Homework 5 - Cryptographic Mashup

Cryptography and Security 2020

- 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 all `Q` values specified in the questions below. You can download your **personal** files from the following link:
`http://lasec.epfl.ch:80/courses/cs20/hw5/index.php`
- You will find an example parameter and answer file on the moodle. You can use this parameters' file to test your code and also ensure that the types of `Q` values you provided match what is expected. **Please do not put any comment or strange character or any new line** in the .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. Notebook files are allowed, but we prefer if you export your code as a text file with a sage/python script.
- 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/corrections 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.
- The homework is due on Moodle on **Friday the 18th of December** at 23:59.

Exercise 1 You'll Never (Random) Walk Alone

Your Local Fun Club (LFC) competes in the national tournament of rock-paper-scissors, which they haven't won in 30 years. This season, because of the pandemic, the games cannot take place in person and need to be played remotely. As a result, the league hired the Crypto Apprentice to design a secure system based on commitment schemes. Then, in a match each team would choose its moves for several rounds, convert them into an integer x and commit to this value. The Crypto Apprentice heard about Pedersen Commitment, which works in a subgroup of $\mathbb{Z}_p^*$ generated by g, where g has prime order $q = \frac{p-1}{2}$. You can find the corresponding `Q1_p`, `Q1_g`, `Q1_q` in your parameters file.

Question a)

The Crypto Apprentice first considers the original commitment scheme which works as follows.

- Setup: Generate p, q, g as defined above. Pick a random $\tau \in \mathbb{Z}_q^*$ and compute $h \leftarrow g^\tau \bmod p$. The public parameters are (p, g, q, h).
- $\mathsf{Commit}(x; r)$: Output $g^x h^r \bmod p$, where r is picked at random from $\mathbb{Z}_q^*$.
- $\mathsf{Open}(c; (x', r'))$: Output x' iff $c = g^{x'} h^{r'} \bmod p$ otherwise output an error $\perp$.

Show that given a **valid** commitment/opening tuple (`Q1a_c`, (`Q1a_x`, `Q1a_r`)), one can recover the value h. Put it under `Q1a_h` in your answer file.

Question b)

Based on this observation, the Crypto Apprentice removes h from the public parameters and decides that it can be computed during the commitment phase (i.e. as before by picking a random value τ and computing $h \leftarrow g^\tau \bmod p$).

The coach of LFC's greatest rivals, Paper Granola, heard about this modification and decides to cheat by committing to a value x_0 but providing a valid opening for a different value x_1. Show how he can do it by forging an opening value r_{forge} s.t. (x_1, r_{forge}) is a valid opening of the commitment $c = \mathsf{Commit}(x_0; r_0)$. You can find the parameters `Q1b_tau`, `Q1b_x_0`, `Q1b_x_1`, `Q1b_r`, `Q1b_c` in your parameters file. Report the forged opening r_{forge} under `Q1b_r_forge` in the answer file.

Question c)

Given the failure of the previous construction, LFC captain Penderson proposes another variant of the commitment scheme, based on a hash function. The scheme works as follows.

- Setup: As in Question a). Generate p, q, g as defined above. Pick a random $\tau \in \mathbb{Z}_q^*$ and compute $h \leftarrow g^\tau \bmod p$. The public parameters are (p, g, q, h).
- $\mathsf{Commit}(x; r)$: Output $g^x h^{H(x)} g^r \bmod p$, where r is picked at random from $\mathbb{Z}_q^*$ and $H(x)$ is defined as $(\mathsf{SHA256}(k \| x))_{40}$, where k is some parameter and $(\cdot)_{40}$ denotes the 40 leftmost bits. More details on H are given below.
- $\mathsf{Open}(c; (x', r'))$: Output x' iff $c = g^{x'} h^{H(x')} g^{r'} \bmod p$ otherwise output an error $\perp$.

It is the final, substitute Dirock (famous for his scissors moves) is on, your favourite team must not loose because of a bad commitment scheme!
Show that Penderson Commitment is not binding. More precisely, given a commitment $c = \mathsf{Commit}(x_0; r_0)$, find $(x_1; r_{forge})$ s.t. $\mathsf{Open}(c; (x_1, r_{forge}))$ succeeds. In your parameters file, you will find `Q1c_h`, `Q1c_x_0`, `Q1c_r_0`, `Q1c_c` and the parameter k of the hash function in `Q1c_hash_key`. In your answer file, you must report x_1 and r_{forge} under `Q1c_x_1` and `Q1c_r_forge`, respectively.

Hint 1: You should find a collision on $H(\cdot)$ involving x_0. **Use Floyd algorithm** (slide 699) with the initial value `Q1c_x_start` provided in the parameters file. Do not bruteforce.

Hint 2: Here is a code snippet showing how to compute $H(x)$ in Sage. Note that k is the hash key, x is an integer and $h = H(x)$ returns an integer as well.

```
>>> import hashlib
>>> x = 1062019
>>> k = 'sgtpepper'
>>> h_bytes = hashlib.sha256((k+str(x)).encode()).digest()[:5]
>>> h = int.from_bytes(h_bytes, "big")
```

Exercise 2 Tongue twister

Throughout this exercise, we denote by $\mathtt{LSB}(x, \omega) \triangleq x \mathbin{\&} (1 \ll \omega) - 1)$ *the lowest* ω *bits of* $x \in \{0,1\}^*$ *and by* $\overline{x}$ *the bitwise negation of* $x \in \{0,1\}^*$*. The addition and multiplication operators in* $\mathbf{Z}$ *are denoted by* $+$ *and* $\cdot$ *respectively.*

Given an elliptic curve E/κ defined over a finite field $\kappa = \mathbb{F}_q$, a well-known result states that $E(\kappa)$ is either cyclic or is a product of two cyclic groups. As such, we say that $G \in E(\kappa)$ is a *pseudo-cyclic generator* if it generates a cyclic component. Let $[n]\colon E(\kappa) \longrightarrow E(\kappa)$ be the multiplication-by-n map in $E(\kappa)$ and consider the following algorithm:

Algorithm 1: `challenge`

Input: A security parameter $\lambda \geq 3$ and a bound B.
Output: A challenge tuple (p, E, ν, P, Q).

$p \leftarrow \mathsf{random\_prime}(\lambda)$ ▷ `an` λ`-bit prime`
repeat
  pick a random elliptic curve E over $\mathbb{F}_p$
  pick a random pseudo-cyclic generator G of E
  $\nu \leftarrow \mathsf{ord}(G)$
  $\nu \rightarrow f_1^{s_1} \cdots f_r^{s_r}$ ▷ `prime factorization of` ν
  $\varepsilon \leftarrow \prod_{f_i < B} f_i^{s_i}$
until $\varepsilon \geq B$
$e_{max} \leftarrow \min(B, \nu - 1)$
$e_{min} \leftarrow \lfloor \frac{e_{max}}{2} \rfloor$
$e \leftarrow_{\$} \{e_{min}, e_{min} + 1, \ldots, e_{max} - 1\}$
$P \leftarrow G$
$Q \leftarrow [e]P$
return (p, E, ν, P, Q)

The Mersenne Twister is a pseudo-random number generator (PRNG) based on a *twisted generalised feedback shift register of rational normal form.* The algorithm is parametrized[1] by public algorithmic constants $\mathfrak{C} = (w, n, m, r, a, u, d, s, b, t, c, \ell, f)$. In practice, those constants are chosen so that they satisfy some conditions, e.g. the widely-used mt19937 asks for $2^{wn-r} - 1$ to equal the Mersenne prime $2^{19937} - 1$. Although the parameters for this exercise do *not* necessarily satisfy this primality condition, we show that it is possible to predict future outputs based on sufficiently many previous ones.

- ▷ Given a challenge tuple (p, E, ν, P, Q) generated by Algorithm 1 with inputs $\lambda = 160$ and $B = 2^{32}$ as (Q2a_p, Q2a_a, Q2a_b, Q2a_n, Q2a_P, Q2a_Q), where E is an elliptic curve defined over $\mathbb{F}_p$ by the Weierstrass equation $E : y^2 = x^3 + ax + b$, recover the discrete logarithm e of Q in base P and report it under Q2a_e.

  Given algorithmic constants $\mathfrak{C}$ as a *dictionary* Q2a_C and an integer R as Q2a_R, report the output of Algorithm 2 with inputs $(\mathfrak{C}, \sigma = e, R)$ under Q2a_y.

  Hint: Observe that e satisfies some bound conditions on the factors of ν. Using Pohlig-Hellman to solve the DLP directly may take while, so think of the *Sunzi Suanjing*.

- ▷ Given algorithmic constants $\mathfrak{C} = (w, n, m, r, a, u, d, s, b, t, c, \ell, f)$ as a *dictionary* Q2b_C and a list Q2b_out of the $N \geq n$ last outputs $(y_1, \ldots, y_N)$ of the Mersenne Twister parametrized by $\mathfrak{C}$, predict the next output y_{N+1} and report it under Q2b_y.

  Hint: Study the nature of each operation when $N = n = 1$.

Algorithm 2: random

Input: Some algorithmic constants $\mathfrak{C}$, an integral seed $\sigma > 0$ and an integer $R > 0$.
Output: The R-th random generated value y of the PRNG seeded with σ.

$\mathfrak{C} \to (w, n, m, r, a, u, d, s, b, t, c, \ell, f)$
$\Sigma \leftarrow \mathsf{seed}(\sigma, w, n, f)$ ▷ see Algorithm 3
$y \leftarrow \perp$
for $i = 1, \ldots, R$ **do**
  $\mathsf{next}(\Sigma, w, n, m, r, a, u, d, s, b, t, c, \ell) \to (\Sigma', y')$ ▷ see Algorithm 4
  $\Sigma \leftarrow \Sigma'$
  $y \leftarrow y'$
return y

Algorithm 3: seed

Input: A seed σ and constants w, n and f.
Output: A state $\Sigma = (\iota, \sigma_0, \ldots, \sigma_{n-1})$.

$\sigma_0 \leftarrow \sigma$
for $i = 1, \ldots, n-1$ **do**
  $z \leftarrow f \cdot (\sigma_{i-1} \oplus (\sigma_{i-1} \gg (w-2))) + i$
  $\sigma_i \leftarrow \mathsf{LSB}(z, w)$
$\iota \leftarrow n$
return $(\iota, \sigma_0, \ldots, \sigma_{n-1})$

[1]See the Wikipedia page for the significance of those constants, although irrelevant for this exercise.

Algorithm 4: next

Input: A state $\Sigma = (\iota, \sigma_0, \ldots, \sigma_{n-1})$ and constants $w, n, m, r, a, u, d, s, b, t, c$ and ℓ.
Output: An updated state $\Sigma' = (\iota', \sigma'_0, \ldots, \sigma'_{n-1})$ and a random integer y.

$\Sigma \rightarrow (\iota, \ldots)$
if $\iota \geq n$ **then**
  if $\iota > n$ **then**
    error "The generator was never seeded"
  $\Sigma' \leftarrow \mathsf{twist}(\Sigma, w, n, m, r, a)$ ▷ `see Algorithm 5`
  $\Sigma \leftarrow \Sigma'$ ▷ `update the state`
$\Sigma \rightarrow (\iota, \sigma_0, \ldots, \sigma_{n-1})$ ▷ `reparse the state if a twist occurred`
$y' \leftarrow \sigma_\iota$
$y' \leftarrow y' \oplus ((y' \gg u) \;\&\; d)$
$y' \leftarrow y' \oplus ((y' \ll s) \;\&\; b)$
$y' \leftarrow y' \oplus ((y' \ll t) \;\&\; c)$
$y' \leftarrow y' \oplus (y' \gg l)$
$\iota' \leftarrow \iota + 1$
$\Sigma' \leftarrow (\iota', \sigma_0, \ldots, \sigma_{n-1})$
$y \leftarrow \mathsf{LSB}(y', w)$
return (Σ', y)

Algorithm 5: twist

Input: A state $\Sigma = (\iota, \sigma_0, \ldots, \sigma_{n-1})$ and constants w, n, m, r and a.
Output: An updated state $\Sigma' = (\iota', \sigma'_0, \ldots, \sigma'_{n-1})$.

$\mu_L \leftarrow (1 \ll r) - 1$
$\mu_U \leftarrow \mathsf{LSB}(\overline{\mu_L}, w)$
for $i = 0, \ldots, n-1$ **do**
  $x \leftarrow (\sigma_i \;\&\; \mu_U) + (\sigma_{(i+1) \bmod n} \;\&\; \mu_L)$
  $z \leftarrow x \gg 1$
  if $x \equiv 1 \pmod 2$ **then**
    $z \leftarrow z \oplus a$
  $\sigma_i \leftarrow \sigma_{(i+m) \bmod n} \oplus z$
$\iota' \leftarrow 0$
$\Sigma' \leftarrow (\iota', \sigma_0, \ldots, \sigma_{n-1})$
return Σ'

Exercise 3 Cryptographer's Bizarre Adventures

Let $d \geq 1$ and $c = (c_1, \ldots, c_d) \in \mathbb{F}_q^d$. The feedback polynomial $\phi_c \in \mathbb{F}_q[x]$ associated with c is defined by $\phi_c(x) \triangleq 1 + c_1 x + \ldots + c_d x^d$ and the linear feedback shift register (LFSR) associated with ϕ_c is the map $\Phi_c \colon \mathbb{F}_q^d \longrightarrow \mathbb{F}_q^d$ defined by $\Phi_c(x) \triangleq {}^t(x\boldsymbol{C}_{\phi_c})$, where $\boldsymbol{C}_{\phi_c}$ is the companion matrix of ϕ_c. The associated LFSR sequence $L_{\phi_c}(\tau)$ with initial state $\tau = (x_0, \ldots, x_{d-1}) \in \mathbb{F}_q^d$ is defined to be the sequence $(x_n)_{n \geq 1}$, where x_n is the first coefficient of ${}^t(\tau \boldsymbol{C}_{\phi_c}^n) \in \mathbb{F}_q^d$. For instance, the feedback polynomial of $c = (1, 1, 0, 1)$ is $\phi_c(x) = 1 + x + x^2 + x^4$ and the first terms of $L_{\phi_c}(\tau)$ for $\tau = (0, 1, 1, 0)$ are $(1, 1, 0, 1, 0, 0, 0, 1, 1, \ldots)$.

Consider the composition of two symmetric ciphers as a single symmetric cipher. Any legitimate receiver would *a priori* need to know the secret keys of each cipher to decrypt the ciphertexts. The purpose of this exercise is to show that this is not necessarily the case. Informally, the key-generation and encryption algorithms are described as follows:

1. (*keygen*) Generate a random LFSR feedback polynomial $\phi \in \mathbb{F}_2[x]$ of degree 16, a 16-bit state τ and a 128-bit key K. The pair (ϕ, τ) is called an LFSR configuration and the secret key is defined to be the triplet (ϕ, τ, K).

2. (*encrypt*) The encryption algorithm is described by Algorithm 6. At a high-level point of view, the encryption algorithm performs the following steps: first encrypt the plaintext using Algorithm 7 and the secret key (ϕ, τ) to get an intermediate ciphertext z. Inject some additional data around z and encrypt the whole with Algorithm 8 (which is similar to Telegram's Infinite Garble Extension mode) and K to get the final ciphertext ct.

After having solved the challenges from Mr. Copper and Ms. Smith, our traveller continues their journey through the wilderness and finds another village where they can rest for the night. The inn's manager is a very careful person and loves symmetry. As such, every personal client messages (e.g. the bill) is encrypted using the cipher described above (observe that it would have been much more reasonable to use a PKC instead, but this manager is stubborn).

The traveller books a chamber for the night and gets some shared keys (ϕ, τ, K). Before going to sleep, they decide to spend some time outside, drinking the best booze (don't forget to always drink with moderation) while gazing at the stars. The day after, they noticed a message left by the manager on their table. Unfortunately, the ink has faded away (this is not a good ink) and the ciphertext is corrupted. Since misfortunes never come alone, the traveller discovered that ϕ and τ have disappeared (Heavens' mysteries are really profound) along with a portion of the key K. Would you be able to help our traveller?

- ▷ Given a corrupted key K_{corr} as Q3a_k and a corrupted ciphertext ct_{corr} as Q3a_y, both written in *hexadecimal* with the corruption character being ?, recover the key K and the hexadecimal intermediate ciphertext z. Report the uncorrupted *hexadecimal* key K under Q3a_K and the hexadecimal intermediate ciphertext z under Q3a_z.

  Hint: Observe that the intermediate ciphertext z is split into chunks of 32 bytes, each of which being used as an IV for Algorithm 8. Write down the circuit and use the properties of ECB mode. Furthermore observe that z can be fully recovered as soon as the key K is known, even if the ciphertext is not known.

- ▷ Using the intermediate ciphertext z recovered in the previous question, recover the original plaintext pt and report it under Q3b_pt as a string (see known answer tests for the correct format).

  Hint: Observe that a known header has been added before encrypting. Use this known header to recover the LFSR feedback polynomial and the first bits of the LFSR sequence.

We furthermore define the following routine:

1. hex(x): create a string of hexadecimal numbers from a bytestring x. The inverse operation is denoted by hex^{-1}. Note that $\mathsf{len}(\mathsf{hex}(x)) = 2 \cdot \mathsf{len}(x)$. In Python 3.x, these operations are achieved via `x.hex()` and `bytes.fromhex(x)` respectively.

2. encode(x): encode a string x into a bytestring. The length fo output is expected to be the same as the length of the input. Achieved in Python 3.x via `x.encode()`.

3. PKCS7(x, ℓ): apply a standard PKCS7 padding to a *byte string* x so that the output length is guaranteed to be a multiple of ℓ. The pycryptodome package provides such tool via Crypto.Util.Padding.pad as well as implementations of AES and mode of operations, and hence is strongly recommended[2].

Since LFSR were not seen officially during the courses, we provide the following tools. Parameters were technically generated so that you do not encounter some uniqueness problems that may arise, so please contact us if you encounter any issue.

1. The sage.matrix.berlekamp_massey.berlekamp_massey function which recovers the feedback polynomial given a list of LFSR outputs.

2. The sage.crypto.lfsr.lfsr_sequence function creates an LFSR sequence given the *reverse* feedback polynomial and the initial terms of the LFSR sequence.

3. The pylfsr Python package for constructing LFSR sequences (see official documentation).

```
#!/usr/bin/env sage

from pylfsr import LFSR
from sage.matrix.berlekamp_massey import berlekamp_massey

# c = (1, 1, 0, 1), tau = (0, 1, 1, 0)
# fpoly is a list of indices for which c[i-1] = 1
# fpoly = [1, 2, 4] represents the polynomial 1 + x + x^2 + x^4
L = LFSR(fpoly=[1, 2, 4], initstate=[0, 1, 1, 0])

# pick the first N=10 terms (require the cast to int because of SAGE)
S = L.runKCycle(int(10))
# S is actually a numpy array of float64
S = S.astype(int).tolist()
assert S == [1, 1, 0, 1, 0, 0, 0, 1, 1, 0]

# inputs to ‘berlekamp_massey‘ must be in a field
T = list(map(GF(2), S))
# output of Berlekamp-Massey is the reverse feedback polynomial
g = berlekamp_massey(T) # g = 1 + x^2 + x^3 + x^4
f = g.reverse()         # f = 1 + x + x^2 + x^4
```

[2]Run `sage -python3 -m pip install package_name` to install a Python pip package within SAGE.

Algorithm 6: `encrypt`

Input: A secret key $\mathsf{sk} = (\phi, \tau, K)$ and a human-readable message pt.
Output: A ciphertext ct.

$\mathsf{sk} \rightarrow (\phi, \tau, K)$
$m \leftarrow \mathsf{0x4E414E49213F0D0A}||\mathsf{hex}(\mathsf{pt})$
$z \leftarrow \mathsf{encrypt1}(\phi, \tau, m)$ ▷ z is the hexadecimal intermediate ciphertext
$\widehat{z} \leftarrow \mathsf{PKCS7}(z, 32)$
$\widehat{z} \rightarrow \widehat{Z}_0|| \dots ||\widehat{Z}_{N-1}$ ▷ split $\widehat{z}$ into 32-byte blocks
for $i = 0, \dots, N-1$ **do**
  $h_i \leftarrow \mathsf{PKCS7}(\mathsf{0x11}, 128)$ ▷ byte string of length 128
  $x_i \leftarrow \mathsf{hex}(h_i||K)$ ▷ hexadecimal string of length 288
  $X_i \leftarrow \mathsf{encode}(x_i)$ ▷ encoding of x_i via x_i.encode()
  $y_i \leftarrow \mathsf{encrypt2}(K, \widehat{Z}_i, X_i)$
$\mathsf{ct} \leftarrow y_0|| \dots ||y_{N-1}$
return ct

Algorithm 7: `encrypt1`

Input: An LSFR configuration (ϕ, τ) and an hexadecimal string $x = x_0|| \dots ||x_{n-1}$.
Output: An hexadecimal string $y = y_0|| \dots ||y_{n-1}$.

pick the first $4n$ bits $s_0, \dots, s_{4n-1}$ of the LFSR sequence $L_\phi(\tau)$
convert the *binary* string $s_0|| \dots ||s_{4n-1}$ into an integer s ▷ $s_0 = \mathsf{MSB}(s)$
write the *hexadecimal* expansion $\mu = \mu_0|| \dots ||\mu_{n-1}$ of s
$Y \leftarrow \mathsf{hex}^{-1}(x) \oplus \mathsf{hex}^{-1}(\mu)$
$y \leftarrow \mathsf{hex}(Y)$
return y

Algorithm 8: `encrypt2`

Input: An 128-bit key K, a 32-byte IV μ and an n-byte plaintext $x = X_0|| \dots ||X_{n-1}$.
Output: An N-byte ciphertext $y = Y_0|| \dots ||Y_{N-1}$.

$z \leftarrow \mathsf{PKCS7}(x, 16)$
$z \rightarrow Z_0|| \dots ||Z_{N-1}$ ▷ split z into 16-byte blocks
$y \leftarrow \perp$
$\mu \rightarrow \mu_c||\mu_p$ ▷ split μ into two 16-byte blocks
for $i = 0, \dots, N-1$ **do**
  $c \leftarrow Z_i \oplus \mu_c$
  $c \leftarrow \mathsf{AES\text{-}ECB}(K, c)$
  $Y_i \leftarrow \mu_p \oplus c$
  $\mu_c \leftarrow c$
  $\mu_p \leftarrow Z_i$
$y \leftarrow Y_0|| \dots ||Y_{N-1}$
return y