# 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**

Com o objetivo de implementar dois programas, um que cifre e outro que decifre um dado ficheiro segundo um esquema híbrido, de forma a fornecer autenticidade e confidencialidade, mantendo ainda assim um rápido processamento do ficheiro, começou por se estudar o que é criptografia simétrica, assimétrica, e de que forma ambas se relacionam de forma a obter um esquema híbrido. De seguida, procedeu-se à implementação dos programas em Python e descreveu-se como foram implementadas as funções que processam a componente criptográfica dos mesmos.

# Conteúdo

**1 Introdução** **1**

**2 Criptografia** **3**
2.1 Funções de Síntese . . . . . 4
2.2 Criptografia Simétrica . . . . . 5
2.2.1 Cifras Simétricas Contínuas . . . . . 6
2.2.2 Cifras Simétricas por Blocos . . . . . 7
2.3 Criptografia Assimétrica . . . . . 8
2.4 Criptografia Híbrida . . . . . 9

**3 Implementação de Programas** **11**
3.1 Cifragem . . . . . 12
3.1.1 sigGenerator . . . . . 13
3.1.2 keyGenerator . . . . . 13
3.1.3 encipher . . . . . 14
3.2 Decifragem . . . . . 14
3.2.1 sigVerification . . . . . 15
3.2.2 keyReader . . . . . 15
3.2.3 decipher . . . . . 15

**4 Conclusões** **17**

# Lista de Figuras


2.1 Processos básicos de cifragem e decifragem na criptografia. . . 4
2.2 Criação de assinaturas digitais com funções de síntese. . . . . 5
2.3 Uso de chave partilhada na criptografia simétrica. . . . . 6
2.4 Processos de cifragem na cifra por blocos (*Block Cipher*) e contínua (*Stream Cipher*). . . . . 7
2.5 Criptografia assimétrica e confidencialidade. . . . . 8
2.6 Criptografia assimétrica e autenticação. . . . . 9

3.1 Cifragem com AES modo CFB . . . . . 12
3.2 Decifragem com AES modo CFB . . . . . 12

# Capítulo 1

# Introdução

Cada vez mais a criptografia desempenha um processo fundamental na sociedade atual. Veja-se como a decifragem de 'Enigma'[1] teve uma tão grande importância no desenrolar da Segunda Guerra Mundial, ou como o recente ataque à 'Sony Pictures Entertainment' fez os Estados Unidos da América temer pela sua cibersegurança.

Assim, e visto que a cada dia que passa estamos mais ligados uns aos outros através, por exemplo, da Web, é cada vez mais necessário manter a confidencialidade na partilha de informação, onde algoritmos de cifragem e decifragem desempenham um processo crucial. Felizmente, nos dias que correm temos à nossa disposição inúmeros algoritmos deste tipo, de onde se destacam os algoritmos simétricos e assimétricos.

Em relação aos algoritmos simétricos, destaca-se a velocidade com que determinada informação pode ser cifrada e decifrada. Nestes algoritmos é utilizada uma chave partilhada entre emissor e recetor para transmitir informação de uma forma confidencial. Já em relação a algoritmos assimétricos, estes são centenas senão milhares de vezes mais lentos que os métodos simétricos, no entanto visto utilizarem pares de chaves de forma a garantir transmissão de informação de uma forma segura, onde existe uma chave pública partilhada com os demais e uma privada a que apenas o próprio indivíduo deve ter acesso, este método garante, por exemplo, que um ficheiro cifrado com a chave pública de um indivíduo **A** pode apenas ser decifrado pelo próprio **A** com a sua chave privada (note-se que a informação cifrada com uma chave pública só pode ser decifrada pela respetiva chave privada, ou vice-versa). Para além disso, estes métodos por pares de chaves são considerados mais seguros e permitem também dar resposta a questões de autenticidade.

Com base nestas vantagens e desvantagens de um e outro métodos, surgiram esquemas de encriptação híbrida que procuram tirar partido das vanta-

[1] `http://en.wikipedia.org/wiki/Enigma_machine`

gens de ambos os métodos (velocidade da encriptação simétrica e segurança da encriptação assimétrica).

Estes temas relacionados com diferentes formas de cifragem encontram-se desenvolvidos na secção 2 deste relatório.

Com isto em mente foram implementados dois programas escritos em linguagem Python e recorrendo à biblioteca PyCrypto. Um funcionando como programa que cifra, e outro que decifra implementando um esquema de encriptação híbrida garantindo a transmissão de informação de forma eficiente e segura. A análise recorrente da implementação destes programas encontra-se na secção 3.

# Capítulo 2

# Criptografia

A criptografia é a arte ou ciência de escrever uma mensagem de forma a ocultar o seu conteúdo. Dito por outras palavras, a criptografia usa técnicas que permitem que várias entidades possam transmitir mensagens de forma segura e privada, sem que estas sejam intercetadas e aproveitadas por terceiros. Independentemente do método que for usado para ocultar dados, a criptografia moderna pretende satisfazer quatro objetivos fundamentais:

**Confidencialidade:** caso alguém intercete uma mensagem, essa pessoa não deve ser capaz de ler e perceber o seu conteúdo.

**Integridade dos dados:** o destinatário da mensagem deve ser capaz de verificar se a mensagem foi alterada durante a transmissão, acidental ou deliberadamente.

**Autenticação:** o destinatário da mensagem deve ser capaz de verificar a sua origem.

**Não-repúdio:** o remetente não deve ser capaz de negar mais tarde que foi quem realmente enviou a mensagem para o destinatário.

Existem alguns conceitos que são essenciais para a compreensão das operações envolvidas em criptografia. O termo **cifra** é o processo que transforma um texto em claro num texto cifrado, ou também designado por criptograma. O termo **decifra** é a operação inversa da cifra, ou seja, transforma o criptograma no texto em claro original. As operações de cifra e decifra usam algoritmos e chaves, sendo estas utilizadas como parâmetros nos ditos algoritmos. Os algoritmos são modelos matemáticos que contêm um conjunto de operações (ou regras) que permitem transformar os dados, ou seja, neste contexto possibilitam as operações de cifra e decifra. A figura 2.1 mostra os processos básicos de cifragem e decifragem na criptografia.

A mensagem original (*plain-text*) é cifrada usando um algoritmo e uma chave, produzindo uma mensagem cifrada (*cipher-text*). A mensagem cifrada é por sua vez decifrada usando outro algoritmo e outra chave (que até

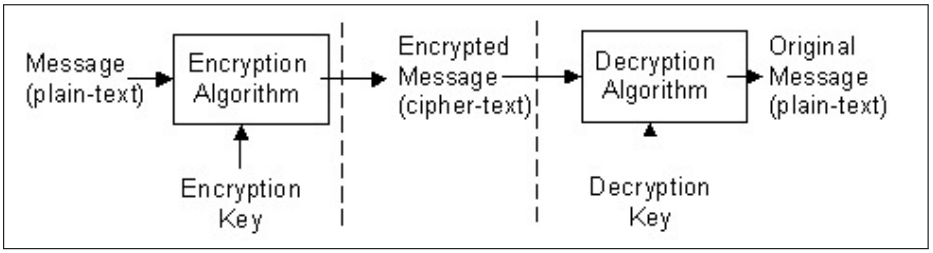


Figura 2.1: Processos básicos de cifragem e decifragem na criptografia.

poderá ser igual à usada no processo de cifragem), obtendo-se a mensagem original (*plain-text*).

A criptografia pode ser dividida em várias áreas, entre as quais as funções de síntese e as criptografias simétrica, assimétrica e híbrida.

## 2.1 Funções de Síntese

Existem três tipos de algoritmos criptográficos: os com chaves simétricas, assimétricas e as funções de síntese (*Digest functions* ou *Hash functions*). Ao contrário dos que utilizam chaves, as funções de síntese apenas processam um determinado ficheiro, sem necessitarem de qualquer chave. Estas funções produzem um valor de dimensão constante (*hash value*) a partir de um volume arbitrário de bits. Além disso, devem garantir a obtenção de valores diferentes mesmo para ficheiros semelhantes (se bem que é extremamente dificil garantir a não ocorrência de colisões). O principal objetivo destas funções na criptografia é a integridade dos dados. O valor que é produzido fornece uma impressão digital (*digital fingerprint*) do conteúdo da mensagem, o que permite assegurar que caso a mensagem for alterada por pessoas mal-intencionadas ou, por exemplo, vírus, isso não passa despercebido. São vários os algoritmos que produzem estas sínteses, entre os quais o MD5 e o SHA-1 com 128 e 160 bits de tamanho, respetivamente. Estes algoritmos podem ser considerados mais ou menos eficientes em função da facilidade com que se consegue obter um mesmo valor de hash para ficheiros diferentes.

Uma das situações onde se podem encontrar as funções de síntese é nas assinaturas digitais. A figura 2.2 esquematiza de forma simples os processos associados à produção de documentos assinados digitalmente.

O remetente da mensagem cria uma assinatura digital aplicando uma função de síntese (*Hash function*) no documento (*Data*), produzindo-se um ficheiro sintetizado (*Hash*). Este ficheiro é cifrado usando a chave privada do emissor de forma a produzir uma assinatura digital que é depois anexada ao documento (*Digitally signed data*). Após a receção do documento assinado,

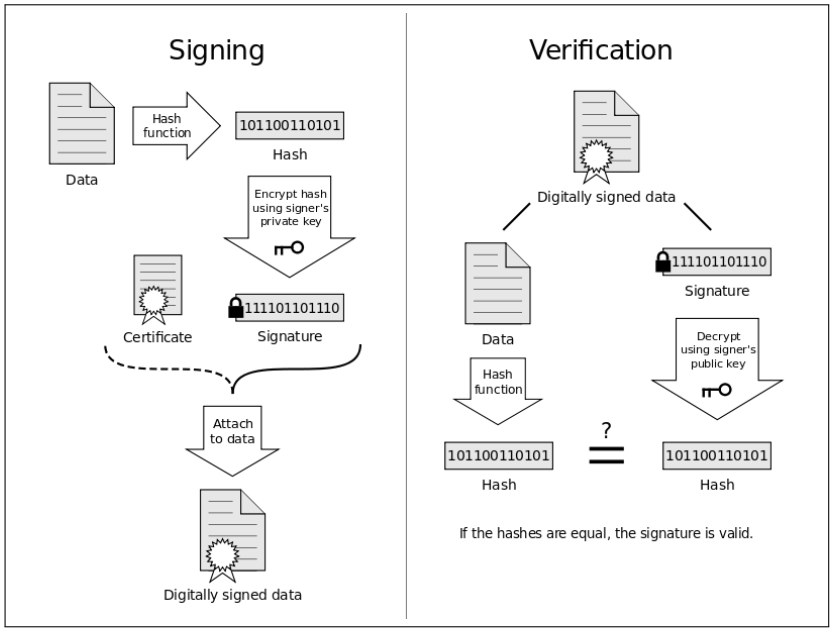


Figura 2.2: Criação de assinaturas digitais com funções de síntese.

o destinatário decifra a assinatura através da chave pública do assinante e executa uma síntese ao próprio ficheiro. São produzidos dois valores 'hash' e ambos comparados, i.e., se forem iguais a assinatura é considerada válida.

De referir que no contexto das assinaturas digitais são aplicados dois objetivos da criptografia moderna: o do não-repúdio e garatia de integridade de dados. Ou seja, o remetente não deverá ser capaz de negar mais tarde que foi realmente a pessoa que enviou a mensagem, e o recetor deverá ser capaz de verificar, para além da origem da mesma, que esta se encontra no mesmo estado em que foi enviada.

## 2.2 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 2.3 ilustra de forma simples o uso de chaves partilhadas na criptografia simétrica.

O remetente (*sender*) cifra o texto em claro (*plaintext*) através da chave simétrica (*secret-key*), produzindo-se um texto cifrado (*ciphertext*). O destinatário (*recipient*) da mensagem decifra-a usando a chave partilhada por

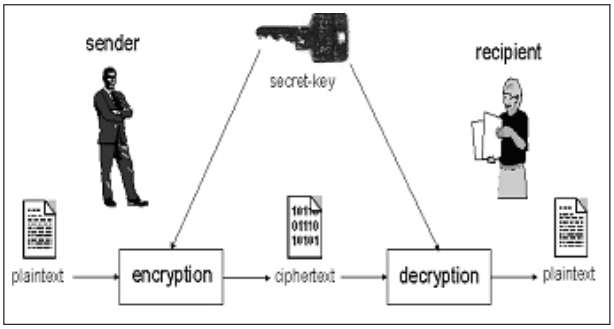


Figura 2.3: Uso de chave partilhada na criptografia simétrica.

ambos. O processo contrário também se pode realizar, ou seja, a chave tanto pode servir para cifrar como para decifrar mensagens.

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

### 2.2.1 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)[1] visto que é uma função facilmente invertível.

A operação de cifra consiste em aplicar a função XOR entre a mensagem a transmitir e a chave utilizada na operação. A expressão 2.1 mostra como é feita a operação de cifra.

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

Onde C é o criptograma, T é o texto em claro e Ks a chave contínua (*keystream*).

A operação de decifra pode ser executada aplicando novamente a função XOR. A expressão 2.2 mostra como é feita a operação de decifra.

$$T = C \oplus Ks \tag{2.2}$$

Onde T é o texto em claro, C é o criptograma e Ks a chave contínua (*keystream*).

[1] `http://en.wikipedia.org/wiki/Exclusive_or`

De referir que é usada a mesma chave nas duas equações (2.1 e 2.2).

### 2.2.2 Cifras Simétricas por Blocos

Na cifra por blocos, a mensagem é dividida em blocos de igual dimensão sendo cada um deles processados de forma independente dos restantes (se bem que podem existir mecanismos de feedback). Conforme o algoritmo aplicado para cifrar e decifrar mensagens, os blocos podem ter dimensões diferentes. Por exemplo, o DES (*Data Encryption Standard*) e o AES (*Data Encryption Standard*) são dois algoritmos bastante usados neste tipo de cifra, mas em que ambos utilizam blocos de tamanhos distintos, 64 e 128 bits respetivamente. O criptograma é criado após a criação de vários pequenos criptogramas representando cada um dos blocos cifrados.

Um conceito bastante importante neste tipo de cifra é o alinhamento, ou *padding*. Existem modos de cifra por blocos que põem particularmente em prática este conceito. Um deles é o ECB (*Electronic Code Book*), que possibilita a separação de ficheiros em blocos de dimensões iguais. Por exemplo, se um determinado ficheiro for separado em blocos e verificar-se que o último não tem uma dimensão igual à dos restantes, é adicionado uma "string" de bytes (ou bits) no final desse bloco de forma a obter o tamanho pretendido. Esse acréscimo de bits forma um excipiente, sendo importante indicar a sua existência e comprimento de forma a facilitar a decifra por parte de quem recebe a mensagem cifrada. A figura 2.4 ilustra muito basicamente os processos de cifragem nas cifras contínuas e por blocos.

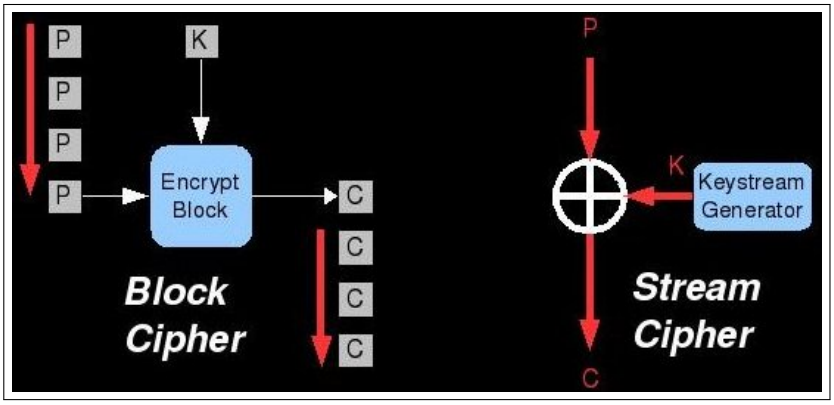


Figura 2.4: Processos de cifragem na cifra por blocos (*Block Cipher*) e contínua (*Stream Cipher*).

Na cifra por blocos, o texto em claro é 'partido' em blocos de dimensões iguais (P1, P2, ..., Pn) sendo cada um deles cifrados de forma independente dos restantes, obtendo-se vários criptogramas (C1, C2, ..., Cn) que todos juntos formam o criptograma principal. Na cifra contínua, é gerada uma

chave contínua e combinando-a com o texto em claro com a função XOR gera-se o criptograma (C).

## 2.3 Criptografia Assimétrica

A criptografia assimétrica é uma área da criptografia onde cada um dos interlocutores possui um par de chaves, uma privada (ou secreta) e outra pública. A chave pública é usada principalmente para cifrar mensagens ou para verificar assinaturas digitais. O termo **assimétrico** vem do facto de se usarem duas chaves, mas ambas com funções opostas. A criptografia assimétrica usa cifras por blocos e pode assegurar a confidencialidade e a autenticação nas mensagens transmitidas dependendo de como se usam as chaves.

Se se pretender comunicar confidencialmente, a chave pública do destinatário é usada para cifrar e a chave privada do destinatário é usada para decifrar, ou seja só o destinatário conseguirá decifrar o criptograma através da sua chave privada. Por outro lado, o destinatário não tem conhecimento de quem remeteu a mensagem cifrada (pelo menos isso não pode ser garantido), e sendo assim a autenticação não pode ser verificada. A figura 2.5 ilustra o uso correto das chaves assimétricas para garantir a confidencialidade na transmissão das mensagens entre emissor e recetor.

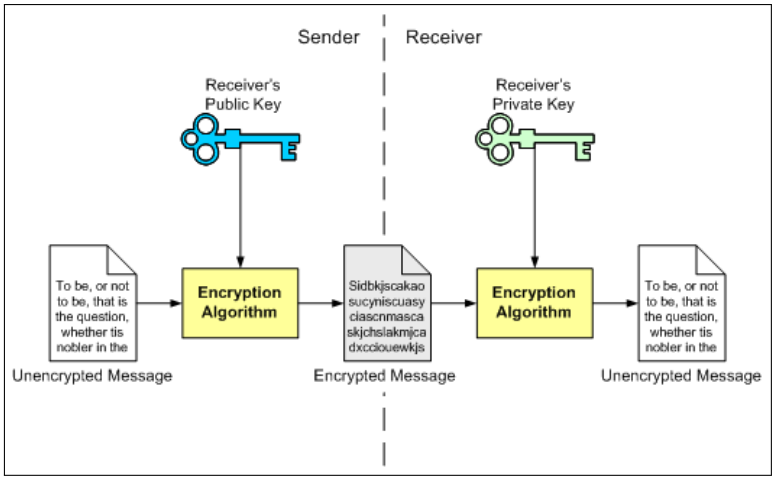


Figura 2.5: Criptografia assimétrica e confidencialidade.

O remetente (*sender*) deve conhecer a chave pública do destinatário (*receiver*) para cifrar as suas mensagens e assim comunicar de forma confidencial. O destinatário usa por sua vez a sua chave privada para decifrar o criptograma. Neste caso, só é usado o par de chaves do lado do destinatário.

Por outro lado, se o remetente pretender garantir a autenticação da origem da mensagem recebida, por exemplo na transmissão de documentos com assinaturas digitais, as chaves são usadas de forma contrária, ou seja, a chave privada do remetente é usada para cifrar as mensagens e a sua chave pública usada para as decifrar. O destinatário decifra o criptograma usando a chave pública do remetente e verifica que foi este último quem gerou o criptograma. Neste caso não há confidencialidade visto que a chave usada para decifrar é pública (chave pública do remetente), ou seja, quem a conhecer poderá decifrar o criptograma. A figura 2.6 ilustra a transmissão de mensagens usando chaves assimétricas para garantir a autenticação da origem das mesmas.

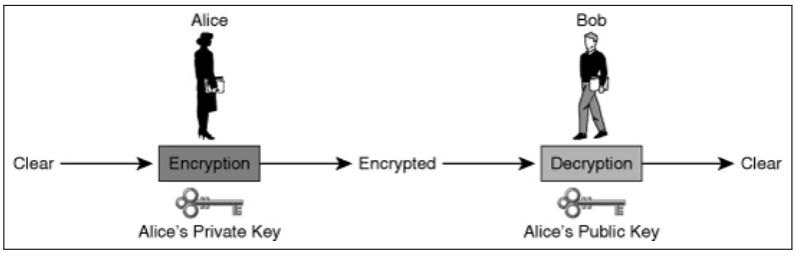


Figura 2.6: Criptografia assimétrica e autenticação.

O emissor (*Alice*) cifra a mensagem através de um algoritmo assimétrico e da sua chave privada (*Alice's Private Key*) e envia para o destinatário (*Bob*) o criptograma gerado. Este, por sua vez, decifra-o através da chave pública do emissor (*Alice's Public Key*) verificando a autenticação do autor. Neste caso, só se usa o par de chaves do lado do emissor.

Um dos algoritmos mais utilizados neste método é o RSA[2] que tira partido de certas propriedades de números primos e da dificuldade e exigência de recursos necessários para factorizar grandes inteiros.

## 2.4 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, este método procura juntar o melhor de ambos os processos. O que geralmente é permitido 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 método de cifragem simétrica. No entanto, a chave utilizada para cifrar o ficheiro é (geralmente) gerada aleatoriamente, sendo guardada noutro ficheiro e cifrada segundo um processo de chaves assimétricas. Quando o ficheiro

[2] `http://en.wikipedia.org/wiki/RSA_(cryptosystem)`

principal é enviado, juntamente envia-se a chave simétrica utilizada para o cifrar por exemplo num ficheiro auxiliar (relembre-se que esta chave simétrica encontra-se também ela cifrada).

Veja-se um simples exemplo de utilização, em que um certo indivíduo **A** pretende enviar uma mensagem cifrada a outro **B**, segundo este esquema 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. Ambas a mensagem e a chave cifradas são enviadas 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 objetivo de cifrar e o outro de decifrar uma certa mensagem. Ambos utilizam um esquema híbrido (2.4), 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 para além de ser enviado em anexo é também disponibilizado (em anexo) no final deste relatório para facilitar o confronto do código face à análise levada a cabo sobre ele em cada uma das secções seguintes, secções onde se procura explicar como se resolveu o problema inicialmente exposto.

Os programas foram escritos na linguagem de programação `Python (2.7)`, e recorreu-se extensamente à biblioteca `PyCrypto`[1] para implementar os processos relacionados com cifragem e decifragem de informação.

Visto estar a ser utilizado um esquema de encriptação híbrida, o ficheiro principal a transmitir é cifrado recorrendo-se a um algoritmo com chaves partilhadas, neste caso AES. Na utilização deste algoritmo, escolheu-se o modo de operação CFB, que é um método de encriptação contínua auto-sincronizada (*self-synchronizing stream encryption*), garantindo assim que a geração do criptograma não é feita apenas pela aplicação do operador XOR entre a mensagem e uma chave, mas em que há uma relação de feedback entre o criptograma a ser gerado e o resto do criptograma a gerar (um pouco como cifras por blocos). Isto juntamente com a existência de um valor `iv` (*initialization vector*) que serve como fator aleatório extra neste processo de feedback, garante que um mesmo ficheiro que seja cifrado várias vezes com uma mesma chave dará origem a criptogramas diferentes e portanto é mais difícil desvendar o que se está a transmitir. Um exemplo da implementação deste algoritmo é visível nas figuras 3.1 e 3.2.

[1] `https://www.dlitz.net/software/pycrypto/`

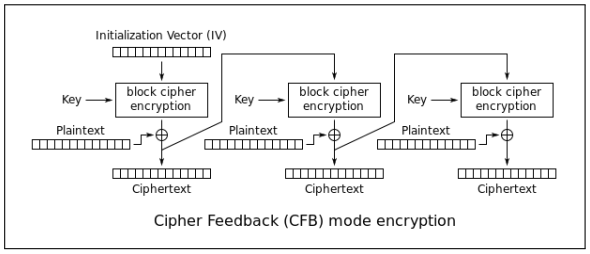


Figura 3.1: Cifragem com AES modo CFB

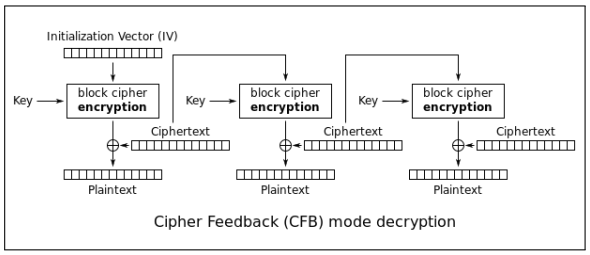


Figura 3.2: Decifragem com AES modo CFB

No entanto, devido à forma como é processado este aumento de segurança também acarreta um decréscimo na velocidade de cifragem e decifragem. Apesar disso escolheu-se usá-lo já que se o maior fator de interesse nestes programas fosse meramente a velocidade, então não se estaria sequer a seguir um esquema híbrido.

Para facilitar certas explicações abaixo, parte-se do princípio de que o indivíduo **A** quer enviar um 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 **encipher.py**. O código do programa é incluido neste relatório em 4. Este programa depende de várias funções para sua execução, destancando-se as `sigGenerator`, `keyGenerator` e `encipher`. Outras funções nele presentes estão relacionadas com a robustez do programa 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 algoritmos de cifragem com chaves simétricas, mas manter as funcionalidades e segurança extra que os com chaves assimétricas permitem. Com isso em mente, é 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 se foi adulterado ou não), e o ficheiro em si é cifrado com a chave simétrica gerada, sendo portanto muito mais rápida a sua cifragem e decifragem.

Apesar de nas funções a seguir expostas se falar na criação de ficheiros auxiliares (`*.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 irá decifrar) é de facto a mensagem enviada (ou, no caso de não ser, pelo menos isso será facilmente verificado). Para além disso, o recetor saberá também com certeza que a mensagem foi enviada pelo indivíduo **A**.

Para isto recorre-se a uma função de síntese, `SHA-256`[2], para calcular a `hash` do ficheiro a cifrar. De seguida recorre-se ao módulo `PKCS1_v1_5`[3] para gerar a assinatura do ficheiro com a chave privada de **A** e este `hash`, 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`[4], 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.

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

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

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

Para além da chave cifrada, é também guardado um código `iv` (*initialization vector*) necessário à correta 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 modo de encriptação de AES CFB[5].

### 3.1.3 encipher

Por último, processa-se de facto a cifragem do ficheiro recorrendo-se para isso à função `encipher`. Esta recorre às duas funções anteriormente expostas de forma a obter os resultados nelas expostos, gera também um código de 16 bytes chamado de `iv`[6] e necessário pelo modo de cifragem em uso, e finalmente cifra o ficheiro original recorrendo ao módulo de cifragem AES[7], modo CFB (*Cipher FeedBack*), sendo o resultado 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 e poderá utilizar a chave pública de **A** para garantir a sua origem. 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 **decipher.py** também disponível em 4. Também este recorre essencialmente a três funções para decifrar o ficheiro recebido, possuindo no entanto mais código responsável por aumentar a sua robustez e utilidade. O seu principal ojectivo é ser capaz de decifrar a chave simétrica utilizada inicialmente 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 analisada a integridade do ficheiro comparando a sua síntese `SHA-256` com a originalmente guardada. De seguida irão ser analisadas em maior detalhe cada uma das funções relacionadas com a decifragem do ficheiro.

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

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

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

### 3.2.1 sigVerification

Esta é a função encarregue de verificar a autenticidade e integridade do ficheiro recebido. É de notar que mesmo que a verificação falhe a decifragem prossegue.

Para garantir a autenticidade e integridade do ficheiro, quando este foi cifrado foi gerada uma assinatura, guardada num ficheiro auxiliar `*.sig`. Desta forma, é agora possível ao utilizador **B**, por meio do programa `decipher.py`, inferir sobre a origem da mensagem recebida bem como se foi ou não adulterada. Para isto, recorre-se novamente ao módulo `PKCS1_v1_5`, em especial à sua função `RSASSA-PKCS1-V1_5-VERIFY`[8] que permite, tal como o nome indica, correr as verificações pretendidas.

Caso a verificação seja concluída 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 com chaves simétricas), e obtida recorrendo-se à função `keyReader`. No final da decifragem, é levada a cabo a verificação do ficheiro final através da

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

função `sigVerification`.

No final, o ficheiro decifrado é guardado num ficheiro com o mesmo nome que o indicado para decifragem, sem extensão.

# Capítulo 4

# Conclusões

Foi-nos possível verificar que ao adotar um esquema híbrido podemos tirar partido da velocidade que um método de encriptação com chaves partilhadas permite, bem como por exemplo da segurança extra e autenticação que métodos com pares de chaves oferecem. Posto isto, pudemos implementar dois programas, um que cifra e outro que decifra, que utilizam este mesmo conceito para garantir a transmissão de informação de forma confidencial e de onde se possa inferir sobre a sua autenticidade e integridade. Isto é possível pois recorre-se a um algoritmo simétrico para cifrar (e decifrar) o ficheiro a transmitir (o que poderia não ser viável caso se se recorresse a um método assimétrico visto este ser muito mais lento e enventualmente o ficheiro poder ser de muito grandes dimensões), mas ainda assim sendo a chave simétrica usada neste processo de cifragem ela própria cifrada com a chave pública do receptor (que a decifra com a sua chave privada), e sendo gerada uma assinatura do ficheiro a partir de uma sua síntese e da chave privada do emissor, não se perdem as vantagens de um método de encriptação assimétrica. Isto garante que apenas o recetor será capaz de decifrar a chave que permite decifrar a informação enviada, bem como que este será capaz de verificar o ficheiro recorrendo à sua assinatura, podendo ter certezas sobre a sua origem e se foi ou não adulterado.

Assim sendo, pudemos verificar que nem métodos simétricos nem assimétricos são perfeitos na transmissão de informação de forma segura. No entanto, apesar dos seus possíveis problemas, ambos têm importantes vantagens e desta forma um método de encriptação que se baseie num esquema híbrido poderá ser um bom compromisso entre as qualidades e desvantagens de um, e as qualidades e desvantagens de outro.

# Anexos

## encipher.py

```
1  import os
2  import sys
3  import zipfile
4  from Crypto import Random
5  from Crypto.Cipher import AES, PKCS1_OAEP
6  from Crypto.Hash import SHA256
7  from Crypto.PublicKey import RSA
8  from Crypto.Random import random
9  from Crypto.Signature import PKCS1_v1_5
10
11
12 # Define public and private key names for faster usage
13
14 # Sender's private key:
15 priKey = "A_PrivateKey.pem"
16 # Receiver's public key:
17 pubKey = "B_PublicKey.pem"
18
19 # File name to encrypt
20 f_name = ""
21
22 # Private key password:
23 priPass = ""
24
25
26 def usage():
27     print "python encipher.py <file_name>"
28     sys.exit(-1)
29
30
31 def sigGenerator(priKey_fname, f_name, priPass):
32     # Opening and reading file to encrypt
33
34     f = open(f_name, "r")
35     buffer = f.read()
36     f.close()
37
38     # Creating hash of the file. Using SHA-256 (SHA-512 rose problems)
```

```
39
40      h = SHA256.new(buffer)
41
42      # Reading private key to sign file with
43
44      keyPair = RSA.importKey(open(priKey_fname, "r").read(), passphrase=priPass)
45      keySigner = PKCS1_v1_5.new(keyPair)
46
47      # Saving signature to *.sig file
48
49      f = open(f_name.split('.')[0] + ".sig", "w")
50      f.write(keySigner.sign(h))
51      f.close()
52
53
54  def keyGenerator(pubKey_fname, f_name, iv):
55      # Generating 1024 random bits, and creating SHA-256 (for 32 bits compatibility with AES)
56
57      h = SHA256.new(str(random.getrandbits(1024)))
58
59      # Reading public key to encrypt AES key with
60
61      keyPair = RSA.importKey(open(pubKey_fname, "r").read())
62      keyCipher = PKCS1_OAEP.new(keyPair.publickey())
63
64      # Saving encrypted key to *.key file
65
66      f = open(f_name.split('.')[0] + ".key", "w")
67      f.write(iv + keyCipher.encrypt(h.digest()))
68      f.close()
69
70      # Returning generated key to encrypt file with
71
72      return h.digest()
73
74
75  def encipher(keyA_fname, keyB_fname, f_name, priPass):
76      # Opening file to encrypt in binary reading mode
77
78      f = open(f_name, "rb")
79      buffer = f.read()
80      f.close()
81
82      # Generating file's signature (and saving it)
83
84      sigGenerator(keyA_fname, f_name, priPass)
85
86      # Generating initializing vector for AES Encryption
87
88      iv = Random.new().read(AES.block_size)
89
90      # Generating symmetric key for use (and saving it)
91
92      k = keyGenerator(keyB_fname, f_name, iv)
```

```
93
94      # Encrypting and saving result to *.bin file. Using CFB mode
95
96      keyCipher = AES.new(str(k), AES.MODE_CFB, iv)
97      f = open(f_name.split('.')[0] + ".bin", "wb")
98      f.write(keyCipher.encrypt(buffer))
99      f.close()
100
101
102 def auxFilesZip(sig, key, bin):
103     # Opening file to contain all bin, sig and key files
104
105     f = zipfile.ZipFile(bin.split('.')[0] + ".all", "w")
106
107     # Writing each of the arguments to the created file
108
109     f.write(sig)
110     f.write(key)
111     f.write(bin)
112
113     # Closing the file
114
115     f.close()
116
117     # Running clean up to the bin, sig and key files
118
119     cleanUp(sig, key, bin)
120
121
122 def cleanUp(sig, key, bin):
123     # Deleting each of the files generated during ciphering
124
125     os.remove(sig)
126     os.remove(key)
127     os.remove(bin)
128
129
130 def checkFiles(f_name, pubKey, priKey):
131     # Checking for encrypting file's existence and access
132
133     if not os.path.isfile(f_name) or not os.access(f_name, os.R_OK):
134         print "Invalid file to encrypt. Aborting..."
135         sys.exit(1)
136
137     # Checking for each of the files to create existence and, in case they exist,
138     # if they are writable
139
140     else:
141         s = f_name.split('.')[0]
142         if os.path.isfile(s + ".sig") and not os.access(s + ".sig", os.W_OK):
143             print "Can't create temporary file: *.bin. Aborting..."
144             sys.exit(2)
145         if os.path.isfile(s + ".key") and not os.access(s + ".key", os.W_OK):
146             print "Can't create temporary file: *.key. Aborting..."
```

```
147             sys.exit(3)
148         if os.path.isfile(s + ".bin") and not os.access(s + ".bin", os.W_OK):
149             print "Can't create temporary file: *.bin. Aborting..."
150             sys.exit(4)
151         if os.path.isfile(s + ".all") and not os.access(s + ".all", os.W_OK):
152             print "Can't create output file. Aborting..."
153             sys.exit(5)
154 
155     # Checking for public key's existence and access
156 
157     if not os.path.isfile(pubKey) or not os.access(pubKey, os.R_OK):
158         print "Invalid public key file. Aborting..."
159         sys.exit(6)
160 
161     # Checking for private key's existence and access
162 
163     if not os.path.isfile(priKey) or not os.access(priKey, os.R_OK):
164         print "Invalid private key file. Aborting..."
165         sys.exit(7)
166 
167 
168 # Gathering encrypting file name
169 
170 if len(sys.argv) > 2:
171     usage()
172 elif len(sys.argv) == 1:
173     print "File name:"
174     f_name = raw_input(">>> ")
175 else:
176     f_name = sys.argv[1]
177 
178 # Gathering names of keys
179 
180 if priKey == "":
181     print "Sender's private key file name:"
182     priKey = raw_input(">>> ")
183 if pubKey == "":
184     print "Receiver's public key file name:"
185     pubKey = raw_input(">>> ")
186 
187 # Running checks to files
188 
189 #checkFiles(f_name, pubKey, priKey)
190 
191 # Reading password if not assigned:
192 
193 if priPass == "":
194     print "Private key password (ENTER for empty value):"
195     priPass = raw_input(">>> ")
196 
197 # Ciphering file (and generating all auxiliary files)
198 
199 encipher(priKey, pubKey, f_name, priPass)
200 
```

```
201 # Generating output file and clean up
202
203 auxFilesZip(f_name.split('.')[0] + ".sig", f_name.split('.')[0] + # Continues on next line
204  + ".key", f_name.split('.')[0] + ".bin")
```

## decipher.py

```
import os
import sys
import 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 for faster usage

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

# File name to decrypt
f_name = ""

# Private key password:
priPass = ""


def usage():
    print "python decipher.py <file_name>"
    sys.exit(-1)


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

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

    # Reading public key 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, priPass):
    # Reading private key to decipher symmetric key used

```

```
52      keyPair = RSA.importKey(open(privKey_fname, "r").read(), passphrase=priPass)
53      keyDecipher = PKCS1_OAEP.new(keyPair)
54
55      # Reading iv and symmetric key used during encryption
56
57      f = open(f_name.split('.')[0] + ".key", "r")
58      iv = f.read(16)
59      k = keyDecipher.decrypt(f.read())
60
61      return k, iv
62
63
64  def decipher(keyA_fname, keyB_fname, f_name, priPass):
65      # Getting symmetric key used and iv value generated at encryption process
66
67      k, iv = keyReader(keyB_fname, f_name, priPass)
68
69      # Deciphering the initial information and saving it to file with no extension
70
71      keyDecipher = AES.new(k, AES.MODE_CFB, iv)
72      bin = open(f_name + ".bin", "rb").read()
73      f = open(f_name.split('.')[0], "wb")
74      f.write(keyDecipher.decrypt(bin))
75      f.close()
76
77      # Running a Signature verification
78
79      sigVerification(keyA_fname, f_name.split('.')[0])
80
81
82  def auxFilesUnzip(all):
83      # Opening the input file
84
85      f = zipfile.ZipFile(all + ".all", "r")
86
87      # Extracting all of its files
88
89      f.extractall()
90
91
92  def cleanUp(sig, key, bin, all):
93      # Removing all of the files created, except for the final deciphered file
94
95      os.remove(sig)
96      os.remove(key)
97      os.remove(bin)
98      os.remove(all)
99
100
101 def checkFiles(f_name, pubKey, priKey, first_run):
102     # Checking for decrypting file's existence and access, keys, aux and output files
103
104     if first_run:
105         # Checking for decrypting file's existence and access
```

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

```
160         priKey = raw_input(">>> ")
161
162     f_name = f_name.split('.')[0]
163
164     # Checking for *.all file and keys' files
165
166     checkFiles(f_name, pubKey, priKey, True)
167
168     # Unzipping all files
169
170     auxFilesUnzip(f_name)
171
172     # Checking for *.sig, *.key, *.bin files
173
174     checkFiles(f_name, pubKey, priKey, False)
175
176     # Reading password if not assigned
177
178     if priPass == "":
179         print "Private key password (ENTER for empty value):"
180         priPass = raw_input(">>> ")
181
182     # Deciphering file
183
184     decipher(pubKey, priKey, f_name, priPass)
185
186     # Cleaning all files but the deciphered file
187
188     cleanUp(f_name + ".sig", f_name + ".key", f_name + ".bin", f_name + ".all")
```

# Referências


[1] *The cryptography api, or how to keep a secret*, https://msdn.microsoft.com/en-us/library/ms867086.aspx, 2015.

[2] *Cryptography: a short tutorial*, http://simulator-tempur.tripod.com/publikasi/cryptography.html, 2015.

[3] *Crypto basics*, http://www.theamazingking.com/crypto-basics.php, 2015.

[4] *Linux security*, http://hamza-mega.blogspot.pt/2011_07_01_archive.html, 2015.

[5] *Chapter 4: fundamentals of cryptography (part03)*, http://ciscodocuments.blogspot.pt/2011/05/chapter-04-fundamentals-of-cryptography_702.html, 2015.

[6] *Digital signature*, http://en.wikipedia.org/wiki/Digital_signature, 2015.

[7] *Pycrypto*, https://www.dlitz.net/software/pycrypto/, 2015.

[8] *Self-synchronizing stream cipher (section "2. self-synchronizing stream encryption")*, http://www.ecrypt.eu.org/stream/ciphers/mosquito/mosquito.pdf, 2015.