Applied Cryptography: Project

Sylvain Ruhault

This project requires that you run `python`. Ensure that you have last versions of files `algebra.py`, `rfc7748.py`, as sent by your teacher. Ensure also that you have installed `pyCryptodome` cryptographic library, as explained by your teacher.

Note: several questions are given in this document, you do not need to answer them in your report, they are here as hints to help your developement tasks.

1 Introduction

Objective of this project is to implement an electronic voting system, based on cryptographic mechanisms. This voting system is simple and covers only a few properties of a real electronic voting system:

- Vote privacy.
- Vote elligibility.
- Homomorphic tally.

In particular, this voting system does *not* cover other important properties, such that:

- Voter authentication.
- Proof of valid vote.
- Individual verifiability.
- Universal verifiability.

2 DSA algorithm

2.1 DSA signature implementations

You will find in file `dsa.py` prototypes for the following algorithms:

- DSA key generation
- DSA signature generation
- DSA signature verification

Assume we use **SHA256** as hash function and **MODP Group 24** for public parameters. For each algorithm, what are the inputs and the outputs, what are their length in bits ? Complete these implementations with `mod_inv` algorithm from `algebra.py` and use the following imports:

```
from algebra import mod_inv
from Crypto.Hash import SHA256
from random import randint
```

2.2 Signature implementation test

We still use **SHA256** as hash function and **MODP Group 24** for public parameters. Let m (a message), k (the nonce used in signature generation) and x (signature private key) defined with:

$m =$ `An important message !`
$k =$ `0x7e7f77278fe5232f30056200582ab6e7cae23992bca75929573b779c62ef4759`
$x =$ `0x49582493d17932dabd014bb712fc55af453ebfb2767537007b0ccff6e857e6a3`

Use your implementation of DSA signature algorithm and verify that you obtain (r, s) as signature, defined with:

$r =$ `0x5ddf26ae653f5583e44259985262c84b483b74be46dec74b07906c5896e26e5a`
$s =$ `0x194101d2c55ac599e4a61603bc6667dcc23bd2e9bdbef353ec3cb839dcce6ec1`

3 El Gamal encryption algorithm

3.1 Multiplicative version

You will find in file `elgamal.py` prototypes for the following algorithms:

- El Gamal key generation
- El Gamal encryption
- El Gamal decryption

Assume we use **MODP Group 24** for public parameters. For each algorithm, what are the inputs and the outputs, what are their length in bits ? Complete these implementations with `mod_inv` algorithm from `algebra.py` and use the following imports:

```
from algebra import mod_inv
from random import randint
```

3.2 Homomorphic encryption : multiplicative version

We still use **MODP Group 24** for public parameters. Let m_1 and m_2 two messages defined with:

$m1 =$ `0x2661b673f687c5c3142f806d500d2ce57b1182c9b25bfe4fa09529424b`
$m2 =$ `0x1c1c871caabca15828cf08ee3aa3199000b94ed15e743c3`

Use your implementation of El Gamal encryption algorithm to encrypt these two messages and compute the following process, where **EG_Encrypt** denotes El Gamal encryption and **EG_Decrypt** denotes El Gamal decryption:

- $(r_1, c_1) = \textbf{EG\_Encrypt}(m_1)$
- $(r_2, c_2) = \textbf{EG\_Encrypt}(m_2)$
- Let $(r_3, c_3) = (r_1 \times r_2, c_1 \times c_2)$
- Let $m_3 = \textbf{EG\_Decrypt}(r_3, c_3)$
- Assess $m_3 = m_1 \times m_2$
- Decode m_3 with `int_to_bytes` from `algebra.py` !

3.3 Homomorphic encryption : additive version

You have implemented El Gamal encryption such that for two messages m_1 and m_2, $\textbf{EG\_Encrypt}(m_1) \times \textbf{EG\_Encrypt}(m_2) = \textbf{EG\_Encrypt}(m_1 \times m_2)$.

Explain how you can turn your previous implementation into an *additive* version, i.e: $\textbf{EGA\_Encrypt}(m_1) \times \textbf{EGA\_Encrypt}(m_2) = \textbf{EGA\_Encrypt}(m_1 + m_2)$. Implement it !

In the context of electronic voting, we will use the additive version of El Gamal where messages are 0 or 1. We still use **MODP Group 24** for public parameters.

Let $m_1 = 1$, $m_2 = 0$, $m_3 = 1$, $m_4 = 1$, $m_5 = 0$ five messages.

Use your implementation of El Gamal encryption algorithm (*additive version*) to encrypt these five messages and compute the following process, where **EGA_Encrypt** denotes El Gamal encryption (additive version) and **EG_Decrypt** denotes El Gamal decryption:

- $(r_1, c_1) = \textbf{EGA\_Encrypt}(m_1)$
- $(r_2, c_2) = \textbf{EGA\_Encrypt}(m_2)$
- $(r_3, c_3) = \textbf{EGA\_Encrypt}(m_3)$
- $(r_4, c_4) = \textbf{EGA\_Encrypt}(m_4)$
- $(r_5, c_5) = \textbf{EGA\_Encrypt}(m_5)$
- Let $(r, c) = (r_1 \times r_2 \times r_3 \times r_4 \times r_5, c_1 \times c_2 \times c_3 \times c_4 \times c_5)$
- Compute $\textbf{EG\_Decrypt}(r, c)$. Beware that the result of this computation is not directly $m = m_1 + m_2 + m_3 + m_4 + m_5$! Indeed you obtain g^m and not m (where g is **MODP Group 24** public parameter). But as in this situation m is small, a direct brute force search is possible. You will find in file `elgamal.py` an implementation of a brute force search, named `bruteLog`. Use it to compute m.
- Assess $m = m_1 + m_2 + m_3 + m_4 + m_5 = 3$

4 Elliptic Curves Cryptography

Note: this section is informative only.

Consider RFC 7748 (`https://www.rfc-editor.org/rfc/rfc7748`). This RFC contains some python code and some pseudocode. These allow to implement scalar multiplication in `curve25519`, which is usually named **X25519**.

In file `rfc7748.py` you will find implementations for these functions. They both use an internal function, named **mul**, whose pseudocode is given in RFC 7748. Function **mul** itself uses another function, named **cswap**, for which a proposed implementation is given in file `rfc7748.py`. Implementation also uses encoding and decoding functions, whose python code is given in RFC 7748 and in file `rfc7748.py`.

5 ECDSA signature algorithm

5.1 ECDSA signature implementations

In file `rfc7748.py` you have an optimized implementation of scalar multiplication which is adapted from RFC 7748. This implementation does not use point addition and point doubling to ensure security against side-channels attacks. This implementation is usefull in contexts where ECDH is performed.

However, to compute ECDSA, we need to have access to point addition on the curve. In file `rfc7748.py` you will find implementations for three additionnal functions:

- Function `add`, that implements point addition. Beware that this function also implements point doubling !
- Function `mult`, that implements (unoptimized and unprotected) scalar multiplication. This function uses internally `add` function.
- Function `computeVcoordinate`, that computes v coordinate of a point given its u coordinate (a point on a curve has (u, v) coordinates).

With these functions, you have all material to implement ECDSA signature and verification algorithms on `curve25519`. In file `ecdsa.py` you will find prototypes for the following algorithms:

- ECDSA key generation
- ECDSA signature generation
- ECDSA signature verification

Assume we use **SHA256** as hash function. For each algorithm, what are the inputs and the outputs, what are their length in bits ? Complete these implementations with `mod_inv` algorithm from `algebra.py`, with `add` ans `mult` algorithms from `rfc7748.py` and use the following imports:

```
from rfc7748 import x25519, add, computeVcoordinate, mult
from Crypto.Hash import SHA256
from random import randint
from algebra import mod_inv
```

5.2 Signature implementation test

We still use **SHA256** as hash function. Let m (a message), k (the nonce used in signature generation) and x (signature private key) defined with:

$m =$ `A very very important message !`
$k =$ `0x2c92639dcf417afeae31e0f8fddc8e48b3e11d840523f54aaa97174221faee6`
$x =$ `0xc841f4896fe86c971bedbcf114a6cfd97e4454c9be9aba876d5a195995e2ba8`

Use your implementation of ECDSA signature algorithm and verify that you obtain (r, s) as signature, defined with:

$r =$ `0x429146a1375614034c65c2b6a86b2fc4aec00147f223cb2a7a22272d4a3fdd2`
$s =$ `0xf23bcdebe2e0d8571d195a9b8a05364b14944032032eeeecd22a0f6e94f8f33`

Check also your signature verification algorithm.

6 EC El Gamal encryption algorithm

El Gamal encryption algorithm can be computed on elliptic curves. We will use `curve25519` implementation from file `rfc7748.py` to implement this algorithm. Our purpose is to use it for electronic voting, so messages to encrypt are equal to 0 or 1.

6.1 Implementation

You will find in file `ecelgamal.py` prototypes for the following algorithms:

- EC El Gamal key generation
- EC El Gamal encryption
- EC El Gamal decryption

In order to perform EC El Gamal encryption, we need to map messages 0 and 1 to points on the elliptic curve. Furthermore, we need to use points that have an additive property, as have 0 and 1 for integers. One solution is to map 0 to point at infinity of coordinates $(1, 0)$ and to map 1 to the base point of the elliptic curve. We also need to use point substraction in order to compute these functions. In files `rfc7748.py` and `ecelgamal.py` you will find implementation for additionnal functions:

- Function `sub`, that implements point substraction.
- Function `ECencode`, that maps 0 and 1 to the correct points on the elliptic curve.

Complete propotypes with `mod_inv` algorithm from `algebra.py`, with `add`, `sub` and `mult` algorithms from `rfc7748.py` and use the following imports:

```
from rfc7748 import x25519, add, sub, computeVcoordinate, mult
from random import randint
from algebra import mod_inv
```

6.2 Homomorphic encryption : additive version

EC El Gamal is already additive ! Explain why !

Let $m_1 = 1$, $m_2 = 0$, $m_3 = 1$, $m_4 = 1$, $m_5 = 0$ five messages.

Use your implementation of EC El Gamal encryption algorithm to encrypt these five messages and compute the following process, where **ECEG_Encrypt** denotes EC El Gamal encryption and **ECEG_Decrypt** denotes EC El Gamal decryption. Beware that in the process, r_i and c_i are points on elliptic curve (hence they are composed of two coordinates). Furthermore, + here denotes points addition on elliptic curve !

- $(r_1, c_1) =$ **ECEG_Encrypt**(m_1)
- $(r_2, c_2) =$ **ECEG_Encrypt**(m_2)
- $(r_3, c_3) =$ **ECEG_Encrypt**(m_3)
- $(r_4, c_4) =$ **ECRG_Encrypt**(m_4)
- $(r_5, c_5) =$ **ECEG_Encrypt**(m_5)
- Let $(r, c) = (r_1 + r_2 + r_3 + r_4 + r_5, c_1 + c_2 + c_3 + c_4 + c_5)$
- Compute **ECEG_Decrypt**(r, c). Beware that as before the result of this computation is not directly $m = m_1+m_2+m_3+m_4+m_5$! As before you obtain $m \times G$ and not m (where G is base point of **curve25519**). But as before, in this situation m is small, a direct brute force search is possible. You will find in file `ecelgamal.py` an implementation of this brute force search, named `bruteECLog`. Use it to compute m.
- Assess $m = m_1 + m_2 + m_3 + m_4 + m_5 = 3$

7 Electronic Voting

With all previous algorithms, you have all material to implement a simple electronic voting system, that ensures vote privacy and voters' elligibility.

In this system, assume a voter has to choose between five candidates C_1, C_2, C_3, C_4, C_5. A vote for a candidate C_i generates a list composed of 0 (four times) or 1 (one time). In this system, blank vote is not possible. For example:

- Vote for candidate C_1 generates list $(1, 0, 0, 0, 0)$
- Vote for candidate C_2 generates list $(0, 1, 0, 0, 0)$
- Vote for candidate C_4 generates list $(0, 0, 0, 1, 0)$

7.1 Privacy

To ensure vote privacy, the voting system encrypts **each message** of the list. Hence there are five encrypted messages for each vote! Furthermore, to ensure that voters' choice cannot be linked with their encrypted ballot, the voting system **does not** decrypt each voter's ballot, but use homomorphic property to decrypt the sum of all choices.

Describe the previous process for two voters, e.g with **EGA_Encrypt** seen previously.

7.2 Elligility

To ensure elligibility, each voter signs its encrypted ballot with its own signature key distributed by the voting system. At reception, ballot signature are verified by the voting system. Hence a ballot finally contains five encrypted messages (as seen before) and a signature of these five messages ! All asymmetric signature algorithms can be used to perform this operation.

Depending on the signature algorithm, signature size will be different !

7.3 Implementation

As seen previously, the voting system can either use El Gamal or EC El Gamal encryption for vote privacy, and DSA or ECDSA signature for voter elligibility. Your work is to implement all combinations. As before, parameters are the following:

- **MODP Group 24** for El Gamal encryption and DSA signature algorithm.
- **SHA256** for hash function for DSA and ECDSA signature algorithms.
- **curve25519** for EC El Gamal encryption and ECDSA signature algorithms.

To fix parameters, you can assume the following:

- There are ten (10) voters.
- There are five (5) candidates for the election.

Hence you need to implement, for each algorithm:

- Voters' signature key generation and distribution : one signature key pair for each voter.
- Ballot generation: each ballot contains 5 encrypted messages.
- Ballot multiplication (for El Gamal) or addition (for EC El Gamal): this generates 5 encrypted messages (one for each candidate).
- Five decryptions and brute force searches to recover election result (number of votes per candidate).

8 Deliverables

Expected deliverables for this project are:

- Completed files `dsa.py`, `elgamal.py`, `ecdsa.py` and `ecelgamal.py`.
- An implementation of the described voting system, that uses as import functions that you have implemented in files `dsa.py`, `elgamal.py`, `ecdsa.py` and `ecelgamal.py`.
- A report that describes and explains your implementation.

Note that you can write your report in English or in French. You can also work as a team (three students max.).

Due Date: April 26^{th}, 23h59m