Laboratórios de Informática 2014/2015

DETI - Universidade de Aveiro

Docentes
João Paulo Barraca <jpbarraca@ua.pt>
Diogo Gomes <dgomes@ua.pt>
João Manuel Rodrigues <jmr@ua.pt>
Mário Antunes <mario.antunes@ua.pt>

TEMA 13

Criptografia em Python

Objetivos:

- Módulos e bibliotecas Python para lidar com criptografia
- Cálculo de sínteses de ficheiros
- Cálculo de sínteses de senhas
- Cifra de ficheiros com criptografia simétrica e assimétrica

13.1 Introdução

A criptografia é um conceito antigo de transformação de conteúdos, que até à cerca de duas décadas era sobretudo usado em ambientes onde a segurança é um elemento fundamental (ambientes militares, serviços de informações, etc.). Hoje em dia, em virtude da massificação do uso da informática e da Internet para os mais variados fins, a segurança criptográfica faz parte do dia-a-dia do cidadão comum, mesmo que disso ele não se aperceba (por exemplo, comunicações seguras usando HTTPS).

O objetivo de desta aula é o de mostrar como se conseguem realizar as transformações criptográficas mais comuns usando módulos Python.

13.2 Funções de síntese

As funções de síntese (*digest*) não são funções de cifra convencionais, como as que veremos nas demais secções, mas são funções que usam princípios relacionados com a criptografia no seu modelo de operação. Para além disso, são muitas vezes usadas em conjugação com funções de cifra para as completar de alguma forma (por exemplo, para calcular chaves de cifra de dimensão fixa a partir de senhas de dimensão arbitrária).

O objetivo de uma função de síntese é o que calcular um valor de dimensão fixa (em número de bits) a partir de conteúdos constituídos por conjunto arbitrários de bits. Normalmente diz-se que estas funções permitem criar *impressões digitais informáticas* (*digital fingerprints*) de conteúdos, porque é muito difícil (por requisito) encontrar dois conteúdos que tenham a mesma síntese, assim como é difícil encontrar dois humanos com as mesmas impressões digitais. Também é comum designar o valor calculado por estas funções como *soma de controlo* (*checksum*), muito embora existam inúmeras funções de cálculo de somas de controlo que não possuem as propriedades (criptográficas) das funções de síntese[1].

As funções mais usadas são a MD5, a SHA-1, SHA-256 e SHA-512, mas há muitas mais. A sua exploração é similar, mas produzem resultados de dimensão diferente (128, 160, 256 e 512 bits, respetivamente).

A forma usual de explorar uma função de síntese num programa consiste sem seguir os 4 passos seguintes:

1. Iniciar o seu contexto interno;
2. Adicionar dados para serem processados pela função;
3. Repetir o passo anterior até que tenham acabado todos os dados do conteúdo a processar;
4. Calcular a síntese resultante de todos os dados fornecidos à função.

Em Python o módulo **`hashlib`**[2] possui as funções de síntese mais usuais. O exemplo seguinte mostra a exploração da função MD5 para calcular a síntese de uma frase longa dividida em duas partes.

```
$ python
>>> import hashlib
>>>
>>> h = hashlib.md5()
>>> h.update( "A long sentence " )
```

[1] Por outras palavras, as funções de síntese podem ser consideradas como funções de cálculo de somas de controlo, enquanto o inverso não é verdade.

[2] `https://docs.python.org/2/library/hashlib.html`

```
>>> h.update( "broken in two halves" )
>>> print h.hexdigest()
f2b8308b3e84e032f5d7e6dee84e647a
>>>
```

O resultado apresentado por este programa é uma sequência de 32 algarismos hexadecimais (algarismos decimais e letras de A a F). Cada um desses algarismos representa 4 bits (0 → 0000, 1 → 0001, 2 → 0010, ···, 8 → 1000, 9 → 1001, a→ 1010, ···, d→ 1101, e→ 1110, f→ 1111). Logo, o resultado possui 32×4 bits, ou seja, 128 bits.

O resultado apresentado seria o mesmo se a frase tivesse sido fornecida apenas de uma vez e não dividida em duas metades:

```
$ python
>>> import hashlib
>>>
>>> h = hashlib.md5()
>>> h.update( "A long sentence broken in two halves" )
>>> print h.hexdigest()
f2b8308b3e84e032f5d7e6dee84e647a
>>>
```

Porém, qualquer pequena alteração do texto processado pela função de síntese muda de forma radical o resultado:

```
$ python
>>> import hashlib
>>>
>>> h = hashlib.md5()
>>> h.update( "A long sentence" )      # trailing space removed!
>>> h.update( "broken in two halves" )
>>> print h.hexdigest()
1d0b93b21eb945593abab4b1a04456d6
>>>
```

Exercício 13.1

Faça um programa que calcule e apresente a síntese de ficheiros usando a função de síntese SHA-1 (cujo nome, no módulo **`hashlib`**, é **`sha-1`**). Os nomes dos ficheiros deverão ser indicados através da variável **`sys.argv`**. Confirme os resultados apresentados pelo seu programa confrontando-os com os apresentados pelo comando **`sha-1sum`**.

Exercício 13.2

Reescreva o programa anterior para calcular a síntese de cada ficheiro usando blocos 512 octetos de cada vez. Use, para esse fim, a função **read** para ler octetos dos ficheiros:

```
$ python
f = open(name, "r")
buffer = f.read(512)

# len(buffer) == 0 --> End-of-file reached
# len(buffer) > 0  --> buffer has len(buffer) bytes

while len(buffer) > 0:
    ...
    buffer = f.read(512)
```

Verifique se obtém os mesmos resultados que antes (não deverão mudar!).

13.3 Biblioteca pycrypto

A biblioteca (*toolkit*) **pycrypto**[3] possui uma coleção muito interessante de funcionalidades relacionadas com criptografia, incluindo funções de síntese. Daqui em diante iremos usar esta biblioteca.

13.3.1 Instalação

Método Automático: Python pip ou easy_install

O interpretador de Python tipicamente é acompanhado por duas ferramentas que auxiliam a instalação de módulos adicionais, necessários aos programas. Estes programas são respetivamente o **easy_install** e **pip**. Para esta aula será necessário instalar a biblioteca **pycrypto**, que fornece mecanismos para a utilização de métodos criptográficos.

Para instalar a biblioteca, no caso de se utilizar um sistema pessoal com Ubuntu ou Debian, será necessário executar:

```
sudo apt-get install build-essential python-dev
```

[3] https://www.dlitz.net/software/pycrypto/

depois é necessário executar:

```
pip install --user pycrypto
```

ou em alternativa,

```
easy_install --user pycrypto
```

De notar a utilização da opção **-user**, indicando que as bibliotecas adicionais deverão ser instaladas para o utilizador atual. Isto é importante pois os utilizadores comuns não possuem permissões para instalar bibliotecas no sistema.

Pode-se verificar a existência da biblioteca executando os seguintes comandos:

```
$ python
>>> from Crypto.Hash import MD5
>>>
>>> h = MD5.new()
>>> h.update( "A long sentence " )
>>> h.update( "broken in two halves" )
>>> print h.hexdigest()
f2b8308b3e84e032f5d7e6dee84e647a
>>>
```

Caso não seja possível instalar a biblioteca **pycrypto** desta forma, terá de ser utilizado o método manual, que se descreve de seguida.

Método Manual

Apenas execute este passo caso o método anterior não seja possível!

Nesta instalação vamos colocar a biblioteca **pycrypto** na diretoria onde iremos desenvolver todos os programas que a usam. Alternativamente poderia ser instalada num local onde pode ser usada por qualquer aplicação Python da máquina.

Descarregue o ficheiro com extensão **tar.gz** da versão estável da biblioteca **pycrypto** da página Web `https://www.dlitz.net/software/pycrypto`. Extraia a árvore completa dos ficheiros desse arquivo para um local qualquer (por exemplo, para a diretoria **/tmp**) e, num interpretador de comandos, avance nessa árvore de diretorias (com o comando **cd**) até chegar à diretoria onde está o ficheiro **README**.

Nessa diretoria execute o comando:

```
python ./setup.py build
```

Este comando irá criar, na diretoria **build**, uma diretoria com um nome que começa por **lib**. Nesta última encontrará a diretoria **Crypto**. Copie esta diretoria e todos os conteúdos abaixo da mesma para a diretoria onde irá desenvolver os programas Python que usam criptografia através da biblioteca **pycrypto**.

Exercício 13.3

No interpretador de comandos mude para a diretoria onde irá desenvolver os seus programas, na qual deverá existir a diretoria **Crypto** que acabou de copiar. Para verificar se está tudo instalado apropriadamente execute alguns dos comandos de autoverificação, como por exemplo:

```
python Crypto/SelfTest/Hash/test_SHA.py
python Crypto/SelfTest/Cipher/test_ARC4.py
python Crypto/SelfTest/Cipher/test_pkcs1_15.py
python Crypto/SelfTest/PublicKey/test_RSA.py
```

O programa antes indicado que usava MD5 agora escrever-se-á assim para usar a função MD5 da biblioteca **pycrypto**:

```
$ python
>>> from Crypto.Hash import MD5
>>>
>>> h = MD5.new()
>>> h.update( "A long sentence " )
>>> h.update( "broken in two halves" )
>>> print h.hexdigest()
f2b8308b3e84e032f5d7e6dee84e647a
>>>
```

Exercício 13.4

Altere o programa que antes desenvolveu com a função SHA-1 do módulo **hashlib** para usar a função SHA-256 da biblioteca **pycrypto**. Confirme os resultados confrontando-os com os produzidos pelo comando **sha256sum**.

13.4 Cifras simétricas

As cifras simétricas são funções de transformação (reversível) de conteúdos que usam duas chaves iguais na cifra e na decifra. Ou seja, se se cifrar um conteúdo original T com a função de cifra E e a chave K, produzindo o criptograma C, poder-se-á recuperar T a partir de C com a função de decifra D e a mesma chave K.

As cifras simétricas subdividem-se em duas grandes famílias: as contínuas (*stream*) e as por blocos.

13.4.1 Cifras contínuas (*stream*)

As cifras contínuas produzem um criptograma C por mistura de um conteúdo original T com uma chave contínua (*keystream*) KS. A decifra consiste em retirar do criptograma a componente KS que lhe foi misturada usando uma função inversa da de mistura. Por simplicidade, a função de mistura e a sua inversa são exatamente a mesma: a adição módulo 2 de bits, vulgarmente designada por XOR (de *eXclusive OR*, cujo símbolo matemático é $\oplus$).

Se A e B forem bits, que podem tomar os valores 0 e 1, a operação $\oplus$ é a seguinte:

A	B	A $\oplus$ B
0	0	0
0	1	1
1	0	1
1	1	0

É fácil de ver que sempre que um operando é 1, o resultado é o inverso do outro operando; pelo contrário, sempre que um operando é 0, o resultado é o valor exato do outro operando. Daqui resulta a seguinte propriedade, para qualquer valor de X:

$$\mathrm{A} \oplus \mathrm{X} \oplus \mathrm{X} = \mathrm{A}$$

Logo, é fácil de mostrar que a cifra e decifra se podem fazer usando a operação XOR e a mesma chave contínua KS:

$$\begin{aligned} \mathrm{C} &= \mathrm{T} \oplus \mathrm{KS} \\ \mathrm{T} &= \mathrm{C} \oplus \mathrm{KS} = \mathrm{T} \oplus \mathrm{KS} \oplus \mathrm{KS} = \mathrm{T} \end{aligned}$$

A diferença entre as várias cifras contínuas está na forma como é produzido o valor de KS. A grande maioria das cifras contínuas usa um gerador pseudoaleatório, controlado por uma chave de dimensão fixa, para gerar KS (tanto para a operação de cifra como para a de decifra).

As cifras contínuas são muito usadas em comunicações rádio envolvendo equipamentos móveis, porque existem geradores muito simples de realizar em hardware e de baixo consumo. É o caso dos geradores A5 (usado nas comunicações GSM) e RC4 (usado

inicialmente nas comunicações WiFi). Neste trabalho iremos usar esta última. O seu nome, porém, é diferente: ARC4 (de *Alleged ARC4*). A razão de ser desta diferença de nomes vem do facto do algoritmo RC4 ainda ser oficialmente secreto, sendo o ARC4 uma sua versão que alegadamente (e comprovadamente) é compatível com o RC4 (e que, por isso, é obviamente igual).

A forma usual de explorar uma função de cifra/decifra num programa consiste sem seguir os 3 passos seguintes:

1. Iniciar o seu contexto interno, normalmente indicando uma chave;
2. Adicionar dados para serem processados pela função e recolher o resultado da sua operação;
3. Repetir o passo anterior até que tenham acabado todos os dados do conteúdo a processar.

Note-se que com uma cifra contínua o processamento dos dados (na cifra ou na decifra) é por omissão feito sequencialmente, não se podendo processar zonas arbitrárias dos dados a cifrar ou decifrar por qualquer ordem. Há cifras contínuas que permitem essa liberdade, mas a sua utilização é diferente. Neste guião não vamos explorar essa faceta.

Exercício 13.5

Faça um programa em Python (**cifraComRC4.py**) que use a cifra RC4 (ou ARC4, na biblioteca **pycrypto**). O programa deverá receber como argumentos o nome de um ficheiro a cifrar e uma chave textual.

O RC4 suporta chaves com uma dimensão entre 40 e 2048 bits (5 a 256 octetos), pelo que deverão ser tomados cuidados para adaptar a chave fornecida pelo utilizador a algo que seja aceitável. Sugere-se a seguinte política: se a chave tiver menos do que 5 octetos (letras), deverá ser usada uma síntese da mesma (calculada, por exemplo, com SHA-1). Se tiver mais dos que 256 octetos, deverão ser usados apenas os primeiros 256. Caso contrário, deverão ser usados exatamente os octetos fornecidos.

O programa deverá escrever o criptograma para o **stdout** (por omissão, a consola), o qual poderá ser redirigido para um ficheiro usando os mecanismos do interpretador de comandos:

```
python cifraComRC4.py ficheiro chave > criptograma
```

Em Python a escrita para o stdout pode ser feita da seguinte maneira:

```
import sys
from Crypto.Cipher import ARC4

cipher = ARC4.new("chave")
decipher = ARC4.new("chave")

cryptogram = cipher.encrypt("Text")
sys.stdout.write( result )
decrypted = decipher.decrypt(cryptogram)
sys.stdout.write( decrypted )
```

Exercício 13.6

Cifre vários ficheiros executando várias vezes o seu programa e verifique se o comprimento dos ficheiros resultantes, contendo os criptogramas, têm a mesma dimensão dos originais (têm de ter!).

Exercício 13.7

Verifique que o programa está a funcionar corretamente decifrando o criptograma usando novamente a mesma chave:

```
python cifraComRC4.py criptograma chave
```

Após executar este comando deverá surgir, listado no ecrã, o conteúdo do ficheiro original (**`ficheiro`**).

Exercício 13.8

Execute o comando anterior usando uma chave diferente e veja o resultado. Explique o sucedido.

13.4.2 Cifras por blocos

As cifras por blocos consideram que os dados a transformar (e o resultado da sua transformação) são constituídos por blocos contíguos de dimensão constante; esta dimensão é imposta pela cifra. O modo mais simples de explorar uma cifra (ou decifra) por blocos, denominado EBC (*Electronic Code Book*) consiste em realizar os seguintes passos:

1. Iniciar o seu contexto interno, normalmente indicando uma chave;
2. Selecionar o primeiro bloco dos dados a transformar para serem processados pela função e recolher o bloco resultante da sua operação;
3. Repetir o passo anterior para os blocos seguintes até que tenham acabado todos os dados do conteúdo a processar.

Este processo implica que a dimensão total dos dados a transformar seja múltipla da dimensão do bloco (diz-se que estão alinhados ao bloco). Porém, é natural que nem sempre assim aconteça, o que implica que se tenha de forçar esse alinhamento acrescentando dados extra, denominados excipiente (*padding*). Estes dados extra são fornecidos na cifra e removidos na decifra. Há várias maneiras de lidar com os excipientes, mas independentemente do método usado, é preciso indicar ao decifrador a sua presença

e dimensão. Uma forma padrão de o fazer, denominada PKCS #7 [1], consiste em fazer o seguinte:

- Acrescentar sempre excipiente, mesmo quando à partida não é necessário (por os dados a cifrar já estão alinhados);
- Cada octeto do alinhamento tem um valor igual ao comprimento desse alinhamento.

Neste exercício vamos usar a cifra por blocos que é atualmente o padrão, denominada AES (*Advanced Encryption Standard*), que foi a vencedora de um concurso de cifras que terminou pouco depois do início do atual milénio. Esta cifra processa blocos de 128 octetos usando para o efeito chaves de 128, 192 ou 256 octetos. quando se usa a biblioteca **pycrypto**, a dimensão do bloco de uma cifra por blocos pode ser obtida através da variável **blocksize** de um objeto de cifra:

```
$ python
>>> from Crypto.Cipher import AES
>>>
>>> key = '1234567890abcdef'     # Must provide a valid key (with 16, 24 or 32 bytes)
>>> cipher = AES.new( key )
>>> print cipher.block_size      # Prints the number of ytes in each block
16A
>>> print cipher.mode            # Prints the cipher mode (1 for ECB)
1
>>>
```

Para se cifrar ou decifrar dados devem-se usar os métodos **encrypt** ou **decrypt**, respetivamente, do objeto de cifra:

```
$ python
>>> from Crypto.Cipher import AES
>>>
>>> key = '1234567890abcdef'
>>> cipher = AES.new( key )
>>> x = cipher.encrypt( "texto pra cifrar" )
>>> print cipher.decrypt( x )
texto pra cifrar
>>>
```

Exercício 13.9

Faça um programa em Python (`cifraComAES.py`) que use a cifra AES. O programa deverá receber como argumentos o nome de um ficheiro a cifrar e uma chave textual.

O AES suporta chaves com uma dimensão exata de 16, 24 ou 32 octetos, pelo que deverão ser tomados cuidados para adaptar a chave fornecida pelo utilizador a algo que seja aceitável. Sugere-se a seguinte política: se a chave tiver menos do que 16 octetos (letras), deverá ser usada uma síntese da mesma (calculada, por exemplo, com SHA-1), de cujo resultado serão usados apenas os 16 primeiros octetos. Caso contrário, deverão ser usados apenas os primeiros 16 octetos da senha fornecida.

O programa deverá escrever o criptograma para o `stdout` (por omissão, a consola), o qual poderá ser redirigido para um ficheiro usando os mecanismos do interpretador de comandos

Exercício 13.10

Faça o programa correspondente de decifra (`decifraComAES.py`). Note que o programa será fundamentalmente igual ao de decifra mas deverá ter os seguintes cuidados:

- Porque os criptogramas estão necessariamente alinhados, não deverá aceitar fazer a decifra de ficheiros que não tenham um comprimento alinhado à dimensão do bloco de cifra do AES. A dimensão de um ficheiro pode ser obtida com a função `os.path.getsize(nome_do_ficheiro)`.
- Não se esqueça de retirar (não escrever) o excipiente colocado durante a cifra. Não se esqueça de que, se usou o método de colocação de excipiente descrito, existe sempre excipiente no último bloco do ficheiro cifrado!

13.5 Cifras Assimétricas

As cifras assimétricas são cifras que usam duas chaves, uma para cifrar e outra para decifrar (designadas por par de chaves). Uma destas chaves designa-se por privada e a outra por pública. A privada só é conhecida por uma entidade, que é dona do respetivo

par de chaves; a pública pode ser universalmente conhecida. O conhecimento da chave pública não permite a dedução da correspondente chave privada. Estas cifras também são por vezes designadas por *cifras de chave pública*.

As cifras assimétricas, ou de chave pública, são historicamente muito recentes. Enquanto as cifras simétricas são tão antigas quanto a própria escrita, as assimétricas só existem desde meados de década de 1970 do século passado. Neste exercício iremos usar a primeira cifra assimétrica que foi publicada, denominada RSA, que é atualmente a mais usada.

13.5.1 RSA

Como se disse, as cifras assimétricas usam pares de chaves, uma privada e outra pública. O RSA permite cifrar com a pública e decifrar com a privada (para obter confidencialidade) o inverso, cifrar com a privada e decifrar com a pública (para obter autenticidade, sendo o conceito que está na base das assinaturas digitais).

O RSA opera através da realização de operações matemáticas com números inteiros de grande dimensão (centenas ou milhares de bits). As operações são a exponenciação e o resto da divisão por um número inteiro. À combinação destas duas operações dá-se o nome de exponenciação modular.

Um par de chaves RSA possui 3 elementos:

- Um módulo, n, comum às componentes privada e pública;
- Um expoente, d, pertencente à componente privada;
- Um expoente, e, pertencente à componente pública.

Assim, a chave privada é formada pelo par de valores (d, n), enquanto a chave pública é formada pelo par de valores (e, n). As operações de transformação de dados usando estas chaves são as seguintes:

$C = T^e \mod n$	Cifra com a chave pública (confidencialidade)
$T = C^d \mod n$	Decifra com a chave privada
$C = T^d \mod n$	Cifra com a chave privada (autenticidade)
$T = C^e \mod n$	Decifra com a chave pública

onde a expressão $x \mod n$ representa o resto da divisão inteira de x por n (em Python seria calculado com a expressão `x % n`).

Ao contrário das cifras anteriores, as cifras assimétricas não usam chaves indicadas por uma pessoa, nem uma pessoa é capaz de memorizar um par de chaves. Os pares de chaves são gerados por programas, usando para o efeito geradores aleatórios de bits, e

as chaves geradas por esses programas têm de ser guardadas algures (por exemplo, em ficheiros) para poderem ser usadas mais tarde.

```
>>> from Crypto.PublicKey import RSA
>>>
>>> keypair = RSA.generate( 1024 )
>>> f = open( "keypair.pem", "w" )
>>> f.write(keypair.exportKey( "PEM", "senha" ))
>>> f.close()
...
>>> f = open( "keypair.pem", "r" )
>>> keypair = RSA.importKey( f.read(), "senha" )
```

O exemplo acima mostra como se gera um par de chaves RSA com um módulo de 1024 bits e se guarda o mesmo num ficheiro (**keypair.pem**) codificando o seu conteúdo em PEM (um formato textual) e protegendo a chave privada da observação de terceiros através da cifra com a senha **senha**. O conteúdo do ficheiro terá alguma semelhança com o seguinte:

```
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: DES-EDE3-CBC,A8751314737B2A42

5JbCg5KVtxYUfheGJBVS1lG7VkFf9OMO+Q/1FgKRluV5DJWRfw9IzYgylHBX8VL8
JdmqHo9HRmxdvLUZWSg3nnOUpVaRjiRzFjC+rxtHEWto9o8izmxBhozecNlMWGEe
Kmt6B2+rrm/KFSrDonuz3r+GiitfzGRM2nT+09DIaYQQsS+L7ZZBk/nZ5hLBo82S
SWTEQZtrUEQho8o872GVANeA9M2A5urjdPKdav0Dw/CS8a/pD7s96cDEwF0V2Kyw
eG+TMeDDSG7SR+uzQGuEWPzuxciRmwxlmB8C4pXdWArhqu+/2feTt1uDH+PeBQS1
J9/WreLJVwNleo/ZXKhVbp+zL9+IT08b72NmrKlFsFOzk6heh9GIEFJNL6oGVRn8
eD2b1bs6KWVp0XoZrdMdLPDmbjbfaAn+RYl5RvgnOm2Jfs8g+iD1UeLhhaO5VTIq
3fYOL+QU3scaRUMvkyNbbVZ9/6Tj7GNgWZRMFN5oiKhEtkH9Hsa6esSmAoLmYUol
vYhs1eRklLnhfiMlg8itfEaX1RKTi5BNe7GxgW63picPf9ryeKnITifpJS+G19+j
3uYOMqCohJqJwk+smYT/QSua7hRXjHLqwZAubhQ4iAVislWMBUVOFtYbF4K6Hub1
jQa/Mgf9hh1jynxoh0RUvAv+OkuNSICkQcyB2LGf4iKcHbj/5lf5xhpc1NoZVYCw
fARDGXFFHFnURBJ12XjKv8TIDuRFPeA6UxaKhEeD8QDWHc6GGQ7qApMpj8f6ODbk
cPfLAheh9yx7i+xU9rAfHeyu43rHaFf4m3XonstPsNxtQROp586SAQ==
-----END RSA PRIVATE KEY-----
```

Exercício 13.11

Faça um programa em Python que gere um par de chaves e que o guarde num ficheiro. O programa deverá ter como parâmetros o nome do ficheiro para o par de chaves e o número de bits da chave.

A cifra RSA não é usada diretamente tal como indicado anteriormente nas expressões matemáticas. Com efeito, muito embora se usem as operações de exponenciação modular referidas, os valores que as mesmas processam na operações de cifra são pré-processados para obter algumas funcionalidades adicionais (controlo de erros e randomização). Há duas formas fundamentais de fazer esse pré-processamento, designadas por PKCS #1 v1.5 e PKCS #1 OAEP (*Optimal Asymmetric Encryption Padding*). Neste guião vamos considerar apenas esta última.

```
>>> from Crypto.PublicKey import RSA
>>> from Crypto.Cipher import PKCS1_OAEP
>>>
>>> f = open( "keypair.pem", "r" )
>>> keypair = RSA.importKey( f.read(), "senha" )
>>> cipher = PKCS1_OAEP.new( keypair )
>>> # Encryption w/ public key
>>> x = cipher.encrypt( "The quick brown fox jumps over the lazy dog" )
>>> # Decryption with private key
>>> print cipher.decrypt( x )
The quick brown fox jumps over the lazy dog
>>>
```

Exercício 13.12

Altere os programas que usam AES para usarem também o RSA. O objetivo genérico é fazer a cifra do ficheiro com uma chave pública, recorrendo à privada para fazer a sua decifra. Na prática, vai-se recorrer à chamada cifra mista que tem uma resultado semelhante mas um custo muito menor em termos de desempenho:

- Gera-se uma chave simétrica aleatória para cifrar os dados do ficheiro;
- Cifra-se a chave simétrica com a chave pública do destinatário e acrescenta-se o resultado ao ficheiro cifrado.

Para gerar a chave simétrica aleatória use a função **`os.urandom()`** com um parâmetro que indique o número de octetos aleatórios desejados. Use a senha fornecida pelo utilizador para decifrar a chave privada do par de chaves RSA.

Glossário

AES Advanced Encryption Standard, cifra simétrica por blocos

MD5 Message Digest 5

OAEP Optimal Asymmetric Encryption Padding

RSA Cifra assimétrica, acrónimo dos nomes dos criadores (Rivest, Shamir, Adleman)

SHA-1 Secure Hashing Algorithm (versão 1)

SHA-256 Secure Hashing Algorithm (versão 2 com resultado de 256 bits)

SHA-512 Secure Hashing Algorithm (versão 2 com resultado de 512 bits)

Referências

[1] R. Housley, *Cryptographic Message Syntax (CMS)*, RFC 5652 (Standard), Internet Engineering Task Force, set. de 2009.