Técnicas Criptográficas – Exercícios TP

Nuno Carvalho pg22815 Milton Nunes pg22797

18 de Junho de 2013

Capítulo 1

Grupo I

O primeiro passo para decifrar os quatro criptogramas é o cálculo do índice de coincidência. A partir desse valor, é possível distinguir entre cifras mono-alfabéticas e poli-alfabéticas, ou em que linguagem se encontra o texto. A fórmula usada para o cálculo do índice de coincidência é a seguinte:

$$\mathsf{IC} = \frac{\sum_{i=0}^{26} f_i(f_i - 1)}{n(n-1)}$$

Sendo n o tamanho do criptograma e f_i a frequência (número de ocorrências) da letra i do alfabeto de 26 letras. Com base nesta fórmula, obtivemos os seguintes valores, sabendo que o IC do inglês e do francês são, respectivamente, 0.0667 e 0.0746:

- IC Texto1: 0.0623
- IC Texto2: 0.0409
- IC Texto3: 0.0799
- IC Texto4: 0.0414

O texto 1 e o texto 3 parecem ser cifras de substituição mono-alfabética em inglês e francês, respectivamente. O texto 2 e 4 foram provavelmente cifradas com uma cifra de substituição poli-alfabética, nomeadamente com a cifra Vigenère. Ambos são textos em inglês.
O texto 1 foi o único que ainda não conseguimos decifrar. De seguida apresentamos os passos dados para cifrar os quatro textos.

1.1 Affine

A chave de uma cifra Affine são dois números a e b. A operação de cifração é $E(x) = (ax + b) \pmod{26}$ e a de decifração $D(x) = a^{-1}(x - b) \pmod{26}$, de tal forma que a é coprimo de m.
Para se atacar a cifra Affine, pode-se usar um método *bruteforce* juntamente com um teste do qui-quadrado para testar a validade da chave obtida. Ou seja, o valor mais baixo do qui-quadrado de todos os candidatos a chave é efectivamente a chave da cifra. Segue-se o código Java para atacar a cifra Affine.

Código 1.1: *Método para obter a chave utilizada na cifra Affine.*

```
public static int[] bruteForceChiSquared(String c, Language l) {
  double smallestChi = -1.0;
  int[] key = new int[2];
  for(int a : Affine.A) { # Affine.A contem todos os coprimos de 26
    for(int b = 0; b < 26; b++) {
      String isM = Affine.dec(a, b, c);
      double chi = Frequencies.chiSquaredStatistic(isM, l);
      if(smallestChi == -1.0) {
```

```
        smallestChi = chi;
        key[0] = a; key[1] = b;
      }
      if(chi < smallestChi) {
        smallestChi = chi;
        key[0] = a; key[1] = b;
      }
    }
  }
  return key;
}
```

O chave obtida para o texto 3 é $a = 19, b = 4$ e o texto limpo obtido é:
"o canada terre de nos aieux ton front est ceint de fleurons glorieux car ton bras sait porter lepee il sait porter la croix ton histoire est une epopee des plus brillants exploits et ta valeur de foi trempee protegera nos foyers et nos droits"

1.2 Vigenère

O processo para atacar o Vigenère já não foi tão simples quanto o usado para o Affine. Na nossa abordagem, o primeiro passo foi calcular o tamanho da chave, sabendo que este valor estaria entre dois números não muito grandes (4 a 7). Para o Vigenère, o IC é calculado de forma diferente: pega-se no criptograma, roda-se x posições para a direita e colocam-se ambos os textos em paralelo; o IC é o número de posições onde a mesma letra ocorre tanto no texto original como no texto rodado, a dividir pelo tamanho do criptograma. No nosso caso, o valor de x varia entre 4 e 7. No final, o x cujo IC seja o mais próximo do IC da linguagem é o que representa o tamanho da chave.
Tendo o tamanho d da chave, obter a chave torna-se bastante fácil. Basta para isso dividir o texto em d blocos com a forma: para todo o i tal que $1 \leq i \leq d$, $c_i, c_{i+d}, c_{i+2d}, \ldots$, e sabe-se que cada um destes blocos foi cifrado usando um *shift*. Para descobrir qual o melhor *shift* usa-se a seguinte forma, onde q_i representa a frequência de ocorrências da letra i do alfabeto de 26 letras:

$$\sum_{i=1}^{26}(q_i - f_i)^2$$

O valor mais pequeno corresponde ao melhor *shift*. No final, a chave é o conjunto dos d melhores *shifts*.
A chaves obtidas para os textos 2 e 4 foram, respectivamente, crypto e theory. Os textos limpos são:
"i learned how to calculate the amount of paper needed for a room when i was at school you multiply the square footage of the walls by the cubic contents of the floor and ceiling combined and double it you then allow half the total for openings such as windows and doors then you allow the other half for matching the pattern then you double the whole thing again to give a margin of error and then you order the paper"
"i grew up among slow talkers men in particular who dropped words a few at a time like beans in a hill and when i got to minneapolis where people took a lake wobegon comma to mean the end of a story i couldnt speak a whole sentence in company and was considered not too bright so i enrolled in a speech course taught by orville sand the founder of reflexive relaxology a self hypnotic technique that enabled a person to speak up to three hundred words per minute"

1.3 Mono-Substituição Alfabética

O processo utilizado para decifrar o criptograma 1 baseou-se na análise de ocorrência de cada letra do criptograma comparativamente à ocorrência das letras do alfabeto em textos em inglês.

Assim, partimos da letra com maior ocorrência, a letra *c*, e substituímos esta pelo *e*. De seguida procuramos a ocorrência dos trigramas mais comuns, acabados em *c*, no criptograma e substituímos pelo trigrama mais comum em inglês *the*. A seguir, substituímos a segunda letra mais frequente do criptograma, *g*, pela letra em inglês mais comum e ainda não utilizada, o *a*. Baseado no *a* procuramos os trigramas mais frequentes começados por esta letra de forma a encontrar ocorrências da palavra *and*. Neste momento, e depois de algumas tentativas de substituição sem sucesso e que não nos levaram a pista nenhuma, verificamos a ocorrência da palavra *dead* que pode indicar que estamos no caminho certo.
O resto do processo é semelhante ao descrito até este ponto. A partir dele foram também surgindo partes de palavras que intuitivamente indicam algumas letras a substituir, como *warden* para *garden*, de modo a encontrar o texto limpo. Assim, o texto limpo correspondente ao criptograma 1 é o seguinte:
"i may not be able to grow flowers but my garden produces just as many dead leaves old overshoes pieces of rope and bushels of dead grass as anybodys and today i bought a wheel barrow to help in clearing it up i have always loved and respected the wheel barrow it is the one wheeled vehicle of which i am perfect master"

Capítulo 2

Grupo II

O One-Time Pad é uma cifra que consiste em fazer o XOR do texto limpo com uma chave aleatória do mesmo tamanho que a mensagem a transmitir. Esta cifra, no caso de utilizar uma chave verdadeiramente aleatória e esta se mantiver em segredo torna o criptograma indecifrável, garantido desta forma segurança perfeita. Contudo, no caso de uma chave ser usada mais do que um vez, essa segurança perfeita deixa de existir e a mensagem pode então ser decifrada por um atacante.
Assim, se $C_1 = m_1 \oplus k$ e $C_2 = m_2 \oplus k$ e for feita a operação $C_3 = C_1 \oplus C_2$ temos $C_3 = m_1 \oplus m_2$. Desta forma realizamos esta operação para todos os pares possíveis de criptogramas dados. De assinalar que a referida operação de xor neste exercício corresponde à função decifrar.
Posto isto, temos um conjunto de novos criptogramas em que um deles deixa de ter uma distribuição de caracteres verdadeiramente aleatória, o que permite que através algumas técnicas usadas em criptoanálise seja possível identificar os criptogramas com maior probabilidade de corresponderem ao que se pretende encontrar. As técnicas que optámos por utilizar foram duas já referidas no guião anterior: Qui-quadrado e o Índice de Coincidência. Os resultados obtidos não foram tal como se esperava muito claros, pois trata-se de um xor entre dois textos limpos apenas composto por letras do alfabeto.
Análisamos os resultados obtidos para os 15 criptogramas que apresentam maior probabilidade nas duas técnicas de corresponderem ao criptograma que se pretende descobrir e apenas um é comum a ambas. O criptograma em questão corresponde ao xor dos criptogramas 6 e 18 e apresenta o valor de 878.14 na técnica do Qui-quadrado e de 0.041 relativamente ao Índice de Coincidência. Podemos então concluir que estes dois criptogramas foram muito provavelmente cifrados com a mesma chave.

Em relação ao segundo desafio, optámos, em primeiro lugar, por seguir as mesmas técnicas utilizados no desafio anterior. Contudo verificamos que todos os novos criptogramas apresentam valores semelhantes para o Índice de Coincidência. Posto isto, decidimos analisar a questão da frequência de ocorrência das letras do alfabeto nos criptogramas de forma a podermos encontrar indícios de qual será o criptograma que pretendemos identificar.
Depois de nos focarmos sobretudo na ocorrência média de cada letra nos criptogramas chegamos a uma conclusão que consideramos poder indicar o criptograma que representa o xor de dois textos limpos. A nossa teoria baseia-se em fazer a média de ocorrência de cada uma das letras do alfabeto no conjunto dos criptogramas, de seguida para cada criptograma é calculada a diferença de ocorrência de cada letra relativamente à média, a soma das diferenças de cada letra do alfabeto num criptograma dá-nos o que denominamos por desvio. Esta teoria baseia-se no facto de todos os criptogramas, exceptuando o que é composto pelo xor de dois textos limpos, apresentarem uma distribuição aleatória de letras, assim sendo o criptograma a encontrar será um dos que apresentará um desvio maior relativamente à média, pelo facto de não ter uma distribuição verdadeiramente aleatória.

Optámos então por utilizar esta técnica em conjunto com o Qui-quadrado e comparar os resultados obtidos. Assim, tal como no primeiro desafio, comparando os 15 resultados que apresentam maior probabilidade em cada uma das técnicas, alcançamos 3 resultados em comum: o xor dos criptogramas 5 e 7, 1 e 18 e por fim 17 e 18. Acreditamos então que os criptogramas cifrados com a mesma chave são muito provavelmente um destes três pares.

Capítulo 3

Grupo III

3.1 Objectivo

O objectivo principal deste grupo era usar o modo de cifras por blocos ECB em imagens, mais propriamente em imagens que são guardadas como uma matriz de pontos não comprimidas (*bitmap images*). No final, pretendia-se verificar que neste modo é possível retirar informação útil do criptograma, ou seja, é possível verificar certos padrões da imagem original no criptograma.

3.2 Bitmap image

Existem vários formatos de *bitmap images*, tais como GIF, PNG, TIFF ou JPEG. Mas nenhum destes formatos serve para o propósito deste grupo, pois são formatos comprimidos. Um outro formato mais conhecido em ambientes *Windows* é o BMP. Existem várias formas de representar um `BMP`, sendo que é comum a todas elas é o facto de no início do ficheiro estar o cabeçalho. De seguida apresenta-se o formato de um cabeçalho:

Posição	**Tamanho**	**Descrição**
0x0	2 bytes	Serve para identificar o tipo de imagem (normalmente é 0x42 que equivale a BMP)
0x2	4 bytes	Tamanho do ficheiro em bytes
0x6	2 bytes	Reservado; depende da aplicação que cria a imagem
0x8	2 bytes	Reservado; depende da aplicação que cria a imagem
0xA	4 bytes	Indica em que posição começa a matriz de pontos

No âmbito deste trabalho, a informação que mais nos interessa é a posição em que começa a matriz de pontos. Sendo assim, os passos a seguir são:

1. Abrir a imagem em modo hexadecimal
2. Obter a posição onde se encontra a matriz de pontos
3. Cifrar imagem original em modo ECB
4. Tendo a imagem cifrada, copiar o conteúdo do cabeçalho da imagem original para o criptograma, caso contrário a imagem não é reconhecida por nenhum leitor de imagens

3.3 Implementação

Para se obter o resultado das operações descritas anteriormente, são fundamentalmente precisas três ferramentas:

od Para abrir um ficheiro em modo hexadecimal, podendo assim manipulá-lo *byte* a *byte*

OpenSSL Para cifrar a imagem em modo ECB

dd Para copiar certas quantidades de *bytes* de um ficheiro para outro

Estas três ferramentas são combinadas entre si usando vários comandos Bash tais como `head`, `tail` ou o redireccionamento de *streams*.
Para se obter a posição inicial da matriz de pontos, usa-se o seguinte comando:

```
od -t x -An $IFILE  | head -n1 | cut -d' ' -f4,5 | tr -d ' ' | tail -c15 | cut -c1-8
```

De seguida explica-se o que cada um dos comandos faz:

od -t x -An lê todo o ficheiro `$IFILE` em modo hexadecimal

head -n1 apresenta apenas a primeira linha do ficheiro; essa primeira linha contém os primeiros 16 *bytes* do ficheiro, representados por 4 blocos de 4 *bytes*. Vamos usar os seguintes 16 *bytes* como exemplo a partir de agora: `687a4d42 00000010 007a0000 006c0000`

cut -d' ' -f4,5 para se obterem os terceiro e quarto blocos de *bytes*. No exemplo, obtém-se `007a0000 006c0000`

tr -d ' ' apagar os espaços em branco. Fica-se com `007a0000006c0000`

tail -c15 obtem os últimos 15 caractéres (ou seja, 7 *bytes* e um `\n`). Fica-se com `7a0000006c0000`

cut -c1-8 obtem os primeiros 8 caractéres, ou seja, 4 *bytes*. Fica-se com `7a000000`

O resultado obtido anteriormente está no formato *little endian*, ou seja, é necessário convertê-lo para *big endian*, para depois o converter para decimal. Eis como o fizemos no nosso script, estando em `$HEADERSIZE_LE` o resultado anterior `7a000000`:

```
HEADERSIZE=""
for (( i = 0; i < ${#HEADERSIZE_LE}; i+=2 )); do
  BYTE=${HEADERSIZE_LE:$i:2} # get substring starting in $i with 2 chars
  HEADERSIZE=$BYTE$HEADERSIZE
done
# convert header size to decimal
HEADERSIZE=$(echo "obase=10; ibase=16; $HEADERSIZE" | bc)
```

No exemplo anterior, o valor de `HEADERSIZE` é 122.
Nesta altura, já podemos cifrar a matriz de pontos. Para isso usamos o dd para obter e cifrar apenas os *bytes* necessários.
Em primeiro lugar, copiámos apenas o *header* da imagem original para o ficheiro de destino `OFILE`.

```
dd if="$IFILE" of="$OFILE" bs=1 count="$HEADERSIZE" conv=notrunc 2> /dev/null
```

De seguida, ciframos a matriz de pontos e redireccionámos o resultado para o ficheiro de destino. A matriz de pontos de pontos é lida directamente a partir do ficheiro original, usando também o dd, e o seu resultado é enviado através do stdin para o OpenSSL.

```
dd if="$IFILE" bs=1 skip="$HEADERSIZE" conv=notrunc 2> /dev/null | openssl enc -e "$CIPHER"
    >> "$OFILE"
```

De forma a automatizar este processo, criou-se um pequeno *script* em Bash. O Código 3.1 e a sua invocação é a seguinte:

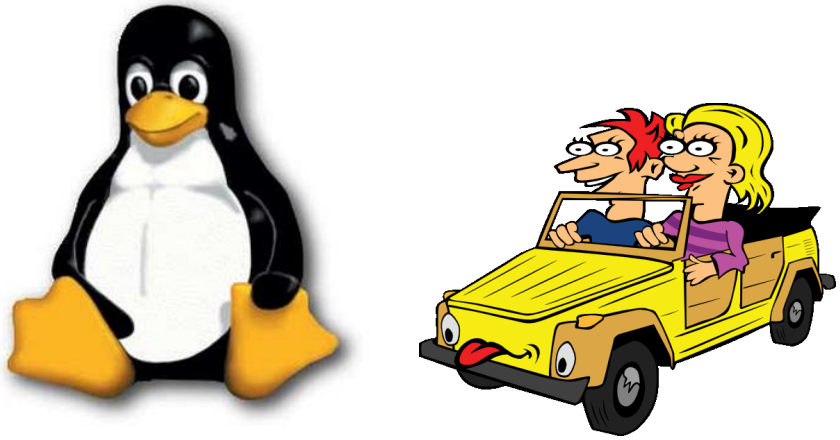
```
$ ./script infile.bmp outfile.bmp ecb-cipher
$ ./script car.bmp car-des.bmp -des-ecb
```

Código 3.1: *Script utilizado para cifrar imagens com uma cifra por blocos.*

```
#!/bin/bash

usage() {
  echo "Chamaste mal o script"
  exit
}

if [ -z "$1" ] || [ -z "$2" ] || [ -z "$3" ]; then
  usage
fi
IFILE="$1"
if [ ! -f "$IFILE" ]; then
  echo $IFILE" does not exist."
  exit
fi
OFILE="$2"
if [ -f "$OFILE" ]; then
  echo $OFILE" already exists. Cannot proceed"
  exit
fi
CIPHER="$3"

HEADERSIZE_LE=$(od -t x -An "$IFILE" | head -n1 | \
  cut -d' ' -f4,5 | tr -d ' ' | tail -c15 | cut -c1-8)
HEADERSIZE_LE="${HEADERSIZE_LE^^}"

HEADERSIZE=""
for (( i = 0; i < ${#HEADERSIZE_LE}; i+=2 )); do
  BYTE=${HEADERSIZE_LE:$i:2}
  HEADERSIZE=$BYTE$HEADERSIZE
done
HEADERSIZE=$(echo "obase=10; ibase=16; $HEADERSIZE" | bc)
echo "Header size of '"$IFILE"' is "$HEADERSIZE" bytes, located at position 0x0A"

dd if="$IFILE" of="$OFILE" bs=1 count="$HEADERSIZE" \
  conv=notrunc 2> /dev/null

dd if="$IFILE" bs=1 skip="$HEADERSIZE" \
  conv=notrunc 2> /dev/null | \
  openssl enc -e "$CIPHER" >> "$OFILE"
```

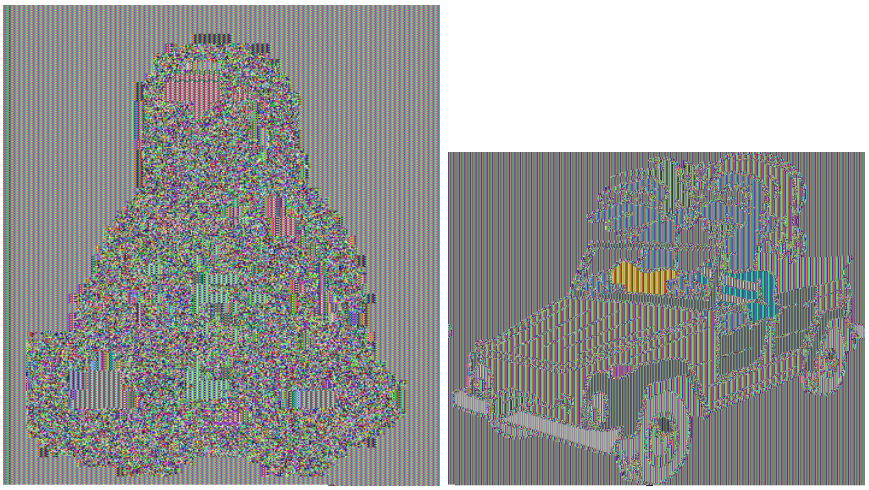
3.4 Exemplos e conclusões

Decidimos usar como exemplos as seguintes imagens: Vamos cifrar com DES e com AES para ver se existem diferenças claras entre ambas. Em todas elas, a *passphrase* utilizada é passphrase-quase-quase-boa-para-usar-com-o-modo-ecb.
Cifrando com o DES e AES, obtêm-se os seguintes resultados: Como se pode ver, não existe uma diferença clara entre o AES e o DES. Da mesma forma, também não existe nenhuma diferença entre o AES com 192 e 256 bits. No entanto, é claro que em todas as cifras, em conjunto com o ECB, é possível encontrar certos padrões da mensagem original.



(a) Famosa imagem do pinguim (b) Carro

Figura 3.4.1: *Exemplos utilizados*

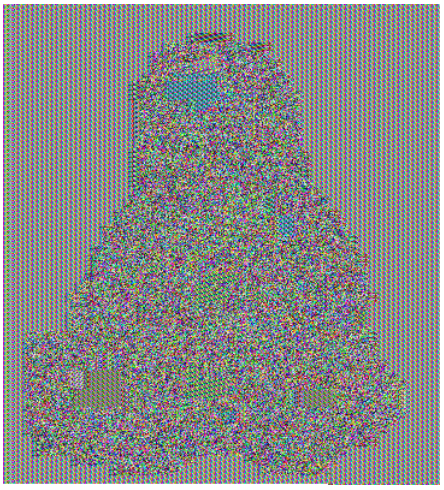


(a) Pinguim, utilizando o comando `./script penguin.bmp penguin-des.bmp -des-ecb`

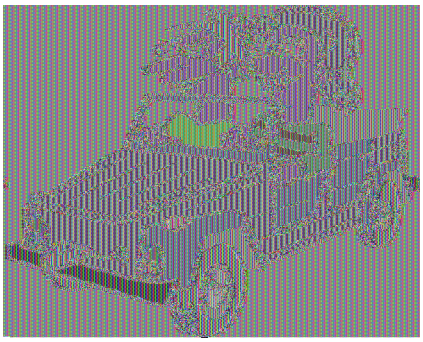
(b) Carro, utilizando o comando `./script car.bmp car-des.bmp -des-ecb`

Figura 3.4.2: *Cifração com DES em modo ECB*

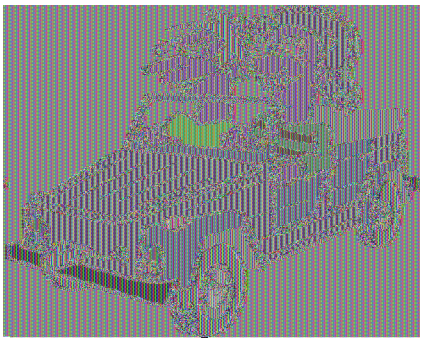
(a) Pinguim, utilizando o comando `./script penguin.bmp penguin-aes192.bmp -aes-192-ecb`



(b) Pinguim, utilizando o comando `./script penguin.bmp penguin-aes192.bmp -aes-192-ecb`



(c) Carro, utilizando o comando `./script car.bmp car-aes192.bmp -aes-256-ecb`



(d) Carro, utilizando o comando `./script car.bmp car-aes192.bmp -aes-256-ecb`

Figura 3.4.3: *Cifração com AES em modo ECB, com 192 e 256 bits*

Capítulo 4

Grupo IV

4.1 Implementação

A implementação do CBC-MAC em Sage é bastante simples. Usa-se como função pseudo-aleatória a cifra por blocos AES, através da biblioteca `pycrypto`. Vamos usar como valores *default* chaves de $n = 128$ bits e $\ell(n) = 5$, ou seja, mensagens de comprimento 640 bits (80 caractéres) divididas em blocos de 128 bits (16 caractéres). Considerem-se a mensagem $m = m_1 \,||\, m_2 \,||\, \cdots \,||\, m_{\ell(n)}$, com as *tags* $t = t_1 \,||\, t_2 \,||\, \cdots \,||\, t_{\ell(n)}$ ($\mathsf{Mac}_\mathsf{k}(m)$ apenas retorna o bloco $t_{\ell(n)}$).

4.2 Retornar todos os blocos

Retornar todos os blocos da *tag* torna o CBC-MAC inseguro pois torna-se bastante fácil falsificar MAC's. Na imagem 4.2.1 é demonstrado o mecanismo de falsificação de MAC's para uma mensagem em que é criada uma mensagem falsa $m' = (t_1 \oplus m_2) \,||\, m_3 \,||\, \cdots \,||\, m_\ell(n) \,||\, (t_{\ell(n)} \oplus m_1)$. A *tag* usada para verificação é apenas $t_{\ell(n)}$.

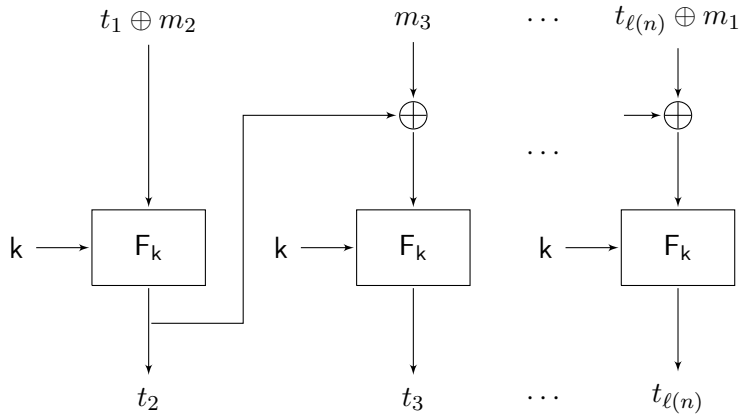


Figura 4.2.1: *Falsificação de MAC's quando todos os blocos da tag são retornados por* Mac.

4.3 Usar um IV aleatório

Usando um IV aleatório iv ainda se torna mais fácil falsificar mensagens (sendo que Mac apenas retorna o último bloco como *tag*). Basta para isso criar a nova mensagem $m' = (m_1 \oplus iv) \,||\, m_2 \,||\, \cdots \,||\, m_{\ell(n)}$ e a tag $t' = t$ tal que $\mathsf{Vrfy}_\mathsf{k}(m', t') = \top$.

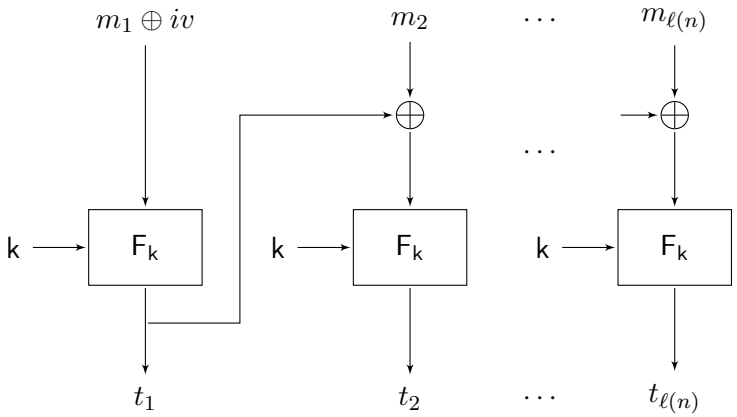


Figura 4.3.1: *Falsificação de MAC's quando se utiliza um IV aleatório.*

4.4 Implementação e execução Sage

Depois de implementado o CBC-MAC, criaram-se dois métodos na classe `CBCMAC` para falsificar mensagens:

ForgeMacRandomIV(msg, tag, iv) Retorna o par (msg', tag)

ForgeMacAllTags(msg, tag) Retorna o par (msg', tag')

Além disso, o método Mac tem uma variante insegura, bastante para isso incluir o argumento `secure=False` na sua invocação.
Segue-se o exemplo de uma sessão Sage em que se falsificam algumas mensagens.

Código 4.1: *Exemplo de falsificação de MACs em Sage*

```
sage: CBC = CBCMAC(128,5)
sage: key = MyKey(length=128)
sage: m = 'Isto e o CBC-MAC. Vamos forgar esta mensagem usando dois metodos. Wish me luck!!'
sage: iv = randomIV()
sage: t = CBC.Mac(key, m, iv=iv) # auth msg using random IV
sage: (m2, t2) = CBC.ForgeMacRandomIV(m, t, iv) # forge new msg and tag
sage: CBC.Vrfy(key, m, t, iv) # Vrfy using the random IV
True
sage: m2 ; CBC.Vrfy(key, m2, t2)
'<\xd0w\xc2#\xc6\xdaa[\x8eH\x94\x0el0\xb2. Vamos forgar esta mensagem usando dois metodos.
    Wish me luck!!'
True
sage: CBC.Vrfy(key, m, t) # Vrfy using IV = 0
False
sage: (iv,t) = CBC.Mac(key, m, secure=False) # auth msg and return all blocks
sage: (m3, t3) = CBC.ForgeMacAllTags(m, t) # forge new msg and tag
sage: m3 ; CBC.Vrfy(key, m3, t3) # Vrfy new forged msg
'\xad\x11q\xe0X IG\xf8\xd4\x1c\x10\x10\x85\xfaMsta mensagem usando dois metodos. Wish me
    luck!!\x0f\xdd0\x9fh=I\xdbB=\x8f"g\x88\x00\x96'
True
```

Capítulo 5

Grupo V

5.1 Congruências

Para resolver os sistemas de congruências podemos utilizar o Teorema Chinês dos Restos (CRT). O CRT serve exactamente para resolver os sistemas de congruências dados no enunciado, ou seja, tendo um conjunto $n_1, \ldots, n_k$ de inteiros mutualmente coprimos e dada uma sequência de inteiros $a_1, \ldots, a_k$, existe um inteiro x que resolve o seguinte sistema de congruências:

$$\begin{aligned} x &\equiv a_1 \pmod{n_1} \\ x &\equiv a_2 \pmod{n_2} \\ &\vdots \\ x &\equiv a_k \pmod{n_k} \end{aligned}$$

Para calcular o CRT precisamos de utilizar o algoritmo de Euclides estendido. Com o Euclides estendido (extendedgcd) podemos encontrar os inteiros r_i e s_i tal que $r_i n_i + s_i(N/n_i) = 1$. Ambos os algoritmos são apresentados em 5.1.1 e 5.1.2.

Optámos por implementar os algoritmos em Sage, sendo o 5.1.1 implementado como `extended_gcd(a, b)`

Algoritmo 5.1.1 Algoritmo de Euclides estendido

Input: Inteiros $a \geq b > 0$
Output: (g, X, Y) com $g = \gcd(a, b)$ e $g = Xa + Yb$

```
x ← 0, y ← 1
lastx ← 1, lasty ← 0
while b ≠ 0 do
    quot ← a div b
    (a, b) ← (b, a mod b)
    (x, lastx) ← (lastx − quot × x, x)
    (y, lasty) ← (lasty − quot × y, y)
end while
g ← (lastx × a_pre) + (lasty × b_pre)        ▷ a_pre e b_pre valores originais de a e b
return (g, lastx, lasty)
```

e o CRT como `solve_congruences(a, n)`. Note-se que os parâmetros `a` e `n` de `solve_congruences` deverão ser listas inteiros de igual comprimento.

Usando a função `solve_congruences`, é bastante simples resolver os dois primeiros exercícios deste grupo. Mas para resolver o segundo exercício, é preciso chegar à seguinte conclusão:

$$\begin{cases} 13x \equiv 4 \pmod{99} \\ 15x \equiv 56 \pmod{101} \end{cases} \Leftrightarrow \begin{cases} x \equiv 4/13 \pmod{99} \\ x \equiv 56/15 \pmod{101} \end{cases} \Leftrightarrow \begin{cases} x \equiv 46 \pmod{99} \\ x \equiv 98 \pmod{101} \end{cases}$$

Algoritmo 5.1.2 Teorema Chinês dos Restos

Input: $x \equiv a_i \pmod{n_i}$ para $i = 1, \ldots, k$ tal que n_i e N/n_i são coprimos
Output: x tal que resolve o sistema de congruências

```
N ← ∏_{i=1}^{k} n_i
x ← 0
for i = 1 to k do
    (g, r_i, s_i) ← extendedgcd(n_i, N/n_i)        ▷ verifica-se sempre g = 1
    e_i ← s_i × (N/n_i)
    x ← x + a_i × e_i
end for
return x mod N
```

em que $\mathrm{mod}(4/13, 99) = 46$ e $\mathrm{mod}(56/15, 101) = 98$.
Sendo assim, basta executar os seguintes comandos em Sage:

```
sage: solve_congruences([12, 9, 23], [25, 26, 27])
14387
sage: solve_congruences([46, 98], [99, 101])
7471
```

5.2 RSA

São nos dadas duas chaves públicas com módulos n pequenos e de fácil factorização. Logo, para decifrar os textos basta:

1. factorizar n para obter o os factores $p \cdot q = n$
2. calcular $\phi(n) = (p-1)(q-1)$
3. calcular expoente privado d tal que $d^{-1} \equiv e \pmod{\phi(n)}$. Neste momento, temos a chave privada e podemos decifrar os textos
4. decifrar os textos número a número, ou seja, decifra-se um número de cada vez para se obter o trio de letras original
5. concatenar os resultados obtidos para criar o texto original

5.2.1 Factorizar n

Dado que, como referido, a factorização é feita sobre n pequenos a função de factorização é bastante simples:

```
def factorN(n):
  for i in range(5, 500):
    for j in range(5, 500):
      if i*j == n:
        return (i,j)
  return -1
```

5.2.2 Cálculo expoente privado d

Para calcular o expoente privado d é, em primeiro lugar calculado o valor de $\phi(n)$:

```
phi = (p-1)*(q-1)
```

De assinalar que p e q são o resultado da função de factorização de n. De seguida, é então calculado o expoente privado:

```
d = Integer(e).inverse_mod(phi)
```

5.2.3 Decifração e descodificação

De forma a recuperar o texto limpo foram desenvolvidas as seguintes funções que permitem decifrar e descodificar os criptogramas dados:

```
def dec(sk, c):
  return power_mod(c, sk[1], sk[0])

def decText(sk, text):
  return ''.join(map(lambda x : decodeTri(dec(sk, x)) , text))

def intToChr(i):
  return abc[i]

def decodeTri(t):
  for i in xrange(0,26):
    for j in xrange(0,26):
      for k in xrange(0,26):
        if ((i*(26**2)) + (j*26) + k) == t:
          return str(intToChr(i) + intToChr(j) + intToChr(k))
```

Assim foi possível recuperar textos limpos dos criptogramas 1 e 2 indicados no enunciado, respectivamente:
"lake wobegon is mostly poor sandy soil and every spring the earth heaves up a new crop of rocks piles of rocks ten feet high in the corners of fields picked by generations of us monuments to our industry our ancestors chose the place tired from their long journey sad for having left the mother land behind and this place reminded them of there so they settled here forgetting that they had left there because the land wasnt so good so the new life turned out to be a lot like the old except the winters are worsez"
"i became involved in an argument about modern painting a subject upon which i am spectacularly ill informed however many of my friends can become heated and even violent on the subject and i enjoy their wrangles in a modest way i am an artist myself and i have some sympathy with the abstractionists although i have gone beyond them in my own approach to art i am a lumpist two or three decades ago it was quite fashionable to be a cubist and to draw every thing in cubes then there was a revolt by the vorticists who drew every thing in whirls we now have the abstractionists who paint every thing in a very abstracted manner but my own small works done on my telephone pad are composed of carefully shaded strangely shaped lumps with traces of cubism vorticism and abstractionism in them for those who possess the seeing eye as a lumpist i stand alone"

5.3 Calcular Símbolos de Jacobi

Os Símbolos de Jacobi, que consistem numa generalização dos Símbolos de Legendre, foram criados por Jacobi em 1837 e revelaram-se bastante úteis em vários ramos da teoria dos números, especialmente na teoria computacional dos números, nomeadamente no teste de primalidade e factorização de inteiros, e portanto também muito importantes em criptografia.
De forma a ser possível calcular os Símbolos de Jacobi e testar o programa para os *inputs* dados foi desenvolvida em *Sage* uma função recursiva que implementa as quatro propriedades pretendidas. Abaixo é apresentado o código desenvolvido:

```
def jacobi(m,n):
  if not is_odd(n):
    print "n not odd"
    return
  if m == 0:
    return 0
```

```
  if m == 1:
    return 1
  #propriedade a)
  if m>=n:
    return jacobi(m%n, n)
  #propriedade b)
  if m == 2:
    nm8 = n%8
    if nm8 == 1.mod(8) or nm8 == (-1).mod(8):
      return 1
    elif nm8 == 3.mod(8) or nm8 == (-3).mod(8):
      return -1
  #propriedade c)
  f = list(factor(m))
  if len(f) == 2 and f[1][1] == 1 and f[0][0] == 2:
    return ((jacobi(2, n)**f[0][1]))*jacobi(f[0][1], n)
  if m%2 == 0:
    return jacobi(2,n)*jacobi(m//2,n)
  #propriedade d)
  if m%4 == 3 and n%4 == 3 and is_odd(m):
    return -jacobi(n,m)
  if is_odd(m):
    return jacobi(n,m)
```

Para os *inputs* fornecidos, os resultados obtidos pelo programa desenvolvido são os seguintes:

- $(\frac{610}{987}) = -1$
- $(\frac{20694}{1987}) = 1$
- $(\frac{1234567}{11111111}) = -1$

5.4 Encontrar bases para as quais n é Pseudo-Primo de Euler

No último exercício do guião V pretende-se desenvolver um programa que encontre as bases b, para as quais n dado é um pseudo-primo de Euler. Para se determinar se n é um pseudo-primo de Euler na base b, deve-se verificar se o Símbolo de Jacobi $(\frac{b}{n})$ é igual a $b^{(n-1)/2}$ (mod n). Desta forma a solução consiste em testar para sucessivos b se $(\frac{b}{n})$ é um Símbolo de Jacobi, se simultaneamente verifica a condição descrita e ainda se o máximo divisor comum entre b e n é igual a 1. De seguida apresenta-se o código desenvolvido que permite resolver o exercício:

```
def eulerPseudoPrime(b,n):
  j = jacobi(b, n) % n
  p = pow(b, (n-1)//2) % n
  if j == p and gcd(b,n) == 1:
    print (b,n), n, "is an euler pseudoprime to base", b
    return true
  else:
    return false

def eulerBase(n):
  i = 2
  ct = 0
  while ct <= 5:
    if eulerPseudoPrime(i,n):
      ct += 1
    i += 1
  return
```

Posto isto, os resultados obtidos foram os seguintes:

- 837 é um Pseudo-Primo de Euler nas bases 836, 838, 1673, 1675, 2510, 2512.
- 851 é um Pseudo-Primo de Euler nas bases 850, 852, 1071, 1703, 2552 e 2554.
- 1189 é um Pseudo-Primo de Euler nas bases 204, 278, 360, 829, 911 e 985.

De assinalar que optámos por apresentar apenas 5 bases para cada n dado, contudo existem muitas mais bases para as quais n é um Pseudo-Primo de Euler.

Capítulo 6

Grupo VI

Neste grupo pretendia-se decifrar um criptograma obtido com a cifra ElGamal. Como já temos acesso às chaves pública e privada, basta implementar a função Dec, e no final aplicar a função de descodificação definida em 5.2.3. Optámos mais uma vez pela utilização do Sage. De seguida apresentam-se as funções Enc e Dec:

```
def enc(pk, p, g, msg):
  k = floor( 1 + (p-2) * random())
  return ( Mod(g, p)**k, msg * Mod(pk**k, p))
def dec(sk, p, g, c1, c2):
  return Mod(c2, p) * Mod(c1, p)**(-sk)
```

Estas funções retornam, respectivamente, $(c_1, c_2) = (g^k, m \cdot h^k)$ e $m = c_2 \cdot c_1^{-x}$.
Tendo estas duas funções, bastou-nos implementar outro método para decifrar o texto dado no enunciado. Essa função pode ser codificada numa única linha com:

```
def decText(sk, p, g, text):
  return ''.join(map(lambda (c1,c2) : (decodeTri(dec(sk, p, g, c1, c2))), text))
```

O resultado final obtido foi:
"she stands up in the garden where she has been working and looks into the distance she has sensed a change in the weather there is another gust of wind a buckle of noise in the air and the tall cypresses sway she turns and moves up hill towards the house climbing over a low wall feeling the first drops of rain on her bare arms she crosses the loggia and quickly enters the house"