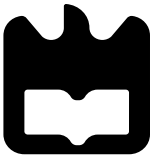
Criptografia

Laboratórios de Informática

Universidade de Aveiro

Nelson Costa 42983
Ricardo Jesus 76613

15 de Março de 2015



Criptografia

Departamento de Electrónica, Telecomunicações e Informática

UNIVERSIDADE DE AVEIRO

Nelson Costa 42983, Ricardo Jesus 76613
nelson.costa@ua.pt, ricardojesus@ua.pt

15 de Março de 2015

Resumo

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

Capítulo 2

Criptografia

2.1 Criptografia Simétrica

A cifra simétrica é a forma mais básica de criptografia na qual é partilhada uma chave entre o emissor e o recetor da mensagem, geralmente designada por chave simétrica, e usada por ambos para cifrar e decifrar mensagens. Além de possuírem a mesma chave, os intervenientes também devem possuir o mesmo algoritmo, neste caso designado por algoritmo simétrico. A figura (?) ilustra de forma simples o uso de chave partilhada na realização da criptografia simétrica.

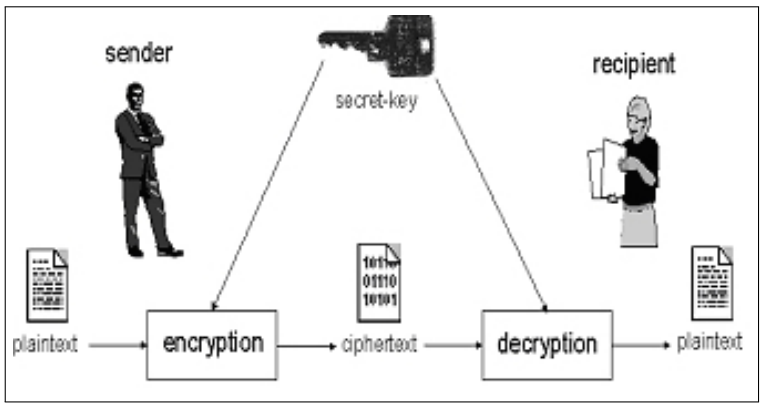


Figura 2.1: Figura (?) – Uso de chave simétrica na criptografia simétrica. O remetente (sender) cifra o texto em claro (plaintext) através da chave privada (secret -key), produzindo-se um texto cifrado (ciphertext). O destinatário (recipient) da mensagem decifra-a usando a chave partilhada por ambos. O processo contrário também funciona, ou seja, a chave tanto pode servir para cifrar como para decifrar mensagens.

A criptografia simétrica divide-se em dois tipos de cifras: contínua e por blocos.

2.1.1 Cifras Simétricas Contínuas

Na cifra contínua, a mensagem em claro é tratada como uma sequência de bytes (ou bits) onde cada sequência é cifrada sequencialmente. Usa uma operação relativamente simples para misturar a mensagem original com uma chave contínua pseudo-aleatória de dimensão finita também designada por keystream. Esta operação é tipicamente a função XOR (Exclusive- OR) visto que é uma função facilmente invertível.

A operação de cifra consiste em aplicar a função XOR para cada sequência de bytes do texto em claro juntamente com a correspondente sequência de bytes da chave contínua (as sequências também podem ser de 1 bit). A expressão (1) mostra como é feita a operação de cifra.

$$C = T \oplus Ks \tag{2.1}$$

2.2 Criptografia Híbrida

A cifragem híbrida tira partido não só da segurança extra e autenticidade que algoritmos assimétricos permitem, como também da velocidade oferecida por algoritmos simétricos. Assim, procura juntar-se o melhor de um e de outro processos. Assim sendo, o que este método geralmente permite é a cifragem da mensagem a transmitir, que poderá ser um ficheiro bastante grande e onde, portanto, torna-se muito mais proveitoso utilizar um algoritmo simétrico, segundo um metódo de cifram assimétrica. Veja-se um simples exemplo de utilização, em que um certo indivíduo A pretendo enviar uma mensagem cifrada a outro, B (segundo este metódo de cifragem híbrida):

1. A terá de obter a chave pública de B.
2. Gera-se uma chave simétrica aleatória.
3. Cifra-se a mensagem segundo um algoritmo de cifragem simétrica como AES.
4. A chave simétrica utilizada é cifrada segundo um algoritmo assimétrico (utilizando a chave pública de B)
5. Ambos a mensagem e a chave cifradas são enviados a B.

O que B terá de fazer quando receber os ficheiros, de forma a decifrar a mensagem, deverá ser algo como:

1. B utiliza a sua chave privada para decifrar a chave simétrica utilizada para cifragem da mensagem.

2. É utilizada essa chave simétrica para decifrar a mensagem enviada.

Capítulo 3

Implementação de Programas

Neste capítulo irá ser abordada a implementação de dois programas, um com o objectivo de cifrar e o outro de decifrar uma certa mensagem. Ambos utilizam um esquema híbrido 2.2, portanto fornecendo confidencialidade e autenticidade mas permitindo uma rápida cifragem e decifragem da mensagem a ser enviada. Os pares de chaves assimétricas utilizados nos testes destes programas foram gerados utilizando o programa **generateKeys.py**. O código de ambos os programas é disponibilizado em apêndixe no final deste relatório para facilitar o confronto do código exposto face à análise levada a cabo sobre ele em cada uma das seções seguintes, secções em que se procura explicar como se procurou resolver o problema inicialmente exposto.
Para facilitar certas explicações abaixo, parte-se do princípio que que o indivíduo A quer enviar em ficheiro cifrado a B, que só este último pode decifrar, e em que é possível verificar a autenticidade do ficheiro.

3.1 Cifragem

O programa responsável pela cifragem da mensagem a transmitir é **encipherPy.py**. O código do programa é incluido neste relatório em .1. Este programa depende de várias funções para sua execução, destancando-se as funções `sigGenerator`, `keyGenerator` e `encipher`. Outras funções nele presentes estão relacionadas com a robustês deste e não com a cifragem da mensagem em si, e portanto não irão ser abordadas.
O objectivo deste programa é tirar partido da velocidade de cifram com chaves simétricas, mas manter as funcionalidades e segurança extra que chaves assimétricas permitem. Com isso em mente, em gerada uma chave simétrica pelo programa, que é cifrada com a chave pública de B (para que apenas este a possa decifrar), a assinatura do ficheiro é cifrada com a chave privada de A (para que B possa ter certezas sobre a origem do ficheiro), e o ficheiro em si é cifrado com uma chave simétrica, sendo portanto muito mais rápida a sua cifragem e decifragem.

Apesar de nas funções a seguir expostas se falar na criação de ficheiros auziliares (`*.sig` e `*.key`) e dum ficheiro principal `*.bin`, estes não estão presentes no final da execução do programa já que se optou por juntar todos estes ficheiros num único `*.all`, facilitando assim o envio do ficheiro cifrado. Este processo é um dos implementados pelas funções auxiliares referidas acima, não estando relacionado com a cifragem em si.

3.1.1 sigGenerator

Esta função é responsável pela criação de uma assinatura do ficheiro, de forma a garantir ao recetor da mensagem que o que ele está a receber (e decifrar) é de facto a mensagem enviada (ou, no caso de não ser, pelo menos isso será facilmente verificado), e que esta foi enviada pelo indivíduo A.
Para isto recorre-se a uma função de síntese, `SHA-256`[1], para calcular o `hash` do ficheiro a cifrar. De seguida recorre-se ao módulo `PKCS1_v1_5`[2] para gerar uma assinatura com a chave privada de A, sendo esta gravada num ficheiro `*.sig`, onde `*` simboliza o nome do ficheiro original sem extensão.

3.1.2 keyGenerator

Esta função tem como objectivo gerar e guardar a chave simétrica usada pelo programa, cifrando-a com a chave pública de B (de forma a que apenas este a possa decifrar). Para isso, inicialmente gera-se uma sequência aleatória de 1024 bits. Visto que o método de encriptação AES necessita de chaves com 16, 24 ou 32 bytes, é calculada a síntese (SHA-256) do código aleatório gerado de forma a garantir que a chave final é de 256 bits (ou seja, 32 bytes). De seguida esta chave é cifrada com a chave pública de B recorrendo-se ao módulo `PKCS1_OAEP`[3], garantindo-se assim que apenas B pode decifrar esta chave (com a sua chave privada), obtendo a chave necessária à decifragem da mensagem enviada. O resultado é depois guardado num ficheiro `*.key`, onde mais uma vez `*` simboliza o nome do ficheiro a cifrar sem extensão.
Para além da chave cifrada, é também guardado um código IV (Initialization Vector) necessário à correcta descodificação da mensagem. Este código é gerado na função `encipher`, sendo que a função `keyGenerator` apenas o guarda no mesmo ficheiro onde a chave é guardada. Este código é necessário em resultado da utilização do mode de encpritção de AES CFB[4].

[1] `http://en.wikipedia.org/wiki/SHA-2`

[2] `https://www.dlitz.net/software/pycrypto/api/current/toc-Crypto.Signature.PKCS1_v1_5-module.html`

[3] `https://www.dlitz.net/software/pycrypto/api/current/toc-Crypto.Cipher.PKCS1_OAEP-module.html`

[4] `http://goo.gl/RxmaQC`

3.1.3 encipher

Por último, processa-se de facto a cifragem do ficheiro recorrendo-se para isso à função `encipher`. Esta implementa as duas funções anteriormente expostas, de forma o obter os resultados nelas exposto. Para além disso, gera também um código de 16 bytes chamado de `iv`[5], necessário pelo modo de cifragem em uso. Finalmente o ficheiro é cifrado recorrendo-se ao módulo de cifragem AES[6], modo CFB, e o resultado é guardado num ficheiro `*.bin`. O recetor da mensagem, indivíduo B, poderá utilizar a sua chave privada para obter a chave simétrica gerada para cifrar este ficheiro. Desta forma tira-se partido da velocidade de cifragem com chaves simétricas, mas mantém-se a segurança extra (e autenticação) que chaves assimétricas permitem.

3.2 Decifragem

A decifragem da mensagem cifrada enviada é feita pelo programa **decipherPy.py**.2. Também este programa recorre essencialmente a três funções para decifrar o ficheiro pedido, possuindo no entanto mais código responsável por aumentar a sua robustez e utilidade. O seu ojectivo é ser capaz de decifrar a chave simétrica utilizada para cifrar a mensagem (com a chave privada do utilizador do programa decifrador, ou seja, com a chave privada do indivíduo B), e com ela decifrar a mensagem inicial. Posto isto, é corrida uma função de verificação que visa garantir não só a integridade do ficheiro como também a sua autenticidade. Para isto, o ficheiro onde foi guardada a assinatura do ficheiro original é decifrado com a chave pública de A (quem envia a mensagem), sendo depois analizada a integridade do ficheiro comparando a sua síntese SHA-256 com a originalmente guardada. De seguida irão ser analizadas em maior detalhe cada uma das funções mais relacionadas com a decifragem do ficheiro.

3.2.1 hashVerification

Esta é a função encarregue de verificar a autenticidade e integridade do ficheiro recebido. Note-se que mesmo que mesmo que a verificação falhe a decifragem prossegue.
Para garantir a autenticidade e integridade do ficheiro, quando este foi cifrado foi gerada uma sua assinatura, guardada num ficheiro auxiliar `*.sig`. Desta forma, é agora possível ao utilizador B, por meio do programa `decipherPy.py` inferir sobre a origem deste, bem como se foi ou não adulterado. Para isto, recorre-se novamente ao módulo `PKCS1_v1_5`, em especial à sua função

[5] `http://en.wikipedia.org/wiki/Initialization_vector`

[6] `https://www.dlitz.net/software/pycrypto/api/current/toc-Crypto.Cipher.AES-module.html`

`RSASSA-PKCS1-V1_5-VERIFY`[7] que permite, tal como o nome indica, permite correr as verificações pretendidas.
Caso a verificação seja concluida com sucesso será imprimida uma mensagem para o terminal de forma a indicar isso mesmo, acompanhada da síntese SHA-256 do ficheiro. Caso contrário, apenas surgirá uma mensagem indicando que a verificação falhou.

3.2.2 keyReader

Esta função é a que permite a obtenção da chave (de 32 bytes) usada aquando da cifragem simétrica, bem como do código IV (de 16 bytes) utilizado no mesmo processo. Ambas as chaves encontram-se guardadas no ficheiro `*.key`, sendo que a chave de 32 bytes está cifrada com a chave pública do recetor (utilizador B) enquanto que o código de 16 bytes não. De forma a obter ambas estas chaves, o programa lê os primeiros 16 bytes do ficheiro `*.key`, onde o código IV foi guardado, procedendo-se depois à leitura do resto do ficheiro. Este "resto"é depois decifrado com a chave privada de B (recorrendo a `PKCS1_OAEP`), obtendo-se assim a chave simétrica utilizada para cifragem do ficheiro original.

3.2.3 decipher

Esta é a função que permite a verdadeira decifragem do ficheiro (no entanto pelo menos a função `keyReader` é também essencial para que esta decifragem seja possível). Esta decifragem é feita de modo inverso à cifragem feita pela função homóloga `encipher` (do programa de cifragem), e, sendo assim, recorre-se também ao algoritmo AES, modo CFB, com o mesmo valor IV para decifragem do ficheiro. A chave utilizada nesta decifragem é também a mesma para a cifragem (já que estamos perante um algoritmo de cifragem simétrica), e obtida recorrendo-se à função `keyReader`. No final da decifragem, é levada a cabo a verificação do ficheiro final através da função `hashVerification`.
No final, o ficheiro decifrado é guardado num ficheiro com o mesmo nome que o indicado para decifragem, sem extensão.

[7] `http://goo.gl/OPdiHU`

Capítulo 4

Conclusões

.1 encipherPy.py

```
import os, sys, zipfile
from Crypto import Random
from Crypto.Cipher import AES, PKCS1_OAEP
from Crypto.Hash import SHA256
from Crypto.PublicKey import RSA
from Crypto.Random import random
from Crypto.Signature import PKCS1_v1_5


# Define Public and Private key names!

# Sender's private key:
priKey = "A_PrivateKey.pem"
# Receiver's public key:
pubKey = "B_PublicKey.pem"

# File name to encrypt
f_name = ""

def usage():
    print "python encipherPy.py <File_Name>"


def sigGenerator(priKey_fname, f_name):
    # Opening and reading file to encrypt

    f = open(f_name, "r")
    buffer = f.read()
    f.close()

    # Creating Hash of the file. Using SHA-256 (there was a problem using SHA-512)

    h = SHA256.new(buffer)

    # Reading PrivateKey to sign file with

    keyPair = RSA.importKey(open(priKey_fname, "r").read())
    keySigner = PKCS1_v1_5.new(keyPair)

    # Saving Signature to *.sig File

    f = open(f_name.split('.')[0] + ".sig", "w")
    f.write(keySigner.sign(h))
    f.close()


def keyGenerator(pubKey_fname, f_name, iv):
    # Generating 1024 random bits, and creating SHA-256 (for 32 bits compatibility with AES)

    h = SHA256.new(str(random.getrandbits(1024)))

```

```
52      # Reading PublicKey to encrypt AES key with
53
54      keyPair = RSA.importKey(open(pubKey_fname, "r").read())
55      keyCipher = PKCS1_OAEP.new(keyPair.publickey())
56
57      # Saving encrypted key to *.key File
58
59      f = open(f_name.split('.')[0] + ".key", "w")
60      f.write(iv + keyCipher.encrypt(h.digest()))
61      f.close()
62
63      # Returning generated key to encrypt file with
64
65      return h.digest()
66
67
68  def encipher(keyA_fname, keyB_fname, f_name):
69      # Opening file to encrypt in binary mode
70
71      f = open(f_name, "rb")
72      buffer = f.read()
73      f.close()
74
75      # Generating file's Signature (and saving it)
76
77      sigGenerator(keyA_fname, f_name)
78
79      # Generating initializing vector for AES Encryption (there were problems when using different AES
80      # Needs to be saved in, for example, .key File!!!
81
82      iv = Random.new().read(AES.block_size)
83
84      # Generating symmetric key for use (and saving it)
85
86      k = keyGenerator(keyB_fname, f_name, iv)
87
88      # Encrypting and saving result to *.bin File. Using CFB mode
89
90      keyCipher = AES.new(str(k), AES.MODE_CFB, iv)
91      f = open(f_name.split('.')[0] + ".bin", "wb")
92      f.write(keyCipher.encrypt(buffer))
93      f.close()
94
95
96  def auxFilesZip(sig, key, bin):
97      # Opening file to contain all bin, sig and key files
98
99      f = zipfile.ZipFile(bin.split('.')[0] + ".all", "w")
100
101     # Writing each of the arguments to the created file
102
103     f.write(sig)
104     f.write(key)
105     f.write(bin)
```

```
106
107     # Closing the file
108
109     f.close()
110
111     # Running clean up to the bin, sig and key files
112
113     cleanUp(sig, key, bin)
114
115
116 def cleanUp(sig, key, bin):
117     # Deleting each of the files generated during ciphering
118
119     os.remove(sig)
120     os.remove(key)
121     os.remove(bin)
122
123
124 def checkFiles(f_name, pubKey, priKey):
125     # Checking for encrypting file's existence and access
126
127     if not os.path.isfile(f_name) or not os.access(f_name, os.R_OK):
128         print "Invalid file to encrypt. Aborting..."
129         sys.exit(1)
130
131     # Checking for each of the files to create existence and, in case they exist, if they are writabl
132
133     else:
134         s = f_name.split('.')[0]
135         if os.path.isfile(s + ".sig") and not os.access(s + ".sig", os.W_OK):
136             print "Can't create temporary file: *.bin. Aborting..."
137             sys.exit(2)
138         if os.path.isfile(s + ".key") and not os.access(s + ".key", os.W_OK):
139             print "Can't create temporary file: *.key. Aborting..."
140             sys.exit(3)
141         if os.path.isfile(s + ".bin") and not os.access(s + ".bin", os.W_OK):
142             print "Can't create temporary file: *.bin. Aborting..."
143             sys.exit(4)
144         if os.path.isfile(s + ".all") and not os.access(s + ".all", os.W_OK):
145             print "Can't create output file. Aborting..."
146             sys.exit(5)
147
148     # Checking for public key's existence and access
149
150     if not os.path.isfile(pubKey) or not os.access(pubKey, os.R_OK):
151         print "Invalid public key file. Aborting..."
152         sys.exit(6)
153
154     # Checking for private key's existence and access
155
156     if not os.path.isfile(priKey) or not os.access(priKey, os.R_OK):
157         print "Invalid private key file. Aborting..."
158         sys.exit(7)
159
```

```
160
161 # Gathering encrypting file name
162
163 if len(sys.argv) > 2:
164     usage()
165 elif len(sys.argv) == 1:
166     print "File name:"
167     f_name = raw_input(">>> ")
168 else:
169     f_name = sys.argv[1]
170
171 # Gathering keys names
172
173 if priKey == "":
174     print "Sender's private key file name:"
175     priKey = raw_input(">>> ")
176 if pubKey == "":
177     print "Receiver's public key file name:"
178     pubKey = raw_input(">>> ")
179
180 # Running checks to files
181
182 checkFiles(f_name, pubKey, priKey)
183
184 # Ciphering file (and generating all auxiliary files)
185
186 encipher(priKey, pubKey, f_name)
187
188 # Generating output file and clean up
189
190 auxFilesZip(f_name.split('.')[0] + ".sig", f_name.split('.')[0] + ".key", f_name.split('.')[0] + ".bi
```

.2 decipherPy.py

```
import os, sys, zipfile
from Crypto.Cipher import PKCS1_OAEP, AES
from Crypto.Hash import SHA256
from Crypto.PublicKey import RSA
from Crypto.Signature import PKCS1_v1_5


# Define Public and Private key names!

# Sender's public key:
pubKey = "A_PublicKey.pem"
# Receiver's private key:
priKey = "B_PrivateKey.pem"

# File name to decrypt
f_name = ""

def usage():
    print "python decipherPy.py <File_Name>"


def hashVerification(pubKey_fname, f_name):
    # Generating decrypted file's SHA-256

    h = SHA256.new()
    h.update(open(f_name, "r").read())

    # Reading PublicKey to check Signature with

    keyPair = RSA.importKey(open(pubKey_fname, "r").read())
    keyVerifier = PKCS1_v1_5.new(keyPair.publickey())

    # If Signature is right, prints SHA-256. Otherwise states that the file is not authentic

    if keyVerifier.verify(h, open(f_name.split('.')[0] + ".sig", "r").read()):
        print "The signature is authentic."
        print "SHA-256 -> %s" % h.hexdigest()
    else:
        print "The signature is not authentic."


def keyReader(privKey_fname, f_name):
    # Reading PrivateKey to decipher Symmetric key used

    keyPair = RSA.importKey(open(privKey_fname, "r").read())
    keyDecipher = PKCS1_OAEP.new(keyPair)

    # Reading iv (initializing vector) used to encrypt and saving Symmetric key used to 'k'

    f = open(f_name.split('.')[0] + ".key", "r")
    iv = f.read(16)
```

```
52      k = keyDecipher.decrypt(f.read())
53
54      return k, iv
55
56
57  def decipher(keyA_fname, keyB_fname, f_name):
58      # Getting Symmetric key used and iv value generated at encryption process
59
60      k, iv = keyReader(keyB_fname, f_name)
61
62      # Deciphering the initial information and saving it to file with no extension
63
64      keyDecipher = AES.new(k, AES.MODE_CFB, iv)
65      bin = open(f_name + ".bin", "rb").read()
66      f = open(f_name.split('.')[0], "wb")
67      f.write(keyDecipher.decrypt(bin))
68      f.close()
69
70      # Running a Signature verification
71
72      hashVerification(keyA_fname, f_name.split('.')[0])
73
74
75  def auxFilesUnzip(all):
76      # Opening the input file
77
78      f = zipfile.ZipFile(all + ".all", "r")
79
80      # Extracting all of its files
81
82      f.extractall()
83
84
85  def cleanUp(sig, key, bin, all):
86      # Removing all of the files created, except for the final deciphered file
87
88      os.remove(sig)
89      os.remove(key)
90      os.remove(bin)
91      os.remove(all)
92
93  def checkFiles(f_name, pubKey, priKey, first_run):
94      # Checking for decrypting file's existence and access
95
96      if first_run and (not os.path.isfile(f_name + ".all") or not os.access(f_name + ".all", os.R_OK)):
97          print "Invalid file to decrypt. Aborting..."
98          sys.exit(1)
99
100     elif not first_run:
101         # Checking if all of the necessary files exist and are accessible
102
103         if not os.path.isfile(f_name + ".sig") or not os.access(f_name + ".sig", os.R_OK):
104             print "Invalid *.sig file. Aborting..."
105             sys.exit(2)
```

```
106             if not os.path.isfile(f_name + ".key") or not os.access(f_name + ".key", os.R_OK):
107                 print "Invalid *.key file. Aborting..."
108                 sys.exit(3)
109             if not os.path.isfile(f_name + ".bin") or not os.access(f_name + ".bin", os.R_OK):
110                 print "Invalid *.bin file. Aborting..."
111                 sys.exit(4)
112 
113             # Checking if in case of output file's existence, it is writable
114 
115             if os.path.isfile(f_name) and not os.access(f_name, os.W_OK):
116                 print "Can't create output file. Aborting..."
117                 sys.exit(5)
118 
119         # Checking for public key's existence and access
120 
121         if not os.path.isfile(pubKey) or not os.access(pubKey, os.R_OK):
122             print "Invalid public key file. Aborting..."
123             sys.exit(6)
124 
125         # Checking for private key's existence and access
126 
127         if not os.path.isfile(priKey) or not os.access(priKey, os.R_OK):
128             print "Invalid private key file. Aborting..."
129             sys.exit(7)
130 
131 
132     # Gathering encrypting file name
133 
134     if len(sys.argv) > 2:
135         usage()
136     elif len(sys.argv) == 1:
137         print "File name:"
138         f_name = raw_input(">>> ")
139     else:
140         f_name = sys.argv[1]
141 
142     # Gathering keys names
143 
144     if pubKey == "":
145         print "Sender's public key file name:"
146         pubKey = raw_input(">>> ")
147     if priKey == "":
148         print "Receiver's private key file name:"
149         priKey = raw_input(">>> ")
150 
151 
152     f_name = f_name.split('.')[0]
153     checkFiles(f_name, pubKey, priKey, True)
154     auxFilesUnzip(f_name)
155     checkFiles(f_name, pubKey, priKey, False)
156     decipher(pubKey, priKey, f_name)
157     cleanUp(f_name + ".sig", f_name + ".key", f_name + ".bin", f_name + ".all")
```