中国地质大学（武汉）

信息可生存原理与技术报告

姓　　名：	郑召作	学号：	20161001451
院（系）：	计算机学院	专业：	信息安全
指导教师：	程　池	职称：	副教授

2019 年 11 月

姓名： 郑召作 学号：20161001451

选题难度	内容深度	文笔流畅	格式	英文摘要	总成绩

评语：

关于同态加密算法的学习研究

郑召作 19216107
（中国地质大学（武汉）计算机学院）

摘要：

随着云计算技术的广泛应用，它在提高使用效率的同时，为实现用户信息资产安全与隐私保护带来极大的冲击与挑战。用户希望云计算平台在带来方便的同时，能够保证数据的机密性与完整性。同态加密算法由于其同态特性，云端可以实现对任意加密数据进行运算，即可以直接对密文进行操作而不需要将密文解密后再处理，不可信的云端利用这种同态特性，使得对加密数据进行可信计算成为可能。

关键词： 同态加密；改进算法；网络信息；安全保护

Study on Homomorphic Encryption Algorithm

Zheng ZhaoZuo
(China University of Geoscience Computer Campus)

Abstract：

With the extensive application of cloud computing technology, it not only improves the use efficiency, but also brings great impact and challenges to the realization of user information asset security and privacy protection.Users hope that the cloud computing platform can guarantee the confidentiality and integrity of data while bringing convenience.Due to its homomorphic characteristics, the cloud can perform operations on any encrypted data, that is, the ciphertext can be directly operated without decrypting the ciphertext and then processing it. The untrusted cloud can make use of this homomorphic characteristics to make it possible to conduct trusted computing on the encrypted data.

Keywords：

Homomorphic Encryption; Improved Algorithms; Network Information; Security Protection

0 引言

近几年，用户的信息隐私受到了越来越多的关注，尽管可以利用云端服务器强大的能力，用户还是希望自己外包给云端的信息的隐私性能够受到保护，不被任何未授权方（包括服务提供者）知道。因此，隐私保护性是安全应用的重要研究内容。常规的网络信息通常用不加密的方式存储及处理，所以在同态加密算法中，网络信息在处理过程中有着很大隐患，如被窃取、篡改及被破坏等安全隐患。

网络信息存储及处理安全性是个难题，传统的加密网络信息无法进行直接处理，必须将其转换成明文信息再进行处理，为了保证网络信息的安全性，处理之后还需要转换成为不加密的网络信息，效率非常低，对服务器资源也是一种极大地浪费。同态加密是解决此问题的理想方法，而且实验数据表明，提出的基于同态加密算法的网络信息安全保护方法相比于传统安全保护方法，对网络信息具有较高的安全保护性能。本文对传统同态加密算法进行了实现，并对其进行了验证。

1 同态加密介绍

1978 年由 RIVEST 等人提出了被称为“学术皇冠上的明珠”的同态加密概念。同态加密是对加密的网络信息和未加密的网络信息执行相应的操作，首先对加密的网络信息进行特定的加密运算，对未加密的网络信息也进行另一种特定的运算（与前者的运算不一定相同）。本文通过查阅文献，本文在云计算应用场景下面对同态加密进行介绍：

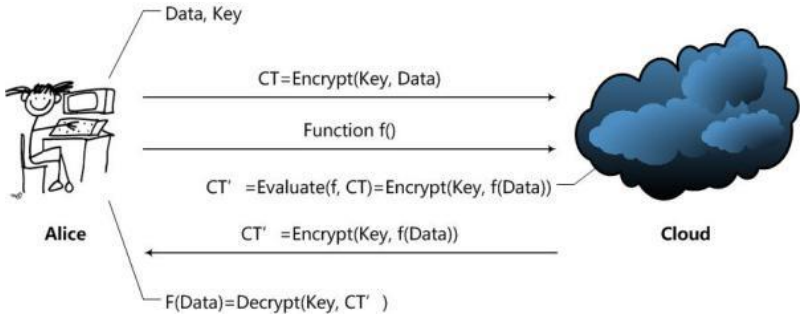
Alice 通过 Cloud，以 Homomorphic Encryption（以下简称 HE）处理数据的整个处理过程大致是这样的：

1、Alice 对数据进行加密。并把加密后的数据发送给 Cloud；

2、Alice 向 Cloud 提交数据的处理方法，这里用函数 f 来表示；

3、Cloud 在函数 f 下对数据进行处理，并且将处理后的结果发送给 Alice；

4、Alice 对数据进行解密，得到结果。



据此，我们可以很直观的得到一个 HE 方案应该拥有的函数：

- KeyGen 函数：密钥生成函数。这个函数应该由 Alice 运行，用于产生加密数据 Data 所用的密钥 Key。同时应该还有一些公开常数 PP（Public Parameter）；
- Encrypt 函数：加密函数。这个函数也应该由 Alice 运行，用 Key 对用户数据 Data 进行加密，得到密文 CT（Ciphertext）；
- Evaluate 函数：评估函数。这个函数由 Cloud 运行，在用户给定的数据处理方法 f 下，对密文进行操作，使得结果相当于用户用密钥 Key 对 f(Data)进行加密。
- Decrypt 函数：解密函数。这个函数由 Alice 运行，用于得到 Cloud 处理的结果 f(Data)。

如果一个加密函数同时满足加法同态和乘法同态，称为全同态加密。那么可以使用这个加密函数完成各种加密后的运算(加减乘除、多项式求值、指数、对数、三角函数)。

第一个满足加法和乘法同态的同态加密方法：2009 年 9 月克雷格·金特里（Craig Gentry）的论文 Fully Homomorphic Encryption Using Ideal Lattices 从数学上提出了“全同态加密”的可行方法，即可以在不解密的条件下对加密数据进行任何可以在明文上进行的运算，使这项技术获取了决定性的突破。

2 同态加密的安全性

HE 方案的最基本安全性是语义安全性（Semantic Security）。直观地说，就是密文（Ciphertext）不泄露明文（Plaintext）中的任意信息。如果用公式表述的话，为：

$$\forall m_0, m_1, Encrypt(PK, m_0) \approx Encrypt(PK, m_1)$$

这里 PK 代表公钥（Public Key），公式中的"约等于"符号，意味着多项式不可区分性，即不存在高效的算法，可以区分两个结果，即使已知 m0, m1 和 PK。这是因为加密算法中还用到一个很重要的量：随机数。也就是说，对于同样的明文 m 进行加密，得到的结果都不一样，即一个明文可以对应多个密文（many ciphertexts per plaintext）。

在密码学中，还有更强的安全性定义，叫做选择密文安全性（Chosen Ciphertext Security）。选择密文安全性分为非适应性（None-Adaptively）和适应性（Adaptively），也就是 CCA1 和 CCA2。HE 方案是不可能做到 CCA2 安全的。那么，HE 方案能不能做到 CCA1 安全呢？至今还没有 CCA1 安全的 FHE 方案，但是在 2010 年，密码学家们就已经构造出了 CCA1 的 SWHE 方案了。

HE 方案还有一方面的安全性，就是函数 f 是否也可以保密？如果能保密，Cloud 不仅不能够得到数据本身的内容，现在连数据怎么处理的都不知道，只能按照给定的算法执行，然后返回的结果就是用户想要的结果。如果 HE 方案满足这样的条件，我们称这个 HE 方案具有 Function-Privacy 特性。不过，现在还没有 Function-privacy FHE，甚至 Function-privacy SWHE 也没有。

3 实验实现

实验环境：Ubuntu 14.04 LTS,Vmware g++编译器

引入头文件：HElib 库、 NTL、 GMP

HElib 是同态 (homomorphic) 加密方法的实现，用于在不解密的情况下处理加密数据的技术。这将使敏感数据处理变得极其安全：比如说，公司可以加密托管在云端的数据库，无需将记录转换回成明文，就可以处理记录。2018 年进行了一次重写，主要目标是，提高性能，减少自同构的数量，并降低每个自同构的成本。目前可用的是 Brakerski-Gentry-Vaikuntanathan（BGV）方案的实施，以及许多优化，使同形评估运行更快，主要集中在有效使用 Smart-Vercauteren 密文打包技术和 Gentry-Halevi-Smart 优化。

实验步骤如下：

3.1 安装 GMP

在官网下载 gmp-5.0.1 的源代码，并解压至 gmp-5.0.1 目录，su 切换至超级用户权限。分别执行以下命令行：

```
1.  ./configure
2.  make
3.  make check
4.  make install
```

开始编译，安装 gmp 数学库。

3.2 编译 NTL 库：

进入 http://www.shoup.net/ntl/。依次点击 A Tour of NTL->Obtaining and Installing NTL for UNIX。默认按照第一种方式编译：

```
1.  gunzip ntl-11.4.1.tar.gz
2.  tar xf ntl-11.4.1.tar
3.  cd ntl-11.4.1/src
4.  ./configure PREFIX=$HOME/sw
5.  make
6.  make check
7.  make install
```

3.3 安装 Helib 库

```
cd src
按照如下方式修改 Makefile 文件:
   Line 19: CFLAGS = -g -O2 -std=c++11
   Line 85: (Insert the following contents)
       install:
           cp -p fhe.a /usr/local/lib/libfhe.a
           chmod a+r /usr/local/lib/libfhe.a
           rm -rf /usr/local/include/fhe
           mkdir -m 755 /usr/local/include/fhe
           cp -p *.h /usr/local/include/fhe/
           chmod -R a+r /usr/local/include/fhe
make
make check
sudo make install
```

3.4 实现简单密文相加：

接下来我将里用 Helib 库实现两个数相加，我们需要做的第一件事是声明这个 FHE 方案中使用的参数。

```
long m = 0;    // Specific modulus
long p = 1021; // Plaintext base [default=2], should be a prime number
long r = 1;    // Lifting [default=1]
long L = 16;   // Number of levels in the modulus chain [default=heuristic]
long c = 3;    // Number of columns in key-switching matrix [default=2]
long w = 64;   // Hamming weight of secret key
long d = 0;    // Degree of the field extension [default=1]
long k = 128;  // Security parameter [default=80]
long s = 0;    // Minimum number of slots [default=0]
```

这些参数将会在函数 FindM 中寻找一个合适的值 m：

```
m = FindM(k,L,c,p, d, s, 0);
```

之后，创建一个 FHEcontext 对象来保存所有参数：

```
FHEcontext context(m, p, r);
buildModChain(context, L, c);
```

由于加密需要使用一个多项式，因此下一步是从 NTL 类中使用 ZZX 类来创建一个多项式：

```
ZZX G = context.alMod.getFactorsOverZZ()[0];
```

接下来，利用 context 来生成公钥和私钥：

```
FHESecKey secretKey(context);
const FHEPubKey& publicKey = secretKey;
secretKey.GenSecKey(w);
```

到现在为止，结束初始化部分。现在我们可以开始对已加密的数据进行一些操作。Helib 使用 Ctxt 来保存密文：

```
Ctxt ctx1(publicKey);
Ctxt ctx2(publicKey);
```

ctx1 是值 2 的加密，ctx2 是值 3 的加密。注意，这些值以多项式 ZZX 的形式给出加密方法:

```
publicKey.Encrypt(ctx1, to_ZZX(2));
publicKey.Encrypt(ctx2, to_ZZX(3));
```

让我们创建另一个密文来保存和 Enc(2) + Enc(3)，并实际执行这个和。HElib 重载运算符 +=，这使得加密文本求和更简单：

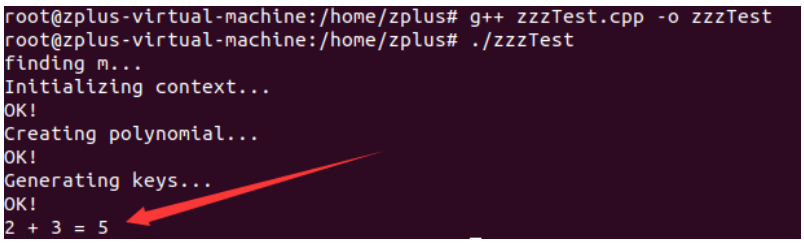
```
Ctxt ctSum = ctx1;
ctSum += ctx2;
```

最后，我们创建一个多项式，它将保存和的解密值，并使用密钥将密文 ctSum 解密为它：

```
ZZX ptSum;
secretKey.Decrypt(ptSum, ctSum);
```

就是这样。如果我们检查 ptSum，应该会看到它的值为 5。但是在创建了 Context 和 Key 之后，求和本身只需使用密码文本上熟悉的+=操作符即可。

3.5 实验结果：



4 总结

本文先通过学习并总结了同态加密算法研究，与一般加密算法相比,同态加密除了能实现基本的加密操作之外,还能实现密文间的多种计算功能,即先计算后解密可等价于先解密后计算。这个特性对于保护信息的安全具有重要意义。然后本文通过 C++的 Helib 库实现了一个简单的密文相加，虽然这是一个简单的例子，但是 HElib 允许我们做一些实际应用中更复杂的事情。大多数可以在 HElib 附带的 Example 中找到。

参考文献：

[1]夏超. 同态加密技术及其应用研究[D]. 安徽大学, 2013.
[2]林如磊，王箭，杜贺. 整数上的全同态加密方案的改进[J]. 计算机应用研究，2013, 30(5):1515-1519.
[3]陈志伟，杜敏，杨亚涛,等. 基于 RSA 和 Paillier 的同态云计算方案[J]. 计算机工程, 2013(7).
[4]汤殿华，祝世雄，王林,等. 基于 RLWE 的全同态加密方案[J]. 通信学报, 2014(01):177-186.
[5]王辉. 同态加密的分析与优化[D]. 杭州电子科技大学, 2013.

附文代码：

```
#include "FHE.h"
#include <iostream>

using namespace std;

int main(int argc, char **argv)
{
   long p = 1021;
   long r = 1;
   long L = 4;
   long c = 2;
   long k = 80;
   long s = 0;
   long d = 0;
   long w = 64;

   cout << "finding m..." << flush;
   long m = FindM(k,L,c,p,d,s,0);
   cout << "m = "<< m << endl;

   cout << "Initializing context..." << flush;
   FHEcontext context(m,p,r);  //initialize context
   buildModChain(context, L, c);  //modify the context
   cout << "OK!" << endl;

   cout << "Creating polynomial..." << flush;
   ZZX G = context.alMod.getFactorsOverZZ()[0];  //creates the polynomial used to encrypte
d the data
   cout << "OK!" << endl;

   cout << "Generating keys..." << flush;
   FHESecKey secretKey(context);  //construct a secret key structure
   const FHEPubKey& publicKey = secretKey;  //An "upcast": FHESecKey is a subclass of FHE
PubKey
```

```
    secretKey.GenSecKey(w);  //actually generate a secret key with Hamming weight w
    cout << "OK!" << endl;


    Ctxt ctxt1(publicKey);
    Ctxt ctxt2(publicKey);

    publicKey.Encrypt(ctxt1, to_ZZX(2));  //encrypt the value 2
    publicKey.Encrypt(ctxt2, to_ZZX(3));  //encrypt the value 3

    Ctxt ctSum = ctxt1;  //create a ciphertext to hold the sum and initialize it with Enc(2)
    ctSum += ctxt2;

    ZZX ptSum;  //create a ciphertext to hold the plaintext of the sum
    secretKey.Decrypt(ptSum, ctSum);

    cout << "2 + 3 = " << ptSum[0] << endl;
    return 0;
}
```