Presentació

En aquesta pràctica treballarem els conceptes tractats en els mòduls 5, 6 i 7 del curs, que inclouen la criptografia de clau pública i els protocols criptogràfics.

D'una banda, a la pràctica treballarem l'algorisme de signatura digital d'ElGamal, un algorisme basat en el problema del logaritme discret. D'altra banda també implementarem un protocol de prova de coneixement nul, en concret el protocol proposat per D. Chaum, J. Evertse i J. Van de Graaf per a demostrar el coneixement d'un logaritme discret.

Objectius

Els objectius d'aquesta pràctica són:

1. Estudiar els conceptes relatius als algorismes de signatura digital.
2. Implementar l'algorisme de signatura d'ElGamal.
3. Treballar els conceptes de proves de coneixement nul.
4. Implementar una prova de coneixement nul.

Descripció de la Pràctica

Aquesta pràctica consta de dues parts. La primera part se centra en els continguts corresponents a la criptografia de clau pública i la segona part en els protocols criptogràfics.

Per tal de realitzar aquesta pràctica, caldrà que realitzeu algunes operacions modulars, com ara calcular inversos modulars, així com treballar amb nombres primers. Per fer tots aquests càlculs, farem servir la llibreria de Python `sympy`[1]. Aquesta llibreria ofereix funcionalitats matemàtiques avançades, que ens seran molt útils per a realitzar les activitats de la pràctica. Per instal·lar la llibreria, només cal que executeu `pip3 install sympy`.

Veureu que a l'esquelet de la pràctica ja s'importen dues funcions d'aquesta llibreria (`isprime` i `is_primitive_root`), que es fan servir en els constructors de les classes que implementen la prova de coneixement nul. Addicionalment, podeu importar altres funcions d'aquesta llibreria que considereu necessàries, així com qualsevol altra funció de les llibreries estàndard de Python.

[1] https://www.sympy.org/

1 Implementació de l'algorisme de signatura ElGamal (3,5 punts)

La primera part de la pràctica consistirà a implementar l'algorisme de signatura d'ElGamal, que trobareu descrit a l'apartat 4.2 del mòdul 5 de l'assignatura "Criptografia de clau pública". L'algorisme de signatura d'ElGamal és un algorisme de signatura probabilístic, que es basa en la dificultat de calcular el logaritme discret.

En concret, implementarem la funció de generació de claus, la funció de signatura, la validació de la signatura, i un atac que es pot realitzar per a recuperar la clau privada en una situació concreta.

1.1 Funció que implementa la generació de claus ElGamal (0,75 punts)

Aquesta funció permetrà generar claus ElGamal aleatòries. La funció rebrà com a paràmetre la mida en bits de la clau, `n_bits`, i generarà una clau aleatòria amb un primer p d'`n_bits` bits de longitud. La funció retornarà una tupla de dos elements, corresponents a les claus privada i pública generades.

- La variable `n_bits` contindrà la mida de la clau (és a dir, la mida del primer p) en bits.
- La funció retornarà una tupla amb les claus privada i pública generades, respectivament. La clau privada serà una tupla de tres elements: el primer p, el valor α, i el secret d. La clau pública serà també una tupla de tres elements: el primer p, el valor α, i el valor públic β.

1.2 Funció que implementa la signatura amb ElGamal (1 punt)

Aquesta funció permetrà generar signatures amb ElGamal. La funció rebrà com a paràmetres una tupla amb la clau privada (tal com la retorna la funció de generació de claus) i el missatge a signar, i generarà una signatura vàlida per a aquest missatge, triant el valor k de manera aleatòria. Opcionalment, la funció pot rebre el valor k com a paràmetre. En aquest cas, en comptes de generar la signatura amb un valor de k seleccionat aleatòriament, la signatura es farà amb el valor k rebut. La funció retornarà una tupla de dos elements, als valors r i s que conformen la signatura digital.

- La variable `k_priv` contindrà una tupla de tres elements amb la clau privada (el primer p, el valor α, i el secret d).
- La variable `m` contindrà un enter amb el missatge a signar.
- La variable `k` contindrà, opcionalment, un enter amb el valor k a utilitzar durant la signatura.
- La funció retornarà una tupla amb els valors r i s que conformen la signatura digital.

1.3 Funció que implementa la validació de la signatura ElGamal (0,75 punts)

Aquesta funció permetrà validar una signatura feta amb l'algorisme de ElGamal. La funció rebrà com a paràmetres la signatura, la clau pública i el missatge, i retornarà un booleà indicant si la signatura és vàlida per a aquest missatge.

- La variable `sig` contindrà una tupla amb la signatura digital, tal com la retorna la funció de signatura.
- La variable `k_pub` contindrà una tupla de tres elements amb la clau pública (el primer p, el valor α, i el valor públic β).
- La variable `m` contindrà un enter amb el missatge a signar.
- La funció retornarà una valor booleà (`True` o `False`), indicant si la validació de la signatura ha resultat correcta o no.

1.4 Funció que implementa la recuperació de clau a partir de dues signatures ElGamal (1 punt)

L'algorisme de signatura ElGamal fa servir un valor aleatori k en el procés de signatura. És important que aquest valor sigui únic per cada signatura generada amb una determinada clau, ja que en cas contrari, un atacant que disposi de dues signatures realitzades per un mateix usuari (amb una mateixa clau privada) sobre dos missatges diferents, serà capaç de recuperar la clau privada utilitzada per crear-les.

Aquesta funció realitzarà doncs l'atac de recuperació de la clau privada a partir de dues signatures fetes sobre dos missatges diferents amb una mateixa clau privada. La funció rebrà com a arguments la clau pública que valida les signatures `k_pub`, les dues signatures (`sig1` i `sig2`) i els dos missatges signats (respectivament, `m1` i `m2`). La funció retornarà la clau privada recuperada quan sigui possible realitzar l'atac, o bé el valor -1 si no és possible realitzar l'atac a partir de les signatures proporcionades.

- La variable `k_pub` contindrà una tupla de tres elements amb la clau pública (el primer p, el valor α, i el valor públic β).
- La variable `m1` contindrà un enter amb el primer missatge.
- La variable `sig1` contindrà una tupla amb la signatura digital del primer missatge, tal com la retorna la funció de signatura.
- La variable `m2` contindrà un enter amb el segon missatge.
- La variable `sig2` contindrà una tupla amb la signatura digital del segon missatge, tal com la retorna la funció de signatura.
- La funció retornarà una tupla amb la clau privada recuperada (en el mateix format que la funció de generació de claus, és a dir, una tupla amb els elements p, α, i d) si s'ha pogut recuperar la clau privada, o -1 si no és possible fer l'atac.

2 Implementació del protocol de coneixement del logaritme discret (3 punts)

En aquesta part de la pràctica implementarem el protocol de prova de coneixement nul que hi ha descrit en l'apartat 4.1 del mòdul 7 "Protocols criptogràfics" dels materials de l'assignatura. Així doncs, el nostre protocol estarà format per dos usuaris, l'Alice i el Bob. L'Alice actuarà de provador i voldrà demostrar al Bob, que serà el verificador, que coneix el valor sense revelar-lo. Per tal que el nostre protocol pugui ser implementat, farem servir programació orientada a objectes [2] i definirem classes per a representar el comportament de cada usuari. Una classe representarà un tipus d'usuari. Les variables d'instància de la classe es faran servir per guardar els valors que cada usuari coneix i necessita per a executar el protocol. Els mètodes de la classe seran les accions que pot executar cada usuari. Aquests mètodes podran cridar funcions que haguem desenvolupat en altres exercicis de la pràctica.

La classe per al provador la definirem d'aquesta manera:

```
class UocZkpProver:

    def __init__(self, p, g, y, x, name="HonestProver"):

        self.p = None
        self.g = None
        self.y = None
        self.x = None
        self.r = None
        self.name = name

        # ...

    def compute_c(self):

        c = None

        # --- IMPLEMENTATION GOES HERE ---

        # --------------------------------

        print_debug("{}:\tI amb sending c = {}".format(self.name, c), LOG_INFO)
        return c

    def compute_h(self, b):

        h = None

        # --- IMPLEMENTATION GOES HERE ---

        # --------------------------------

        print_debug("{}:\tI amb sending h = {}".format(self.name, h), LOG_INFO)
        return h
```

Figura 1: Estructura de la classe provador.

Fixeu-vos que el provador té com a variables d'instància els paràmetres del sistema p i g, el valor del qual vol provar el logaritme discret y i el valor secret x que correspon justament al logaritme discret de y en base g mòdul p. També s'ha de poder guardar el valor r que generarà en el primer pas del protocol i que després utilitzarà en el tercer pas. A l'hora d'inicialitzar aquests paràmetres (en el constructor de la classe), es comprova que siguin vàlids a través de sentències assert. En la definició del provador també hi ha definides dues accions per mitjà de dos mètodes. El primer calcula c, el valor necessari per al primer pas del protocol. El segon mètode calcula el valor h, necessari en el tercer pas del protocol.

[2] A l'aula de laboratori trobareu un petit document amb unes nocions bàsiques sobre classes en Python.

D'altra banda, de forma anàloga definirem una classe per al verificador:

```
class UocZkpVerifier:

    def __init__(self, p, g, y, name="Verifier"):

        self.p = None
        self.g = None
        self.y = None
        self.c = None
        self.b = None
        self.name = name

        # ...

    def choose_b(self, c):

        # --- IMPLEMENTATION GOES HERE ---

        # --------------------------------

        print_debug("{}:\t\tI have chosen b = {}".format(self.name, self.b), LOG_INFO)
        return self.b

    def verify(self, h):

        result = None

        # --- IMPLEMENTATION GOES HERE ---

        # --------------------------------

        print_debug("{}:\t\tThe result of the verification is {}".format(self.name, result), LOG_INFO)
        return result
```

Figura 2: Estructura de la classe verificador.

A l'igual que el provador, el verificador també emmagatzema les variables corresponents als paràmetres del sistema p i g i el valor y del qual el provador coneix el logaritme discret. Evidentment, el verificador no coneix el valor x. D'altra banda, en la definició del verificador també hi ha definides dues accions per mitjà de dos mètodes. El primer tria el bit b necessari en el segon pas del protocol. El segon mètode verifica la igualtat del pas 4 a partir del valors que s'han anat intercanviant al protocol.

Aquesta segona part de la pràctica consistirà doncs en implementar els mètodes de càlcul dels valors c i h de la classe del provador, i els mètodes de selecció del bit b i verificació de la prova de la classe del verificador.

2.1 Funció que implementa el mètode `compute_c` de la classe del provador (0,5 punts)

Aquest mètode implementarà el pas 1 del protocol. El mètode, sense rebre cap paràmetre, generarà un valor aleatori i en retornarà el corresponent valor c.

- El mètode no rebrà cap paràmetre.
- El mètode retornarà el valor c calculat en el pas 1 del protocol (i actualitzarà les variables d'instància pertinents).

2.2 Funció que implementa el mètode `compute_h` de la classe del provador (0,5 punts)

Aquest mètode implementarà el pas 3 del protocol. El mètode rebrà el bit que ha triat el verificador i en retornarà el corresponent valor h.

- La variable `b` rebrà el bit triat
- La funció retornarà el valor h del pas 3 del protocol.

2.3 Funció que implementa el mètode `choose_b` de la classe del verificador (0,25 punts)

Aquest mètode implementarà el pas 2 del protocol. El mètode, sense rebre cap paràmetre, retornarà un bit aleatori.

- El mètode no rebrà cap paràmetre.
- El mètode retornarà el bit aleatori b (i actualitzarà les variables d'instància pertinents).

2.4 Funció que implementa el mètode `verify` de la classe del provador (0,5 punts)

Aquest mètode implementarà el pas 4 del protocol. El mètode rebrà el valor h i retornarà cert o fals en funció de si la validació ha estat correcta o no.

- La paràmetre `h` contindrà el valor proporcionat pel provador.
- El mètode retornarà `True` o `False`, en funció de si la verificació és o no correcta.

2.5 Funció que implementa el protocol (0,75 punt)

Aquesta funció implementarà el protocol sencer entre provador i verificador. La funció `challenge` rebrà com a variables d'entrada una instància de la classe provador, una instància de la classe verificador, i el nombre d'iteracions que ha d'executar el protocol. La funció executarà el protocol de la prova de coneixement nul tantes vegades com s'hagi especificat, i retornarà un booleà indicant si l'execució ha tingut èxit i un float amb la probabilitat que tenia un provador deshonest de superar la prova amb èxit.

- El paràmetre `prover` contindrà una instància de la classe del provador.
- El paràmetre `verifier` contindrà una instància de la classe del verificador.
- El paràmetre `num_times` contindrà el nombre d'iteracions que executarà el protocol.

- La funció retornarà una tupla `(A, B)` on `A` serà `True` o `False` en funció si el provador ha superat la prova, i `B` la probabilitat que un provador que desconegui el secret pugui enganyar al verificador.

3 Implementació d'un usuari trampós en el protocol de coneixement del logaritme discret (3,5 punts)

Sabem que el protocol implementat en l'apartat anterior és just, és a dir, que el provador no pot enganyar al verificador si no coneix x (el valor del logaritme), gràcies al bit aleatori que el verificador tria en el pas 2 del protocol. Ara bé, si el provador pot saber amb anterioritat quin bit triarà el verificador, pot seleccionar els valors adequats per tal que la verificació funcioni sense saber-ne el logaritme discret. En aquest apartat de la pràctica implementarem aquesta funcionalitat de l'usuari maliciós.

Per fer-ho, crearem dues noves classes, la classe del provador preparada per quan rep el bit $b = 0$ en el pas 2:

```
class UocZkpCheaterProverB0(UocZkpProver):

    def __init__(self, p, g, y, name="CheaterProv0"):

        UocZkpProver.__init__(self, p, g, y, None, name=name)

    def compute_c(self):

        c = None

        # --- IMPLEMENTATION GOES HERE ---

        # --------------------------------

        print_debug("{}:\tI amb sending c = {}".format(self.name, c), LOG_INFO)
        return c

    def compute_h(self, b):

        h = None

        # --- IMPLEMENTATION GOES HERE ---

        # --------------------------------

        print_debug("{}:\tI amb sending h = {}".format(self.name, h), LOG_INFO)
        return h
```

Figura 3: Classe del provador trampós per al bit $b = 0$.

i la classe del provador quan rep el bit $b = 1$ en el pas 2:

```python
class UocZkpCheaterProverB1(UocZkpProver):

    def __init__(self, p, g, y):

        UocZkpProver.__init__(self, p, g, y, None, name="CheaterProv1")

    def compute_c(self):

        c = None

        # --- IMPLEMENTATION GOES HERE ---

        # --------------------------------

        print_debug("{}:\tI amb sending c = {}".format(self.name, c), LOG_INFO)
        return c

    def compute_h(self, b):

        h = None

        # --- IMPLEMENTATION GOES HERE ---

        # --------------------------------

        print_debug("{}:\tI amb sending h = {}".format(self.name, h), LOG_INFO)
        return h
```

Figura 4: Classe del provador trampós per al bit $b = 1$.

Si ens hi fixem, aquestes classes tenen la mateixa estructura que la classe del provador de l'apartat anterior, amb la particularitat que en aquesta ocasió, el valor x no estarà mai informat perquè el provador, que és un trampós, no el coneix. En particular, veureu que les dues classes hereten de la classe provador que hem definit a l'exercici anterior.

3.1 Funció que implementa el mètode `compute_c` de la classe del provador trampós per $b = 0$ (0,75 punts)

Aquest mètode implementarà el pas 1 del protocol. El mètode, sense rebre cap paràmetre, retornarà el corresponent valor c.

- El mètode no rebrà cap paràmetre.
- El mètode retornarà el valor c calculat en el pas 1 del protocol (i actualitzarà les variables d'instància pertinents), assumint que el provador sap que el bit b que li arriba és sempre $b = 0$.

3.2 Funció que implementa el mètode `compute_h` de la classe del provador trampós per $b = 0$ (0,75 punts)

Aquest mètode implementarà el pas 3 del protocol. El mètode rebrà el bit que ha triat el verificador i en retornarà el corresponent valor h.

- La variable `b` rebrà el bit triat. Si el valor del bit és 0, aleshores el provador serà capaç de superar el repte.
- La funció retornarà el valor h del pas 3 del protocol, assumint que el provador sap que el bit b que li arriba és sempre $b = 0$.

3.3 Funció que implementa el mètode `compute_c` de la classe del provador trampós per $b = 1$ (0,75 punts)

Aquest mètode implementarà el pas 1 del protocol. El mètode, sense rebre cap paràmetre, generarà un valor aleatori i en retornarà el corresponent valor c.

- El mètode no rebrà cap paràmetre.
- El mètode retornarà el valor c calculat en el pas 1 del protocol (i actualitzarà les variables d'instància pertinents), assumint que el provador sap que el bit b que li arriba és sempre $b = 1$.

3.4 Funció que implementa el mètode `compute_h` de la classe del provador trampós per $b = 1$ (0,75 punts)

Aquest mètode implementarà el pas 3 del protocol. El mètode rebrà el bit que ha triat el verificador i en retornarà el corresponent valor h.

- La variable `b` rebrà el bit triat. Si el valor del bit és 1, aleshores el provador serà capaç de superar el repte.
- La funció retornarà el valor h del pas 3 del protocol, assumint que el provador sap que el bit b que li arriba és sempre $b = 1$.

3.5 Funció que mostra la probabilitat d'engany que té el provador trampós (0,5 punt)

Aquesta funció permetrà mostrar gràficament la probabilitat d'engany que té el provador trampós en funció del número d'iteracions del protocol, és a dir, en funció del valor `num_times` de la funció `challenge`. La funció generarà una gràfica que mostrarà la probabilitat d'engany (a l'eix y) en funció del número d'iteracions del protocol (eix x), fent servir la funció `challenge` desenvolupada anteriorment. Per a cada número d'iteracions del protocol, la gràfica mostrarà 3 valors: la probabilitat d'engany teòrica, la probabilitat d'engany observada en `num_exp` experiments per al provador trampós que assumeix que, en cada iteració del protocol, en el pas 2, sempre li arriba $b = 0$, i la probabilitat d'engany observada en `num_exp` experiments per al provador trampós que assumeix que, en cada iteració del protocol, en el pas 2, sempre li arriba $b = 1$

- El paràmetre `min_num_of_its` definirà el mínim valor del número d'iteracions del protocol a provar (el mínim valor de l'eix x de la gràfica).
- El paràmetre `max_num_of_its` definirà el màxim valor del número d'iteracions del protocol a provar (el màxim valor de l'eix x de la gràfica).
- La variable `num_exp` definirà el número de vegades que es portarà a terme la prova (per a cada valor del número d'iteracions del protocol) amb l'objectiu de calcular la probabilitat d'engany observada.

- La funció no retornarà cap valor (només mostrarà la gràfica amb els resultats).

Nota 1: Podeu fixar els següents valors per a generar la gràfica:
`p, g, y, x = 7687815937255549241, 27, 828418027377238633, 6041213497581640253`

Nota 2: A més de les llibreries estàndard de Python, si voleu, podeu fer servir la llibreria `matplotlib` per realitzar les gràfiques. Per tal de generar una única gràfica on es mostrin diverses línies, podeu consultar els exemples d'aquest enllaç:
https://www.kaggle.com/andyxie/matplotlib-plot-multiple-lines

Criteris d'avaluació

La puntuació de cada exercici es troba detallada a l'enunciat.

D'altra banda, cal remarcar que el codi que es lliuri de la pràctica ha de contenir els comentaris necessaris per poder-lo seguir i entendre. En cas que el codi no inclogui comentaris, la correcció de la pràctica és realitzarà únicament de forma automàtica i no es proporcionarà una correcció detallada. La no inclusió de comentaris també pot ser motiu de reducció de la nota.

Format i data de lliurament

La data màxima de lliurament de la pràctica és el **28/12/2018** (a les 24 hores).

Juntament amb l'enunciat de la pràctica hi trobareu l'esquelet de la mateixa (fitxer amb extensió .py). Aquest fitxer conté les capçaleres de les funcions que cal que implementeu per a resoldre la pràctica. Aquest mateix fitxer és el que heu de lliurar un cop hi codifiqueu totes les funcions.

Addicionalment, també us proporcionarem un fitxer amb testos unitaris per a cadascuna de les funcions que cal que implementeu. Podeu fer servir aquests testos per comprovar que la vostra implementació gestiona correctament els casos principals, així com per obtenir més exemples concrets del que s'espera que retornin les funcions (més enllà dels que ja es proporcionen en aquest enunciat). Noteu, però, que els testos no són exhaustius (no es proven totes les entrades possibles de les funcions). Recordeu que no es pot modificar cap part del fitxer de testos de la pràctica.

El lliurament de la pràctica constarà d'un únic fitxer Python (extensió .py) on hagueu inclòs la vostra implementació.