Homework 5 - Commitment, Integrity, Authentication and Signatures

Cryptography and Security 2019

- 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: `https://lasec.epfl.ch/courses/cs19/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 highly recommend that you export your code as 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 **Monday the 16th of December** at 22h00.

Exercise 1 A full round trip? Ain't nobody got time for that!

The new TLS 1.3 standard includes 0-RTT (zero round-trip) feature to complete the key exchange faster without requiring a response from the receiver. In essence, one can think of *ephemeral-static* DH key exchange between the two parties, where the sender can immediately derive the session key and start encrypting and transmitting its message.

From the application developer perspective, whether or not to use 0-RTT feature might be a question of user experience instead of security. Imagine that a user wants to connect to a web page. If the browser is not utilizing 0-RTT, then an additional delay is incurred because the browser first needs to receive a response from the server, before it can transmit its GET request[1].

In Figure 2, we describe a simplified 0-RTT protocol between a sender and receiver. The sender, in haste, tries to log in to her account with her credentials followed by a text message, without waiting for the server (receiver) to respond (maybe she aced the crypto midterm, and wants to post it on facebook immediately). As soon as she receives a response, her side of the protocol completes the handshake, and adds the second key into its state St, which is used to continuously derive symmetric key. Because the delay of the round trip varies based on network traffic, the number of message blocks (numbered from 0 to t) that are encrypted in 0-RTT fashion are not predetermined. In practice, there is no guarantee that the packages will arrive in the same order they were sent, so you will observe random ordering between the message blocks/packages in this exercise.

In Figure 2, (p, q, g) defines the multiplicative subgroup $\langle g \rangle \subset \mathbb{Z}_p^*$, whose order is prime q. Let encode_n be a function that truncates a group element $x \in \mathbb{Z}_p^*$ to its least significant n bits, i.e. it returns n-bit string $b_{n-1}||b_{n-2}||\dots||b_1||b_0$ such that $x \bmod 2^n = \sum_{i=0}^{n-1} 2^i b_i$ with $\forall i, b_i \in \{0,1\}$. An initial message M is string of characters, and they are mapped to bit a string according to the ASCII table. Then, M is divided into the sequence $m_0|m_1|\dots|m_{\ell-1}$ where each block m_i is 256-bit, e.g. m_0 contains the first 32 characters.

Key derivation function $\mathsf{KDF}(\mathsf{St})$ takes a single 256-bit symmetric state St as input. It returns the tuple (St', k') by making two calls to the hash function by $k' \leftarrow \mathsf{SHA256}(\mathtt{0x00}||\mathsf{St})$ and $\mathsf{St}' \leftarrow \mathsf{SHA256}(\mathtt{0xFF}||\mathsf{St})$, in given order. Note the change in the `0x00` and `0xFF` constants, which was incorrectly described in the previous version of this file.

In your parameters file, you are given the domain parameters (p, q, g) as well as the public key pk that belongs to the subgroup $\langle g \rangle$ (as `Q1_p`, `Q1_q`, `Q1_g`, `Q1_pk` respectively). In addition, a transcript of the communication between the two parties are also provided where the blocks are intact but **the order of packages are completely randomized** and there is no information on the direction of the flow. For instance, a transcript might look like the following bitstring:

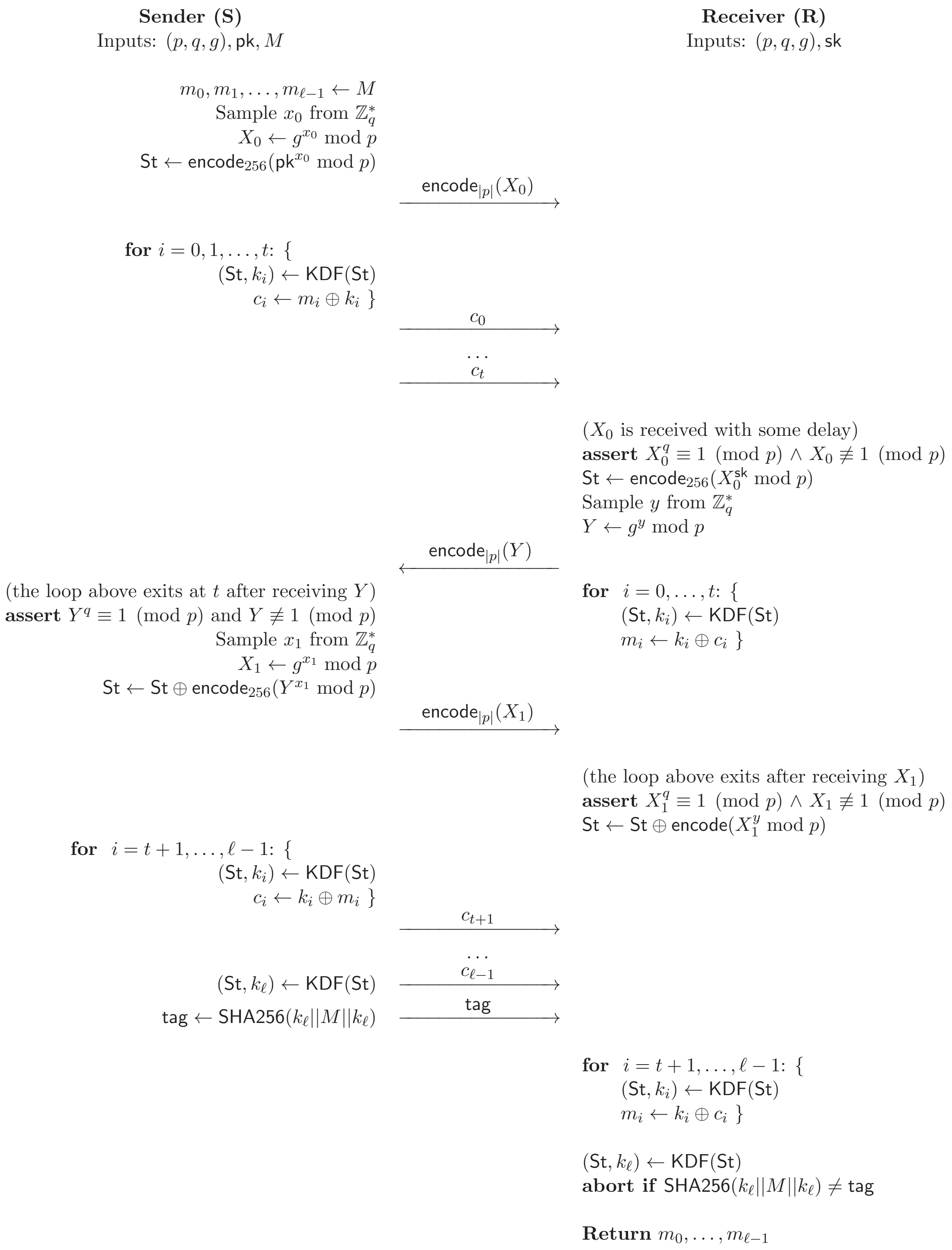
$$c_2 \,||\, \mathsf{encode}_{|p|}(X_1) \,||\, c_0 \,||\, \mathsf{encode}_{|p|}(Y) \,||\, c_3 \,||\, \mathsf{encode}_{|p|}(X_0) \,||\, \mathsf{tag} \,||\, c_1$$

for $\ell = 4$ (ℓ will be larger in your actual queries). The session is based on HTTP POST request, and contains a password of exactly 15 characters. It has the following format:

```
...&password=Abc-Xzz-R93-Zzz&...
```

You are further provided with the long-term secret key sk (as `Q1_sk`) of the receiver to capture the essence of *forward-secrecy*. An oracle is provided at `lasecpc25.epfl.ch` at port `5559`, it receives only your SCIPER number and returns a new session transcript (based on the

[1]This "optional" feature is enabled by default in Firefox. See `security.tls.enable_0rtt_data` option at `about:config` page.



```
Sender (S)                                   Receiver (R)
Inputs: $(p, q, g), \mathsf{pk}, M$          Inputs: $(p, q, g), \mathsf{sk}$

$m_0, m_1, \ldots, m_{\ell-1} \leftarrow M$
Sample $x_0$ from $\mathbb{Z}_q^*$
$X_0 \leftarrow g^{x_0} \bmod p$
$\mathsf{St} \leftarrow \mathsf{encode}_{256}(\mathsf{pk}^{x_0} \bmod p)$
                  $\xrightarrow{\mathsf{encode}_{|p|}(X_0)}$

for $i = 0, 1, \ldots, t$: {
    $(\mathsf{St}, k_i) \leftarrow \mathsf{KDF}(\mathsf{St})$
    $c_i \leftarrow m_i \oplus k_i$ }
                  $\xrightarrow{c_0}$
                  ...
                  $\xrightarrow{c_t}$

                                             ($X_0$ is received with some delay)
                                             assert $X_0^q \equiv 1 \pmod p \wedge X_0 \not\equiv 1 \pmod p$
                                             $\mathsf{St} \leftarrow \mathsf{encode}_{256}(X_0^{\mathsf{sk}} \bmod p)$
                                             Sample $y$ from $\mathbb{Z}_q^*$
                                             $Y \leftarrow g^y \bmod p$
                  $\xleftarrow{\mathsf{encode}_{|p|}(Y)}$

(the loop above exits at $t$ after receiving $Y$)    for $i = 0, \ldots, t$: {
assert $Y^q \equiv 1 \pmod p$ and $Y \not\equiv 1 \pmod p$      $(\mathsf{St}, k_i) \leftarrow \mathsf{KDF}(\mathsf{St})$
Sample $x_1$ from $\mathbb{Z}_q^*$                        $m_i \leftarrow k_i \oplus c_i$ }
$X_1 \leftarrow g^{x_1} \bmod p$
$\mathsf{St} \leftarrow \mathsf{St} \oplus \mathsf{encode}_{256}(Y^{x_1} \bmod p)$
                  $\xrightarrow{\mathsf{encode}_{|p|}(X_1)}$

                                             (the loop above exits after receiving $X_1$)
                                             assert $X_1^q \equiv 1 \pmod p \wedge X_1 \not\equiv 1 \pmod p$
                                             $\mathsf{St} \leftarrow \mathsf{St} \oplus \mathsf{encode}(X_1^y \bmod p)$

for $i = t+1, \ldots, \ell-1$: {
    $(\mathsf{St}, k_i) \leftarrow \mathsf{KDF}(\mathsf{St})$
    $c_i \leftarrow k_i \oplus m_i$ }
                  $\xrightarrow{c_{t+1}}$
                  ...
$(\mathsf{St}, k_\ell) \leftarrow \mathsf{KDF}(\mathsf{St})$      $\xrightarrow{c_{\ell-1}}$
$\mathsf{tag} \leftarrow \mathsf{SHA256}(k_\ell || M || k_\ell)$   $\xrightarrow{\mathsf{tag}}$

                                             for $i = t+1, \ldots, \ell-1$: {
                                                 $(\mathsf{St}, k_i) \leftarrow \mathsf{KDF}(\mathsf{St})$
                                                 $m_i \leftarrow k_i \oplus c_i$ }

                                             $(\mathsf{St}, k_\ell) \leftarrow \mathsf{KDF}(\mathsf{St})$
                                             abort if $\mathsf{SHA256}(k_\ell || M || k_\ell) \neq \mathsf{tag}$

                                             Return $m_0, \ldots, m_{\ell-1}$
```

Figure 1: The oversimplified flow of messages in the 0-RTT protocol, **if we naively assume** that the channel preserves the order of delivery.

same message with new randomization) by repeating the same query. The oracle responses are encoded in hexadecimal. Recover the password as ASCII string and provide it as your answer in `Q1_pwd` variable as string value. Example:

```
Q1_pwd = "Abc-Xzz-R93-Zzz"
```

Exercise 2 SHA2 Sponge

A sponge construction is used for SHA3, but the sponge construction can be used with any other function. In this question, we define the following sponge construction with SHA2:

```
Algorithm SHA2-Sponge($r, c, d, m$)
    $m \leftarrow m \| 10 \| 10^{-(|m|+4) \bmod r} \| 1$
    $m_0, m_1, \ldots, m_{n-1} \overset{r}{\leftarrow} m$
    $s \leftarrow 0^{r+c}$
    for $i = 0$ to $n-1$ do
        $s \leftarrow SHA2(s \oplus (m_i \| 0^c))$
    end for
    $h \leftarrow \perp$
    for $i = 0$ to $\lfloor d/r - 1 \rfloor$ do
        $h \leftarrow h \| s_0 \| s_1 \| \cdots \| s_{r-1}$
        $s \leftarrow SHA2(s)$
    end for
    if $d \not\equiv 0 \pmod r$ then
        $h \leftarrow h \| s_0 \| \cdots \| s_{(d \bmod r)-1}$
    end if
    return $h$
end Algorithm
```

In your parameter file, you will find integers r, c, d in `Q2_r`, `Q2_c`, `Q2_d` and a message m encoded in base64 in `Q2_m`. Compute base64(SHA2-Sponge(r, c, d, m)) and write it under `Q2_h` in your answer file.

Exercise 3 Interactive Password-based Authentication

A server that provides a secret service (in fact, it is streaming videos with meerkats) is using an interactive challenge-response protocol for user authentication. Let $\mathsf{AES}(K, m)$ be the encryption of a message m by using the AES with a key K, $|w|$ be the length of w, and 0^n is a string of n zero bits. When a new user signs up with a username id, the server and the user generate shared secrets s_0 and s_1 from a password w which is chosen by the user upon enrolment where $s_0 = \mathsf{AES}(0^{128}, w \| 0^{128-|w|})$, $s_1 = \mathsf{AES}(1^{128}, w \| 0^{128-|w|})$. I.e., s_0 and s_1 are AES encryptions of $w \| 0^{128-|w|}$ with keys 0^{128} and 1^{128}.

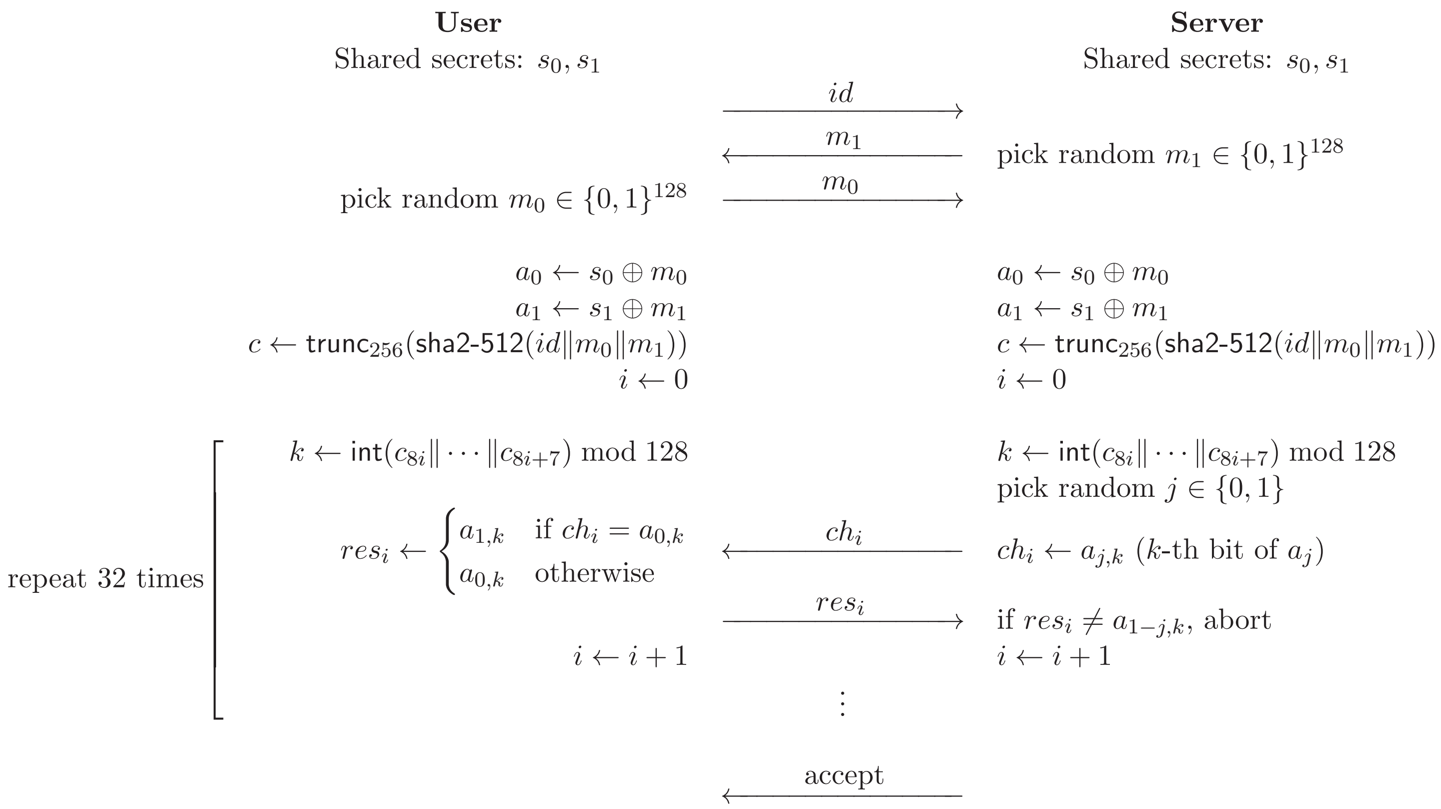
Once the user is registered, the user can authenticate itself to the server with its password w by following the protocol in Figure 2.

For the storage efficiency, the binary vector of dimension n is encoded as byte string of $n/8$. For example, a vector

```
[0, 0, 1, 0, 0, 1, 1, 1]
```

is encoded as

```
0x27
```


User
Shared secrets: s_0, s_1
Server
Shared secrets: s_0, s_1
id
m_1
pick random $m_1 \in \{0,1\}^{128}$
pick random $m_0 \in \{0,1\}^{128}$
m_0
$a_0 \leftarrow s_0 \oplus m_0$
$a_1 \leftarrow s_1 \oplus m_1$
$c \leftarrow \mathsf{trunc}_{256}(\mathsf{sha2\text{-}512}(id \| m_0 \| m_1))$
$i \leftarrow 0$
$a_0 \leftarrow s_0 \oplus m_0$
$a_1 \leftarrow s_1 \oplus m_1$
$c \leftarrow \mathsf{trunc}_{256}(\mathsf{sha2\text{-}512}(id \| m_0 \| m_1))$
$i \leftarrow 0$
repeat 32 times
$k \leftarrow \mathsf{int}(c_{8i} \| \cdots \| c_{8i+7}) \bmod 128$
$k \leftarrow \mathsf{int}(c_{8i} \| \cdots \| c_{8i+7}) \bmod 128$
pick random $j \in \{0,1\}$
$res_i \leftarrow \begin{cases} a_{1,k} & \text{if } ch_i = a_{0,k} \\ a_{0,k} & \text{otherwise} \end{cases}$
ch_i
$ch_i \leftarrow a_{j,k}$ (k-th bit of a_j)
res_i
if $res_i \neq a_{1-j,k}$, abort
$i \leftarrow i + 1$
$i \leftarrow i + 1$
$\vdots$
accept

*$\mathsf{trunc}_n(x)$ returns n most significant bits of x.
*$\mathsf{int}(b_0 \| b_1 \| \ldots \| b_7)$ returns $\sum_{i=0}^{7} b_{7-i} \cdot 2^i$.

Figure 2: The challenge-response protocol for user authentication

The crypto-apprentice is one of clients of the secret meerkat-streaming service, and you would like to log into his account. You can access to the server lasecpc25.epfl.ch on port 7777 using Sage to initiate the protocol. There are three different type of queries: one to initialize the protocol, one to send m_0 and one to send a response.

In order to initialize the protocol, you need to send your username. This query should have the following format:

```
username\n
```

Then, you will get a response of the following format (You shouldn't close the connection to respond to the challenge.):

```
base64(m_1)\n
```

where m_1 is the encoded binary vector of dimension 128. Then, you can send a response to the server with a query of following format:

```
base64(m_0)\n
```

The server then will send a challenge of following format:

```
a\n
```

where a is either 0 or 1 which corresponds to ch_i. Then, you can send a response res_i to the server with the following query:

```
b\n
```

where b is either 0 or 1 which corresponds to res_i. Then, you will get a response of the following format:

```
c\n
```

where c is either an abort message, an accept message or ch_{i+1}.

In your parameter file, you will find your username under `Q3_id`. Recoever your password w and write in under `Q3_w` in your answer file.

Exercise 4 I have commitment issues.

Our adventurous apprentice is bored of usual groups, so she wants to try a more intriguing group. She picks a large strong prime p (i.e. $q = (p-1)/2$ is also prime) and defines the group $\mathcal{G}$ over the set $\mathbb{Z}_p^* \times \mathbb{Z}_p$ with the following operation:

$$\forall (a,u),(b,v) \in \mathbb{Z}_p^* \times \mathbb{Z}_p \quad (a,u) * (b,v) = (ab, bu + av).$$

Since she does not intend to use the group for public-key encryption, but only for Pedersen's commitment, she ignores whether the usual problems (e.g. DL, DDH, CDH) are hard over this particular group $\mathcal{G}$.

She generates the parameters (p, g, h) and publishes them. Here, $g, h \in \mathcal{G}$ are chosen such that $|\langle g \rangle| = |\langle h \rangle| = 2p$. She chooses the message domain as $\mathbb{Z}_{2p}$ and randomness domain as $\mathbb{Z}_p$. She tweaks the Pedersen's commitment scheme over $\mathcal{G}$:

- To commit to an integer $m \in \mathbb{Z}_{2p}$, she picks a random $r \in \mathbb{Z}_p$ and computes the commitment $c = g^m * h^r$.

- To open c, she reveals (m, r). The opening to m is valid if $g^m * h^r = c$ and $(m, r) \in \mathbb{Z}_{2p} \times \mathbb{Z}_p$.

You are given the public parameters (p, g, h) (as `Q4_p`, `Q4_g` and `Q4_h` respectively). For the first part of the exercise, you need to disprove her *unconditionally hiding* claim of the scheme (according to the definition in the page 821 of the course slides). She gives you a list of tuples (as `Q4_list`) $(c_0, m_0), (c_1, m_1), \ldots, (c_\ell, m_\ell)$ where c_i is a claimed commitment and m_i is a randomly sampled message. You are expected to tell whether $c_i \in \mathcal{G}$ can be a commitment of m_i (encode as 1) or if it is impossible to commit to m_i and obtain c_i (encode as 0). By combining your answers for each i, return a list of integers in `Q4_hiding`.

For the binding property, you are given a commitment c (as `Q4_c`) in your parameter file. Provide a valid opening (m, r) to given commitment c such that m is your SCIPER number. r must an integer and must be given as `Q4_r` in your answer file.