Instruction Manual for **ECLIB**: Encrypted Controller Library

Minryoung Kim

June 9, 2020

1 Overview

"**ECLIB**" library provides C++ modules to convert a linear SISO controller into an encrypted controller.

The main goal of this library is to give designers an easy way to implement an encrypted controller from their original linear SISO controller and simulate its performance.

LWE-based cryptosystem allows homomorphic encryption and arithmetics. **Homomorphic Encryption** refers to a type of encryption technology that allows computation to be directly on encrypted data, without requiring any decryption in the process.

For a control system, if the controller is homomorphically encrypted, all the control operations are performed in encrypted state. So the controller doesn't have to know the secret key and plaintext of signals, thus the system attacker can't get information from controller access.

The structure of the control system that ECLIB will construct is depicted in the Figure 1.

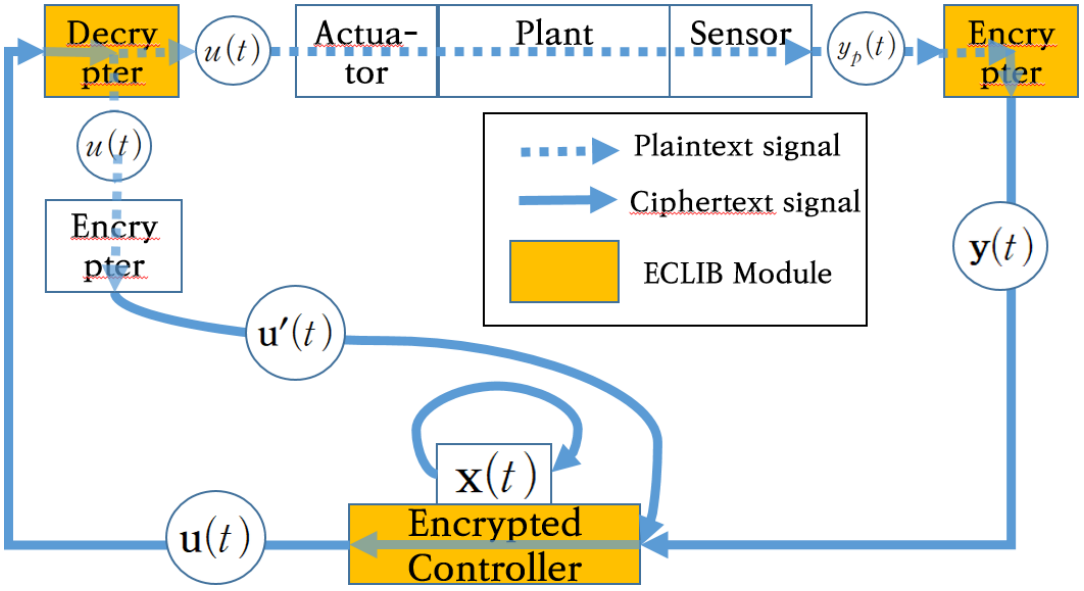

Figure 1: Control system structure of ECLIB
(Bold symbol represents encrypted signal.)

A linear SISO controller

(x: controller state, u: controller output, y: plant output)

$$x^+ = Fx + Gy$$
$$u = Hx + Jy$$

This library is composed of **three main modules**.

- **Encrypter**

  Encrypt signals(and also encrypt controller matrices while building system).

- **Decrypter**

  Decrypt signals.

- **Encrypted Controller**

  Performs control operation with encrypted data.

You can embed these modules into your devices to implement an encrypted controller.

Also, to help system implementation and test simulation, **ECLIB** provides auxiliary modules.

- **System Builder**

  With user input, decide proper parameters for system and build it.

- **Plant**

  Receives signal from actuator, updates plant state with the signal, and sends the plant output to the sensor.

- **Sensor**

  Receives the plant output signal, encrypts it, and then sends it to the encrypted controller.

- **Actuator**

  Receives controller output signal, decrypts it, and then sends it to the plant.

During controller building process, this library determines several conflicting parameters to ensure the performance desired by the user. Encrypted controller constructed by **ECLIB** will satisfy the following conditions.

[Performance conditions]

- Complete control operation within the controller's sampling cycle.
- Performance degradation due to controller encryption is relatively insignificant than user's desired bound.
- Guarantee the maximum security level while satisfying the above conditions.

2 Getting Started

This section explains a quick way to run a encrypted controller simulation in a virtual control system. If you are interested in implementation on a real system, skip to the next section.

2.1 How to build a simulation

Windows environment

- Open the project solution file '**EncryptedSystem.sln**' with Visual Studio.
- To build this project, choose **Build Solution**(F7) from the **Build** menu.
- To run the code, on the menu bar, choose **Debug** → **Start without debugging**(Ctrl+F5).

Also, you can run a simulation of a control system operation right away with the prepared example codes.

2.2 Making an input file

Input file '**parameters.txt**' specify the controller and the LWE-based cryptosystem. Each field in a line must be separated by a tab. Following parameters are entries of the input file 'parameters.txt'.

[Controller Parameters]

- $\boldsymbol{F, G, H, J}$: state space matrices in following form of controller
  (x: controller state, u: controller output, y: plant output)

$$x^+ = Fx + Gy$$
$$u = Hx + Jy$$

- **T_s**(T_s): sampling time(second)
- **1/r_y**($1/r_y$): resolution of plant output sensor
  (e.g. if y=1.46, then $1/r_y = 100$)
- **1/r_u**($1/r_u$): resolution of actuator
  (e.g. if u=4.821, then $1/r_u = 1000$)
- **U**(U): size bound of controller output
  (e.g. if possible output of controller range from -32 to 17, then U=32)

[Cryptosystem Parameters]

- **sigma**(σ): standard deviation of Gaussian noise to inject to the ciphertext(highly recommend 1.0 as value)
- **degrade_bound**($degrade$): desired upper bound of performance degradation ratio due to injected noise(recommend 0.01 as value)
  (e.g. if $degrade = 0.05$, then degradation ratio will be under 5 percent)

The input format of the parameters is shown in the Figure 2.

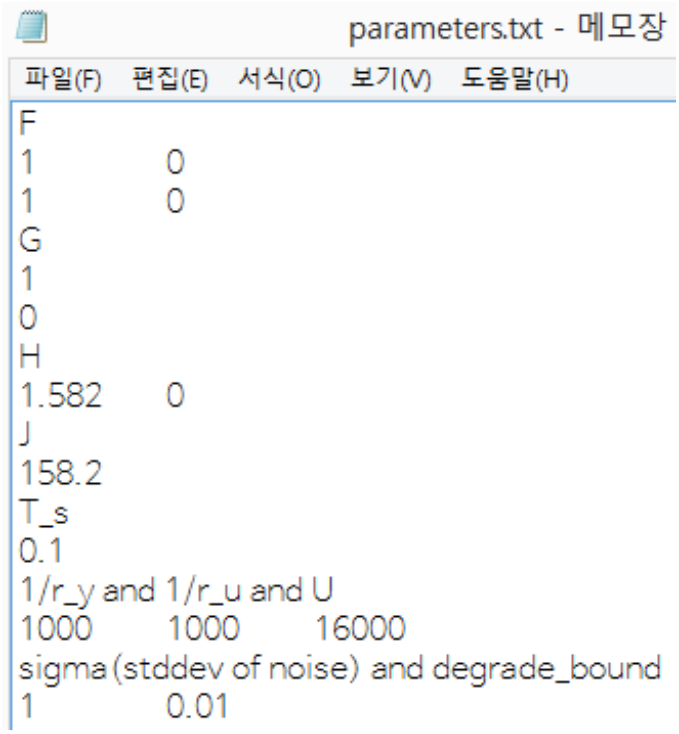

Figure 2: Example of input file '**parameters.txt**'

2.3 Getting an Output file

When running a control simulation, the default plant code generates an output file '**result.txt**' and records the the system status.

	A	B	C	D
1	Time(s)	r	y	r - y
2	0.1	25.75	97.1708	-71.4208
3	2.1	25.75	46.4889	-20.7389
4	4.1	25.75	16.2048	9.54524
5	6.1	25.75	15.3275	10.4225
6	8.1	25.75	22.4036	3.34642
7	10.1	25.75	25.5459	0.204148
8	12.1	25.75	25.3033	0.446725
9	14.1	25.75	24.6682	1.08182
10	16.1	25.75	24.6453	1.10473
11	18.1	25.75	24.9179	0.832139
12	20.1	25.75	25.1371	0.612889
13	22.1	25.75	25.2481	0.501853
14	24.1	25.75	25.3152	0.434795
15	26.1	25.75	25.379	0.370957

Figure 3: Example of output file '**result.txt**'(opened with a spreadsheet software)

You can find and modify the source code to change the output content and format, which is in '**SendOutputToSensor**() function of '**Plant.cpp**'. In the Figure 3, **r** represents reference signal value, **y** represents recorded value of plant output, and **r-y** is difference between the two.

3 API: Modules

There are codes of modules in **ECLIB** project folder. You can use them in your project or physical device by including each header file(.h) and source file(.cpp).

3.1 class Encrypter

Class **Encrypter** is a module for encryption and proper signal scaling. You can create an **Encrypter** object like this:

```
encrypter = new Encrypter(r_y_inverse, s_1_inverse, s_2_inverse, U, L_inverse, sigma,
    n);
```

- **r_y_inverse**($1/r_y$) is the resolution for plant output signal y.
- **s_1_inverse**($1/s_1$) and **s_2_inverse**($1/s_2$) are matrix scaling factors. $1/s_1$ scales matrices G, R, and J. $1/s_2$ scales H and J.
- **U** is the range size of controller output u.
- **L_inverse**($1/L$) is signal scaling factor for controller input signals. It reduces the effect of injected noise.
- **sigma**(σ) is standard deviation for Gaussian distribution of noise to inject to the ciphertext.
- **n** is ciphertext dimension.

With an **Encrypter** object, you can encrypt a signal like following line of code.

```
// y: a MatrixXd object, i.e. real number matrix
// Enc(signal, true) function auto-scale plant output y and return encryption of it.
MatrixXu encrypted_y = encrypter->Enc(y, true);
```

3.2 class Decrypter

Class **Decrypter** is a module for decryption, and proper signal re-scaling. You can create an **Decrypter** object from pre-made **Encrypter** like this:

```
decrypter = encrypter->GenerateDecrypter();
```

GenerateDecrypter() function of **Encrypter** class automatically generates a **Decrypter** object with the same secret key and parameters, and then return it.

With an **Decrypter** object, you can decrypt an encrypted signal like following line of code.

```
// u: a MatrixXu object, i.e. integer ciphertext matrix
// Dec_u(encrypted_signal) function decrypt ciphertext u, auto-rescale it, and then
    return it.
MatrixXd u = decrypter->Dec_u(enc_u);
```

3.3 class EncryptedController

```
controller = new EncryptedController(encm_FGR, encm_HJ, enc_x_con_init, q, actuator);
```

This function returns an encrypted controller object that was built with following parameters.

- **encm_FGR** and **encm_HJ** are state space matrices of the encrypted controller. They are encrypted matrices of converted and then scaled controller.
- **enc_x_con_init** is a ciphertext which has message of the initial state of controller.
- **q** is the size of cipherspace(ex: q:=2^{48}).
- **actuator** must be an actuator object that will receive controller output signal, decrypt it, and then send it to the plant.

3.4 class Actuator

This virtual **Actuator** object receives controller output signal, decrypts it, and then sends it to the plant.

```
actuator = new Actuator(encrypter);
actuator->SetPlant(plant);
actuator->SetController(controller);
```

- **encrypter** must be an **Encrypter** object that will decrypt the controller output signal, and also re-encrypt the signal.

- **plant** is expected to be an **Plant** object that will receive actuator signal from this actuator.
- **controller** is expected to be an **EncryptedController** object that will send an output $\mathbf{u}$ to this actuator and receive re-encrypted signal $\mathbf{u}'$ in return.

3.5 class Sensor

This virtual **Sensor** object receives the plant output signal, encrypts it, and then sends it to the controller.

```
sensor = new Sensor(controller, encrypter);
```

- **controller** is expected to be an **EncryptedController** object that will be get the sensor signal.
- **encrypter** must be an **Encrypter** object that will encrypt the plant output signal.

3.6 class Plant

This virtual **Plant** object receives signal from **Actuator**, updates plant state with the signal, and sends the plant output to the **Sensor**.

```
plant = new Plant(sensor);
```

- **sensor** must be a **Sensor** object that will get the plant output signal.

4 Building An Encrypted Controller System

4.1 class SystemBuilder

Class **SystemBuilder** provides default codes for encrypted system construction.

```
SystemBuilder* controlSystem = new SystemBuilder();
```

The following is the process of work that **SystemBuilder** does during above line of code.

1. Read the input file.
   - Get the state-space representation of original controller.
   - Get cryptosystem settings.
2. Build an **EncryptedController** object.
   - Build a controller with parameters read from input file.

     ```
     BuildController(T_s, F_precision, G_precision, H_precision, J_precision);
     ```

     (X_precision: precision of the matrix X from the input file)
   - Convert the controller to have integer state matrix, and then scale it by powers of 10 to quantize other matrices(process explained in 5.1).
   - Encrypt the controller.
3. Simulate one step control routine and compare that time cost with sampling time $\boldsymbol{T_s}$. Adjust $\boldsymbol{n}$ so that it has the largest possible value while guaranteeing control time constraint.
4. Construct an **Encrypter** object with $\boldsymbol{n}$ which was determined above.
5. Construct an **Actuator**, a **Sensor**, and a **Plant** objects and connect each of them.

4.2 Control loop

You can start control loop with following line of codes and the loop will continue forever until forced termination.

```
SystemBuilder* controlSystem = new SystemBuilder();
controlSystem->ControlLoop();
```

5 Brief Explanations For Processes

5.1 Converting process of controller

To encrypt a controller by LWE-based cryptosystem, its state-space representation must be composed of only integer matrices to be encrypted. Further, its state matrix(ex: F) needs to be integer matrix without scaling, because otherwise encrypted controller can't perform recursive state update infinitely.
Following routine shows the process of building an encrypted controller from the original controller model.

Colors of matrix symbols represent:

- real matrix

- integer matrix

- encrypted matrix

$$x(t+1) = F'x(t) + G'y(t)$$
$$u(t) = H'x(t) + J'y(t)$$

$$\Downarrow$$

Convert state-space to the observable canonical form

$$\Downarrow$$

$$x(t+1) = F''x(t) + G''y(t)$$
$$u(t) = Hx(t) + Jy(t)$$
$$x(t+1) = \begin{bmatrix} -a_1 & 1 & 0 & \cdots & 0 \\ -a_2 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -a_{l_1-1} & 0 & 0 & \cdots & 1 \\ -a_{l_1} & 0 & 0 & \cdots & 0 \end{bmatrix} x(t) + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{l_1} \end{bmatrix} y(t)$$
$$u(t) = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \end{bmatrix} x(t) + \begin{bmatrix} b_0 \end{bmatrix} y(t)$$

$$\Downarrow$$

Introduce u as a new input to make state matrix F an integer matrix

$$\Downarrow$$

$$x(t+1) = Fx(t) + Gy(t) + Ru(t)$$
$$u(t) = Hx(t) + Jy(t)$$
$$x(t+1) = \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ 0 & 0 & 0 & \cdots & 0 \end{bmatrix} x(t) + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{l_1} \end{bmatrix} y(t) + \begin{bmatrix} -a_1 \\ -a_2 \\ \vdots \\ -a_{l_1} \end{bmatrix} u(t)$$

$\Downarrow$

Scale matrices except for F, with $1/s_1$ and $1/s_2$,
so that they become integer matrices

$\Downarrow$

$$\bar{x}(t+1) = F \cdot \bar{x}(t) + \bar{G} \cdot \bar{y}(t) + \bar{R} \cdot \bar{u}(t)$$
$$\bar{u}(t) = \bar{H} \cdot \bar{x}(t) + \bar{J} \cdot \bar{y}(t)$$

$\Downarrow$

Encrypt the matrices and signals

$\Downarrow$

$$\mathbf{x}(t+1) = \mathbf{F} \cdot \mathbf{x}(t) + \mathbf{G} \cdot \mathbf{y}(t) + \mathbf{R} \cdot \mathbf{u}(t)$$
$$\mathbf{u}(t) = \mathbf{H} \cdot \mathbf{x}(t) + \mathbf{J} \cdot \mathbf{y}(t)$$