Tutorial #5

João Freitas up202100373
Rui Gonçalves up202103077

O presente relatório tem como objetivo descrever o processo seguido para a resolução da ficha `Tutorial #5`, disponibilizada no âmbito da disciplina de Criptografia Aplicada. As seções numeradas em baixo representam cada um dos exercícios resolvidos.

Poly1305, AES-GCM e Complexidade Computacional

1) Implemente a universal hash function do Poly1305 no Sage

Sabendo que o a função do Poly1035 se descreve como:

```
H((K1, K2), (M1, M2, ..., Mn)) = K1 + P(K2)
P(X) = K1 + M1*X + M2*X^2 + M3*X^3 + ... + Mn*X^n
```

E que o valor primo *p* usado na função tem o valor `2^130 - 5`, então este pode ser representado no Sage como:

```
F = FiniteField(2**(130-5))

PR.<X> = PolynomialRing(F)

K = (F.random_element(), F.random_element())
M = [F.random_element() for x in range(5)]

def uhf(K, M):
    K1 = K[0]
    K2 = K[1]
    X = K2

    return K1 + PR(M)
```

Dado que a construção Poly1305 é uma construção MAC, então isso signfica que existem colisões após `2^n/2` hashes geradas, ou seja `2^64`, dado que o tamanho dos blocos em Poly1305 é 128 bits, o que indica uma probabilidade de colisão de `1/2^64`.

2) Utilize o Python para cifrar um ficheiro com AES-GCM, garantido que o mesmo pode ser decifrado usando o OpenSSL e que uma vez alterado, não pode ser decifrado de novo

Usando a biblioteca `cryptography` do Python, facilmente implementa-se a cifração de uma mensagem usando a cifra de bloco AES-GCM:

```
from cryptography.hazmat.primitives.ciphers.aead import AESGCM

def encrypt_aes_gcm(input: bytes, key: bytes, aead: bytes, nonce: bytes):
    return AESGCM(key).encrypt(nonce, input, aead)

def decrypt_aes_gcm(input: bytes, key: bytes, aead: bytes, nonce: bytes):
    return AESGCM(key).decrypt(nonce, input, aead)

def pad(m):
    return m+(chr(16-len(m) % 16)*(16-len(m) % 16)).encode()

block_size = 128
key = os.urandom(16)
aead = os.urandom(16)
nonce = os.urandom(12)

Path('input-key').write_bytes(key)
Path('input-iv').write_bytes(nonce)
Path('input-tag').write_bytes(aead)
```

```
input = Path(sys.argv[1]).read_bytes()
cipher_input = encrypt_aes_gcm(pad(input), key, aead, nonce)
output = Path(sys.argv[2]).write_bytes(cipher_input)
```

Se executarmos o script Python, seguido do comando que recorre à ferramente `aesgcm` para decifrar o ficheiro produzido, podemos verificar que o mesmo é decifrado:

```
python3 aes_gcm.py <input_path> <output_path>
./aesgcm dec -key input-key -iv input-iv -in <input_path> -out <output_path> -tag input-tag
```

Contudo, se modificarmos o ficheiro produzido e tentarmos decifrar o mesmo, verificamos que não é possível, visto que o ficheiro é autenticado na cifração. Se substituírmos a cifra utilizada pelo `AES-CTR`, é possível decifrar o ficheiro, apesar de este se apresentar modificado, pois o bloco modificado não pode ser recuperado.

3) Complexidade Computacional usando a Notação Big O

No presente exercício é pedido que um conjunto de comparações usando a notação Big O sejam confirmadas. Estas são:

```
1) 2n = O(n) // V
2) n^2 = O(n) // F
3) n^2 = O(nlog(n)) // F
4) nlog(n) = O(n^2) // V
5) 3^n = O(2^n) // F
6) O(2^(n^2)) = 2^O(n^2) // F
```

A afirmação 1) é verdadeira, pois a notação Big O especifica que funções com crescimento semelhante (n - linear), mas com constantes diferentes (2), tem a mesma complexidade dada pelo crescimento da função (n). No mesmo sentido, a afirmação 2), 3) são falsas, pois o crescimento de uma função exponêncial (n^2) é diferente do crescimento de uma função linear (n) e logarítmica (nlog(n)).

A afirmação 4) é verdadeira, pois apesar do crescimento de `n^2` ser maior que `nlog(n)`, como ambas não estão expressas com a notação Big O, esta afirmação não pode ser confirmada. Como nlog(n) "instruções cabem" dentro da função `n^2`, podemos considerar esta afirmação como verdadeira.

A afirmação 5) é falsa, pois tal como referido na 4), `3^n` cresce mais rápido do que `2^n`. Para finalizar, a afirmação 6) também é falsa pois `2^O(n^2)` tem um crescimento superior.