Homework 4 - Symmetric Cryptography, Integrity and Authentication and Public-Key Cryptography

Cryptography and Security 2021

- 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/courses/cs21/hw4/index.php`
- You will find an example parameter and answer file on the moodle. You can use this 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, 31st December** at 23h59.

Exercise 1 I think they're sending us Signals

In class, we looked at Signal (slides 989-993), which is used by instant messaging apps like WhatsApp and the eponymous Signal app by billions of people each day, and the two main protocol components: the key agreement protocol X3DH and the Double Ratchet continuous key exchange protocol (slide 993). In this exercise, we explore different properties of the Double Ratchet portion of the Signal protocol.

We consider two parties, Alice and Bob, that execute the Signal protocol, and in particular the Double Ratchet component of the protocol. For the purposes of this exercise, we will consider the case that only one party sends and the other receives. Let Alice (A) be the sender and Bob (B) the receiver hereafter. We assume functions of the form $c \leftarrow\$\, send(m)$ and $m \leftarrow recv(c)$ for Alice and Bob respectively for sending/receiving; note that $send$ is probabilistic. We assume both parties agree on the following primitives/abstractions:

- A cyclic group G of (prime) order p generated by group element g. We use multiplicative notation for G.
- An AEAD scheme which has two functions, Enc and Dec, with syntax $AEAD.Enc(k, ad, m) \to c$ and $AEAD.Dec(k, ad, c) \to m$, where k, ad, m and c denote the input key, associated data, message and ciphertext, respectively. Note that decryption can fail (intuitively, when authentication fails), in which case the algorithm Dec outputs $\perp$.
- Two KDFs. One is of the form $KDF_{RK}(k, g') \to (k', k'')$ for length λ strings k, k, k'' and group element g'. The other is of the form $KDF_{CK}(k) \to (k', k'')$ for length λ strings k, k, k''.

We recall relevant features of the Double Ratchet protocol. We assume that A's initial state contains the value g^{x_0} and B's initial state contains x_0 for uniformly sampled $x_0 \in \mathbb{Z}_p$. We assume that both A and B also agree on a uniformly random value RK_{init} of length λ. We describe the protocol below.

When A sends her first message (i.e. first invokes $send(m_1)$), she executes as follows:

- A samples $x_1 \leftarrow\$\, \mathbb{Z}_p$, sets $g_{a'} \leftarrow g^{x_1}$ and $DH_{new} \leftarrow (g^{x_0})^{x_1}$.
- A sets $KDF_{RK}(RK_{init}, DH_{new}) \to (RK_{new}, CK_0)$ and $KDF_{CK}(CK_0) \to (MK_0, CK_1)$.
- A sets $c \leftarrow AEAD.Enc(MK_0, (g_{a'}, 1), m_1)$.
- A **deletes** RK_{init}, CK_0 and MK_0.
- A outputs $(c, (g_{a'}, 1))$.

When A sends her i-th message m_i thereafter:

- A sets $KDF_{CK}(CK_{i-1}) \to (MK_{i-1}, CK_i)$.
- A sets $c \leftarrow AEAD.Enc(MK_{i-1}, (g_{a'}, i), m_i)$.
- A deletes CK_{i-1} and MK_{i-1}.
- A outputs $(c, (g_{a'}, i))$.

In the following, for a given point in time, let max, initialised to -1, be the maximum value of i such that B has previously called $recv(c_i)$ where $c_i = (c, (g_{a'}, i))$ which output $m \neq \perp$. Then, when B receives his first message of the form $(c_i, (g_{a'}, i))$:

- B computes $DH'_{new} = g_{a'}^{x_0}$ and computes $KDF_{RK}(RK_{init}, DH'_{new}) \rightarrow (RK_{new}, CK_0)$ as A does.
- B computes $KDF_{CK}(CK_j) \rightarrow (MK_j, CK_{j+1})$ for each $j \in \{0, 1, ..., i-1\}$.
- B sets $m \leftarrow AEAD.Dec(MK_{i-1}, (g_{a'}, i), c_i)$. If $m = \perp$, B rolls back their state and returns $\perp$. Otherwise, B proceeds.
- B deletes RK_{init}, $CK_0, CK_1, ..., CK_{i-1}$, MK_{i-1} and x_0.
- B returns m.

When B receives subsequent messages of the form $(c_i, (g_{a'}, i))$:

- If $i < max$, then B computes $m \leftarrow AEAD.Dec(MK_{i-1}, (g_{a'}, i), c_i)$, rolls back their state given failure, and otherwise deletes MK_{i-1} and returns output m given success.
- Otherwise, if $i > max$ (we trivially ignore $i = max$) then B iteratively computes CK_j and MK_j values, attempts decryption, deletes all values CK_j s.t. $j < i$ and MK_{i-1} given decryption succeeds, and returns the response to Dec m.

With that lengthy but necessary description complete, we move onto the sub-exercises. In your parameters file you are given an array of the form `Q1_ops = [op1, ..., opn]` denoting a (valid) sequence of operations executed by A and B, where each `opi` is either of the form:

- `'send'`, denoting Alice sending some message, or
- `j`, denoting the fact that Bob receives the j-th ciphertext output by Alice (the sequence is always such that this exists and the same j is never included twice).

Your tasks are as follows. Assume that Alice is disallowed from sending more than 70 messages over the lifetime of protocol execution, and recall that Bob never sends.

▷ Consider the sequence of operations `Q1_ops`. In `Q1_a1`, write the indices of messages as a set (e.g. `{1, 3, 5}`) that can be easily derived using the learnt keying material as a result of the **exposure** of Alice's state after all operations are executed.[1] An exposure is an event where the adversary immediately learns the entire state of a given party. In `Q1_b1`, write the set of indices given Bob's state is instead exposed.

▷ Consider `Q1_ops` again. Write in `Q1_a2` (respectively `Q1_b2`) the set of indices of messages that can be trivially (i.e. easily) *forged* as a result of state exposure of Alice (respectively Bob) at the end of the sequence of operations. We emphasise that we are still assuming that only Bob can receive messages.

▷ Suppose now that messages that are sent are additionally *signed* by the sender. In particular, assume that Alice and Bob agree on a signature key pair before the beginning of protocol execution, where only Alice holds the secret key in her state. Then, the value $v = (c, (g', i))$ is additionally signed at the end of $send(\cdot)$; the signature is then sent

[1] We are implicitly assuming here that the underlying primitives are secure under relevant security notions, and that discrete logarithms are infeasible to compute (among other assumptions required for G).

alongside v, and *recv* additionally verifies the signature. With this modified protocol in mind, write in `Q1_a3` (respectively `Q1_b3`) the set of indices of messages that can be trivially *forged* as a result of state exposure of Alice (respectively Bob).

▷ We return to the original protocol and disregard `Q1_ops`. Assume now that Alice performs k sends; your value k is stored in `Q1_k`. Consider a sequence of at most 10 **receive** operations (performed by Bob). In `Q1_B`, write the (non-empty) sequence of operations, i.e. a sequence of 1 to 10 operations, that maximises Bob's state size (i.e. the number of keys that he has stored), as an array using the notation for operations followed in `Q1_ops` (i.e. of the form `[1, 3, 5]`).

Exercise 2 MBCGA

After losing the beauty contest, our favorite protagonist Papy McDonald has decided to find some remedy by making Blockciphers networks great again. He decided to mimic what the AES S-box for his design. The block-size and key size in this block-cipher are equal to eachother and are both 128-bits. The round function of this block cipher looks as shown in the following diagram.

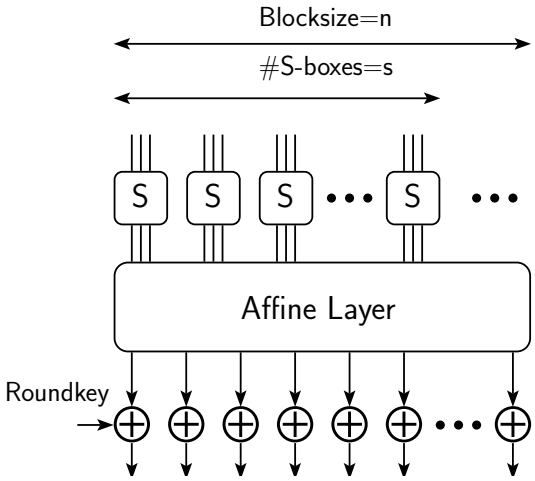

Figure 1: A single round of the block cipher, note that only the $3s$ left most wires past through the S-box and the rest goes directly into the affine layer.

To simplify things he decided to keep s, the number of S-boxes in each round, to be 1. The k^{th} round Affine layer basically multiplies the state vector by the k^{th} Matrix L_k given in the cipher parameters, and the k^{th} layer roundkey is basically the master key multiplied by A^k, where A is the key-update matrix, also given in the parameters. The Sbox $S(x, y, z)$ is simply given by casting $0bxyz$ to $F_8 = \mathbb{Z}_2[x]/\langle x^3 + x + 1\rangle$ and inverting the element in the final field, a.k.a $F(a, b, c) = \text{coefficients}(ax^2 + bx + c)^{-1}$. You are asked to answer the following 3 questions about this cipher.

Enc(m, k, s, r, R, KUM):

1. $st \leftarrow m$
2. For round $\in [r]$
   - $st[0], st[1], st[2] \leftarrow \text{Sbox}(st[0], st[1], st[2])$
   - $st = R[\text{round}] \times st$
   - $st+ = KUM^{\text{round}} \times k$
3. **Return**(st)

Question a)

In your parameters file you are given a plaintext Q2a_pt, a master secret key Q2a_K, the affine matrices Q2a_R and the key update matrix Q2a_A. You are asked to implement McDonald's protocol and find the corresponding ciphertext Q2a_ct as a bit string, e.g. "1101101 . . . 01".

Question b)

In your parameters file you can find a plaintext Q2b_pt1, the corresponding ciphertext Q2b_ct1, encrypted using a 2-round variant of the McDonald cipher. You are also given the round affine matrices Q2a_R and the key update matrix Q2a_A. You are asked to decrypt the given ciphertext Q2b_ct2 and find the corresponding plaintext Q2b_pt2, which is a printable string.

Question c)

The task of this question is exactly the same with the previous one! But this time a 10-round variant of the cipher is used. Decrypt Q2c_ct2 and find the corresponding plaintext Q2c_pt2, which is a printable string.

Hint 1: Try to write the Sbox as a function from $\mathbb{Z}_2^3 \to \mathbb{Z}_2^3$.

Hint 2: What can we say about $S(a, b, c)$ if we know the value of $\mathsf{Maj}(a, b, c)$, $\mathsf{Maj}(*, *, *)$ being the majority function. Can you simplify the S-box given the majority?

Exercise 3 Merry Christmhash

Throughout this exercise, we denote by `xxd`(s) *the hexadecimal representation of a* `bytes` *object. In Python, this is achieved by* `binascii.hexlify` *and the reverse transformation is given by* `binascii.unhexlify`*. Both functions are present in the built-in module* `binascii`*. We denote by* `pack`(k, n) *the function which returns a* `bytes` *of length* $k/8$ *representing an integer* $0 \leq n < 2^k$*. In Python, this is achieved by importing the built-in module* `struct` *and calling* `struct.pack('>L', n)` *if* $k = 32$ *and* `struct.pack('>Q', n)` *if* $k = 64$*:*

```
assert list(struct.pack('>L', 1)) == [0, 0, 0, 1]
assert list(struct.pack('>Q', 1)) == [0, 0, 0, 0, 0, 0, 0, 1]
```

We denote by $\odot$ *and* $\oplus$ *the* AND *and* XOR *operators defined over the integers. In Python, these operations are accessible via* `&` *(e.g.* `a & b`*) and* `operator.xor`[2] *from the* `operator` *built-in module (e.g.* `operator.xor(a, b)`*). The concatenation operator is denoted by* $||$*.*

[2]Using this function avoids conflicts with the SAGE exponentiation operator `^`.

Parameters and answers values are expected to either be Python `int` *or* `str` *instances. We also recall that* `binascii.hexlify` *outputs* `bytes` *objects that can be decoded into strings by calling their* `decode` *method (i.e.* `binascii.hexlify(d).decode()`*).*

Having successfully escaped the ruins, our archaeologist travels back to the Academy of Science to report his findings. After his presentation, one of his peers approaches him to share their concerns with: they recently found a locked chest which can only be opened by recovering some internal secrets of a hash function $\mathcal{H}$ described by Algorithm 1. Unfortunately this time, the archaeologist does not come up with any solution better than exhaustive search, and therefore asks you to help him in this task.

- ▷ Given the parameters (H, K, S) as `Q3a_H`, `Q3a_K` and `Q3a_S` respectively, compute the digest $d = \mathcal{H}(M, H, K, S)$ of the UTF-8 message[3] M given as `Q3a_M` and report $\texttt{xxd}(d)$ in the answers file under `Q3a_d` as a Python `str`.

- ▷ Let $K' = (k_0, \ldots, k_7) \in \mathbf{Z}_{2^{32}}^8$ be a vector of unknowns and $K'' = (k_8, \ldots, k_{63}) \in \mathbf{Z}_{2^{32}}^{56}$ be a vector of round keys given as `Q3b_K2`. Let t be an 8-byte string such that $\texttt{xxd}(t)$ is given as `Q3b_t`. Let $H = (h_0, \ldots, h_7) \in \mathbf{Z}_{2^{32}}^8$ be an initial state given as `Q3b_H` and let $S \in \mathbf{Z}_{2^5}^{2\times 3}$ be a matrix given as `Q3b_S`. Given $\texttt{xxd}(d)$ as `Q3b_d` where $d = \mathcal{H}(t, H, K'||K'', S)$, recover the vector $K' \in \mathbf{Z}_{2^{32}}^8$ and report it in the answers file under `Q3a_K1` as a Python `list` of *positive* integers.

  Hint: Carefully study which variables are known and can be easily recovered. For recovering the unknown keys, study the evolution of the initial state and how the values are propagated. What happens when a round key is 0?

Hint: For the implementation of the algorithm, be careful to work with the correct modulus, i.e. reduce the result of the operations modulo 2^{32} whenever needed. If one of the step is not properly reduced, the final output might be different from what it is expected.

Algorithm 1: $\mathcal{H}(m, H, K, S)$

Input: An **encoded** message m and parameters $H \in \mathbf{Z}_{2^{32}}^8$, $K \in \mathbf{Z}_{2^{32}}^{64}$ and $S \in \mathbf{Z}_{2^5}^{2\times 3}$.
Output: The digest d of m.

$\ell \leftarrow (-|m| - 9) \bmod 64$
$m \leftarrow m||\texttt{0x80}||\texttt{0x00} * \ell||\texttt{pack}(64, 8|m|)$ ▷ `0x00` $* \ell$ `means` ℓ `zero bytes`
$m \rightarrow M_0|| \cdots ||M_{L-1}$ ▷ M_i `is a 64-byte chunk.`
$d \leftarrow H$
for $i = 0, \ldots, L-1$ **do**
  $W \leftarrow \texttt{extend}(M_i)$
  $y \leftarrow \texttt{compress}(d, W, K, S)$
  $d \leftarrow d + y \bmod 2^{32}$
$d \rightarrow (d_0, \ldots, d_7)$
$d \leftarrow \texttt{pack}(32, d_0)|| \ldots ||\texttt{pack}(32, d_7)$
return d

[3]The input to $\mathcal{H}$ is a `bytes` object, and thus `Q3a_M.encode()` should be called beforehand.

Algorithm 2: `extend`(M)

Input: A 64-byte block $M = m_0||\dots||m_{15}$ split into 32-bit chunks $m_i \in \mathbf{Z}_{2^{32}}$.
Output: A vector $W = (w_0, \dots, w_{63}) \in \mathbf{Z}_{2^{32}}^{64}$ of 32-bit integers.

for $i = 0, \dots, 15$ **do**
  $w_i \leftarrow m_i$
for $i = 16, \dots, 63$ **do**
  $t_1 \leftarrow \mathsf{OR}(w_{i-15} \gg 7, w_{i-15} \ll 25) \oplus \mathsf{OR}(w_{i-15} \gg 18, w_{i-15} \ll 14) \oplus (w_{i-15} \gg 3)$
  $t_2 \leftarrow \mathsf{OR}(w_{i-2} \gg 17, w_{i-2} \ll 15) \oplus \mathsf{OR}(w_{i-2} \gg 19, w_{i-2} \ll 13) \oplus (w_{i-2} \gg 10)$
  $w_i \leftarrow w_{i-16} + w_{i-7} + t_1 + t_2$
$W \leftarrow (w_0, \dots, w_{63})$
return W

Algorithm 3: `compress`(x, W, K, S)

Input: $x = (x_0, \dots, x_7) \in \mathbf{Z}_{2^{32}}^{8}$, $W = (w_i)_i$, $K = (k_i)_i \in \mathbf{Z}_{2^{32}}^{64}$ and $S = (S_{ij})_{ij} \in \mathbf{Z}_{2^5}^{2\times 3}$.
Output: $y = (y_0, \dots, y_7) \in \mathbf{Z}_{2^{32}}^{8}$.

$y \leftarrow x$
for $i = 0, \dots, 63$ **do**
  $y \to (y_0, \dots, y_7)$
  $s_0 \leftarrow \bigoplus_{j=1}^{3} \mathsf{OR}(y_0 \gg S_{1j}, y_0 \ll (32 - S_{1j}))$
  $s_1 \leftarrow \bigoplus_{j=1}^{3} \mathsf{OR}(y_4 \gg S_{2j}, y_4 \ll (32 - S_{2j}))$
  $t_1 \leftarrow y_4 \odot y_5 \oplus \mathsf{NOT}(y_4) \odot y_6$
  $t_2 \leftarrow s_1 + t_1 + y_7 + k_i + w_i$
  $t_3 \leftarrow y_0 \odot y_1 \oplus y_0 \odot y_2 \oplus y_1 \odot y_2$
  $y \leftarrow (s_0 + t_2 + t_3, y_0, y_1, y_2, y_3 + t_2, y_4, y_5, y_6)$
return y