Tutorial #2

João Freitas up202100373
Rui Gonçalves up202103077

O presente relatório tem como objetivo descrever o processo seguido para a resolução da ficha `Tutorial #2`, disponibilizada no âmbito da disciplina de Criptografia Aplicada. As seções numeradas em baixo representam cada um dos exercícios resolvidos.

Entropia e Aleatoriedade

1) Considere o conjunto de números entre o intervalo [0..250] e o número primo `p=251`. Produza um valor *b* aleatório uniforme (8-bit) e calcule o seu módulo (b (mod(p))). Repita o mesmo para o valor *w* (64-bit)

Calcule a probabilidade destes valores pertencerem ao intervalo `[0..250]` e comente se as distribuições são uniformes:

Utilizando o Python, definimos a função `distribuicao`, que permite calcular a probabilidade de cada valor no intervalo [0... 250] ocorrer, calculando o valor *resto*.

```
def distribuicao(valor_max,p):
    res = {}

    q = valor_max // p  #quociente
    r = valor_max % p   #resto

    dif = 1/valor_max

    if q > 0:
        for i in range(p):
            res[i] = q * dif

    if r >= 0:
        i=0
        while r >= 0:
            try:
                res[i] += dif
            except:
                res[i] = dif
            i += 1
            r -= 1
    return res

res = distribuicao(2**8 -1, 251)
print(res)
```

Depois de executar a função anterior para o valor *b*, percebemos que a distribuição não é uniforme (i.e., há valores que ocorrem com maior probabilidade do que outros).

No caso do valor *w*, o comportamento mantém-se.

```
res = distribuicao(2**64 -1, 251)
print(res)
```

2) Repita o exercício anterior para `p=2^8`

Sendo *p* uma potência de 2, então a distribuição é uniforme.

```
res = distribuicao(2**8 -1, 2**8)
print(res)
```

3) Use o Sage para calcular a entropia das duas distribuições do exercício 1)

Traduzindo a fórmula de calculo de entropia representado no ficheiro, obtemos a seguinte função:

```
def entropia(d: dict):
    res = 0
    for (_,b) in d.items():
        res += (-1) * b * float(log(b,2))
    return res
entropia_1 = entropia(distribuicao(2**8 -1,251))
entropia_2 = entropia(distribuicao(2**64 -1,251))

print(entropia_1)
print(entropia_2)
```

5) Explique o seguinte comando faz: `hexdump -n 32 -e '1/4 "%0X" 1 "\n"' /dev/urandom`

O comando `hexdump` transforma um dado input (string, file) num formato específico. A opção `-n` permite especificar o tamanho do input a ser considerado (i.e., primeiros *n* bytes do input) e a opção `-e` o formato do output. Em suma, o comando usa como input o ficheiro `/dev/urandom`, que contem bitstrings produzidas pelo PRG do sistema operativo, e transforma os primeiros 32 bytes no formato hexadecimal.

Alternativamente podemos usar o comando `dd` para obter um resultado semelhante e escrever o mesmo num ficheiro:

```
dd if=/dev/urandom of=<output_file> count=<bytes_count> openssl rand <bytes_count> -out <output_file>
```

6) Utilize o seguinte comando OpenSSL para produzir um par chave, onde a chave privada esta protegida com uma password: `openssl genrsa -aes128 4096`

> O que acontece quando se aumenta/diminui o tamanho da chave?

Se aumentarmos/diminuirmos o tamanho da chave, a produção da mesma demora mais/menos. Se esta for menor que 128 bits, a produção falha.

> Investiga como o openSSL converte a password em uma chave criptográfica para cifração

Para o seguinte comando, o OpenSSL utiliza a password como a chave secreta para o algoritmo de cifra AES.

7) Utilize o seguinte comando OpenSSL para produzir parâmetros aleatórios Diffie-Hellman: `openssl gendh -aes128 2048`

> O que acontece quando se aumenta/diminui o tamanho da chave?

Se aumentarmos/diminuirmos o tamanho da chave, a produção dos parâmetros demora mais/menos. Se esta for menor que 128 bits, a produção falha. O operação demora muito mais do que a primeira, pois nesta é necessário encontrar um valor primo com o tamanho de bits especificado, e que seja um primo seguro (i.e., safe prime = p=primo e (p-1) / 2 = primo).