B.Tech Project Report, 2011
Autumn Semester

Fully Homomorphic Encryption

Amit Datta (08CS1045)

Guide: Prof. Debdeep Mukhopadhyay
Department of Computer Science and Engineering
IIT-Kharagpur

.

Acknowledgment

This report is the result of the project work performed under the able guidance of Prof. **Debdeep Mukhopadhyay** at the Department of Computer Science and Engineering of the Indian Institute of Technology, Kharagpur.

I am deeply grateful to my supervisor for having given me the opportunity of working under him and guiding me selflessly for the whole period. He gave me exposure to all the related research going on in this field, which helped me enormously.

Without the help, encouragement and patient support I received from my guide, this report would never have materialized.

.

Certificate

This is to certify that the project report entitled Fully Homomorphic Encryption, which is being submitted by Mr. Amit Datta (Roll No:-08CS1045) in partial fulfillment for the requirements of the degree of Bachelor of Technology in Computer Science and Engineering is a bona-fide record of investigations carried out by him under my supervision and guidance.

(Debdeep Mukhopadhyay)

Place:

Date:

.

Contents

1 Introduction

Fully Homomorphic Encryption (FHE) has been a field of extensive research, more so in the recent years. And, we are talking about an Encryption Scheme which is also efficient. By efficient we mean that the encryption and the decryption must take place in time polynomial in some security parameter λ. There have been several attempts at constructing an FHE scheme that is efficient. Describe some attempts here. But before 2009, no efficient scheme for FHE existed. Gentry et.al. gave the first working edition of the FHE in [1]. It was a scheme based on lattices. An implementation of the scheme was made by Smart and Vercauteren [4], but in this implementation, the key and cipher-text sizes had to be relatively small. A conceptually simpler version of this scheme was developed in [2] where simple integer operations were used instead of lattices. A working implementation of the FHE was finally developed by Gentry and Halevi [3].

2 Mathematical Development

An encryption scheme $\mathcal{E}$ consists of three algorithms: $\mathsf{KeyGen}_{\mathcal{E}}$, $\mathsf{Encrypt}_{\mathcal{E}}$and $\mathsf{Decrypt}_{\mathcal{E}}$. Each of these algorithms must be efficient, i.e. they must all run in poly(λ), where λ is the security parameter which specifies the bit-length of the keys. $\mathsf{KeyGen}_{\mathcal{E}}$generates a key, which is used in both $\mathsf{Encrypt}_{\mathcal{E}}$and $\mathsf{Decrypt}_{\mathcal{E}}$. The homomorphic scheme discussed here includes a fourth algorithm $\mathsf{Evaluate}_{\mathcal{E}}$, which takes in a function $f \in \mathcal{F}_{\mathcal{E}}$ and a set of cipher-texts $\{c_1, c_2, ..c_t\}$ such that $c_i \leftarrow \mathsf{Encrypt}_{\mathcal{E}}(pk, m_i)$, and generates a cipher-text c which encrypts $f(m_1, m_2, .., m_t)$, i.e. $\mathsf{Decrypt}_{\mathcal{E}}(sk, c) = f(m_1, m_2, .., m_t)$. Here $\mathcal{F}_{\mathcal{E}}$ is the set of permitted functions. $\mathcal{E}$ can handle functions only in $\mathcal{F}_{\mathcal{E}}$. For all other functions, the output of $\mathsf{Evaluate}_{\mathcal{E}}$may not be meaningful.

2.1 A "Somewhat" Homomorphic Scheme

Consider the following encryption scheme. Here λ is the security parameter. We set $N = \lambda$, $P = \lambda^2$ and $Q = \lambda^5$.

- $\mathsf{KeyGen}_{\mathcal{E}}(\lambda)$: Generate a random P-bit odd integer p, which acts as the key.
- $\mathsf{Encrypt}_{\mathcal{E}}(p, m)$: Output a cipher-text $c \leftarrow m' + p * q$, where m' is a random N-bit number such that $m' = m \mod 2$ and q is any random Q-bit number.
- $\mathsf{Decrypt}_{\mathcal{E}}(p, c)$: Output $(c \mod p) \mod 2$.
- $\mathsf{Evaluate}_{\mathcal{E}}(f, c_1, ..., c_t)$: The boolean function f is first converted to an equivalent function f' with only AND and XOR gates. Then the AND and XOR operators are replaced with multiplication and addition operators respectively to generate the function f''. Compute and return $f''(c_1, ..., c_t)$.

One can observe that the cipher-texts from $\mathcal{E}$ are *near-multiples* of p. $(c \mod p)$ is referred to as the noise associated with the cipher-text. It is the difference from the nearest multiple of p. Since the noise has the same parity as the message encrypted. Operations like $\mathsf{Add}_{\mathcal{E}}(c_1, c_2)$, $\mathsf{Sub}_{\mathcal{E}}(c_1, c_2)$ and $\mathsf{Mult}_{\mathcal{E}}(c_1, c_2)$ are computed as $(c_1 + c_2)$, $(c_1 - c_2)$ and $(c_1 * c_2)$ respectively.

In order to make it more convincing, an example is presented. For the computation of $\mathsf{Mult}_{\mathcal{E}}(c_1, c_2)$, where $c_1 \leftarrow \mathsf{Encrypt}_{\mathcal{E}}(p, m_1) = m_1' + p * q_1$, and $c_2 \leftarrow \mathsf{Encrypt}_{\mathcal{E}}(p, m_2) = m_2' + p * q_2$. The cipher-text output by $\mathsf{Evaluate}_{\mathcal{E}}$is $c = c_1 * c_2$. So,

$$c = m_1' * m_2' + p * q'$$

where q' is some integer. As long as the noise $m_1' * m_2'$ is small, and not comparable to p, we have:

$$c \mod p = m_1' * m_2'$$

and therefore, $(c \mod p) \mod 2 = m_1.m_2$. This scheme works as long as the noise does not blow up too much and start affecting the result. Now observe the following function

- $\mathsf{Recrypt}_{\mathcal{E}}(pk_2, D_{\mathcal{E}}, \bar{sk}_1, c_1)$: Compute $\bar{c_1} \leftarrow \mathsf{Encrypt}_{\mathcal{E}}(pk_2, \langle c_1 \rangle)$ over the bits of c_1. Then output $c \leftarrow \mathsf{Evaluate}_{\mathcal{E}}(pk_2, D_{\mathcal{E}}, \bar{sk}_1, \bar{c_1})$.

Here c_1 is the cipher-text encrypted under pk_1, $\bar{sk}_1$ are the bits of the secret key sk_1 encrypted under pk_2, and $D_{\mathcal{E}}$ is the decryption circuit. If we observe carefully, we will find that $\mathsf{Recrypt}_{\mathcal{E}}$takes in a cipher-text encrypted in pk_1 and outputs another new cipher-text encrypted under pk_2. This recryption procedure basically refreshes the noise element, so that it once again becomes small enough. But for that, the scheme must be able to handle the decryption function $\mathsf{Decrypt}_{\mathcal{E}}$. But the scheme that we just discussed does not have this property, and so the scheme is not appropriate.

2.2 A Fully Homomorphic Scheme

In order to make the above scheme Fully homomorphic, Gentry introduced the concept of bootstrapping. He defined a scheme to be bootstrappable if the set $\mathcal{F}_{\mathcal{E}}$ of permitted function includes the $\mathsf{Decrypt}_{\mathcal{E}}$function. The scheme $\mathcal{E}$ is transformed to $\mathcal{E}^*$ so that the new scheme becomes bootstrappable. This scheme requires two integer parameters $0 < \alpha < \beta$

- $\mathsf{KeyGen}_{\mathcal{E}^*}(\lambda)$: Run $\mathsf{KeyGen}_{\mathcal{E}}(\lambda)$ to obtain p. Generate a set $\vec{y} = \{y_1, y_2, ..., y_\beta\}$ such that $y_i \in [0, 2)$. Out of these elements, there must exist a sparse subset $\mathcal{S} \subset \vec{y}$ of α elements, such that $\sum_{y_j \in \mathcal{S}}(y_j) = \frac{1}{p} \mod 2$. Set sk to be a binary encoding s of the sparse subset $\mathcal{S}$, where $s = \{0, 1\}^\beta$. Set $pk \leftarrow \{p, \vec{y}\}$.

- $\mathsf{Encrypt}_{\mathcal{E}^*}(pk, m)$: Run $\mathsf{Encrypt}_{\mathcal{E}}(p, m)$ to obtain the cipher-text c. Generate $\vec{z} : z_i \leftarrow c.y_i \mod 2$. Return $c^* = \{c, \vec{z}\}$

- $\mathsf{Decrypt}_{\mathcal{E}^*}(sk, c^*)$: Output $\mathsf{LSB}(c)$ XOR $\mathsf{LSB}(\lfloor \sum_t \mathcal{S}_t z_t \rceil)$, where $\mathsf{LSB}()$ returns the least significant bit of th input, and $\lfloor . \rceil$ returns the nearest integer to the input. Decryption works since (up to small precision errors) $\sum_t \mathcal{S}_t z_t = \sum_t c.\mathcal{S}_t y_t = c/p \mod 2$.

The computations also get modified.

- $\mathsf{Add}_{\mathcal{E}^*}(c^*, c_1^*, c_2^*)$: Obtain c by running $\mathsf{Add}_{\mathcal{E}}(c, c_1, c_2)$. Compute $\vec{z}$ from c and $\vec{y}$ in a manner similar to the one explained in $\mathsf{Encrypt}_{\mathcal{E}^*}$.

$\mathsf{Mult}_{\mathcal{E}^*}$and $\mathsf{Sub}_{\mathcal{E}^*}$will be computed similarly.

3 Implementation

3.1 Expression Evaluator

The first and foremost function that had to written was an *Expression Evaluator*. This function would take in any expression E as an input in the form of a string, and a array containing the values of the variables in the expression. The function would then output the result of the expression, by replacing the variables by their values. For convenience, E needs to be fed in the form $\Big[m_1[O_1]m_2[O_2]...[O_k]m_{k+1}\Big]$, where O_i are operators, and m_i are variables.

The expression E is then converted to another where the variables are replaced by their actual values. If the values are taken from an array values, the new expression is represented by E[values]. An *InfixToPostfix Converter* is used to convert the *infix* expression E[values] to its *postfix* form E_p[values]. This postfix expression is then evaluated using a method based on stacks. This *Expression Evaluator* is used for implementing the $\mathsf{Evaluate}_{\mathcal{E}}()$ and $\mathsf{Evaluate}_{\mathcal{E}^*}()$ functions.

3.2 "Somewhat" Homomorphic Encryption

Implementing the somewhat homomorphic encryption scheme did not pose many difficulties. On obtaining the security parameter λ from the user, $\mathsf{KeyGen}_{\mathcal{E}}$ is run. Then, the number v of variables in the expression is scanned from the user. Thereafter, the boolean expression E is requested. One constraint which is imposed on E is that it may contain only AND and XOR operations. But this is not a problem, since any expression can be converted to a form having only ANDs and XORs. This conversion function has still not been coded, and is included as a future work in Section 4.

Once the expression is scanned, the bit values for each of the v variables are scanned and stored in the array bits, and consequently $\mathsf{Encrypt}_{\mathcal{E}}$ is run on each of the bits with the key generated from $\mathsf{KeyGen}_{\mathcal{E}}$ and the cipher-texts stored in ciphers. The expression E is now transformed into the expression E', where the ANDs and XORs are replaced with multiplications and additions respectively. E' and ciphers are fed into the *Expression Evaluator*, which emulates the $\mathsf{Evaluate}_{\mathcal{E}}$ function. The result from the $\mathsf{Evaluate}_{\mathcal{E}}$ function is fed into $\mathsf{Decrypt}_{\mathcal{E}}$ which returns the decrypted bit, which is stored in result. The correct value of the expression is computed from E and bits using the *Expression Evaluator*. If this value matches with result, then we have a success.

3.3 Fully Homomorphic Encryption

For the fully homomorphic scheme, a new structure was defined- PublicKey. This structure has an integer, and an array of reals. This structure is used to represent each cipher-text, as well as pk. sk is represented by an array of booleans. Integer parameters α and β are randomly generated. For implementing $\mathsf{KeyGen}_{\mathcal{E}^*}$, a sparse subset-sum problem is to be developed. In order to maintain the relation $\sum_{y_j \in \mathcal{S}}(y_j) = 1/p \mod 2$, each element that is to be part of the subset sum solution is assigned a value $\big(1/p + 2 * (\mathsf{rand} \mod \alpha)\big)/\alpha$.

For implementing the $\mathsf{Encrypt}_{\mathcal{E}^*}$ function, a new function has been written, which was named mod2. This function handles reals, and computes their values modulo 2. This was required for post-processing the values of $\vec{y}$ to compute $\vec{z}$. For the $\mathsf{Decrypt}_{\mathcal{E}^*}$ function, the LSB and the $\lfloor . \rceil$ functions were written, which weren't too difficult. $\mathsf{Evaluate}_{\mathcal{E}^*}$ remains mostly

the same as $\mathsf{Evaluate}_{\mathcal{E}}$, except that it, as well as $\mathsf{Encrypt}_{\mathcal{E}^*}$ and $\mathsf{Decrypt}_{\mathcal{E}^*}$, had to be modified to be compatible with the new structure PublicKey. The stack required to implement the *Expression Evaluator* had to be modified as well. What still remains to be implemented is the $\mathsf{Recrypt}_{\mathcal{E}}$ function, and has been added in Section 4

3.4 Security

The above mentioned implementation could only handle a security parameter of 2 (two) in the true sense, since for $\lambda = 3$, $Q \leftarrow \lambda^5 = 243$, which means handling a 243-bit number! So, initially, for initial implementation, Q was set to λ^2, for testing purposes. As the simpler version started giving satisfactory results, the next objective was to increase the security of the implemented scheme.

For this purpose, Matt McCutchen's C++ Big Integer Library [1] was first tried out, since it was easy to incorporate, and used the same operators for $+, -, *, /, \%$. But this library gave various compilation as well as run-time errors which could not be resolved. After this debacle, a more standardized and well-documented library was the need of the hour. The GNU Multiple Precision Arithmetic Library, popularly known as gmp, was the perfect winner. Although the installation required a lot of effort, but once it was done, there were no more complications. This is a free library for arbitrary-precision arithmetic, operating on signed integers, rational numbers, and floating point numbers, and there are no practical limits to the precision except the ones implied by the available memory in the machine gmp runs on. The gmp integer type is mpz_t, and that for real numbers is mpz_f, and every simple operation has its gmp-counterpart. This required major restructuring of the already written functions.

3.5 Timing Observations

λ	$\mathsf{KeyGen}_{\mathcal{E}^*}$	$\mathsf{Encrypt}_{\mathcal{E}^*}$	$\mathsf{Decrypt}_{\mathcal{E}^*}$
3	0.405 ms	0.145 ms	0.125 ms
5	0.421 ms	0.337 ms	3.43 ms
7	0.422 ms	4.2 ms	16.36 ms
9	0.438 ms	33.37 ms	24.54 ms
11	0.437 ms	187.02 ms	89.16 ms
13	0.434 ms	474.29 ms	215.94 ms
15	0.433 ms	0.99 sec	0.5 sec

Fig1: Table showing the variation of the Key Generation, Encryption and Decryption times with the security parameter λ

[1] https://mattmccutchen.net/bigint/

4 Future Work

The things that still need to be done have been classified into the following short term and long term goals.

4.1 Short Term Goals

AND-XOR Converter

As was mentioned in Section 3.3, a function which takes in any boolean function and outputs an equivalent circuit with only AND and XOR gates needs to be written.

$\mathsf{Recrypt}_{\mathcal{E}}$

The published implementations of the scheme so far [4] [3] have used lattices. So, there isn't enough documentation on how to implement the $\mathsf{Recrypt}_{\mathcal{E}}$ function using just integers. The other functions were comparatively simpler to implement without lattices, but $\mathsf{Recrypt}_{\mathcal{E}}$seems to be too complicated for a naive implementation. So, the theory of lattices needs to be studied, and applied for this scheme.

Extensive Testing

Once the scheme is properly implemented, extensive testing will be done to ensure that the scheme actually is fully homomorphic.

4.2 Long Term Goals

Efficient Implementation

In the 2011 implementation [3] by Gentry himself, it was observed that this scheme took too much time and memory to be used for practical purposes. In a toy-setting with dimension of 2^9, the public-key size was $17MB$, and recryption time was 6 *sec*, and this increased to 2.3 *GB* and 30 *mins* in a large-setting with a dimension of 2^{15}. Hence, we see that for a secure scheme, the requirements are huge, and just not practical for widespread application. One objective of the project is to obtain an efficient scheme for FHE, or perhaps just an efficient implementation of the existing scheme.

Side Channel Attacks

It would also be interesting to come up with some potential side-channel attacks on the scheme.

References

[1] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *STOC*, pages 169–178. ACM, 2009.

[2] Craig Gentry. Computing arbitrary functions of encrypted data. *Commun. ACM*, 53:97–105, March 2010.

[3] Craig Gentry and Shai Halevi. Implementing gentry's fully-homomorphic encryption scheme. In *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, volume 6632, page 129, 2011.

[4] N.P. Smart and F. Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. Cryptology ePrint Archive, Report 2009/571, 2009. `http://eprint.iacr.org/`.