University of Notre Dame
Department of Computer Science and Engineering

CSE 40622 Cryptography
Fall Semester, 2022

# Programming Assignment 01 - RSA

- Please strictly follow the instruction!
- Your source code will be compiled and run at docker images of Gradescope automatically.

## 1 Objectives of this PA

- Install the GMP library.
- Correctly decrypt a ciphertext of RSA.
- Correctly encrypt a message using RSA.

## 2 Instruction

1. Download and install GMP library which supports number theoretic functions.
   - `https://gmplib.org`
   - Please do NOT use CRC machines / student machines for the PAs. You won't be able to complete PA03 and PA04 on student machines.
   - Please install Virtual Box (https://www.virtualbox.org/wiki/Downloads), download an Ubuntu OS image (https://ubuntu.com/), and install Ubuntu in a virtual machine using the Virtual Box.
   - Please do NOT use Window Subsystem for Linux (WSL) for PAs. WSL used to have some issues with the libraries that you will use in PA03 and PA04. I do not know whether they have been fixed or not.
2. Use the library to implement the algorithms of RSA. Generate the pair of public and secret keys – $n$ must be larger than 1,000 bits.
   - For autograding purposes, please write everything in one file named either `pa01.c` or `pa01.cpp`.
   - You will get 0/100 if you name it `PA01.c` or `PA01.cpp`.
3. We will provide you an input string. Process the given string using your implemented algorithms, and generate the required output by following the instructions below.
4. You may debug and resubmit your code as many times as you want until you get 100 pts.

## 3 Input/output requirements

**Format of the provide input** Your code needs to read in an input file with the following format.

$$m, c', d', p', q'$$

An example is available at: `https://notredame.box.com/s/0fnvz49eq3grhotooifg1qzxgjdt8ant`

- The string will be provided to you in the file "`./input`" such that the input file's name is "`input`" and it is available in the directory/folder where your source code is.
  - For example, one can use the following code to open the file pointer in C and use it to read the input file.

```
FILE *fp;
fp = fopen("./input", "r");
```

1. Encrypt $m$ and generate the ciphertext of it (denoted as $c$) using your own keys. Denote the private key as $d$, the public key as $e$, and the modulus as $n$.
2. Use the provided $p', q'$ to decrypt the $c'$ to get the message, denoted as $m'$.

**Output that needs to be generated** Your code needs to generate an output file with the following format, which has the parameters you generate from the input.

$$c, e, d, n, m'$$

An example is available at `https://notredame.box.com/s/28r4tupznhzksqancs8z13rcsg92hlu8`

- There cannot be space between the values. Only the commas should separate the variables.
- The output file must be generated with the path "`./output`" such that the output file's name is "`output`" and it is generated in the same directory/folder. If your output file is named `output.txt`, you will get 0/100 because my grading program will not recognize it.
  - For example, one can use the following code to open the file pointer in C and use it to write the output file.

```
FILE *fp;
fp = fopen("./output", "w+");
```

# 4 Suggestions/tips

- When you learn how to use the library, looking for example codes/tutorial is helpful.
- GMP library is widely used all over the world. If you have any question, that was probably already asked/answered somewhere in the Internet. Try to find it by Googling. In case you don't find the answer, feel free to contact the instructor by DM/email. You are also encouraged to seek TAs' help during their office hours.
- After successfully installing the library, you may find these chapters useful from their documentation (`https://gmplib.org/manual/`)
  - Integer Functions: All modular operations as well as number theoretic functions are described here.
  - Random Number Functions: Functions in this chapter need to be used to generate random numbers.

  - Formatted Output: Good for debugging.
  - Formatted Input: Good for debugging.

  Of course, other chapters are worth reading too.

- Any big integer needs to be declared and initialized before using. All declared big integers need to be cleared at the end of the program.
  - ... because a big integer variable is in fact a memory area dynamically created with `malloc` in C.

# 5 Rubric of the grading program

Our grading program will parse your output, and the following rubric will be applied.

| **Description** | **Score** |
|---|---|
| $n$ is smaller than 1,000 bits. | -15 |
| $c$ is larger than $n$ (invalid ciphertext format). | -15 |
| The student-provided $e$ cannot be used to generate the student-provided $c$ using the $m$ in the input. | -15 |
| The student-provided $d$ cannot be used to decrypt the student-provided $c$ correctly. | -15 |
| The student-provided $m'$ is not equal to the original message. | -15 |

University of Notre Dame
Department of Computer Science and Engineering

CSE 40622 Cryptography
Fall Semester, 2022

# Programming Assignment 02 - Secure ElGamal

- Please submit the source code to Gradescope. Your code must be named `pa02.c` or `pa02.cpp`.

## 1 Objectives of this PA

- Generate a large-enough prime number.
- Generate a random number with sufficient randomness.
- Find a generator from $\mathbb{Z}_p^*$.
- Defend against QR/QNR attacks.

## 2 Instruction

1. Keep using the GMP library you used for PA01.
2. Use the library to implement all algorithms in the **regular** ElGamal encryption (Section 1 in the note for Lecture 06-08). Your encryption needs to be secure against QR/QNR attacks discussed in Lecture 09-10.
3. Students need to submit their source codes to Gradescope by the deadline.

   Checklists:

   - Is your encryption correct?
   - Is your modulus large enough?
     - As long as your modulus is longer than 2000 bits, I consider it sufficient.
     - Because $p$ is longer than 2000 bits, finding the right prime number may take a long time. It took the instructor's laptop 15 minutes to find the right prime number to use.
   - Are you fully utilizing the group? In other words, did you indeed choose a generator?
   - Does your encryption provide sufficient randomness?
     - As long as you use the random number generator in GMPlib, I consider it sufficient.
   - Is your encryption vulnerable to QR/QNR attack?

## 3 Submission, Input/output requirements, and rubric

**Format of the provided input** Your code needs to read in an input file with the following format.

| $m$ |
|---|

An example is 10000, which is clearly a QR modulo any number larger than 10000.

- The string will be provided to you in the file "`./input`" such that the input file's name is "`input`" and it is generated in the directory/folder where your source code is.

- For example, one can use the following code to open the file pointer in C and use it to read the input file.

```
FILE *fp;
fp = fopen("./input", "r");
```

- The input is the message that you need to encrypt.
- For the grading purpose, please use $x = 1234567890123456789012345678901234567890$ as the private key. This is a large integer that has to be stored in a mpz_t variable.
  - You may hard-code this value.
  - In reality, such a short key is VERY dangerous.
  - This key is still large enough, and it cannot be stored in any normal integer types (e.g., int, long, long int, unsigned int, unsigned long int). If you use gmp functions with `_ui` to handle this key, it will cause correctness issues.

**Output that needs to be generated** You need to generate and submit 3 lines of strings with the following format.

$$c_1^1, c_2^1, p$$
$$c_1^2, c_2^2, p$$
$$c_1^3, c_2^3, p$$

An example is available at `https://notredame.box.com/s/z53gotsqy9nbjh0k115p5018mlsrvht1`.

Encrypt the same message $m$ for 3 times, and provide the output values of $c_1, c_2, p$ from each time in different lines. You need to use the same $p$ once your encryption scheme is set up, so $p$ should be same for all lines. This $p$ is included in every line for the simplicity of our grading.

- There cannot be space between the values. Only the commas should separate the variables.
- 3 lines of strings should be provided.
- Between each line, there is no extra empty line or space. Only one character '\n' should be used to change the line.
- The output file must be generated with the path "`./output`" such that the output file's name is "`output`" and it is generated in the same directory/folder.
  - For example, one can use the following code to open the file pointer in C and use it to write the output file.

```
FILE *fp;
fp = fopen("./output", "w+");
```

The following rubric will be applied in grading (Minimum score is 0).

- Note: We check whether you are reusing the prime number $p$ in our sample output in our grading. Therefore, if you feed our sample output to the grading program, it will give 15-point deduction because of the rubric item *The same prime number $p$ in the sample output is re-used.*

| Description | Deduction |
|---|---|
| Encryption is not correct, *i.e.,* the grading program failed to recover the correct $m$ | -30 |
| Allows DLOG becuase the modulus $p$ is not at least 2000 bits | -15 |
| Did not choose $p$ properly, making it intractable to find a generator of $\mathbb{Z}_p^*$ | -15 |
| * Does not provide sufficient randomness, *i.e.,* did not use random number generator in GMPlib | -15 |
| Vulnerable to QR/QNR attack | -15 |
| The $p$ in the sample output is re-used | -15 |
| Hardcoded $p$ in the source code | -15 |
| 3 lines contain the same ciphertexts or there are no 3 lines of ciphertexts | -15 |

* No program can "measure" randomness from one or just several ciphertexts. I will consider your ciphertexts provide sufficient randomness if you used the random number generators in the GMP library.

## 4 Suggestions/tips

- These chapters in `https://gmplib.org/manual/` will be useful.
  - Integer Functions: All modular operations as well as number theoretic functions are described here.
    - Specially, Integer Random Numbers in this chapter will be useful.
  - Random Number Functions: Functions in this chapter need to be used to generate random numbers.
  - Formatted Output: Good for debugging.
  - Formatted Input: Good for debugging.
- When in doubt about how to use a function, try to google "mpz_FUNCTIONNAME example".
- Try to verify your own result before posting the outputs to Sakai. The function "mpz_legendre" at GMPlib (`https://gmplib.org/manual/Number-Theoretic-Functions.html`) is useful for checking whether your ciphertexts are secure against QR/QNR attacks.

University of Notre Dame
Department of Computer Science and Engineering

CSE 40622 Cryptography
Fall Semester, 2022

# Programming Assignment 04 - HEAAN for $x^{64} + x^{17} - 5x^5$

- Please submit the required source code to Gradescope.

## 1 Objectives of this PA

- Let the students learn how to use the state-of-the-art FHE library for homomorphic evaluation with bootstrapping.

## 2 Instruction

1. Download the HEAAN library from `https://github.com/snucrypto/HEAAN`
2. Install all the libraries that HEAAN depends on and install the HEAAN library.
   - When you install the NTL library, you may need to uninstall GMP-6.2.1 and install GMP-6.1.2.
   - GMP-6.1.2 is available at `https://gmplib.org/download/gmp/gmp-6.1.2.tar.bz2`
3. Follow the instruction in the subsection "How to use this library?" on `https://github.com/snucrypto/HEAAN` to compile the `test.cpp`.
   - The instruction is a bit outdated, but typing "make" in the `/run` directory will still compile the `test.cpp`.
   - You just need to make sure compilation succeeds.
4. Read `HEAAN/src/Scheme.h` to learn what APIs are available for homomorphic evaluation.
5. Read `HEAAN/src/StringUtils.h` to learn what APIs are available for printing/comparing the plain-texts/ciphertexts.
6. Read `HEAAN/src/TestScheme.cpp` to learn how APIs are called.
   - In the testing of bootstrapping in `TestScheme.cpp`, they called all the subprocedures of `Scheme::bootstrapAndEqual` instead of calling it directly. It would be easier to call `Scheme::bootstrapAndEqual` directly in this PA03.
7. Modify the `test.cpp` posted on the course website such that it generates the required output.
   - There is an area in the code where it says "Student input starts." Write your code there. Do NOT modify anything else.
   - Before that code, an input $x$ has been generated and stored in the variable `mval`. Your code needs to encrypt it and homomorphically compute a ciphertext that contains $x^{64} + x^{17} - 5x^5$.
   - You are NOT modifying the `test.cpp` in the original repository. You are modifying the one posted on the course website.
   - I showed some sample codes in the `test.cpp` along with comments. To complete PA03, you will need to read the header files in Instruction 4 and Instruction 5 above to learn other APIs as well.

# 3 Submission

Submit the modified `test.cpp` to Gradescope.

**Rubric** If the code correctly uses homomorphic evaluation (i.e., a series of homomorphic multiplication operations and bootstrapping operations) and generates the ciphertext that contains $x^{64} + x^{17} - 5x^5$, the student gets 100 pts and 0 pts otherwise.

- There may be ways to generate a cipher that passes the automatic grading without a valid homomorphic evaluation. Instructor will check everyone's code and apply 0 pts if this is the case.
- The CKKS scheme is an approximate homomorphic encryption scheme. Your final outcome will not be exactly equal to $x^{64} + x^{17} - 5x^5$. A range test will be done to determine whether your ciphertext contains a value that is close enough.

University of Notre Dame
Department of Computer Science and Engineering

CSE 40622 Cryptography
Fall Semester, 2022

# Programming Assignment 04 - BLS signature

## 1 Instruction

1. Install the PBC library by following the instructions at `https://crypto.stanford.edu/pbc/`.
   - Attention: It is not Palm Beach County Library!
2. Keep using the GMP library, because the PBC library relies on the GMP library.
3. Input will be given as a file `./input`. You need to process it and generates the signatures.

## 2 Objectives of this PA

- Initialize and use the symmetric pairing function – Type A pairing – **without** using the test/demo functions, *i.e.,* without using `pbc_test.h`.
- Use element functions (Chapter 3. in the manual) and pairing functions (Chapter 4. in the manual).
- Sign a given message using the simplified BLS signature scheme in Section 2.3 in the note.
- Verify whether a given message matches a signature or not.

## 3 Suggestions/tips

- After installing the PBC library, follow the tutorial in the manual to compile and run the `bls.c` to make sure you know how to compile and run existing codes. If everything is correct, you should see the following in your terminal:

```
Taehos-MacBook-Pro:grading tjung$ gcc bls.c -o bls.o -lgmp -lpbc -I/usr/local/include/pbc
Taehos-MacBook-Pro:grading tjung$ ./bls.o < ./a.param
Short signature test
system parameter g = [70352909971763275197269830063567177307552945153305309165514609504008352804542064892925309792508755671514981521001668071656835687025603999670173
33808583455, 81145989376691461857450829854659085649021342852075043138195731035722978701322860049832813697093408183895687574496927250820973029914802039702203165668473
37]
private key = 618743979612706426851734146824961347202524188213
public key = [11942670096663133516657803073638121424151119822141256705425014718756425796880321087566504872982950031483426805653824003152206520979560077064359264446665
27, 19692256901580426054709188329410419245512080729051031619500586826463552870935953606791159907674547617010316663182187514479355447708637159865173604529O5271]
message hash = [63080756773523367909062812632949867296616317693870196912577786307205893019756592476322806845140575717237642871126538707348268399299865847109892448262
73472, 61424547010085035811888713078797014264878170743386477964166514580538662193682652347062703040342638904519907575415022232509189070476825549795111247747511O9]
signature = [20885712630562573759970785033721419818543008703300080305720273349388059351834978959123777213699602355905366701539054776789547886636479413559119091766682
37, 21626769502724200946647842659006868375217333606853110951611502487664215922920568335048241267076956940178942885401847111912794035474817819583700211430375966]
compressed = 27E0B943041A26CB7FAF739387A5C61F78EFEBF83863FAEF42C4028CFDD2375478BAA21EA5569D08CBF321AEF5C4A0789C09BC64645B249D6B6193176DC2704D00
decompressed = [20885712630562573759970785033721419818543008703300080305720273349388059351834978959123777213699602355905366701539054776789547886636479413559119091766
68237, 21626769502724200946647842659006868375217333606853110951611502487664215922920568335048241267076956940178942885401847111912794035474817819583700211430375966]
f(sig, g) = [43850552999773237818049992577548251897575077862439596290657637140954174764636456491334239738404829370554172660702950743498513993233076289287097055423268
19, 53995180330943581037093503338007617783321544795432608929440087408567730169462180985784566173746997702872650968806187982151142147915265497521809510545352 27]
f(message hash, public_key) = [43850552999773237818049992577548251897575077862439596290657637140954174764636456491334239738404829370554172660702950743498513993233076
28928709705542326819, 53995180330943581037093503338007617783321544795432608929440087408567730169462180985784566173746997702872650968806187982151142147915265497521809
51054535227]
signature verifies
x-coord = 27E0B943041A26CB7FAF739387A5C61F78EFEBF83863FAEF42C4028CFDD2375478BAA21EA5569D08CBF321AEF5C4A0789C09BC64645B249D6B6193176DC2704D
de-x-ed = [20885712630562573759970785033721419818543008703300080305720273349388059351834978959123777213699602355905366701539054776789547886636479413559119091766682 37
, 66180338493908924277729977188533629782851498387288971158675031505000540385881661235738010527149665274052615702293976055464836778924996617412249185678488 25]
signature verifies on second guess
random signature doesn't verify
Taehos-MacBook-Pro:grading tjung$
```

- You do not need to call functions with preprocessing (those with `_pp_` in the name).
- These chapters in `https://crypto.stanford.edu/pbc/manual/` will be useful.

  - 3. Pairing functions
  - 4. Element functions

- The example code `bls.c` in `pbc-0.5.14/example` does not implement the simplified BLS signature scheme, so please consult it without copying it.

- The order of the group $\mathbb{G}_1$ and $\mathbb{G}_T$ are both $r$, which is a prime number. Therefore, the exponents form an integer set $\mathbb{Z}_r$.

- The library supports both $+$ and $\circ$ (multiplication), and it interprets your input first to decide which operation it performs. If the input arguments are illegal (*e.g.,* trying to add a number in `Zr` to a point in `G1`), the behavior of the functions are not defined. It may be correct by coincidence, or it may output random values, or it may simply return an error message.

  - `element_add(R, P, Q)` is same as `element_mul(R, P, Q)` if $R, P, Q$ are all initialized as points on `G1`. They both perform $R := P \circ Q$.
    - If one of $P$ and $Q$ is initialized as a number in `Zr`, `element_mul(R,P,Q)` performs $R := P^Q$ (if $Q \in \mathbb{Z}_r$) or $R := Q^P$ (if $P \in \mathbb{Z}_r$).
  - `element_neg(P, Q)` is same as `element_invert(P, Q)` if $P, Q$ are initialized as points on `G1`. They both perform $P := Q^{-1}$.
    - If $P, Q$ are both initialized as numbers in `Zr`, `element_neg(P,Q)` performs $P := -Q \mod r$ and `element_neg(P,Q)` performs $P := Q^{-1} \mod r$, which is allowed since $(\mathbb{Z}_r, +_r, \times_r)$ is not only a ring but also a field with a prime $r$.
  - `element_double(P,Q)` is same as `element_square(P,Q)` if $P, Q$ are initialized as points on `G1`. They both perform $P := Q^2$.
    - If $P, Q$ are both initialized as numbers in `Zr`, `element_double(P,Q)` performs $P := 2Q \mod r$ and `element_square(P,Q)` performs $P := Q^2 \mod r$.

# 4 Guides to follow (mandatory)

**If you don't follow this guide, it's likely that the grading program does not correctly recognize your output.** You will not get full credits.

- NEVER perform element arithmetic directly between `element_t` and `mpz_t` or `element_t` and `signed long int`. If you wish to, assign `mpz_t` or `signed long int` to `element_t` first before performing arithmetic calculation.

- In most operating systems, you need to specify the location of the library in compilation. For example, in Linux/Unix-based OS, you may use the following options with gcc.

```
gcc XXX.c -lpbc -lgmp -I/usr/local/include/pbc
```

  The path following `-I` should be the path where the pbc is installed (NOT the path where it is downloaded).

- Please use `a.param` for initializing the `pairing_t` parameter.
  - Please follow the tutorial `https://crypto.stanford.edu/pbc/manual/ch02.html` to learn how elliptic curve/pairing parameters are loaded and how pairing functions are initialized.
  - The source codes in `pbc-0.5.14/examples` use `pbc_demo_pairing_init` in `pbc_test.h` header, which is not allowed in this PA. You may, however, look into `pbc-0.5.14/include/pbc_test.h` to learn how `a.param` is loaded.

- Then, **hardcode the parameters** (i.e., store the entire `a.param` as a string and use it).
- `https://crypto.stanford.edu/pbc/manual/ch08s03.html` describes the details of the elliptic curve and the pairing, but you do not need to know about them.

- Please use
  - `element_printf` and `element_set_str` in `https://crypto.stanford.edu/pbc/manual/ch04s07.html` for printing an element and and reading into an element.
  - `element_from_hash` in `https://crypto.stanford.edu/pbc/manual/ch04s03.html` as the cryptographic hash. When you do so, please apply the hash to the entire line of the message that you read from the input file. This line would include the character '\n' which is intended.

- Please implement a simplified BLS signature scheme in Section 2.3, with further simplifications shown below:
  - $e$ is a symmetric pairing function $\mathbb{G}_1 \times \mathbb{G}_1 \to \mathbb{G}_T$.
  - Consequently, $\mathbb{G}_2$ and $g_2$ are substituted with $\mathbb{G}_1$ and $g_1$ respectively.

# 5 Submission, Input/output requirements, and rubric

### Input

$$g_1$$
$$m$$
$$\mathsf{sk}$$

- The input file will contain three lines of strings:
  - $g_1$, the generator of the group $\mathbb{G}_1$ (which is a point on the elliptic curve described by `a.param`).
    - Recall that the pairing function initialized by `a.param` is symmetric, therefore the pairing is $e : \mathbb{G}_1 \times \mathbb{G}_1 \to \mathbb{G}_T$. You need only one generator for $\mathbb{G}_1$.
  - A string message $m$ to be signed.
  - The private key $\mathsf{sk}$ for signing the message.

- The strings will be provided to you in the file "`./input`" such that the input file's name is "`input`" and it is generated in the directory/folder where your source code is.
  - For example, one can use the following code to open the file pointer in C and use it to read the input file.

```c
FILE *fp;
fp = fopen("./input", "r");
```

  - Please read the entire line for each line of the input file. By doing so, the second line you read in will contain the message and the character '\n', and you need to apply the hash to the entire line including the character '\n'.

**Output that needs to be generated** You need to generate and submit one string with the following format.

| |
|---|
| $\sigma$ |

- Sign the given message $m$ using sk, and provide the signature $\sigma$.
  - $\sigma$ should be given as its full coordinates $[x_\sigma, y_\sigma]$, just like that in the sample input. `element_printf("%B\n", P)` shows a point `P`'s coordinates in that format.
- The output file must be generated with the path "`./output`" such that the output file's name is "`output`" and it is generated in the same directory/folder.
  - For example, one can use the following code to open the file pointer in C and use it to write the output file.

```
FILE *fp;
fp = fopen("./output", "w+");
```

The following rubric will be applied in grading (minimum score is 20).

| Description | Deduction |
|---|---|
| Signature is not correctly generated | -40 |
| Contains `pbc_test.h` in the source codes. | -40 |