $b^{\frac{n-1}{2}}\equiv(\frac{b}{n})(\mathrm{mod}\ n)$，其中 $(\frac{b}{n})$ 是雅可比符号。

(2) 欧拉伪素数：对于奇合数 n，$(b, n)=1$，如果 $b^{\frac{n-1}{2}}\equiv(\frac{b}{n})(\mathrm{mod}\ n)$ 成立，则称 n 为关于 b 的欧拉伪素数。

(3) 定理：令 n 为奇合数，那么在 $\mathbb{Z}^*_n$ 中至少存在一半的数使式(2)成立。

(4) 强伪素数：令 n 为奇合数，$n-1=2^st$，$2\nmid t$，$s\geqq 1$.令 $b\in b\in\mathbb{Z}^*_n$，$(b, n)=1$.如果对于某个 $r\ (0\leqq r<s)$，有 $b^t\equiv 1\ (\mathrm{mod}\ n)$ 或 $b^{2^rt}\equiv -1\ (\mathrm{mod}\ n)$，则称 n 为关于基 b 的强伪素数。

(5) 定理：令 n 为奇合数,b∈$\mathbb{Z}^*_n$，那么 n 为关于基 b 的强伪素数的概率不超过 1/4。

（三）Solovay-Strassen 算法

1. 算法原理

(1) 先输入奇整数 n>=3 和安全参数 t（其实就是计算次数）

(2) 随机选取整数 b ，b 的范围[2，n-2]。

(3) 计算 r =$b^{\frac{n-1}{2}}$(mod n)。

(4) 如果 r != 1 以及 r != n-1，则 n 是合数。

(5) 计算 Jacobi 符号 s=$(\frac{b}{n})$。

(6) 如果 r != s(mod n)，则 n 是合数。

(7) 上述过程重复 t 次。

2. 代码实现

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;
// 计算 (a/b) % p 的雅可比符号
int jacobi(int a, int b) {
    if (b <= 0 || b % 2 == 0) return 0; // 非法输入或b是偶数
    int j = 1;
    if (a < 0) {
        a = -a;
        if (b % 4 == 3) j = -j;
    }
    while (a != 0) {
        while (a % 2 == 0) {
            a /= 2;
            if (b % 8 == 3 || b % 8 == 5) j = -j;
        }
        swap(a, b);
        if (a % 4 == 3 && b % 4 == 3) j = -j;
        a %= b;
    }
    if (b == 1) return j;
    return 0;
}
```

```cpp
// 快速幂模运算
int mod_exp(int base, int exp, int mod) {
    int result = 1;
    base = base % mod;
    while (exp > 0) {
        result = (result * base) % mod;
        exp--;
    }
    return result;
}

// Solovay-Strassen素性检验法
bool solovay_strassen(int n, int k) {
    if (n < 2) return false;
    if (n != 2 && n % 2 == 0) return false;
    srand(time(NULL));
    for (int i = 0; i < k; i++) {
        int a = rand() % (n - 3) + 2;
        int x = jacobi(a, n);
        int y = mod_exp(a, (n - 1) / 2, n);
        if (x == 0 || y != (x + n) % n) return false;
    }
    return true;
}

int main() {
    int n, k;
    cout << "请输入要测试的数: ";
    cin >> n;
    cout << "请输入测试次数: ";
    cin >> k;
    if (solovay_strassen(n, k)) {
        cout << n << " 可能是一个素数。" << endl;
    }
    else {
        cout << n << " 不是一个素数。" << endl;
    }
    return 0;
}
```

（四）Miller-Rabin 算法

1. 算法原理

Miller-Rabin 素性检验的核心思想是利用费马小定理和一些数学性质来检测一个数是否为合数。具体步骤如下：

(1) **基本检查：**
- 如果 n 小于等于 1，则 n 不是素数。
- 如果 n 是 2 或 3，则 n 是素数。
- 如果 n 是偶数且不等于 2，则 n 不是素数。

(2) **分解 n-1：**
- 将 n-1 写成 $2^s \cdot d$ 的形式，其中 d 是奇数。

(3) **选择基数 a：**
- 随机选择一个整数 a，满足 $2 \leqslant a < n-2$。

(4) **计算 x：**
- 计算 $x=a^d \pmod n$

(5) **检查条件：**
- 如果 x=1 或 x=n-1，则 n 可能是素数。
- 否则，对于 r=0 到 s-1 重复以下步骤：
  - 计算 $x=x^2 \pmod n$。
  - 如果 x=n-1，则 n 可能是素数。
- 如果上述条件都不满足，则 n 是合数。

(6) **重复测试：**
- 为了提高准确性，可以多次选择不同的基数 a 进行测试。如果所有测试都通过，则 n 被认为是素数的概率很高。

2. 代码实现

```
#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;

// 快速幂算法 (a^d % n)
long long mod_pow(long long a, long long d, long long n) {
    long long result = 1;
    a = a % n;
    while (d > 0) {
        if (d % 2 == 1) {
            result = (result * a) % n;
        }
```

```
        d = d >> 1;
        a = (a * a) % n;
    }
    return result;
}

// Miller-Rabin 素性测试
bool miller_rabin_test(long long n, int k) {
    if (n <= 1 || n == 4) return false;
    if (n <= 3) return true;

    // 将 n-1 写成 2^s * d 的形式
    long long d = n - 1;
    while (d % 2 == 0) {
        d /= 2;
    }

    // 进行 k 次测试
    for (int i = 0; i < k; i++) {
        // 随机选择一个基数 a
        long long a = 2 + rand() % (n - 4);

        // 计算 a^d % n
        long long x = mod_pow(a, d, n);

        if (x == 1 || x == n - 1) continue;

        bool composite = true;
        while (d != n - 1) {
            x = (x * x) % n;
            d *= 2;

            if (x == 1) return false;
            if (x == n - 1) {
                composite = false;
                break;
            }
        }

        if (composite) return false;
    }

    return true;
}
```

```
int main() {
    srand(time(0)); // 设置随机数种子

    long long n;
    int k;
    cout << "请输入待测试的整数 n: ";
    cin >> n;
    cout << "请输入测试次数 k: ";
    cin >> k;

    if (miller_rabin_test(n, k)) {
        cout << n << " 可能是素数。" << endl;
    }
    else {
        cout << n << " 是合数。" << endl;
    }

    return 0;
}
```

二、大整数分解问题

(一) 定义

给定一个大整数 N,他是两个大素数的乘积，但其因子 p,q 未知，我们将寻找 p 和 q，使其满足 N=p*q 的问题称作大整数分解问题。

(二) 算法思想及特点

1. 试除法（Eratosthences 筛法）

1) 算法原理：

a) 给出一个大整数 N。

b) 对小于$\sqrt{N}$的素数一个个地去验证是否整除 N。

c) 如果 p 整除 N，则同时可知 q=N/p。

d) 故可得 N=p*q。

2) 代码实现：

```
#include <iostream>
using namespace std;
```

```
//判断是否为素数
bool judge(int m) {
    if (m < 2) { return false; }
    else {
        for (int i = 2; i * i <= m; i++) {
            if (m % i == 0) { return false; }
        }
        return true;
    }
}
void Eratosthences(int n,int &p,int &q) {
    for (int i = 2; i * i <= n; i++) {
        if (judge(i)&&n%i==0) {
            p = i, q = n / i;
        }
    }
 }

int main() {
    int n, p, q; cin >> n;
    Eratosthences(n, p, q);
    cout << n << "=" << p << "*" << q;


   return 0;
      }
```

3） **算法评价：**

显而易见，该算法的时间复杂度为 $O(\sqrt{N})$，当 N 比较小的话，我们还是能够在一定的时间内将其进行分解，但是如果 N 当很大的时候，我们就不能够对其进行分解了。所以在大整数的分解问题中，我们往往不采用这种暴力的算法。

2. 费马分解算法

1） **算法原理：**

a） 给出一个大整数 N。

b） 选择一个整数 a，通常从$\sqrt{N}$开始。

c） 计算 b^2 = a^2-N，其中 b 是另一个因子。

d） 如果 b^2 是一个完全平方数，则 a 和 b 都是 n 的因子。

e） 如果 b^2 不是完全平方数，增加 a 的值并重复步骤 2 和 3。

f） 故可得 N=（a+b）*（a-b）。

2） 代码实现：

```
#include <iostream>
#include <cmath>
using namespace std;

// 检查一个数是否为完全平方数
bool isPerfectSquare(int x) {
    int s = sqrt(x);
    return s * s == x;
}

// 使用费马分解算法分解整数n
void fermatFactorization(int n) {
    int a = ceil(sqrt(n)); // 从sqrt(n)开始
    while (true) {
        int b2 = a * a - n; // 计算b^2
        if (isPerfectSquare(b2)) { // 检查b^2是否是完全平方数
            int b = sqrt(b2);
            cout << "Factors of " << n << " are: " << a - b << " and " << a + b << endl;
            break;
        }
        a++; // 如果不是，增加a的值
    }
}

int main() {
    int n;
    cout << "Enter a number to factorize: ";
    cin >> n;
    fermatFactorization(n);
    return 0;
    }
```

3） 算法评价：

费马分解算法的基本思想相对简单，容易在编程中实现，对于某些特定形式的合数（例如那些可以表示为两个较小整数平方和的数），费马分解算法可能非常有效。但对于大多数合数而言，找到合适的 a 和 b 的值可能非常耗时，尤其是当处理非常大的数字时。这会导致算法的效率极低。

3. Pallard ρ分解算法

1） 算法原理：

a） 给出一个大整数 N。

b） 选择一个适当的多项式函数 f(x) =x^2 +c(mod n)，其中 c 是一个常数。

c） 选择一个起始值 x0，并设置两个变量 x 和 y，初始都为 x0。。

d） 重复以下步骤直到找到因子：

计算 x = f(x) mod n。

计算 y = f(f(y)) mod n。

计算 gcd(|x - y|, n)，如果结果大于 1，则它是 n 的一个非平凡因子。

2） 代码实现：

```
#include <iostream>
#include <cmath>
using namespace std;

// 计算最大公约数
int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a % b);
}

// Pollard's Rho算法
void pollardRhoFactorization(int n) {
    if (n % 2 == 0) {
        cout << "Factor found: 2" << endl;
        return;
    }
    int x = rand() % (n - 2) + 2; // 随机选择x0
    int y = x;
    int c = rand() % (n - 1) + 1; // 随机选择c
    int d = 1;
    while (d == 1) {
        x = (x * x % n + c + n) % n; // f(x) = x^2 + c
        y = (y * y % n + c + n) % n; // f(y) = y^2 + c
        y = (y * y % n + c + n) % n; // f(f(y)) = f(y)^2 + c
        d = gcd(abs(x - y), n);
        if (d == n) return; // 失败，重新开始
    }
```

```
    cout << "Factor found: " << d <<" and " <<n/d<< endl;
}

int main() {
    int n;
    cout << "Enter a number to factorize: ";
    cin >> n;
    pollardRhoFactorization(n);
    return 0;
}
```

3） **算法评价：**

对于某些类型的合数，特别是那些可以表示为两个相近素数乘积的数，Pollard ρ 算法非常有效。算法的结果依赖于随机选择的参数，有时可能需要多次尝试才能成功。对于某些类型的合数，尤其是那些不能轻易表示为两个较小整数平方和的数。

4. Pollard ρ-1 分解算法

1） **算法原理：**

a） 给出一个大整数 N。

b） 我们选择一个 a，其满足 1<a<ρ-1。（ρ为 N 的一个素因子）。

c） 计算$a^{k!}$≡m(mod N),且存在整数 j 使 k!=j(ρ-1)。

d） 因此有 m≡$a^{k!}$≡$a^{j(\rho-1)}$≡1(mod ρ)。

e） 故（m-1,N）>1,只要 m 与 1 mod N 不等即可。

f） 这样我们就求出了我们的非平凡因子(m-1，N)。

2） **代码实现（给出伪代码）：**

```
#include <NTL/ZZ.h>

using namespace NTL;

using namespace std;

#include <iostream>

#include <cstdlib>

#include <math.h>

#include <time.h>

#include <stdlib.h>
```

```
void pollard(ZZ N) {
  cout << "N可以分解为： ";
  while (1) {
      ZZ B = ZZ(10);
      ZZ a = ZZ(2);
      a = PowerMod(a, B, N);
      ZZ d = GCD(a - 1, N);
      if (d > 1 && d < N) {
          cout << d << " ";
          N = N / d;
      }
      else {
          cout << N;
          return;
      }
  }
}
int main() {
  ZZ N;
  cout << "请输入待分解的数：";
  cin >> N;
  pollard(N);
}
```

3） 算法评价：

这个算法存在失败的可能，也就是说有可能找不到N的因子，但这并不妨碍N是个合数的事实。只要k！取值合理，可以在多项式时间计算出结果。

但是k！必须满足"大于p － 1 p-1p-1的所有因子"，如果p － 1的因子很大，选择小的k会造成算法求解失败，选择足够大的k才会增加算法成功的概率，但那样的话算法的复杂度不比试除法好，所以说算法的关键是选取合适的k。

三、同态加密算法

（一） 密码原语

同态加密（Homomorphic Encryption, HE）是指满足密文同态运算性质的加密算法，即数据经过同态加密之后，对密文进行特定的计算，得到的密文计算结果在进行对应的同态解密后的明文等同于对明文数据直接进行相同的计算，实现数据的“可算不可见”。

如果满足 f(A)+f(B)=f(A+B)f(A)+f(B)=f(A+B) ， 我们将这种加密函数叫做加法同态。

如果满足 f(A)×f(B)=f(A×B)f(A)×f(B)=f(A×B) ，我们将这种加密函数叫做乘法同态。

全同态加密：种同态加密算法支持对密文进行任意形式的计算（即满足加法和乘法）。

半同态加密：支持对密文进行部分形式的计算，例如仅支持加法、仅支持乘法或支持有限次加法和乘法。

（二） 数学问题

1. Paillier 算法

●**密钥生成**

生成两个大素数p,q，

计算$n=pq$，$g=n+1$，$\lambda=\mathrm{lcm}(p-1,q-1)$，

定义函数$L(x)=\dfrac{x-1}{n}$，

计算$\mu=(L(g^{\lambda} \bmod n^2))^{-1}(\bmod n)$，

得到公钥(n,g)，私钥(λ,μ).

●**加密过程**

选取明文m，

选择随机数$r(0<r<n)$，

计算得到密文$c=g^m r^n \pmod{n^2}$.

●**解密过程**

计算得到明文$m=L(c^{\lambda} \bmod n^2)\cdot\mu(\bmod n)=\dfrac{L(c^{\lambda} \bmod n^2)}{L(g^{\lambda} \bmod n^2)}$.

●正确性分析

由最小公倍数可得$(p-1)|\lambda,(q-1)|\lambda$，可令$\lambda=k_1(p-1)=k_2(q-1)$,

由欧拉定理可得$g^{\varphi(p)}=g^{p-1}\equiv 1(\bmod p)$，故$g^{\lambda}=g^{k_1(p-1)}\equiv 1(\bmod p)$. 同理有$g^{\lambda}\equiv 1(\bmod q)$.

所以$(g^{\lambda}-1)|p,(g^{\lambda}-1)|q$,所以$(g^{\lambda}-1)|pq$,即$(g^{\lambda}-1)|n$. 可得$(g^{\lambda}-1)|n^2, g^{\lambda}\equiv 1(\bmod n^2)$.

$(g^{\lambda} \bmod n^2)\equiv 1(\bmod n)$. 可令$g^{\lambda} \bmod n^2=nk_g+1$,则$L(g^{\lambda} \bmod n^2)=k_g$.

由二项式定理得$(1+kn)^m=C_m^0(kn)^0+C_m^1(kn)^1+C_m^2(kn)^2+\cdots+C_m^m(kn)^m\equiv 1+kmn(\bmod n^2)$,

故$g^{m\lambda}=(nk_g+1)^m\equiv k_g mn+1(\bmod n^2)$,同理$r^{n\lambda}=(nk_r+1)^n\equiv k_r n^2+1\equiv 1(\bmod n^2)$.

故$L(c^{\lambda} \bmod n^2)=L(g^{m\lambda}r^{n\lambda} \bmod n^2)=L(k_g mn+1)=k_g m$,

所以$\dfrac{L(c^{\lambda} \bmod n^2)}{L(g^{\lambda} \bmod n^2)}=\dfrac{k_g m}{k_g}=m.$

●同态性分析

■加法同态

明文m_1,m_2 明文加法$m_1+m_2\Rightarrow$明文加密$\mathrm{Enc}(m_1+m_2)=g^{m_1+m_2}(r)^n \bmod n^2$

加密↓ ↓

密文c_1,c_2 密文乘法$c_1\cdot c_2=g^{m_1+m_2}(r_1r_2)^n \bmod n^2$

■数乘同态

r随机选择

$\mathrm{Enc}(km)=g^{km}(r)^n \bmod n^2$

$(\mathrm{Enc}(m))^k=c^k=(g^m(r)^n \bmod n^2)^k=g^{km}(r^k)^n \bmod n^2$

★可以在密文c上计算任意一次多项式$ac+b$

●安全性分析

■大整数分解问题

破解密码：已知公钥$n=pq$，难以推出素数p和q，故难以破解出私钥λ.

■复合剩余类问题（Decisional Composite Residuolity Assumption，DCRA）

复合剩余类问题指的是：给定一个合数n和整数z，很难确定是否存在一个整数y，使得$z\equiv y^n(\bmod n^2)$.

即判断z是不是模n^2的n阶剩余是很困难的.

复合剩余类问题的困难性是Paillier算法的安全性的基础。由于复合剩余类问题的困难性，Paillier算法能够抵抗多种攻击，包括选择明文攻击（CPA）和选择密文攻击（CCA）。这意味着即使攻击者拥有大量的密文和对应的明文，或者能够选择密文并获取对应的明文，也无法有效地破解Paillier算法的加密。

2. ElGamal 算法

●密钥生成

生成一个大素数p，g是p的原根，
随机选择一个私钥$x(1<x<p-1)$，
计算得公钥 $y \equiv g^x \pmod p$.

●加密过程

选取明文m，选择一个随机数$k(1<k<p-1)$，
计算得到密文$c_1 \equiv g^k \pmod p$,

$$c_2 \equiv m \cdot y^k \pmod p.$$

●解密过程

计算 $k' \equiv k^{-1} \pmod{p-1}$,
计算得到明文 $m \equiv c_2 c_1^{-x} y^{k'c_2} \pmod p$.

●同态性（乘法同态）

明文m_1, m_2

加密↓ ↓

密文c_1, c_2

有$\mathrm{Dec}(c_1c_2)=m_1m_2$

★可以在密文c上计算任意正整数次幂c^k

●安全性分析

■离散对数问题

有限域上的离散对数指的是：设p,q为两个素数，$G=\{g^i \mid 0 \le i \le q-1, g \in \mathbb{Z}_p^*\}$为阶为$q$的有限域 $\mathbb{Z}_p^*$ 上的乘法群. 给定一个元素$y \in G$，找到一个整数 $x \in \mathbb{Z}_q$ 使得$y=g^x$的问题.

离散对数问题的难点在于，对于某些群（特别是大素数阶的群），不存在已知的多项式时间算法来有效地求解x.

一些求解思路：

- 穷举法
- 商客法（小步大步法）
- Pollard ρ算法
- Pohlig-Hellman算法
- 指数积分法

ElGamal加密算法的安全性依赖于离散对数问题的难解性。ElGamal算法在理论上被认为比RSA算法更安全，因为理论上存在更快的方法能够分解大质数，这可能会降低RSA的安全性；而ElGamal算法依赖的离散对数问题在数学上更难解决。

3. BFV 加密方案

●密钥生成

从特定的分布（通常是离散高斯分布）中随机采样一个多项式$\mathbf{s}$作为私钥，
从多项式环 $R_q = \mathbb{Z}_q[x]/(x^n+1)$ 中随机采样一个多项式$\mathbf{a}$，
从噪声分布（通常是离散高斯分布）中随机采样一个多项式$\mathbf{e}$，
计算$\mathbf{b} = -(\mathbf{a}\cdot\mathbf{s}+\mathbf{e}) \pmod q$，公钥为$(\mathbf{a},\mathbf{b})$.

●加密过程

选取明文m，将其编码到多项式环上（一般使用SIMD编码）得到$\mathbf{m} \in R_t$，
从R_q中随机采样一个多项式$\mathbf{u}$，从噪声分布中随机采样两个多项式$\mathbf{e_1},\mathbf{e_2}$，
计算$\mathbf{c_1} = \mathbf{a}\cdot\mathbf{u}+\mathbf{e_1}+\Delta\cdot\mathbf{m} \pmod q$，$\mathbf{c_2} = \mathbf{b}\cdot\mathbf{u}+\mathbf{e_2} \pmod q$，其中$\Delta = \left\lfloor \frac{q}{t} \right\rfloor$，密文为$(\mathbf{c_1},\mathbf{c_2})$.

●解密过程

计算 $\mathbf{m}' = \mathbf{c_0} + \mathbf{c_1}\cdot\mathbf{s} \pmod q$,
计算得到明文 $\mathbf{m} = \left\lfloor \frac{t\mathbf{m}'}{q} \right\rceil \pmod t$.

●同态性（全同态）

■加法同态

明文$\mathbf{m}_1,\mathbf{m}_2$ 加密 密文$(\mathbf{c}_{11},\mathbf{c}_{12}),(\mathbf{c}_{21},\mathbf{c}_{22})$

明文加法$\mathbf{m}_1+\mathbf{m}_2$

密文加法$(\mathbf{c}_{11},\mathbf{c}_{12})+(\mathbf{c}_{21},\mathbf{c}_{22})=(\mathbf{c}_{11}+\mathbf{c}_{21},\mathbf{c}_{12}+\mathbf{c}_{22})$

$=\left(\mathbf{a}(\mathbf{u}_1+\mathbf{u}_2)+(\mathbf{e}_{11}+\mathbf{e}_{21})+\Delta(\mathbf{m}_1+\mathbf{m}_2),\mathbf{b}(\mathbf{u}_1+\mathbf{u}_2)+(\mathbf{e}_{12}+\mathbf{e}_{22})\right)$

■乘法同态

略

原始数据空间		明文空间		密文空间
a	编码 →	$\mathbf{a}$	加密 →	Enc($\mathbf{a}$)
+		+		+
b	编码 →	$\mathbf{b}$	加密 →	Enc($\mathbf{b}$)
↓		↓		↓
c	← 解码	$\mathbf{c}$	← 解密	Enc($\mathbf{c}$)

噪声会在同态运算中不断增大。减小噪声积累的方法：Bootstrapping。

●安全性分析

■带误差学习问题（Learning With Errors，LWE）

扩展：环上带误差学习问题（Ring Learning With Errors，RLWE）

给定一个多项式环$R_q=\mathbb{Z}_q[x]/(x^n+1)$（其中$q$是一个大素数）和一个概率分布$\chi$（通常是高斯分布），再给定多项式$\mathbf{a},\mathbf{b}\in R_q$，找到多项式$\mathbf{s}\in R_q$使得$\mathbf{b}=\mathbf{a}\cdot\mathbf{s}+\mathbf{e}(\mathrm{mod}\,q)$是困难的.

其中$\mathbf{e}$是从分布χ中采样得到的噪声多项式.

BFV加密方案的安全性依赖于RLWE问题的复杂性。RLWE问题的复杂性主要来源于其数学结构的复杂性和噪声的引入。由于RLWE问题是在环上进行的，因此其数学结构比LWE问题更为复杂。同时，噪声的引入也增加了问题的难度，使得从给定的样本中恢复出s变得非常困难。

（三） 应用场景

1. 云计算

用户的数据在加密状态下进行处理，云服务提供商无法直接访问或解读数据的原始内容，由云服务器直接对密文进行计算，计算结果解密后与直接在明文上操作的结果一致，从而确保了用户数据的隐私安全，无需担心数据泄露。

2. 医疗健康

保护患者的病历、医疗影像等敏感信息，安全地分析医疗数据，而无需将数据解密，有效防止患者隐私泄露。

3. 金融服务

处理大量包括个人身份信息、信用卡信息、贷款 信息等的客户数据时保护客户数据的隐私和安全，避免因数据泄露而导致的法律和经济风险。

4. 联邦学习

联邦学习（Federated Learning）是一种分布式机器学习方法，它允许多个客户端（如移动设备、浏览器或分布式服务器）协作训练一个共享模型，同时保持数据的隐私和安全。

联合建模过程中的参数交互计算：同态 加密用于联邦学习中的参数交互计算过程，实现预测模型的联合确立。在联邦学习中，多个参与方可以在保证各自数据隐私的同 时实现联合机器学习建模，即在不获取对 方原始数据的情况下利用对方数据提升自 身模型的效果。

提高数据隐私保护：同态加密允许在加密数据上直接进行计算，这意味着参与方的数据在整个训练过程中始终处于加密状态，从而保护隐私。这种方式可以确保数据的隐私性，同时允许进行有效的模型训练。