MIEI/MI - Estruturas Criptográficas
Trabalho Prático 0

João Alves
a77070@alunos.uminho.pt

Nuno Leite
a70132@alunos.uminho.pt

Universidade do Minho

1 Introdução

A resolução deste trabalho prático tem como objetivo servir de iniciação à componente prática da unidade curricular de Estruturas criptográficas, onde se pretende: - Instalar as ferramentas computacionais necessárias para a realização dos trabalhos práticos. - Demonstrar pequenas aplicações implementadas em **Python** e em **SageMath**.

A aplicação em **Python** deve ser implementada de tal forma que permita a comunicação entre um emissor e um recetor, com as seguintes caracteristicas: - Criptograma e metadados devem ser autenticados. - Utilizar uma cifra simétrica em modo *stream cipher*. - Autenticar previamente a chave.

A aplicação em **SageMath** deve ser implementada de tal forma que: - Crie 4 corpos finitos primos. - Crie um *plot* de uma função em cada um dos corpos finitos primos. - Teste, por amostragem, o facto de que, considerando um expoente n, um elemento primitivo de um corpo g e um número primo p, se $g^n = 1$, então $n = 0 mod(p-1)$.

2 Aplicação Python

2.1 Imports

Esta secção executa os *imports* de **Python** que contêm as funções necessárias para desenvolver a aplicação enunciada.

```
In [1]: import os
        from getpass import getpass
        from multiprocessing import Process,Pipe
        from cryptography.hazmat.backends import default_backend
        from cryptography.hazmat.primitives import hashes, hmac
        from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
        from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
        from cryptography.exceptions import InvalidSignature
        from base64 import b64encode, b64decode
```

2.2 Definição da classe de multiprocessamento

Esta secção tem o propósito de definir a classe de multiprocessamento, que permite uma comunicação bidireccional com o *Emitter* e o *Receiver*, sendo estes dois processos criados e implementados

pela **API multiprocessing**.

```
In [2]: class BiConnection(object):
            def __init__(self,left,right):
                left_side,right_side = Pipe()
                self.timeout = None
                self.left_process = Process(target=left,args=(left_side,))
                self.right_process = Process(target=right,args=(right_side,))
                self.left = lambda : left(left_side)
                self.right = lambda : right(right_side)

            # Execução manual apenas devido ao facto de a password
            # ter que ser lida em ambos os lados do Pipe
            def manual(self):
                self.left()
                self.right()
```

2.3 Definição do Emissor e do Recetor

Nesta secção encontram-se implementados os comportamentos do emissor e do recetor que participam na comunicação bidireccional da aplicação.

```
In [3]: def Emissor(connection):
            con_salt = os.urandom(16)
            passwd = bytes(getpass('Password do emissor: '),'utf-8')
            text_to_send = os.urandom(128)
            print('Texto a cifrar e enviar:')
            print(b64encode(text_to_send))
            try:
                #derivar a chave apartir da password
                derivation = PBKDF2HMAC(
                        algorithm = hashes.SHA256(),
                        length = 64,
                        salt = con_salt,
                        iterations = 100000,
                        backend = default_backend()
                )
                #Separar a password para cifragem e autenticação
                full_key = derivation.derive(passwd)
                cript_key = full_key[:32]
                mac_key = full_key[32:]

                # Utilizar o AES com um dos modos que o torna numa stream cipher
                # para cifrar o criptograma.
                nonce = os.urandom(16)
                cipher = Cipher(algorithm = algorithms.AES(cript_key),mode= modes.CTR(nonce),
                backend = default_backend())
```

```
            cryptogram = cipher.encryptor().update(text_to_send)

            #geração do código de autenticação do criptograma e dos metadados
            message_to_authenticate = cryptogram + nonce + con_salt
            hasher = hmac.HMAC(mac_key,hashes.SHA256(),default_backend())
            hasher.update(message_to_authenticate)
            hash_msg = hasher.finalize()

            #geração do código de autenticação da chave
            #esforço computacional é reduzido no recetor caso insira a password errada.
            hasher_passwd = hmac.HMAC(mac_key,hashes.SHA256(),default_backend())
            hasher_passwd.update(cript_key)
            hash_pass = hasher_passwd.finalize()
            obj = {'cryptogram': cryptogram, 'mac_code': hash_msg,'pass_code':
            hash_pass, 'salt': con_salt, 'nonce': nonce}
            connection.send(obj)
        except Exception as e:
            print(e)
    def Recetor(connection):
        passwd = bytes(getpass('Password do recetor: '), 'utf-8')
        try:
            obj = connection.recv()

            #Obter parâmetros no objeto
            pass_code = obj['pass_code']
            cryptogram = obj['cryptogram']
            mac_code = obj['mac_code']
            salt = obj['salt']
            nonce = obj['nonce']

            #derivar a chave apartir da password lida
            derivation = PBKDF2HMAC(
                    algorithm = hashes.SHA256(),
                    length = 64,
                    salt = salt,
                    iterations = 100000,
                    backend = default_backend()
            )
            #Separar a password para cifragem e autenticação
            full_key = derivation.derive(passwd)
            cript_key = full_key[:32]
            mac_key = full_key[32:]

            #Autenticação prévia da chave
            hasher_passwd = hmac.HMAC(mac_key,hashes.SHA256(),default_backend())
            hasher_passwd.update(cript_key)
            hasher_passwd.verify(pass_code)
```

```
            #Autenticação do criptograma e metadados
            message_to_authenticate = cryptogram + nonce + salt
            hash_msg = hmac.HMAC(mac_key,hashes.SHA256(),default_backend())
            hash_msg.update(message_to_authenticate)
            try:
                hash_msg.verify(mac_code)

                #Decifrar criptograma
                cipher = Cipher(algorithm = algorithms.AES(cript_key),
                mode = modes.CTR(nonce),backend = default_backend())
                plain_text = cipher.decryptor().update(cryptogram)
                print('Texto decifrado:')
                print(b64encode(plain_text))
            except InvalidSignature as i:
                print("Código de autenticação não é válido!")
        except Exception as e:
            print(e)
        connection.close()
```

2.4 Iniciação do processo

Por último, resta apenas criar um objeto de conexão bidireccional passando-lhe como argumentos o emissor e o recetor definidos e, finalmente, prosseguindo com a execução de ambos os processos através da chamada à função manual.

```
In [4]: BiConnection(Emissor,Recetor).manual()
```

```
Password do emissor: ········
Texto a cifrar e enviar:
b'Bp7wjCNmvFlppA98akTDp/GDIHBqUOWiF7rSNuZRFzKEJ3dkCZV7xGeqJt+Y8XSeR820m0AVgckpMAY
+GRcTP1UPT+8osucvorZ1pN4RcWZCFH1Nburz1wcIZRwFOYHmMYFubBToo/ZwlqSoJvcESi5vjiceWxc='
Password do recetor: ········
Texto decifrado:
b'Bp7wjCNmvFlppA98akTDp/GDIHBqUOWiF7rSNuZRFzKEJ3dkCZV7xGeqJt+Y8XSeR820m0AVgckpMAY
+GRcTP1UPT+8osucvorZ1pN4RcWZCFH1Nburz1wcIZRwFOYHmMYFubBToo/ZwlqSoJvcESi5vjiceWxc='
```

Através de uma ligera comparação entre o texto antes de ser cifrado e depois de ser decifrado, pode-se verificar que é exatamente o mesmo.

3 Aplicação SageMath

3.1 Criação dos corpos finitos primos

Nesta secção são criadas duas listas:

- A primeira (P) corresponde à lista de números primos que serão analisados.

- A segunda (*GP*) corresponde à lista de corpos finitos primos, onde cada elemento corresponde ao corpo finito primo de um dos números primos, previamente adicionados à lista *P*.

```
In [1]: P = [37, 163, 263, 1009]
        GP = [GF(p) for p in P]
```

3.2 Definição iniciais

De seguida, definiu-se a função que será aplicada aos pontos dos corpos finitos primos. Esta recebe um ponto *x* e um primo *p* e calcula "*x elevado a p menos 2*". Além disso, é definida também uma lista de listas *L*, onde cada uma das mesmas contém os pontos resultantes da aplicação da função aos pontos de um dos corpos finitos primos.

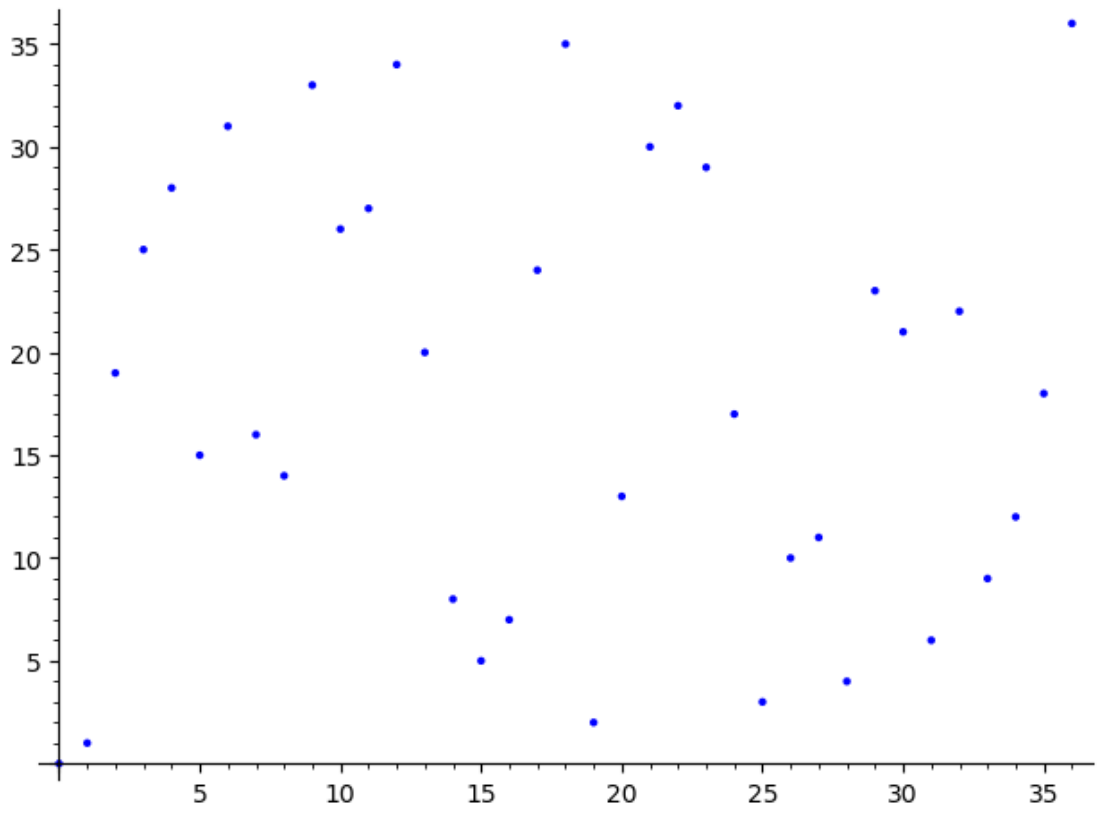
```
In [2]: def f(x,p):
            return x^(p-2)

        L = [[f(x,p) for x in GF(p)] for p in P]
```

3.3 Plot da função aplicada a cada um dos corpos finitos

Esta secção tem como objetivo fazer o **plot** dos pontos resultantes da aplicação da função definida anteriormente como `f(x,p)` a cada um dos corpos finitos primos, sendo que o resultado deverá ser apresentado como um gráfico por cada lista de pontos do corpo finito respetivo.

```
In [3]: # Plot dos pontos resultantes da aplicação da função aos pontos do
In [3]: # corpo finito primo GF(37)
        list_plot(L[0])
```

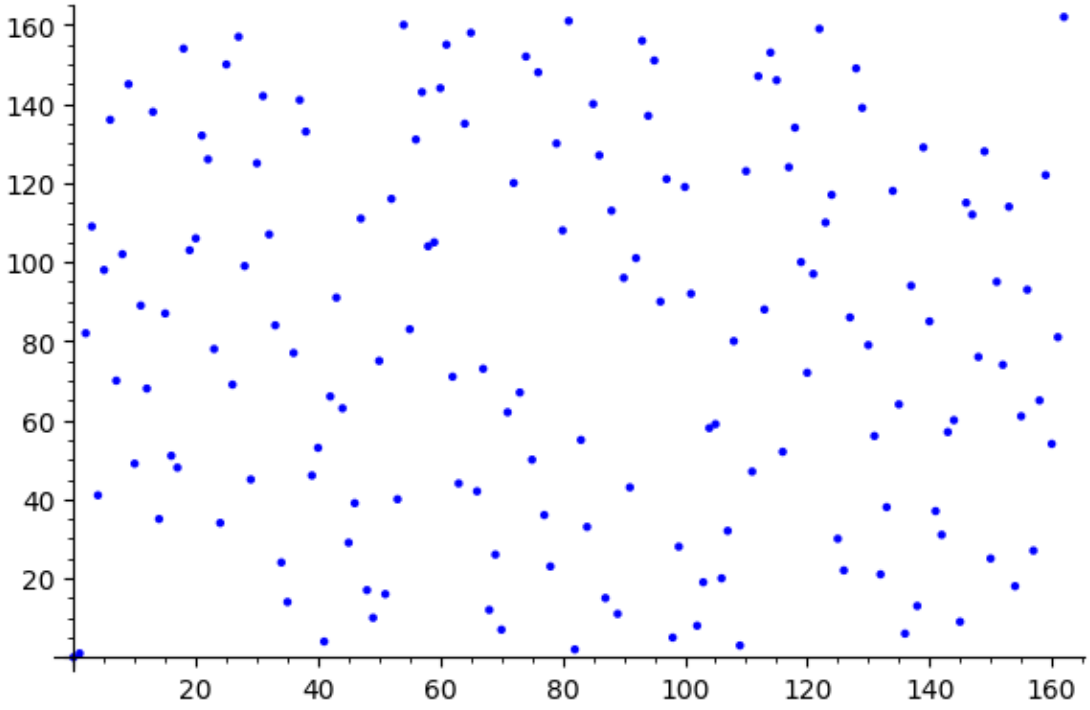
```
Out[3]:
```



In [4]: *# Plot dos pontos resultantes da aplicação da função aos pontos do*
In [4]: *# corpo finito primo GF(163)*

```
list_plot(L[1])
```

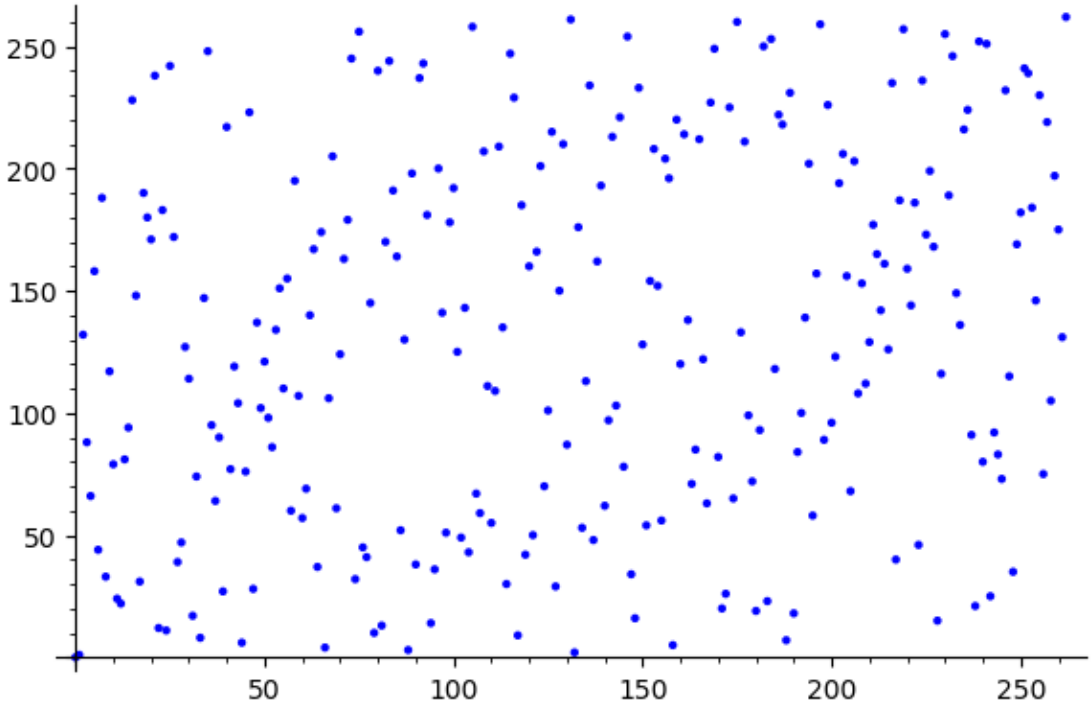
Out[4]:



In [5]: *# Plot dos pontos resultantes da aplicação da função aos pontos do*
In [5]: *# corpo finito primo GF(263)*

```
list_plot(L[2])
```

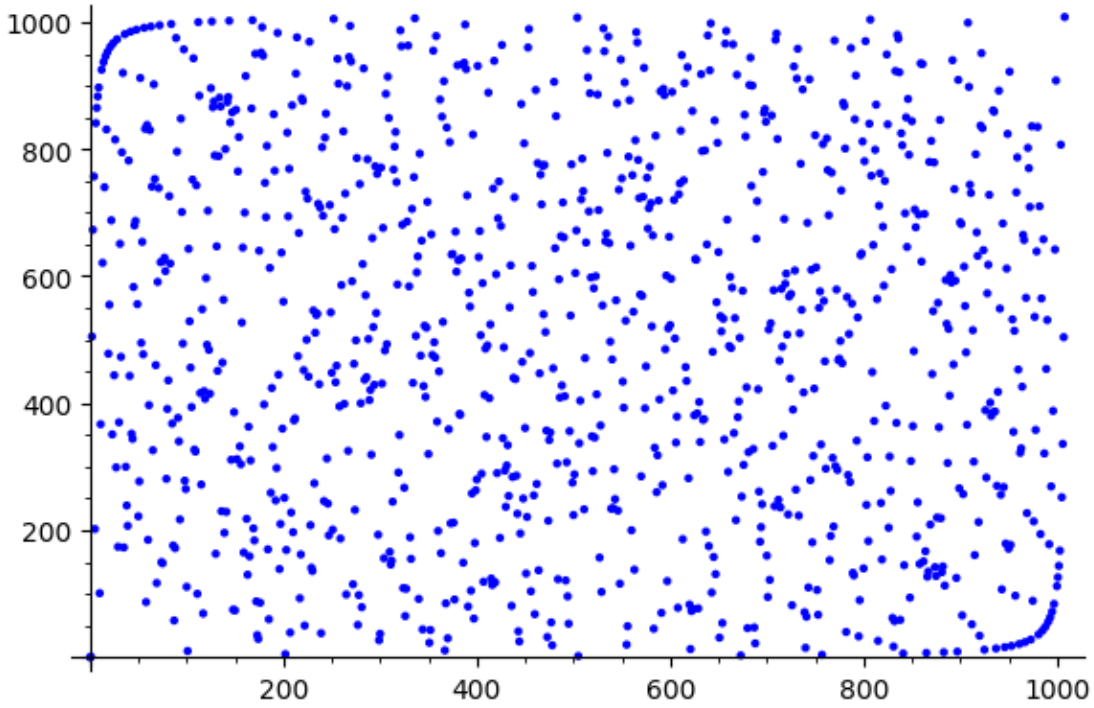
Out[5]:



In [6]: *# Plot dos pontos resultantes da aplicação da função aos pontos do*
In [6]: *# corpo finito primo GF(1009)*

```
list_plot(L[3])
```

Out[6]:



A partir da análise dos gráficos previamente gerados podemos verificar que, em todos eles, à primeira vista, parece existir uma certa aleatoriedade nos pontos desenhados mas, analisando melhor cada um deles, podemos confirmar a existência de um padrão simétrico que existe no gráfico.

3.4 Funções da proposição e auxiliares

Resta agora definir duas funções que são utilizadas para garantir a proposição na totalidade. A função `checker(n,g,p)` testa, com o auxílio das funções anteriormente referidas, a veracidade da proposição para um dado expoente, elemento primitivo e número primo, ao tentar encontrar, por amostragem, elementos que provem que a proposição é incorrecta, seguindo o seguinte algoritmo:

- Se $g^n = 1$:
  - Se $n = 0 mod(p-1)$, a proposição verifica-se, pelo que o resultado retornado é 0 (não existe erro a ser somado).
  - Se $n! = 0 mod(p-1)$, a proposição falha, pelo que o resultado retornado é 1 (proposição é falsa para estes elementos, erro deve ser somado).
- Se $g^n! = 1$, a proposição não é testada.

```
In [7]: def prop1(g,n):
            return g^n

        def prop2(p):
            return Mod(0,p-1)
```

```
        def checker(n,g,p):
            if(prop1(g,n) == 1):
                if(prop2(p) == n):
                    return 0
                else: return 1
            else: return 0
```

3.5 Criação da lista de expoentes aleatórios

Por último criou-se uma lista de expoentes, por amostragem, para que seja averiguada a veracidade da proposição enunciada. São criados 100000 expoentes aleatórios que variam entre 1 e 1 bilião.

```
In [8]: import numpy
        exponents_list = [numpy.random.randint(1,1000000000000) for i in xrange(100000)]
```

3.6 Prova da proposição por amostragem

Nesta secção pretendemos, com o auxílio da função `checker(n,g,p)` definida anteriormente, verificar, para todos os corpos finitos primos criados, que a proposição enunciada se verifica, seguindo o seguinte algoritmo:

- Para todo o número primo p em P, onde P é a lista dos números primos utilizados para criar os corpos finitos:
  - Para todo o expoente n na lista de expoentes previamente calculada:
    - Calcular o resultado de `checker(n,GF(p).primitive_element(),p)`, onde o segundo argumento é o elemento primitivo do corpo finito do primo p.
    - Se a proposição se verificar falsa, a variável *checkFalses* será incrementada de uma unidade, caso contrário será incrementada de 0 (sem falsos).
  - Imprimir o número primo e o número de falsos encontrados na aplicação da proposição ao mesmo.

```
In [9]: checkFalses = 0
        for p in P:
            for n in exponents_list:
                checkFalses += checker(n,GF(p).primitive_element(),p)
            print('Primo:')
            print(p)
            print('Nº de erros na proposição:')
            print(checkFalses)
            print('\n')
            checkFalses = 0

Primo:
37
Nº de erros na proposição:
0
```

```
Primo:
163
Nº de erros na proposição:
0


Primo:
263
Nº de erros na proposição:
0


Primo:
1009
Nº de erros na proposição:
0
```

Como é possível verificar pelo output produzido, a proposição enunciada verifica-se para todos os expoentes gerados por amostragem e para todos os corpos finitos primos criados.

4 Conclusão

Os resultados da resolução deste trabalho prático são, na nossa opinião, bastante satisfatórios, tendo em conta que os mesmos são os que eram esperados. O desenvolvimento das aplicações foi feito de um modo gradual (texto de explicação pelo entre o código), de forma a tornar a leitura e compreensão do trabalho mais agradável.

As maiores dificuldades que surgiram durante a resolução deste trabalho prático resumiram-se, na sua maior parte, aos aspetos que dizem respeito ao desenvolvimento utilizando **sagemath**, visto que este foi o primeiro contacto do grupo com esta tecnologia. Além disso, este é também o primeiro relatório que o grupo produz neste formato, ou seja, utilizando apenas o **jupyter** para a produção do código e do próprio texto do mesmo, pelo que tentámos familiarizar-nos apenas com a ferramenta, especificamente, na produção do ficheiro necessário, que representa o relatório na sua totalidade.

4.1 Referências

1. Worksheets do TP0 fornecidas pelo professor
2. Cryptography - Symmetric encryption
3. Cryptography - Message Authentication Codes
4. Cryptography - Key Derivation Functions
5. SageMath - Finite Prime Fields
6. SageMath - 2D Plotting
7. SageMath - Base Classes for Finite Fields