# Implementing Fully Key-Homomorphic Encryption in Haskell

Maurice Shih
CS 240h

**Abstract**

Lattice based encryption schemes have many desirable properties. These include quantum and classic computer attack resistant and being highly parallelizable. The construction implemented in this project have addition advantages, such as fully key-homomorphic and being an attribute based encryption. In addition, he public key size being only dependent on the depth of the circuit. An haskell implementation was created to encode and decode, using this encryption scheme.

# Contents

# 1 Introduction

We implement the cryptosystem outlined in Fully Key-Homomorphic Encryption, Arithmetic Circuit ABE, and Compact Garbled Circuits by Dan Boneh, Craig Gentry, Sergey Gorbunovx, Shai Halevi, Valeria Nikolaenko, Gil Segev, Vinod Vaikuntanathan, and Dhinakaran Vinayagamurth. Given inputs, the implementation generates the public key and encrypts a message based on an arithmetic circuit. Decryption works if the attribute evaluates to zero when put into the given circuit.

# 2 Motivation

There are many mathematically sound encryption methods published without an actually implementation. Without implementation, a greater understanding of how a complex scheme cannot be achieved. The particular scheme was selected due to many desirable properties of the algorithm. Breaking these systems are equivalent to solving the hardest case of lattice problems. Lattice based cryptographic constructions are resistent to classic and quantum computing attacks. This particular construction is also fully key-homomorphic. The size of the secret key is proportional to the depth of the circuit, not the size. We also reduction in size as previous attribute based encryption constructions required an element for every gate of a circuit, while this construction only has a matrix based on only the final output of the circuit.

This project was also written in Haskell for desirable traits. Since Haskell is functional language, it mirrors mathematical functions very well. It has strong typing, which makes for easier error detection, in addition to immutable variables makes catching and fixing bugs considerably easier. Haskell is also lazy, so it can reduce computation and runtime.

# 3 Background

In this section, we define some mathematical and cryptographic terms relevant to this project.

**Lattices.** A lattice is a set of all points (in modulus of some prime in this paper) that are the linear combination of points.

**Fully Key-Homomorphic.** A scheme is fully key-homomorphic if it is possible to transform a message, $m$ encoded by a a public key $pk_1$, $E(m, pk_1)$, to an encryption of the same message encoded by a different public key, say $pk_2$, $E(m, pk_2)$.

**Attribute Based Encryption.** In attribute based encryption schemes, when a message is encoded, there is an attribute associated with it, $x$. A secret key can only decode if its associated function $f$, with the attribute, $x$ as input evaluates to 0, or $f(x) = 0$ for the particular encoded message.

**Learning with Errors (LWE)** - The learning with seeks to distinguish two distributions,

$$(A, A^T s + e),$$

and

$$(A, u),$$

where $A$, $s$, $e$, and $U$ are independently sampled and the length of $A^T s + e$ is the same as $u$.

**Gadget and trapdoor generation.** A gadget matrix $G$ is a matrix of length $n$ by $nl$ that has $g$ on its diagonals, where $g = [2^0 2^1 ... 2^{2-l}]$. In our construction, we use this gadget matrix to create a trap door, $R$ for $A$ so that for a given $u$ such that $Ax = u$, $x = RG^{-1}(H^{-1}u)$, where $H$ user-generated. To do this, create $R$ such that $AR = HG \pmod q$.

**Arithmetic Circuits.** These circuits have gates of two or more inputs with addition or multiplication. These gates can also be weighted.

# 4 Use

In this section, we discuss the workings of the construction mathematically, as well as the haskell functions that we have created that mirrors these.

## 4.1 Inputs

We have an integer $n$ and a prime $q$. Let $m = O(n \cdot log(q))$, and $l = \lceil \log q \rceil$, $w = nl$ and $\bar{m} = m - n$, note that this should be positive.

We created a data structure `Parameters` that contains $n$, $m$, $\bar{m}$, and $q$.

```
data Parameters = Parameters Integer Integer Integer Integer deriving (Show)
```

In addition, we had functions, `getn`, `getm`, `getmb`, `getq`, `getw`, `getl`, to retrieve the specified value.

## 4.2 Setup

We now create the matrix $A \in \mathbb{Z}_q^{n \times m}$ and its trapdoor, $R \in \mathbb{Z}_q^{n \times m}$. The following steps were taken.

- We first construct a random matrix $\bar{A} \in \mathbb{Z}_q^{n \times \bar{m}}$.
- Let $G \in \mathbb{Z}_q^{n \times nl}$ be the gadget matrix.
- Also let $H \in \mathbb{Z}_q^{n \times n}$ be generated as a random, $\pm 1$ invertible matrix. Our implementation randomly generates a $\pm 1$ matric, and checks its rank.
- Let $\bar{R} \in \mathbb{Z}_q^{\bar{m} \times nl}$ random $\pm 1$ matrix.
- The trapdoor $R$, is then the vertical concatenation of $\bar{R}$ and $I \in \mathbb{Z}_q^{nl \times nl}$.

- The matrix $A \in \mathbb{Z}_q^{n \times m}$ is the horizontal concatenation of $\bar{A}$ and $H \cdot G - \bar{A} \cdot \bar{R}$.

The commands in out package are (respectively) wrote the values to file (hence the IO() output),

- `fgen_abar :: Parameters -> IO()`
- `fgen_gg :: Parameters -> IO()`
- `nfullrankhb :: Parameters -> IO()` (this would check for full rank, hence the longer name)
- `fgen_rbar :: Parameters -> IO()`
- `fget_r,:: Parameters -> IO (Matrix (Integer))` (this did not write to file, but would generate $R$ based on the generation of the above)
- `fget_a,:: Parameters -> IO (Matrix (Integer))` (this did not write to file, but would generate $R$ based on the generation of the above)

Let the public parameters be $A$, and $A_1$,...,$A_l \in \mathbb{Z}_q^{n \times m}$ be random matrices as well as a syndrome $u \in \mathbb{Z}_q^{n \times 1}$.

## 4.3 Encryption

We encryption a message bit $\mu \in \{0, 1\}$ with its attribute vector $x \in \mathbb{Z}_q^{l \times 1}$, which is composed of $0$'s and $1$'s. We first generate the following matrices.

- Let $s \in \mathbb{Z}_q^{n \times 1}$ be a random matrix.
- Let the error vectors $e_0$ and $e_1$ be of length $m$.
- Generate $l$ $\pm 1$ random matrices $S_i$ of size $m \times m$.
- Define $J \in \mathbb{Z}_q^{n \times (l+1)m}$ be $(A|x_1G + B_1|...|x_lG + B_l)$
- Define $e = (I_m|S_1|...|S_l)^T \cdot e_0 \in \mathbb{Z}_q^{(l+1)m}$

Then the ciphertext is

$$c_x = [\bar{c}^t|c_1^t|...|c_l^t|c] = (J^T s + e, (u + e_1)^T s + \lceil q/2 \rceil \mu),$$

where $\bar{c}^t = A^T s + e$ and $c_i^t = (x_i G + B_i)^T s + e$ and $c = (u + e_1)^T s + \lceil q/2 \rceil \mu$. NOte that for small values of $q$ and big values of $n$ in comparison to $q$, the error vectors can actually distort the message so it will not may not be recoverable.

There exists a function to get each $A_i$ and $c_i$ written, but to do all the encoding, only the following function can be used, where the first integer is $q$ and the second is $n$

`encode_m :: Integer -> Integer -> IO()`

### 4.4 Secret Key Generation / Evaluation of Circuit

In order to create a secret key, we must have a arithmetic circuit $f$ such that $f(x) = 0$. Using this circuit we create $A_f$ recursively.

- The base case of $f(x) = x_i$, then $A_f = A_i$
- For addition gates, such as $f(x) = f_1(x) + f_2(x)$, then $A_f = A_{f_1} + A_{f_2}$.
- For multiplication gates, such as $f(x) = f_1(x) \cdot f_2(x)$, then $A_f = A_{f_1} \cdot G^{-1} A_{f_2}$.

Note: the term $G^{-1}M$ for some matrix is not the inverse of $G$ multiplied by $M$, rather we create a matrix $N$, such that $GN = M$.

Using the trapdoor of $A$ (as discussed in the previous section), $R$ we find a solution $z_f$ to

$$[\bar{A}| - A_f] \cdot z_f = -u \pmod{q}$$

Functions were created to return $barA$ and other elements, but all of this is done implicitly when decryptoin is done.

### 4.5 Decryption

To decrypt, we recursively generate $c_f$, similar to the secret key generation with the following ruels,

- In the base case where $f(x) = x_i$, $c_f = c_i$
- For addition gates, such as $f(x) = f_1(x) + f_2(x)$, then $c^t_{f,x} = c^t_{f_1,x} + c^t_{f_2,x}$
- For multiplication gates, such as $f(x) = f_1(x) \cdot f_2(x)$, then $c^t_{f,x} = c^t_{f_1,x} \cdot G^{-1}(A_{f_2}) + c^t_{f_2,x} \cdot f_1(x)$

We then get the message $\mu$ by

$$[\bar{c}^t | c^t_{f,x}] \cdot z_f + c \approx \mu \cdot \lceil q/2 \rceil$$

In the code the following function does this.

```
decode_m ::  Integer -> Integer -> IO()
```

# 5 Future Work

One of the notable feature of lattice based cryptographic construction is the ability parallelize many parts. However, this was not done and can be implemented to reduce runtime. On a related mark, the testing of the runtime and increase based on input size has not been determined yet.