# Homework 4 - Elliptic Curves and Symmetric Key Cryptography

*Cryptography and Security 2018*

- You are free to use any programming language you want, although SAGE is recommended.
- Put all your answers **and only your answers** in the provided SCIPER-answers.txt file. This means you need to provide us with $Q_{1a}$ $Q_{1b}$, $Q_2$, $Q_3$, $Q_4$ and $Q_5$. You can download your **personal** files on `http://lasec.epfl.ch/courses/cs18/hw4/index.php`
- The answer $Q_{1a}$ should be a list of integers, the answer $Q_{1b}$, $Q_4$ and $Q_5$ should be strings in base64, and $Q_2$ and $Q_3$ should be English phrases in ASCII. **Please provide nothing else. This means, we don't want any comment and any strange character or any new line** in the answers.txt file.
- We also ask you to submit your **source code**. This file can of course be of any readable format and we encourage you to comment your code.
- The plaintexts of most of the exercises contain some random words. Don't be offended by them and Google them at your own risk. Note that they might be really strange.
- If you worked with some other people, please list all the names in your answer file. We remind you that **you have to submit your own source code and solution**.
- We might announce some typos in this homework on Moodle in the "news" forum. Everybody is subscribed to it and does receive an email as well. If you decided to ignore Moodle emails we recommend that you check the forum regularly.
- Please contact us as soon as possible if a server is down.
- Denial of service attacks are **not** part of the homework and will be logged (and penalized).
- The homework is due on Moodle on **Friday 30th November** at **22:00**.

## Exercise 1 Hashed Elgamal: On the Ugly but Crucial Face of Decoding/Encoding Functions

Field Elements and Curve Points.
Let $E = (a, b, G, p, q)$ denote an elliptic curve, i.e. $E = \{(x, y) \in \mathbb{F}_p^2 : y^2 = x^3 + ax + b\} \cup \{\mathcal{O}\}$, where $\mathcal{O}$ denotes the neutral element. We use additive notation for groups. Let $G \in E$ be such that $|\langle G \rangle| = q$ for some prime $q$. Suppose $\lambda = |p| = |q|$ and $\lambda$ is divisible by 8. Let $\{\texttt{0x00}, \texttt{0x01}, \ldots, \texttt{0xFF}\}$ denote the set of byte values, whose integer equivalents are $\{0, 1, \ldots, 255\}$ respectively. Let $|\cdot|$ be bitsize operator. The symbol $\perp$ is used for error. $\{0,1\}^\lambda$ denotes the set of bit strings whose size is exactly $\lambda$.

We handle different elements as follows:

- $\mathsf{encode}_{\mathbb{F}} : \mathbb{F}_p \to \{0,1\}^\lambda$. Maps a field element $x$ into a string of bytes $B_{\lambda/8-1} \ldots B_1 B_0$ such that $\sum_{i=0}^{\lambda/8-1} B_i \cdot 256^i = x$. Essentially, any field element $x \in \mathbb{F}_p$ is encoded to fixed-length string, that is of $\lambda/8$ characters just as in SAGE. Recall that SAGE considers strings to be byte arrays incidentally. Note that the definition requires encoding of any *small* element to be left padded with sufficient number of zeros.

- $\mathsf{decode}_{\mathbb{F}} : \{0,1\}^\lambda \to \mathbb{F}_p/\perp$. The inverse of $\mathsf{encode}_{\mathbb{F}}(\cdot)$. On input $y$, it returns a valid field element only if there exists $x \in \mathbb{F}_p$ whose encoding is $y$; **otherwise it raises error** and returns $\perp$ (your code does not necessarily need to return $\perp$ symbol, we use it to denote the occurrence of an error).

- $\mathsf{encode}_{\mathbb{Z}} : \mathbb{Z}_q \to \{0,1\}^\lambda$. Maps a ring element $x \in \mathbb{Z}_q$ into a string of bytes $B_{\lambda/8-1} \ldots B_1 B_0$ such that $\sum_{i=0}^{\lambda/8-1} B_i \cdot 256^i = x$.

- $\mathsf{decode}_{\mathbb{Z}} : \{0,1\}^\lambda \to \mathbb{Z}_q/\perp$. is defined as the inverse of $\mathsf{encode}_{\mathbb{Z}}(\cdot)$. On input $y$, it returns a ring element only if there exists $x \in \mathbb{Z}_q$ whose encoding is $y$; **otherwise it raises error**.

- $\mathsf{encode}_P : E \to \{0,1\}^{\lambda+8}$. Maps any point $P$ of elliptic curve into a string of $(\lambda/8) + 1$ bytes. Let $B_{\lambda/8} \ldots B_1 B_0$ be the encoding of $P$,
  - if $P = \mathcal{O}$, then $B_i = \texttt{0x00}$ for all $i \in \{0, 1, \ldots, \lambda/8\}$.
  - if $P \neq \mathcal{O}$, then let $(x, y)$ be the coordinates of $P$. Then $B_{\lambda/8} \ldots B_2 B_1 = \mathsf{encode}_{\mathbb{F}}(x)$. $B_0$ is set based on the parity of $y$, that is $B_0 = \texttt{0x0F}$ if $y$ is odd, and $B_0 = \texttt{0xFF}$ if $y$ is even.

- $\mathsf{decode}_P : \{0,1\}^{\lambda+8} \to E/\perp$. is the inverse of $\mathsf{encode}_P(\cdot)$. For decoding, any $(\lambda/8 + 1)$-byte string is valid only if there is a point $P$ on $E$ whose encoding returns the given input. For example, $(\texttt{0x00})^{\lambda/8}\texttt{0x10}$ should **raise error**.

- Message domain is $\bigcup_{i=1}^{32} \{0,1\}^{2^i}$, i.e. set of strings whose bitlength can be squeezed into four bytes.

- $\mathsf{encode}_4 : \{0, 1, \ldots, 2^{32} - 1\} \to \{0,1\}^{32}$ truncates/pads given integer value to fit into four bytes as follows: $X$ is mapped to byte sequence $B_0 B_1 B_2 B_3$ such that $\sum_{i=0}^{3} B_i \cdot 256^i = X$. (Notice the reverse order compared to $\mathsf{encode}_{\mathbb{F}}$).

- Since field elements, elliptic curve points and messages can be encoded as strings, any string operation such as concatenation (denoted with $|$) or XOR-ing can be done with them. We further define cut-then-xor operator $\oplus_{\mathsf{L}}$ over two bitstrings $h$ and $m$, that

$\underline{\mathsf{Kg}(E)}$:
1: $\mathsf{sk} \leftarrow_{\$} \mathbb{Z}_q$
2: $\mathsf{sk}_{\mathsf{en}} \leftarrow \mathsf{encode}_{\mathbb{Z}}(\mathsf{sk})$
3: $\mathsf{pk} \leftarrow \mathsf{sk} \cdot G$
4: $\mathsf{pk}_{\mathsf{en}} \leftarrow \mathsf{encode}_P(\mathsf{pk})$
5: **return** $(\mathsf{sk}_{\mathsf{en}}, \mathsf{pk}_{\mathsf{en}})$

$\underline{\mathsf{Enc}(E, \mathsf{pk}_{\mathsf{en}}, m)}$:
1: $\mathsf{pk} \leftarrow \mathsf{decode}_P(\mathsf{pk}_{\mathsf{en}})$
2: $x \leftarrow_{\$} \mathbb{Z}_q$
3: $X \leftarrow x \cdot G$
4: $X_{\mathsf{en}} \leftarrow \mathsf{encode}_P(X)$
5: $K \leftarrow x \cdot \mathsf{pk}$
6: $K_{\mathsf{en}} \leftarrow \mathsf{encode}_P(K)$
7: $\ell \leftarrow \mathsf{encode}_4(|m|)$
8: $h \leftarrow \mathsf{HashStream}(\ell | K_{\mathsf{en}})$
9: $c \leftarrow h \oplus_{\mathsf{L}} m$
10: $ct \leftarrow c | X_{\mathsf{en}}$
11: **return** $ct$

$\underline{\mathsf{Dec}(E, \mathsf{sk}_{\mathsf{en}}, ct)}$:
1: $\mathsf{sk} \leftarrow \mathsf{decode}_{\mathbb{Z}}(\mathsf{sk}_{\mathsf{en}})$
2: **if** $\neg(2^{32} > |ct| - \lambda - 8 > 0)$ **then**
3: **return** $\perp$
4: **end if**
5: $c | X_{\mathsf{en}} \leftarrow ct$ such that $|X_{\mathsf{en}}| = \lambda + 8$
6: $\ell \leftarrow \mathsf{encode}_4(|c|)$
7: $X \leftarrow \mathsf{decode}_P(X_{\mathsf{en}})$
8: $K \leftarrow \mathsf{sk} \cdot X$
9: $K_{\mathsf{en}} \leftarrow \mathsf{encode}_P(K)$
10: $h \leftarrow \mathsf{HashStream}(\ell | K_{\mathsf{en}})$
11: $m \leftarrow c \oplus_{\mathsf{L}} h$
12: **return** $m$

Figure 1: $\mathsf{sk}, \mathsf{pk}$ denotes secret decryption key and public encryption key respectively. We use "$\mathsf{en}$" subscript to refer to the encoded versions of variables. $E$ refers to elliptic curve parameters, that is **brainpoolP384r1**. For brevity, decoding of $E \to (a, b, G, p, q)$ is removed from the code and can be interpreted as global variables. "$\leftarrow_{\$}$" symbol denotes sampling a random element from given domain. "$|\cdot|$ denotes the bitlength (not bytelength).

are not necessarily of same size. It cuts the larger operand and takes its leftmost bits. **We use index 0 to address the leftmost bit of a bitstring**. More formally, let $h = h_0 h_1 \dots h_{|h|-1}$ and $m = m_0 m_1 \dots m_{|m|-1}$ where each $h_i$ and $m_i$ denotes a bit, and let $m \oplus_{\mathsf{L}} h = c_0 c_1 \dots c_{k-1}$. Then $k = \min\{|h|, |m|\}$ and $c_i = h_i \oplus m_i$ for $i \in \{0, 1, \dots, k-1\}$.

- Let $\mathsf{SHA256}$ be SHA256 hash function. Hash stream $\mathsf{HashStream}(seed)$ is a stream cipher producing upto $2^{40}$ bits of padding that works as follows. With ($seed$) input; $\mathsf{HashStream}$ returns the sequence $H_0 | H_1 | \dots | H_{2^{40}-1}$ where each $H_i = \mathsf{SHA256}(\mathsf{encode}_4(i) | seed)$. Each $H_i$ block is exactly 256 bits.

Hashed Elgamal Public-key Encryption Scheme.
Hashed Elgamal over curve $E$ is a triplet of algorithms $(\mathsf{Kg}, \mathsf{Enc}, \mathsf{Dec})$ as given in Figure 1.

We refer to $(\mathsf{sk}_{\mathsf{en}}, m, ct)$ as a valid hashed Elgamal tuple if feeding it to $\mathsf{Dec}$ raises no errors and returns the plaintext $m$. It is crucial that, if $\mathsf{Dec}$ or $\mathsf{Enc}$ encounters a decoding/encoding error, then they must immediately abort and raise error (return $\perp$). It is a rule of thumb that one should always check whether given inputs belong to proper domains.

In your parameter file, you will find a list of five such tuples given as $Q1_{list}$. Each element of the list is a hashed Elgamal tuple $(\mathsf{sk}_{\mathsf{en}}, m, ct)$ and each component is encoded with $\mathsf{base64}$. In other words, $Q1_{list}$ looks like [($\mathsf{base64}$, $\mathsf{base64}$, $\mathsf{base64}$), ...]. You should implement the decryption algorithm fully (including decoding errors) and decide which tuples are valid. Provide your answers in $Q_{1a}$ as a list of integers, i.e. put 1 for valid tuples and 0 for invalid ones. The sequence is randomly generated, therefore it is possible to have all valid or all invalid tuples.

Then, you will also find a public (encryption) key $\mathsf{pk}_{\mathsf{en}}$ (as $\mathsf{base64}$ encoded $Q_{pk}$ on top of $\mathsf{encode}_P$) and a message $m$ (as $\mathsf{base64}$ encoded $Q_m$). You are expected to implement encryption algorithm and encrypt given message **with the random coins provided**, i.e. $x$ (as integer $Q_x$) value of encryption algorithm is given as well. Provide your answer $ct$ (should be encoded to $\mathsf{base64}$ and placed in $Q1_b$).

One valid, $Q1_{good}$, and an invalid, $Q1_{bad}$, tuple is provided for your convenience. It is a part of your task to locate and retrieve **brainpoolP384r1** parameters from the internet[1].

## Exercise 2 Modes of operation with counter

After the lecture about the mode of operation, the crypto-apprentice realized that any mode of operation might be secure with a counter initial vector. So, he decided to implement a system which keeps changing the mode of operations for each message. Since CBC mode cannot take incomplete block as input, the crypto-apprentice decided to accept only complete blocks for any mode. That means that the message must be a multiple of the block size. The exact implementation can be found in Figure 2.

[1] After spending some time with the implementation, ask yourself the following question. Suppose your friend shared his/her public key with you via an out-of-band authenticated channel. If you use this Hashed Elgamal hybrid encryption scheme for messaging, which of the fundamental communication goals (from confidentiality, authentication, integrity) do you obtain? Can an adversary controlling the network mess with the integrity of the messages? Is it IND-CCA secure?

**Algorithm** AES-CTR$(N, K, M)$
  $M_0, M_1, \ldots, M_{m-1} \xleftarrow{128} M$
  **for** $i = 0$ **to** $m - 1$ **do**
    $V \leftarrow \mathsf{BigEnd}(N + i) \| \mathsf{LitEnd}(N + i)$
    $T_i \leftarrow \mathsf{AES}(K, V)$
    $C_i \leftarrow M_i \oplus T_i$
  **end for**
  **return** $C_0 \| C_1 \| \ldots \| C_{m-1}$
**end Algorithm**

**Algorithm** AES-CBC$(N, K, M)$
  $M_0, M_1, \ldots, M_{m-1} \xleftarrow{128} M$
  $C_{-1} \leftarrow \mathsf{BigEnd}(N) \| \mathsf{LitEnd}(N)$
  **for** $i = 0$ **to** $m - 1$ **do**
    $C_i \leftarrow \mathsf{AES}(K, C_{i-1} \oplus M_i)$
  **end for**
  **return** $C_0 \| C_1 \| \ldots \| C_{m-1}$
**end Algorithm**

**Algorithm** AES-CFB$(N, K, M)$
  $M_0, M_1, \ldots, M_{m-1} \xleftarrow{128} M$
  $C_{-1} \leftarrow \mathsf{BigEnd}(N) \| \mathsf{LitEnd}(N)$
  **for** $i = 0$ **to** $m - 1$ **do**
    $C_i \leftarrow \mathsf{AES}(K, C_{i-1}) \oplus M_i$
  **end for**
  **return** $C_0 \| C_1 \| \ldots \| C_{m-1}$
**end Algorithm**

**Algorithm** Enc$(N, K, M)$
  **if** $N \equiv 0 \pmod 4$ **then**
    $C \leftarrow$ AES-CBC$(N, K, M)$
    $N' \leftarrow N + 1$
  **else if** $N \equiv 1 \pmod 4$ **then**
    $C \leftarrow$ AES-CFB$(N, K, M)$
    $N' \leftarrow N + 1$
  **else if** $N \equiv 2 \pmod 4$ **then**
    $C \leftarrow$ AES-CTR$(N, K, M)$
    $N' \leftarrow N + \lfloor 2^{(|M|/128)+1} \rfloor + 1$
  **else if** $N \equiv 3 \pmod 4$ **then**
    $C \leftarrow$ AES-OFB$(N, K, M)$
    $N' \leftarrow N + 1$
  **end if**
  **return** $N', C$
**end Algorithm**

**Algorithm** AES-OFB$(N, K, M)$
  $M_0, M_1, \ldots, M_{m-1} \xleftarrow{128} M$
  $T \leftarrow \mathsf{AES}(K, \mathsf{BigEnd}(N) \| \mathsf{LitEnd}(N))$
  **for** $i = 0$ **to** $m - 1$ **do**
    $C_i \leftarrow M_i \oplus T$
    $T \leftarrow \mathsf{AES}(K, T)$
  **end for**
  **return** $C_0 \| C_1 \| \ldots \| C_{m-1}$
**end Algorithm**

Figure 2: AES-CTR, AES-CBC, AES-CFB (left), Enc and AES-OFB (right). $|M|$ denotes the bit length of the message $M$, $M_0, M_1, \ldots, M_{m-1} \xleftarrow{128} M$ denotes the split of a message $M$ into $m$ blocks of 128 bits. The function $\mathsf{BigEnd}(n)$ denotes the 8 byte representation of the 64 bit integer $n$ in big-endian (e.g. $\mathsf{BigEnd}(81985529216486895) =$ `0x0123456789ABCDEF`), and the function $\mathsf{LitEnd}(n)$ denotes the 8 byte representation of the 64 bit integer $n$ in little-endian (e.g. $\mathsf{LitEnd}(81985529216486895) =$ `0xEFCDAB8967452301`). Note that $N$ is a 64 bit integer and $\mathsf{AES}(K, X)$ means an encryption of a message $X$ by using AES and the key $K$.

In order to not manually store the secret key $K$ and the counter $N$, the apprentice tried to implement it as an encryption server. It permanently stores the secret key $K$ and the initial vector $N$, and when one sends a message $M$ to the encryption server, the encryption server takes $K$ and $N$ from its storage and calls Enc$(N, K, M)$. Then, the server returns the ciphertext $C$ and store new counter $N'$ in its storage.

When he finished the implementation, the apprentice did the manual exchanged of the secret key $K_2$ and the initial value of the counter $N$ with a friend. In order to check if their key is same, the apprentice encrypted a message $M_2$ with the AES by using the same key $K_2$ and a random initial vector $\mathrm{IV}_2$ in CFB mode, and sent $\mathrm{IV}_2$ and $C_2$, which is the encryption of $M_2$, to his friend after encoding them in `base64`. By observing the traffic of the apprentice, you got $\mathrm{IV}_2$ and $C_2$. Moreover, you found that you can access to the apprentice's encryption server. By querying your SCIPER and a plaintext encoded in `base64` with the following `\n` to the server `lasecpc25.epfl.ch` on port 6666 using Sage or ncat, you can get the encryption of

this plaintext, also encoded in base64. The query should have the following format: SCIPER followed by the base64 encoding of the plaintext you want to encrypt. For instance

```
123456 UmVkIEZveCEAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=\n
```

will return the encryption of the ASCII string "Red Fox!" with following 24 zero bytes.

In your parameter file, you can find the initial vector $IV_2$, and the ciphertext $C_2$ encoded in base64. Recover $M_2$ and write it under $Q_2$ in your answer file as a ASCII string. (This means that you have to provide a **"meaningful" English phrase in ASCII!**)

## Exercise 3 Return of Vernam cipher

During the lecture about elliptic curves, the apprentice realized that it is hard to know the cardinality of an elliptic curve and consequently it is hard to know the order of a point on the elliptic curve. From there, the apprentice got a brilliant idea of the key generation of the Vernam cipher. His idea was to use a random point $P = (P_x, P_y)$ of an elliptic curve $E$ as a shared secret between the sender and the receiver and a seed of the key sequence. Let $K = K_0K_1\cdots$ be a key sequence. Then, $K_i = \mathsf{x}((i+k)P) \bmod 2$ ($K_i = 1$ if $(i+k)P$ is the point at infinity $\mathcal{O}$) where $\mathsf{x}(P) = P_x$ for some fixed $k$. Since a point addition on the elliptic curve requires both $P_x$ and $P_y$, it is hard to guess next bit without knowing $P$. Moreover, the apprentice was also worrying the distinguishability between ciphertexts. Since the Vernam cipher preserves the length of the plaintext, it is easy to distinguish two ciphertexts if there length are different. Therefore, the apprentice decided to select the byte size of the ciphertext, and append some zero bytes as paddings to the message. Since the apprentice could find appropriate encoding to send an elliptic curve, he decided to always use the elliptic curve $E_3 = \{\mathcal{O}\} \cup \{(x,y) \in K_3^2 \mid y^2 = x^3 + 1\}$ where $K_3 = GF(p_3)$. With a message $M$, a random point $P$ and a ciphertext byte length $\ell$, the encryption works as on Figure 3.

**Algorithm** $\textsc{Enc}(p_3, M, P, \ell)$
  $k \leftarrow$ random integer in $[1, p_3]$
  $r_1 \leftarrow$ random integer in $[1, \ell - |M|/8]$
  $M' \leftarrow \texttt{0x00}^{r_1} \| M \| \texttt{0x00}^{\ell - |M|/8 - r_1}$
  **for** $i = 0$ **to** $\ell - 1$ **do**
    $K_i \leftarrow 0$
    **for** $j = 0$ **to** $7$ **do**
      $K_i \leftarrow K_i + 2^{7-j} \cdot (\mathsf{x}((8i + j + k)P) \bmod 2)$
    **end for**
    $C_i \leftarrow K_i \oplus M'_i$
  **end for**
  **return** $k, C_0 \| C_1 \| \cdots \| C_{\ell-1}$
**end Algorithm**

Figure 3: Exact implementation of the encryption. $|M|$ denotes the bit length of the message $M$, and $\texttt{0x00}^n$ denotes the concatenation of $n$ 0x00 bytes.

However, the apprentice did not know that the $j$-invariant of $E_3$ is 0, and we can easily compute the cardinality of the curve when its $j$-invariant is 0 or 1728. By observing the traffic of the apprentice, you got a ciphertext $(k_3, C_3)$ which was encrypted by using a random point $P_3$.

In your parameter file, you will find the prime number $p_3$, the cardinality of $E_3$ $n_3$, and the ciphertext $k_3$, $C_3$ where $k_3$ is an integer and $C_3$ is encoded in base64. Decrypt $C_3$ and write it under $Q_3$ in your answer file as a ASCII string. (This means that you have to provide a **"meaningful" English phrase in ASCII!**)
**Hint:** Depending on the choice of $k$, $r_1$ and $\ell$, it becomes impossible to recover the original message. But your ciphertext is one of possible cases. Do not forget to drop leading and following `0x00` bytes.

## Exercise 4 Bytecipher

To invent a *lightweight* symmetric variable-length encryption primitive, the apprentice thought it would be a *tour de force* to tie great concepts together: a blockcipher, a mode of operation and key-scheduling over IV. His design *bytecipher* consisted of mini-blockciphers (call them S-boxes if you will) operating over bytes, and a novel IV-scheduling based mode of operation. His design contains the following internal elements:

- The key of *bytecipher* consists of two random permutations over bytes, denoted by $\mathsf{Perm_K}, \mathsf{Perm_{IV}} : \{0,1\}^8 \to \{0,1\}^8$. Given that there are $(2^8!)^2$ different keys which approximates to 256-bit security, the apprentice found it more than enough in terms of security.
- Initialization vector $\mathsf{IV}_0$ is a single byte. Being the second component of the key, permutation $\mathsf{Perm_{IV}}$ is used for IV-scheduling. Essentially $\mathsf{IV}_{i+1} = \mathsf{Perm_{IV}}(\mathsf{IV}_i)$ for $i = \{0, 1, \dots\}$.
- Plaintext size must be some multiple of bytes. Before encryption, plaintext $m$ of $k$ bytes is chopped up into multiple byte blocks: $m = m_0|m_1|\dots|m_{k-1}$. Then the ciphertext is $c = c_0|c_1|\dots|c_{k-1}$ where $c_i = \mathsf{Perm_K}(\mathsf{IV}_i \oplus m_i)$ for $i \in \{0, 1, \dots, k-1\}$.
- ASCII conversion applies for English sentences. Namely, each character is replaced with its ASCII corresponding byte. The leftmost character is addressed with index 0.
- The final ciphertext of $m$ is a tuple $ct = (\mathsf{IV}_0, c)$.

With an utmost confidence in his design, he picked a random key, that is a tuple of permutations $(\mathsf{Perm_K}, \mathsf{Perm_{IV}})$ over bytes, and allowed you to query *plaintext-checking oracle* $\mathcal{O}_{\mathsf{PCO}}$[2]. $\mathcal{O}_{\mathsf{PCO}}$ takes a candidate plaintext $m$ and a ciphertext $ct = (\mathsf{IV}_0, c)$ and decides whether $ct$ decrypts to $m$ with $(\mathsf{Perm_K}, \mathsf{Perm_{IV}})$ as the symmetric key. $\mathcal{O}_{\mathsf{PCO}}(m, ct)$ returns 1 if plaintext matches the ciphertext, and otherwise 0 (and of course some informative messages when your queries are not properly formatted).

He was not so generous about his choice of hardware for $\mathcal{O}_{\mathsf{PCO}}$, as it was designed for some low-cost device and all his symmetric operations were meant to be *lightweight* anyways. Therefore, the time cost of each oracle query is dominated by the connection establishment rather than symmetric operations. You will be compensated by the fact that you can make bulk queries, i.e. a list of such pairs are all answered at the same time upto 100 pairs. (say on the order of $5 \cdot 10^4$). You can also use the fact that messages are actually multiples of blocks.

[2]Is it realistic assume to we have $\mathcal{O}_{\mathsf{PCO}}$? Very much so. For instance, if such an encryption scheme was deployed to grant access to —nowadays ubiquitous— an Internet-of-Things device facing the internet, anyone could make well-formatted messages the device is expecting to see; and then make a guess for the ciphertext. The device would behave as though $\mathcal{O}_{\mathsf{PCO}}$ oracle: an adversary submitting such ciphertext would see failure messages for his bad guesses and a success message only when the guess is correct.

Your goal is to embarrass him by forging a ciphertext $Q4_{ct} = (Q4_{\mathsf{IV}}, Q4_c)$ such that $Q4_{ct}$ is the encryption of $Q4_m$ with $\mathsf{IV}_0 = Q4_{\mathsf{IV}}$. You will find $Q4_m$ and $Q_{\mathsf{IV}}$ in your parameter file, so you should not use arbitrary $\mathsf{IV}_0$ value. Provide your answer as $Q4_c$ in base64 encoded format, since it will not be ASCII printable. Here are the types of elements for your convenience:

- The oracle is located at `lasecpc25.epfl.ch` and port `5559`. $\mathcal{O}_{\mathsf{PCO}}$ takes a string query of the form with spaces in between:

  `SCIPER INT(N) BASE64(M) INT(IV) BASE64(C) BASE64(M) INT(IV) BASE64(C) ...`

  where `SCIPER` is your SCIPER number, `INT(N)` is as integer for the total number of tuples contained in this query. It is followed by sequence of `3N` space-separated elements (or `N` tuples/triplets), where each triplet consists of `BASE64(M)` as base64 encoded message $m$, `INT(IV)` as integer between 0 and 255 for $\mathsf{IV}_0$; `BASE64(C)` the second component of the ciphertext. Please do take a look at the **SAGE code provided on the moodle**.

- $Q4_{\mathsf{IV}}$ is an integer value between 0 and 255.

- $Q4_m$ is a string value (without base64 encoding).

- $Q_c$ should be a base64 encoded string.

**Hint:** You can test your final answer with the oracle.

## Exercise 5 Breaking the Feistel Network over Small Domain

After he gave it a try with stream ciphers and modes of operation, our crypto apprentice came across a block cipher that consists of only three rounds of Feistel Network (FN). To understand its design, he needed some more details about FN-based block ciphers. Analyzing the slides of the crypto course, he saw that DES is based on 16-round FN. He also learned that DES is broken in the case that one is able to access many messages encrypted with the same key. Later on, he thought that if one can break the 16-round DES, he can possibly break the 3-round FN with a way easier method! He decided to dive in the details.

While digging the details of the course notes, he noticed that the new design of Feistel network is different than the lecture notes. For instance, unlike standard Feistel schemes which use the exclusive or (XOR) (denoted by $\oplus$), this new cipher uses the modular addition that is denoted by $\boxplus$.

More specifically, for a byte domain of size $N = 2^{24}$ of a FN, this 3-round FN is defined with three distinct round functions $F_0, F_1, F_2$ and modular addition. Given $x$ and $y$ in $N$, the new cipher defines:

$$\begin{aligned} c &= x + F_0(y) \mod N, \\ t &= y + F_1(c) \mod N, \\ z &= c + F_2(t) \mod N. \end{aligned} \tag{1}$$

The picture of the 3-round FN is given on Fig. 4. To be more precise about the description of the round functions, let us define the following functions:
$\boldsymbol{STR^b}$ **:** a function that maps an integer $x$ where $0 \leq x < 2^b$ to a bitstring of length $b$ with most significant character first, e.g. $STR^4(5) = 0101$.
$\boldsymbol{NUM_2}$ **:** a function that maps a string $X$ of length $b$ to an integer $x$ such that $STR^b(x) = X$. For instance, $NUM_2(010010) = 18$

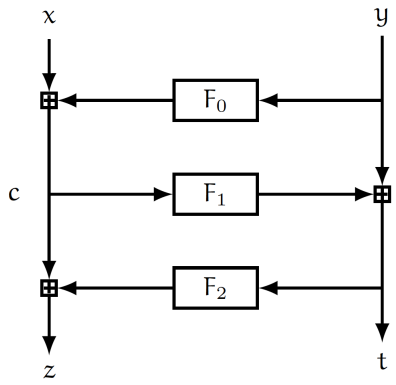


Figure 4: 3-round Feistel Network

Then, we define the $i^{th}$ round function as

$$F_i(x) = NUM_2(AES_K(STR^8(i) \| S \| STR^{100}(x)))$$

where $i$ is a round index (represented as 8-bit), $S$ is a 20-bit sciper number, AES is a block cipher mapping a 128-bit bitstring to a 128-bit bitstring with a secret key $K$.

Having all the details of the 3-round FN design, in your parameter file, you will find a set of known plaintext/ciphertext pairs $S_5$ which are generated with a secret key $K$ and round functions $F_0, F_1, F_2$, and a ciphertext $C_5$. Your task is to decrypt the given ciphertext $C_5$ in your parameter file. Write your answer under $Q_5$ after encoding in base64.
**Hint:** You can set $F_0(a) = 0$ for a fixed $a$.