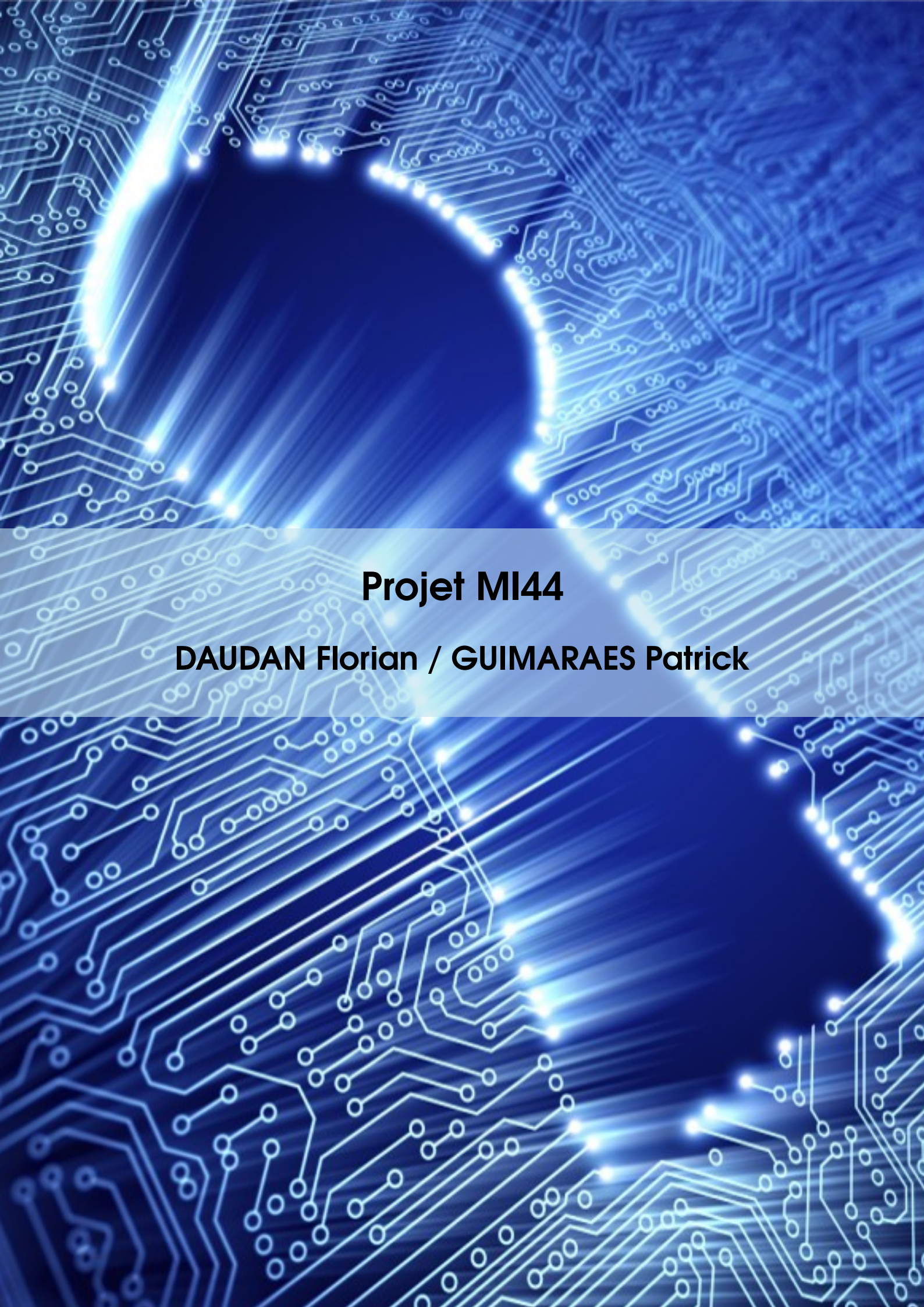
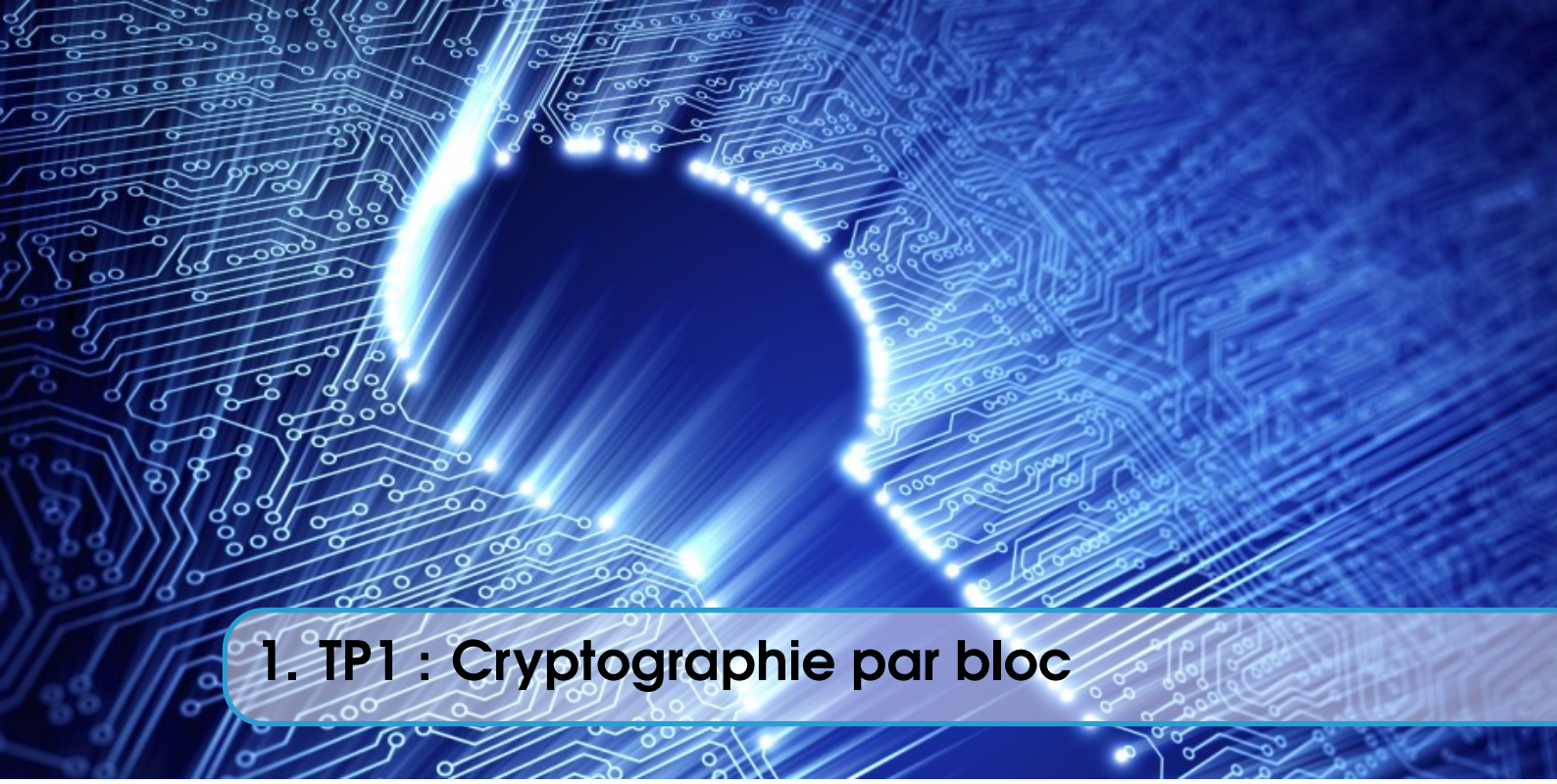


Projet MI44

DAUDAN Florian / GUIMARAES Patrick

Sommaire



1. TP1 : Cryptographie par bloc

1.1 Introduction

Afin de gérer la conversion d'un caractère en un nombre binaire, nous avons mis au point un dictionnaire. En effet, chaque caractère (A,B,C...) correspond à un nombre (A=0, B=1,C=2,...) et le dictionnaire va donc permettre d'attribuer plus rapidement un nombre à un caractère. Et dans le cadre inverse, un second dictionnaire va permettre de passer d'un nombre au caractère correspondant.

Nous avons 32 caractères à gérer ; toutes les lettres de l'alphabet latin ainsi que les caractères espace, point, virgule, apostrophe, point d'exclamation et point d'interrogation. Ce qui nous fait un total de 32 caractère codable sur 5 bits en binaire. Notons la constante NB_BIT permettant d'ajuster le codage des caractères. En outre, elle permet d'égaliser l'écriture de tous les nombres binaires qui auront, de surcroît, le même nombre de bit. Par exemple, B (=1) sera écrit 00001.

1.2 Binaire

1.2.1 Chiffrer

La fonction **encrypt_binaire(lettre)** reçoit en paramètre un caractère et retourne directement le nombre binaire associé. En prime abord, elle demande au dictionnaire de lui retourner la valeur numérique en base 10 du caractère . Puis ensuite elle convertit cette valeur en binaire qui sera elle-même envoyée à la fonction **toilettage_binaire(binaire)** qui, comme son nom l'indique, va se charger d'écrire le binaire sous le bon nombre de bit(cf. constante NB_BIT). Une fois ces opérations terminées, le caractère en binaire sera renvoyé.

Dans la même optique, nous avons conçu la fonction **encrypt_mot_binaire(mot)** qui va recevoir un mot, ou une phrase, et renvoyer directement le mot/phrase converti(e).

```python
dictionnaire = {'A': 0,'B':1,'C':2,'D':3,'E':4,
'F':5,'G':6,'H':7,'I':8,'J':9,
'K':10,'L':11,'M':12,'N':13,'O':14,
'P':15,'Q':16,'R':17,'S':18,'T':19,
'U':20,'V':21,'W':22,'X':23,'Y':24,'Z':25,' ':26,'.':27,',':28,'\'':29,'!':30,'?':31}

dic_bin = {dictionnaire[k]:k for k in dictionnaire.keys()}

def toilettage_binaire(binaire):
    taille = len(binaire) - 2
    cypherText=''
    if(taille < NB_BIT):
        for j in range(NB_BIT-taille):
            cypherText+='0'

    for i in range(2,taille+2):
        cypherText+= binaire[i]

    return cypherText

def encrypt_binaire(lettre):
    return toilettage_binaire(bin(dictionnaire[lettre]))
def encrypt_mot_binaire(mot):
    mot_bin = ''
    for i in mot:
        mot_bin += encrypt_binaire(i)
    return mot_bin
```

Figure 1.1: Fonctions de conversion en binaire

1.2.2 Déchiffrer

Finalement, pour réaliser l'opération inverse, c'est à dire de convertir un binaire en caractère, nous avons écrit la fonction **decrypt_binaire()** qui reçoit un nombre binaire. Cette méthode va en premier lieu convertir le nombre en décimal. Puis appeler le second dictionnaire pour lui demander quel caractère est associé au nombre. Enfin, elle va retourner le caractère trouvé. De façon similaire à encryp_mot_binaire(), nous avons écrit **decrypt_mot_binaire()** qui à partir d'une phrase ou un mot binaire de taille % NB_BIT, va retourner les caractères correspondant.

```python
def decrypt_binaire(binaire):
    #return dictionnaire[int(binaire,2)]
    dic_bin[(int(binaire,2))]
    return dic_bin[(int(binaire,2))]

def decrypt_mot_binaire(mot_bin):
    mot=''
    for i in range(0,len(mot_bin),5):
        mot+=decrypt_binaire(mot_bin[:5])
        mot_bin=mot_bin[5:]

    return mot
```

Figure 1.2: Fonctions de conversion en caractère

1.2.3 Ou Exclusif

La fonction **ouExclusif(A,B)** réalise le ou exclusif en binaire, elle prend en compte deux binaires.

On parcourt le premier binaire bit à bit, on compare le bit avec le bit de l'autre binaire. S'ils sont égals et que l'un d'eux est égal à 1, on retourne 0. Sinon si l'un deux est égal à 1 on retourne 1. Dans les autres cas on retourne 0. Une fois terminé, nous avons le résultat de l'addition binaire.

```
def ouExclusif(A,B):
    resultat = ''
    for i in range(0,len(A)):
        if (A[i] == B[i] and B[i]=='1'):
            resultat += '0'
        elif A[i] == '1' or B[i]=='1':
            resultat += '1'
        else:
            resultat += '0'
    return resultat
```

Figure 1.3: ou exclusif(XOR)

1.3 Function

La fonction **function()** prend en paramètre un message msg de 2 caractères ainsi qu'une clé Key de 2 caractères convertie en binaire. On converti le message en binaire puis on réalise le décalage binaire. Ce dernier se réalise de cette manière : on crée une nouvelle variable, on lui ajoute tous les bits du message binaire en commençant par le second bit. Une fois arrivé à la fin, on lui ajoute le premier bit du message réalisant ainsi le décalage binaire gauche.

```
def function(msg,Key):
    #Decalage binaire
    msg_bin = encrypt_binaire(msg[0]) + encrypt_binaire(msg[1])
    #print msg_bin
    msg_fin=''
    for i in range(1,len(msg_bin)):
        msg_fin+= msg_bin[i]
    msg_fin+=msg_bin[0]
    #print msg_fin
    #ou exclusif
    return ouExclusif(msg_fin,Key)
```

Figure 1.4: Fonction de Feistel

1.4 Réseau de Feistel

1.4.1 Chiffrer

Le réseau de Feistel est symbolisé par la fonction **encrypt_feistel()** qui à partir d'un bloc et d'une clé, va chiffrer un texte. Tout d'abord, nous divisons le bloc en deux, le bloc gauche et le bloc droit. Pour chaque caractère de la clé (i.e. 4 boucles), nous effectuons les opérations suivantes. On réalise la copie du bloc droit, on prend les deux premier caractères de la clé et on les stocke, on décale la clé de 1 caractère, on transforme le bloc droit à travers la fonction function() à laquelle on lui envoie le bloc droit lui-même ainsi que la clé convertie en binaire. Une fois cela réalisé, on effectue le ou exclusif avec le nouveau bloc droit (résultat de la fonction function()) et la partie gauche. Le résultat obtenu est ainsi converti en caractère, on stocke la copie du bloc droit dans le bloc gauche, et on réitère ce traitement. Nous obtenons finalement notre bloc chiffré à partir de la clé.

```
def encrypt_feistel(bloc,Key):
    G = bloc[:2]
    D = bloc[2:]
    #print "encrypt"
    for i in range(4):
        #copie du bloc de droite
        D2 = D
        #la cle prend les deux premiers caracteres
        Key_feistel = Key[:2]
        #La cle globale est décalée de 1 caractere
        Key = Key[1:]+Key[:1]
        #On chiffre la partie droite
        #print "key:"+Key_feistel
        print G+" : "+D +" ////////// "+Key_feistel

        D = function(D,encrypt_mot_binaire(Key_feistel))

        #Maintenant, il faut réaliser le ou exclusif
        D = decrypt_mot_binaire(ouExclusif(D,encrypt_mot_binaire(G)))
        G = D2

        print G+" : "+D
        print Key_feistel

    return G+D
```

Figure 1.5: Réseau de Feistel pour chiffrer

1.4.2 Déchiffrer

Dans l'optique inverse, nous avons conçu **decrypt_Feistel**(), qui à partir d'un bloc chiffré et d'une clé va déchiffrer le message. L'opération est similaire à encrypt_Feistel(), nous divisons notre bloc de 4 caractères en deux blocs : gauche et droite. À la différence de encrypt_Feistel(), nous parcourons la clé à l'envers (i.e. « XK » → « CX » → « XC » → « KX ») mais nous effectuons les mêmes opérations : le bloc droit passe par function(), puis le résultat obtenu subit un ou exclusif avec le bloc gauche, et finalement on inverse le bloc gauche avec le bloc droit.

```
def decrypt_feistel(bloc,Key):
    D = bloc[:2]
    G = bloc[2:]
    Key = Key[3:]+Key[:3]
    #print Key
    #print "Decrypt"
    for i in range(4):
        #copie du bloc de droite
        D2 = D
        #la cle prend les deux premiers caracteres
        Key_feistel = Key[:2]
        #La cle globale est décalée de 1 caractere
        Key = Key[3:]+Key[:3]
        #print Key
        #On chiffre la partie droite
        #print "key:"+Key_feistel
        D = function(D,encrypt_mot_binaire(Key_feistel))

        #Maintenant, il faut réaliser le ou exclusif
        D = decrypt_mot_binaire(ouExclusif(D,encrypt_mot_binaire(G)))
        G = D2

    return D+G
```

Figure 1.6: Déchiffrement par Feistel

1.4.3 Exemple

Afin d'illustrer un exemple concret de fonctionnement du réseau de Feistel, nous avons procédé à une fonction main() permettant de simuler l'échange d'un texte chiffré.

Nous choisissons un mot arbitraire « AAAA??BB » et une clé « KXCX ». Nous nous assurons que les lettres soient en lettres majuscules (mot.upper()). Nous « préparons » le texte de sorte que sa taille soit un multiple de 4 par le fonction tailleTexteMod4() qui va ajouter des espaces à la fin du mot. Par la suite, nous chiffrons bloc par bloc le texte (taille de 4 caractères). Nous obtenons ainsi « MYMWEQG, ». Et en effectuant l'opération inverse par la fonction decrypt_feistel() nous ré-obtenons « AAAA??BB ».

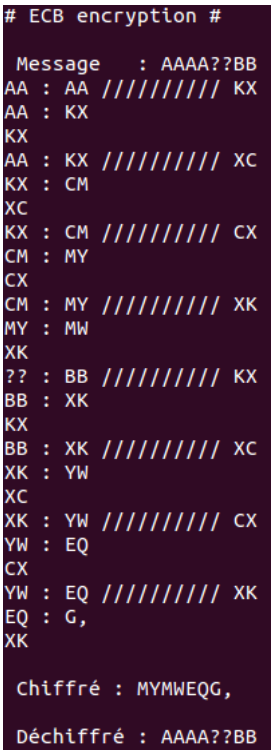


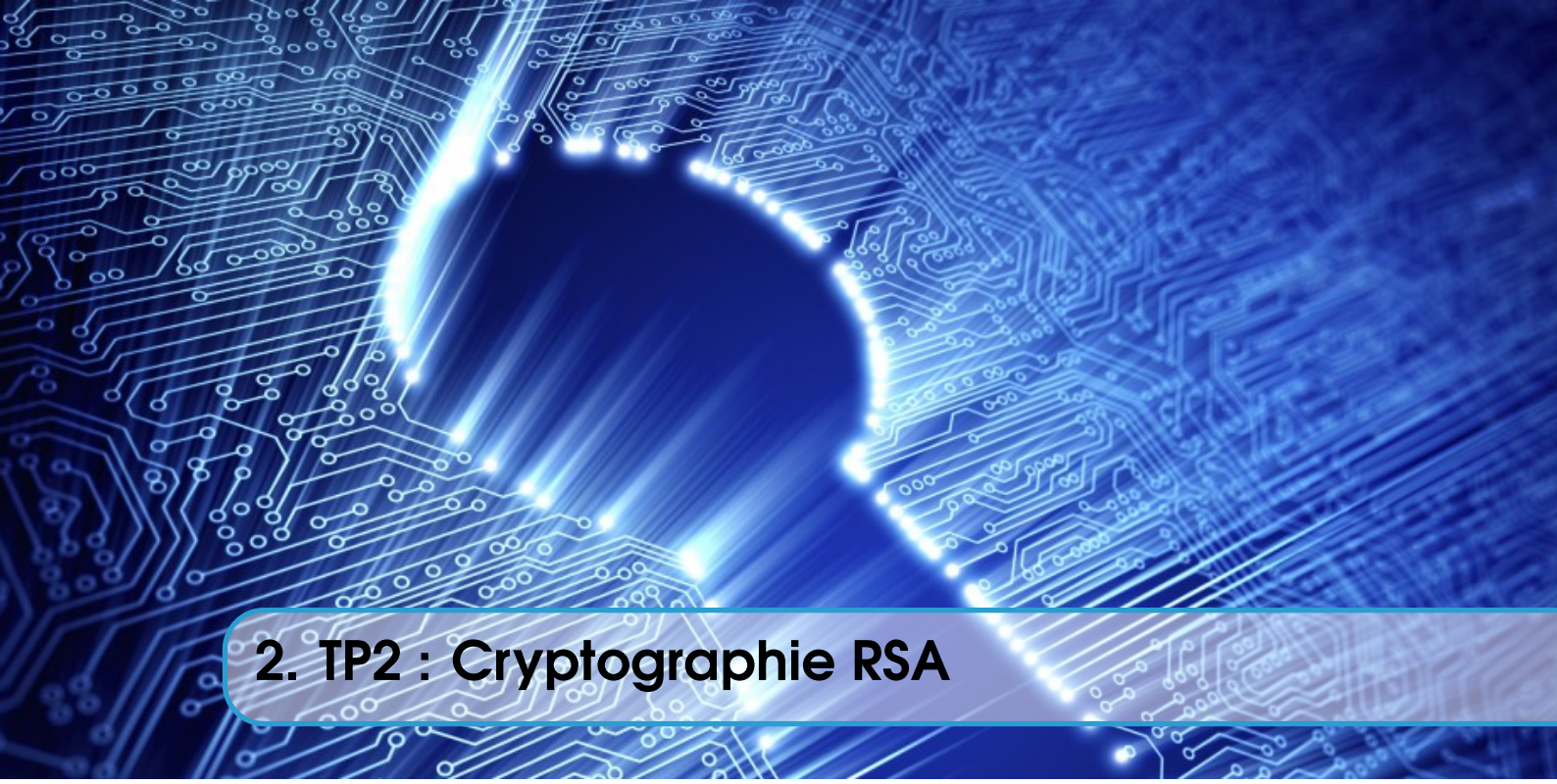
Figure 1.7: Exemple de chiffrement

1.5 Cipher-bloc Chaining

Maintenant intéressons nous au chiffrement et déchiffrement en mode CBC. Tout d'abord, nous avons la fonction vecteur_initialisation() qui va prendre au hasard 4 caractères dans le dictionnaire pour former la clé. Une fois cette clé formée, nous effectuons un ou exclusif avec le texte en clair. Avec le resultat, nous opérons avec la fonction du réseau de feistel. Une fois l'opération terminée, nous recommençons mais en prenant cette fois-ci le résultat de la fonction et le texte en clair.

```python
def vecteur_initialisation():
    length = len(dictionnaire) -1
    # 4 lettre random
    return dic_bin[random.randint(0,length)]+dic_bin[random.randint(0,length)]+dic_bin[random.randi

def encrypt_CBC(mot,Key,VI):
    key = encrypt_mot_binaire(Key)
    res=''
    #print "encrypt"
    VI = encrypt_mot_binaire(VI)

    for i in range (0,len(mot),4):
        #passage des 4 premiers caractères en binaire
        plain_bin = encrypt_mot_binaire(mot[:4])
        #plain text XOR Vecteur d'initialisation
        cypher_text = ouExclusif(plain_bin[:20],VI)
        #passage en caractère
        cypher_text = decrypt_mot_binaire(cypher_text)
        #print cypher_text
        cypher_text = encrypt_feistel(cypher_text,Key)

        res += cypher_text

        VI = encrypt_mot_binaire(cypher_text)

        #supression des 4 premier caractères déjà chiffrés
        mot = mot[4:]

    return res
```

Figure 1.8: CBC



2. TP2 : Cryptographie RSA

2.1 Introduction

Nous avons simulé un échange de messages avec le protocole RSA entre deux identités à l'aide de 3 programmes. Les programmes A et B simulent Alice et Bob, deux personnes souhaitant communiquer de manière secrète, et finalement le programme C représentant l'autorité décernant les clés aux différentes personnes. Nous allons nous intéresser dans un premier temps au programme C qui génère les différentes clés, puis nous étudierons le fonctionnement de l'échange de messages.

2.2 Le programme C

La fonction centrale de ce programme est la fonction **generation()** permettant d'éditer l'ensemble des clés
Tout d'abord, nous choisissons 2 nombres premiers aléatoires à l'aide de la fonction **get_random()** qui va tirer au hasard un nombre puis va vérifier s'il est premier grâce à la méthode **lucas_lehmer()**. Cette dernière fonctionne ainsi : elle reçoit en paramètre un nombre à tester ainsi qu'un nombre de test qui par défaut est 7. Ce nombre correspond aux nombres de boucles de test que nous réalisons pour tester la primalité du nombre. Tout d'abord si le nombre à tester, que nous appellerons n, est inférieur à 1 le test retourne faux. Ainsi que si n est égal à 2 le test retourne vrai. Nous arrivons dans la boucle principale où le traitement sera réitéré plusieurs fois. En premier lieu nous choisissons un nombre aléatoire entre 3 (car nous excluons 1 et 2) et n-2 . Nous réalisons l'exponentiation modulaire de aˆn-1 mod n. Si le résultat est différent de 1, on retourne faux. Sinon nous établissons la liste des primitifs de n(cf plus bas la description de la fonction **primes()**). Nous parcourons cette liste et à chaque primitif nous réalisons aˆ(n-1)/q mod n. Si le résultat est égal à 1, nous retournons faux. A la fin de la boucle, si le test n'a retourné aucune valeur, nous finissons par le retour de vrai, car nous avons trouvé que n est un nombre entier.

Fonction **primes()**. Cette fonction permet de trouver la liste des entiers primifs d'un nombre entier. Tout d'abord, nous allouons un tableau, et un indice de départ d égal à 2 (premier nombre premier). Nous entrons ensuite dans la boucle while principale où la condition est tant que $d^2 \leq n$, avec n le nombre premier. Dans cette boucle, nous entrons dans une sous-boucle while où la condition est (n % d) == 0. Si la condition est respectée, on ajoute d à la liste. Puis on divise n par d, en gardant la partie entière. On sort de la sous-boucle while , on incrémente d et on recommence. Lors de la sortie de la boucle principale while, si n>1 on ajoute n à la liste des entiers primitifs, dans tous les cas on retourne la liste.

Retournons à la génération des clés, nous avons grâce aux fonctions précédentes, 2 nombres entiers que nous nommerons p et q. Nous calculons n = p*q puis phi(n) = (p-1)*(q-1). Maintenant nous cherchons e, qui est un nombre premier aléatoire entre 1 et phi(n). Intéressons nous à la fonction e_aleatoire(phi). Cette fonction tire un nombre premier aléatoire entre 2 et phi-1 uniquement si ce nombre ne possède aucun diviseur commun avec phi.
Maintenant que nous possédons la clé privée (i.e. e), nous pouvons éditer la clé publique .
Pour cela nous devons chercher d, l'inverse modulaire de phi modulo n. Nous faisons appel à la fonction **inverse_modulaire(e,phi)**. Cette fonction utilise l'algorithme suivant :

- les constantes u0 = 1, u1 = 0, v0=0 et v1 = 1
- r0 = (e * u0) + (phi * v0)
- r1 = (e * u1) + (phi * v1)
- Tant que r1 est différent de 0 nous faisons
  - q = r0 / r1
  - r = r0
  - u = u0
  - v = v0
  - on affecte à r0 la valeur de r1, ainsi que u0 reçoit u1 et v0 vaut v1.
  - r1 = r – (q * r1)
  - u1 = u – (q * u1)
  - v1 = v – (q*v1)
- Une fois terminé, nous incrémentons b à u0 afin qu'il devienne positif
- On retourne u, l'inverse modulaire de phi.

La fonction génération retourne le couple e,d,n qui comporte clé publique/privée pour le futur utilisateur.

2.3 Le programme A et B

Maintenons nous allons étudier le fonctionnement des différents opérateurs dans l'échange de leur message. Nommons les A et B.

2.3.1 Etape 1

A souhaite envoyer le message suivant : « AB ?! » vers B. Pour cela il fait appel à la fonction **envoie_message_to_B(message)** qui va encoder le message selon le protocole RSA. C'est à dire que nous allons convertir chaque caractère en nombre. Pour cela nous réutilisons le dictionnaire du TP précédent. Maintenant que nous avons un nombre, nous effectuons le calcul suivant :
soit M le nombre à chiffrer et (d,n) la clé publique de B
C = $M\hat{d}$ mod n

Nota Bene : à des fins pratiques, nous utilisons la fonction d'exponentiation modulaire rapide pour limiter l'utilisation du processeur et pour une résultat plus rapide.

Une fois notre nombre chiffré, nous recommençons avec les autres caractères du message. Ce qui nous donne une liste de nombres. Nous envoyons cette liste à B qui va commencer le déchiffrement. Pour cela, B utilise **decrypt_message_from_A(msg_crypt)** . Cette fonction va prendre chaque nombre dans la liste reçue et le déchiffrer en utilisant la formule suivante :

- soit C le message chiffré, (e,n) la clé privée de B
- M = C^e mod n

2.3.2 Etape 2

Une fois le message lu, B compare si le message reçu est égal à « AB?! ». Si tel est le cas, B prépare le message « AB OK » pour A. Il le chiffre de la même manière que A en faisant C = M$\hat{d}$ mod n avec (d,n) la clé publique de A.

A reçoit le message, le déchiffre et le compare à « AB OK ». S'ils sont similaires, on passe à l'étape 3

2.3.3 Etape 3

A génère 4 caractères aléatoires à partir de son dictionnaire. Puis il les envoie à B en utilisant le protocole RSA. B les déchiffre, puis les chiffre pour les renvoyer à A. A son tour, A déchiffre pour vérifier que le message a bien été reçu.

2.3.4 Etape 4

Puisque A et B possède le même mot de passe, on va pouvoir effectuer les opérations suivantes. On effectue le ou exclusif de « AB OK » et du mot de passe. Puisque que le message possède une taille plus grande que le mot de passe (4 caractères contre 5), nous nous sommes adaptés à la fonction ouExclusif(). En effet nous effectuons un ou exclusif uniquement avec les caractères nécessaires en partant de la droite du message.
Exemple :

mAB OK → message
ABCD → mot de passe
Nous alignons à droite le mot de passe

AB OK
ABCD
puis nous découpons le messages
B OK
ABCD
Maintenant nous effectuons un ou exclusif de B OK et ABCD.
Puisque le XOR de A et 0 donne A, nous prenons A et ajoutons le résultat du XOR à la suite.

2.3.5 Etape 5

Avec notre mot de passe ainsi généré (la clé pour le réseau de Feitel), nous pouvons commencer le chiffrement. Nous avons adapté la fonction **encrypt_feistel** en **encrypt_feistel_2** , elle permet de réaliser le chiffrement avec des tailles différentes de 4 caractères. A chiffre son message puis l'envoie à B qui va le déchiffrer avec le même mot de passe(clé).

```
def encrypt_feistel_2(bloc,key):
    key_blocs = []
    for i in range(0,len(key)):
        if(i<len(key)-1):
            if(i<len(key)-2):
                key_blocs.append(str(key[i])+str(key[i+1])+str(key[i+2]))
            else:
                key_blocs.append(str(key[i])+str(key[i+1])+str(key[0]))
        else:
            key_blocs.append(str(key[i])+str(key[0])+str(key[1]))

    while len(bloc) % (len(key_blocs[0])) > 0:
        tmp  = "A"
        tmp += bloc
        bloc = tmp

    print bloc
    G = bloc[:len(key_blocs[0])]
    D = bloc[len(key_blocs[0]):]
    for i in range(len(key_blocs)):
        print key_blocs[i]
        D2 = D
        D = function(D,encrypt_mot_binaire(key_blocs[i]))
        D = decrypt_mot_binaire(ouExclusif(D,encrypt_mot_binaire(G)))
        G = D2
    return D+G
```

Figure 2.1: Nouvelle version du réseau de Feistel