Travaux d'Études et de Recherche: Annexes

Delbot François Rovedakis Stéphane

13 mai 2005

Table des matières

Chapitre 1

Table des fréquences des caractères

Ces tables permettent de casser très rapidement la majorité des systèmes de chiffrement ne reposant que sur une simple substitution de caractères.

1.1 Dans la langue française

A	0.083944
B	0.007669
C	0.033297
D	0.040699
E	0.145037
F	0.012109
G	0.009495
H	0.007973
I	0.081828
J	0.006377
K	0.000638
L	0.058405
M	0.029355
N	0.075570
O	0.053669
P	0.032087
Q	0.012613
R	0.070209
S	0.080091
T	0.074775
U	0.059808
V	0.015791
W	0.000067
X	0.004098
Y	0.003155
Z	0.001240

1.2 Dans la langue anglaise

A	8.04 %
B	1.54 %
C	3.06 %
D	3.99 %
E	12.51 %
F	2.30 %
G	1.96 %
H	5.49 %
I	7.26 %
J	0.16 %
K	0.67 %
L	4.14 %
M	2.53 %
N	7.09 %
O	7.60 %
P	2.00 %
Q	0.11 %
R	6.12 %
S	6.54 %
T	9.25 %
U	2.71 %
V	0.99 %
W	1.92 %
X	0.19 %
Y	1.73 %
Z	0.09 %

Chapitre 2

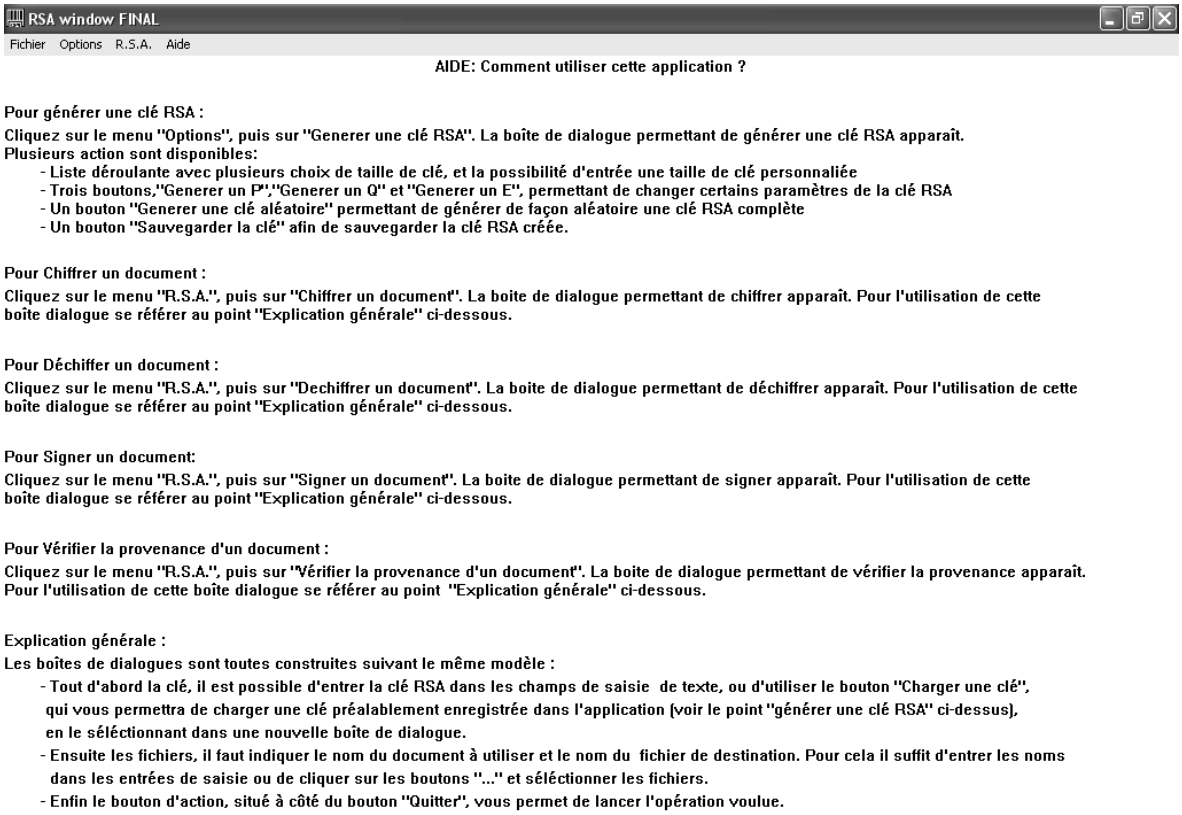
Document utilisateur de l'application RSA

Document utilisateur de l'application RSA

Auteurs : DELBOT François et ROVEDAKIS Stéphane
Tuteur : M.BAYAD

2.1 Fenêtre principale de l'application

Lors du lancement du programme, la fenêtre suivante va apparaître :



Il s'agit de la fenêtre principale de l'application. Son menu permet d'accéder aux différentes fonctions du programme :

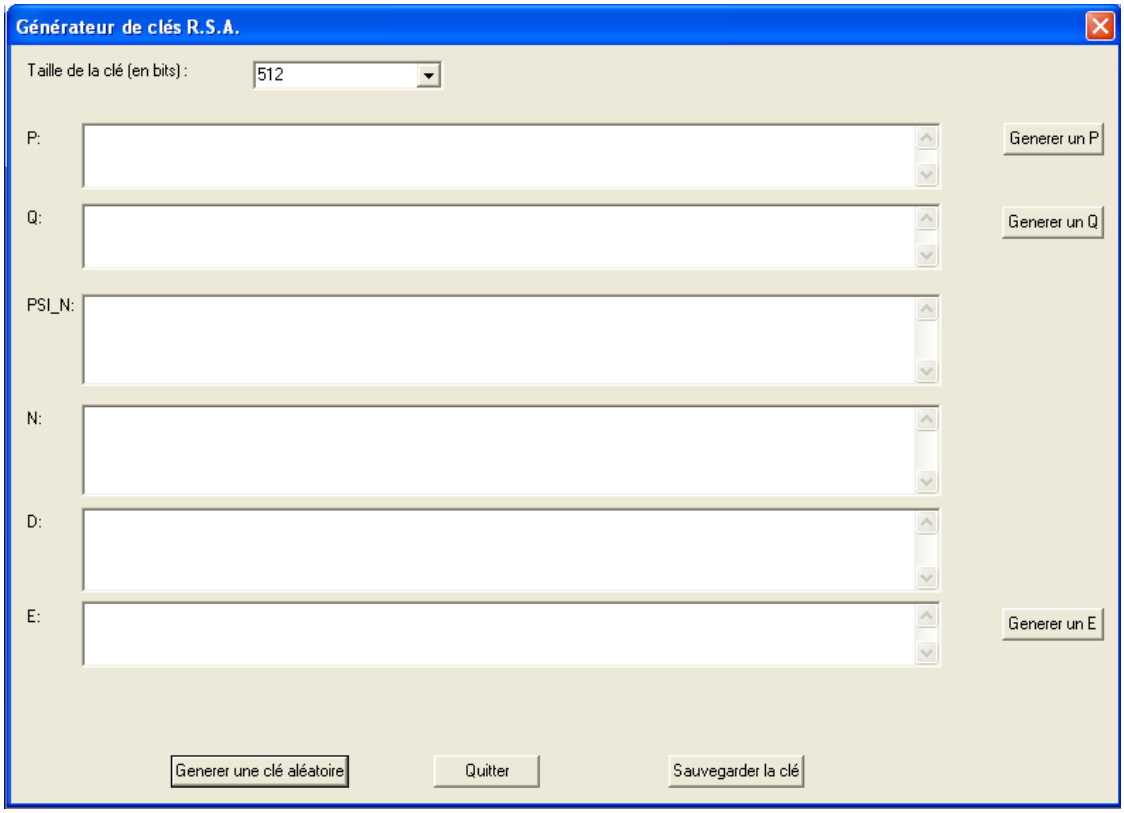
1. Générer une clé RSA
2. Chiffrer un document
3. Déchiffrer un document
4. Signer un document
5. Vérifier la provenance d'un document

Cette fenêtre contient aussi une aide sur comment utiliser l'application.

2.2 Le menu Options

2.2.1 Générateur de clés RSA

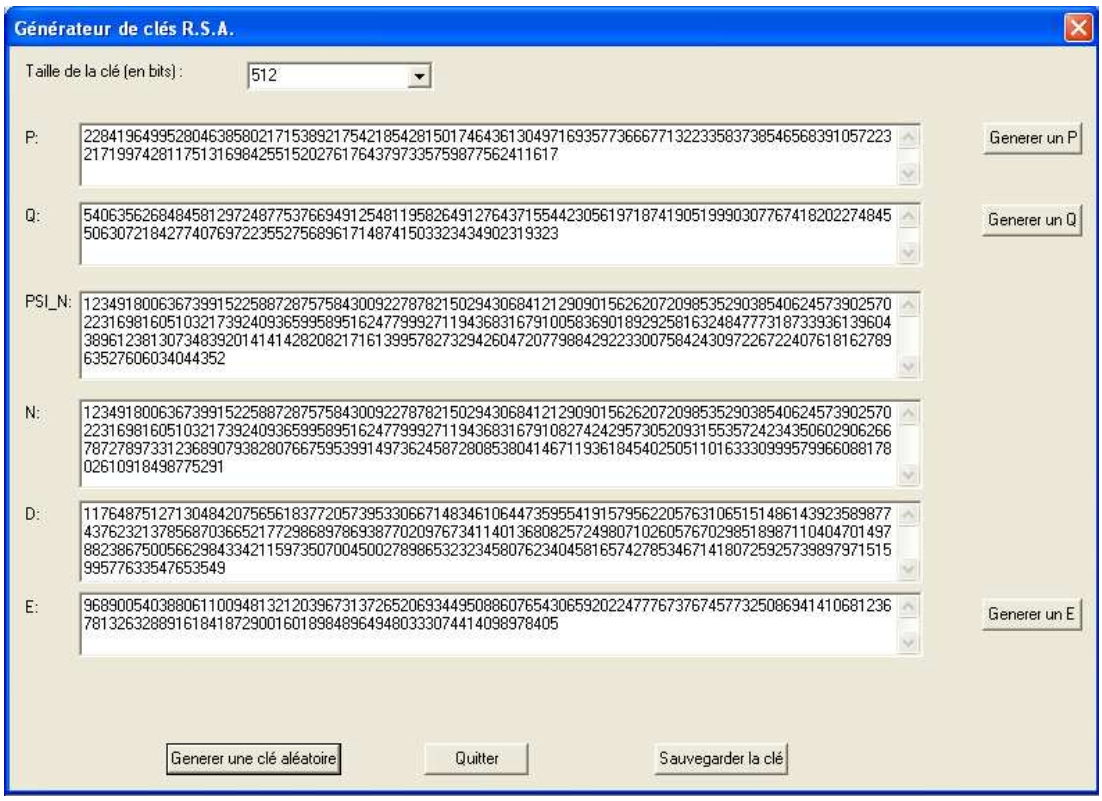
Notre application dispose d'un générateur de clés RSA. Pour y accéder, il suffit de cliquer sur le menu "options", puis de sélectionner le générateur de clé.



Cette fenêtre permet de générer des clés RSA et de les enregistrer.
La génération d'une clé se fait en deux étapes :

1. Sélectionner la taille de la clé (en bits) dans le menu déroulant. Plusieurs choix courants sont disponibles (256, 512, 1024, 2048), mais l'utilisateur peut entrer lui même la taille de clé qu'il aura choisit.
2. Cliquer sur le bouton "Générer une clé".

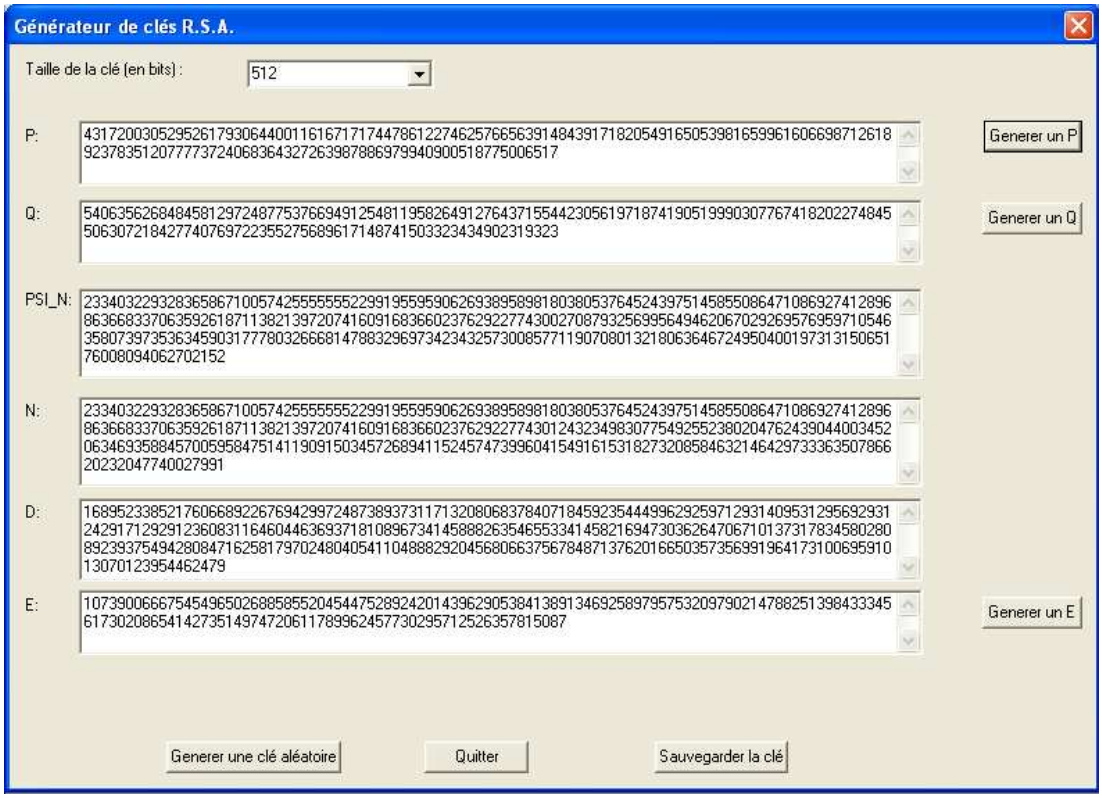
Voici un exemple de clé :



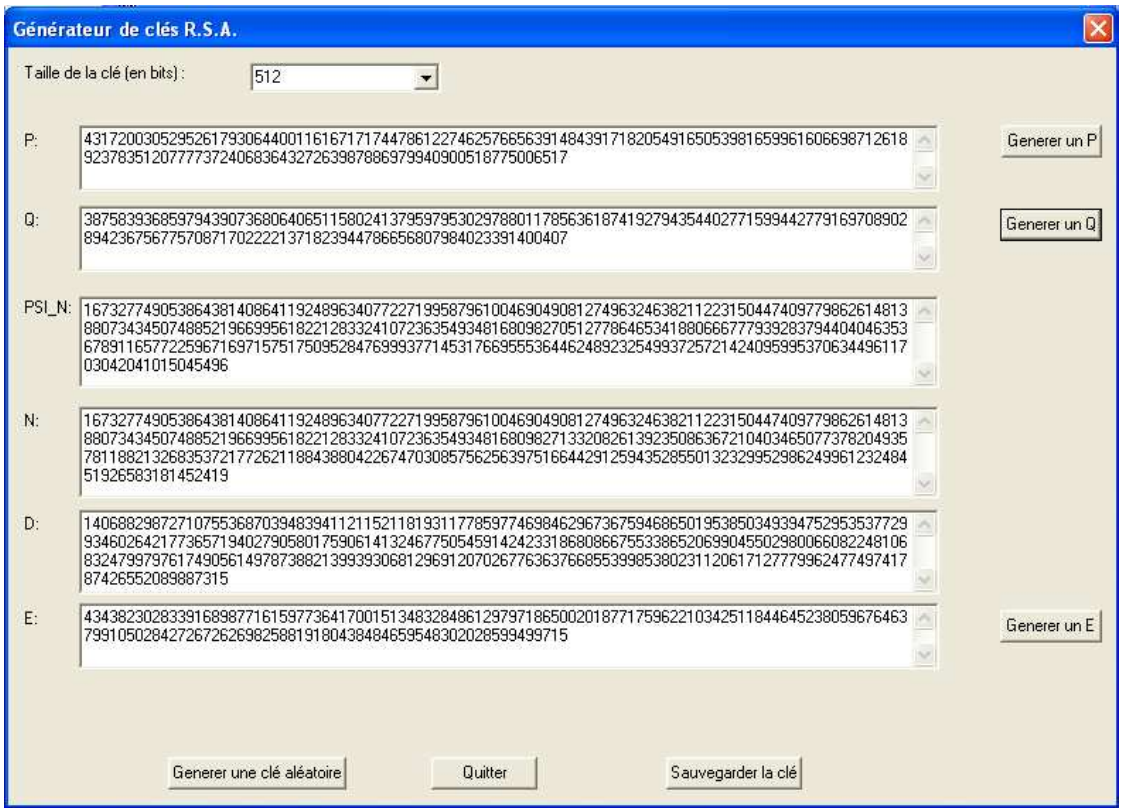
La génération des nombres premiers étant probabiliste, il est possible qu'une clé ne soit pas valide. Ce cas peut se présenter environ une fois sur 600000.

Changer l'un des deux nombres premier (P ou Q)

Nous avons choisis de laisser à l'utilisateur la possibilité de changer les deux nombres premiers sur lesquels est basée la génération de la clé. Ainsi, en cliquant sur le bouton "générer un P", seul le nombre Q sera conservé car toutes les autres valeurs doivent êtres recalculées :

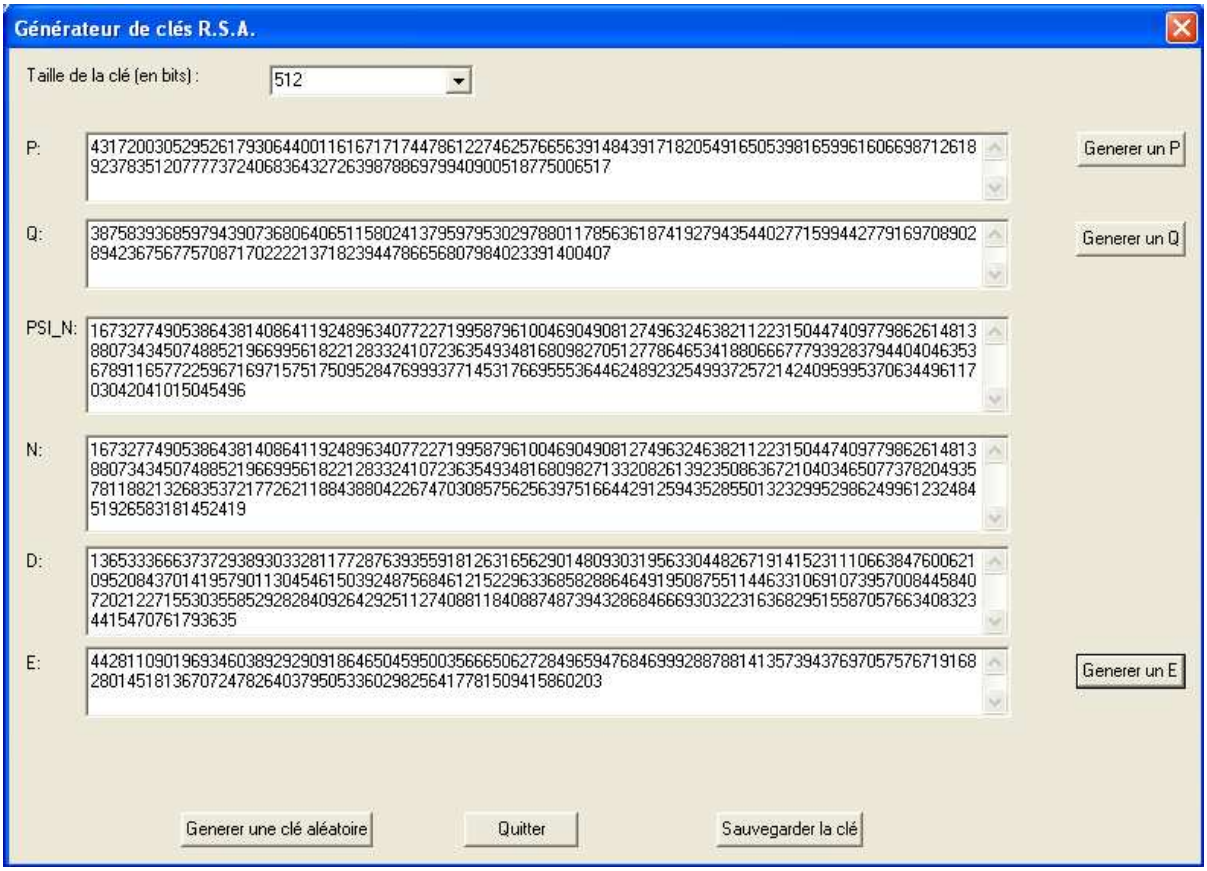


Pour conserver le nombre P et changer le nombre Q, il suffit de cliquer sur le bouton "générer un Q" :



Changer la clé

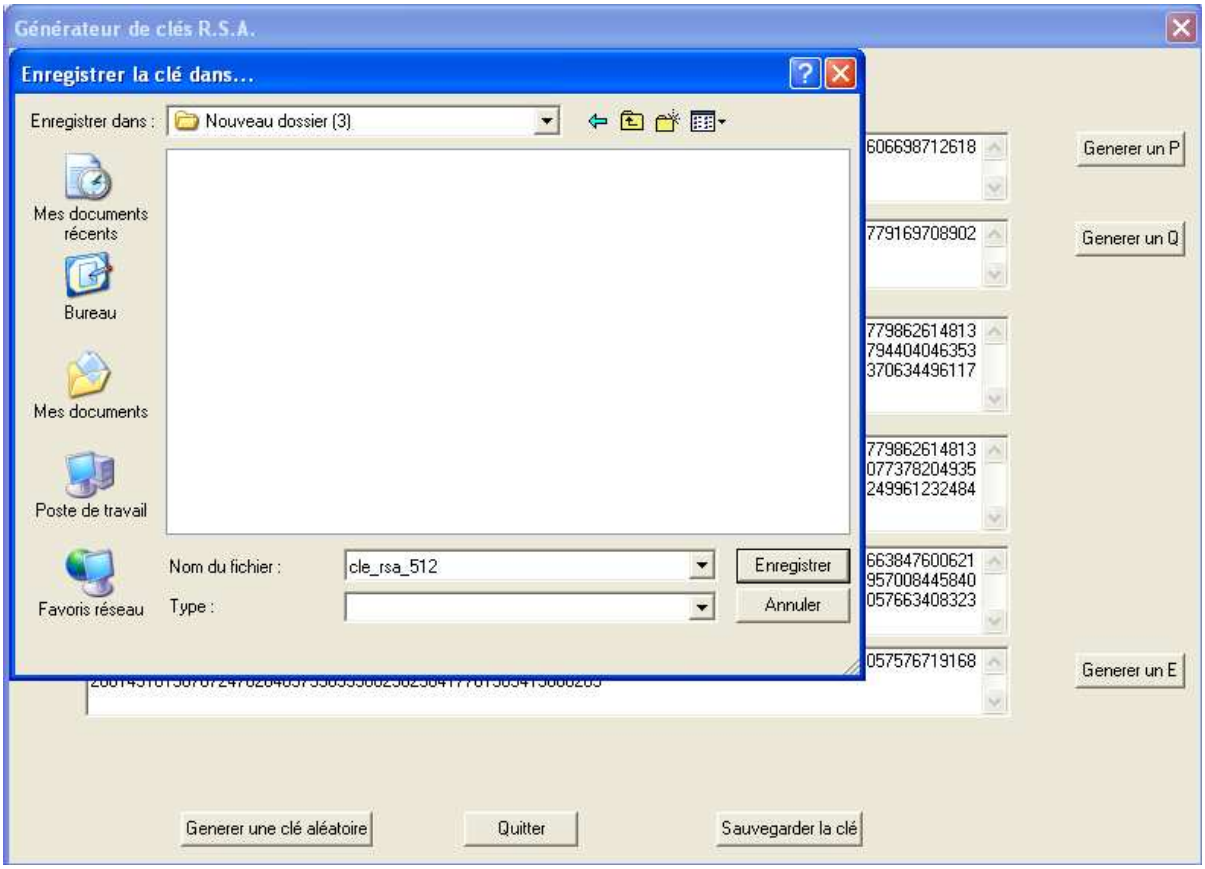
Tout en conservant les deux nombres P et Q, l'utilisateur peut choisir de changer le nombre E précédemment générer (E étant généré de façon aléatoire). En cliquant sur le bouton "générer un E", seuls les valeurs de E et de D seront recalculée.



Enregistrer la clé

Pour enregistrer la clé générée, il suffit de cliquer sur le bouton "sauvegarder la clé". Une boite de dialogue apparaît, il suffit alors d'entrer le nom sous lequel l'utilisateur souhaite sauver la clé. Il est à noter que chaque clé est sauvegardée sous deux fichiers :

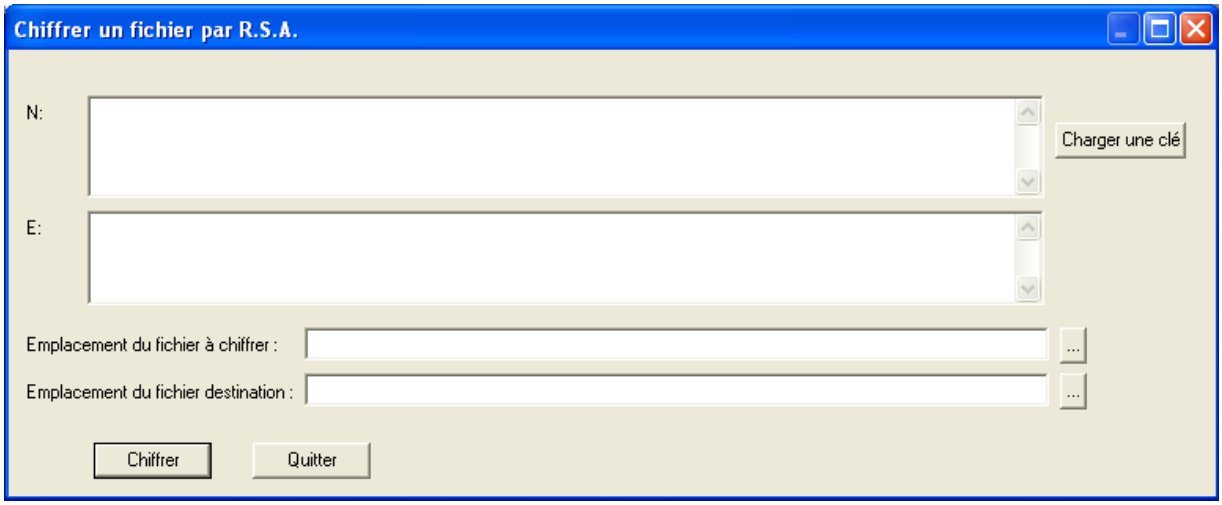
Par exemple, si le nom entré est "cle_rsa_512", les deux fichiers "cle_rsa_512.public" et "cle_rsa_512.private" seront créés. Le fichier d'extension ".public" contiendra la clé publique (N, E) et le fichier d'extension ".private" contiendra la clé privée (N, D).



2.3 Le menu RSA

2.3.1 Chiffrer un document

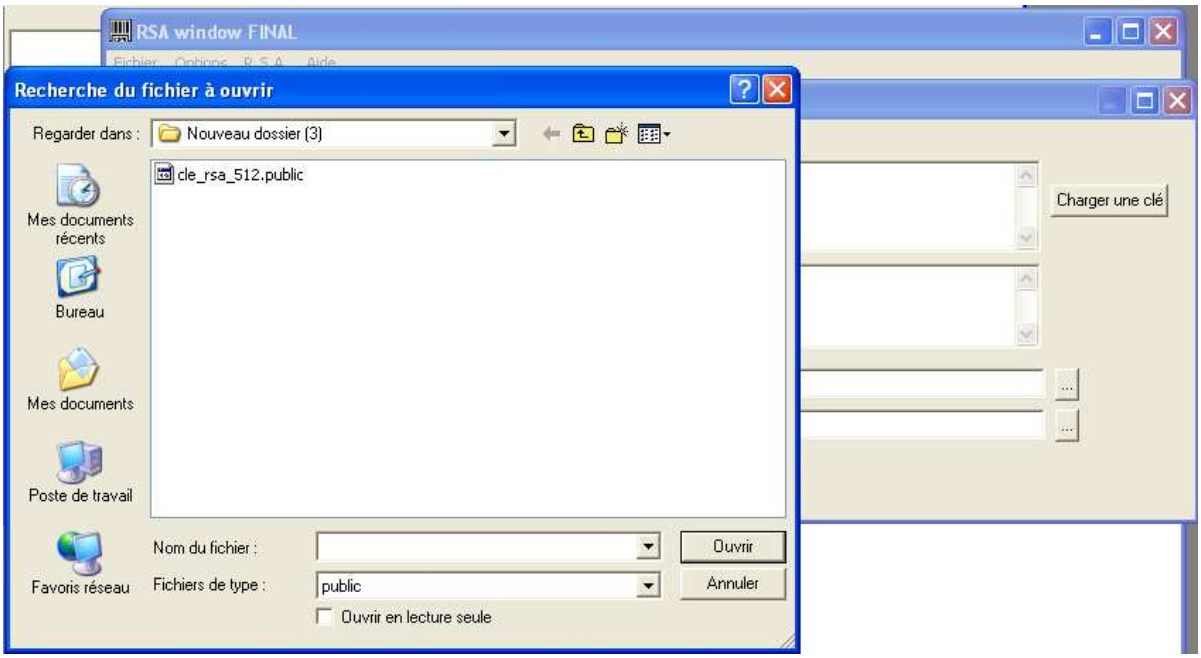
Pour chiffrer un document, il suffit de cliquer sur le menu "R.S.A.", puis de sélectionner "chiffrer un document". La fenêtre suivante apparaît :



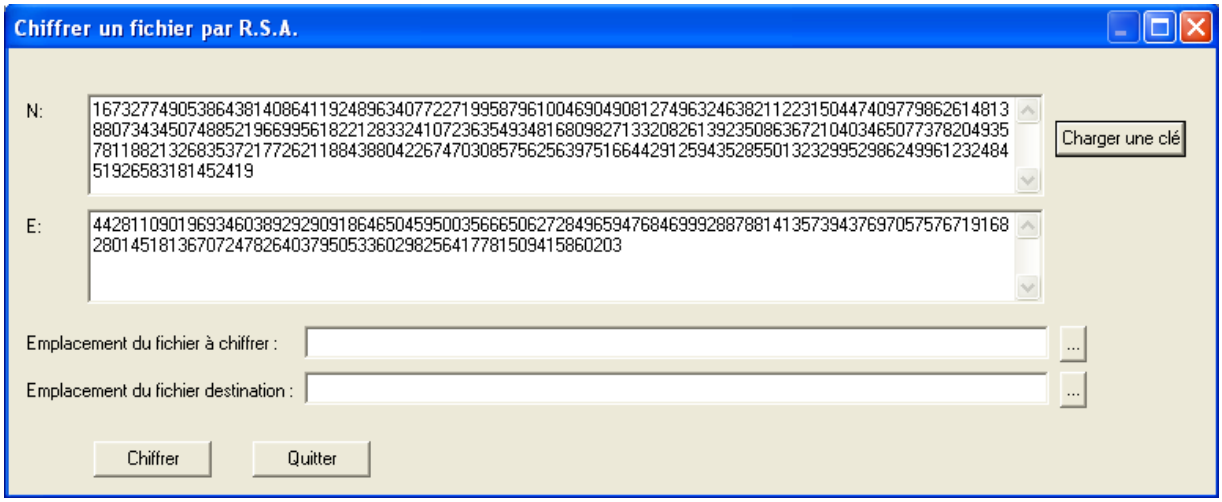
Pour utiliser le chiffrement, il faut entrer une clé publique. Deux méthodes sont possibles :

1. Entrer manuellement la clé

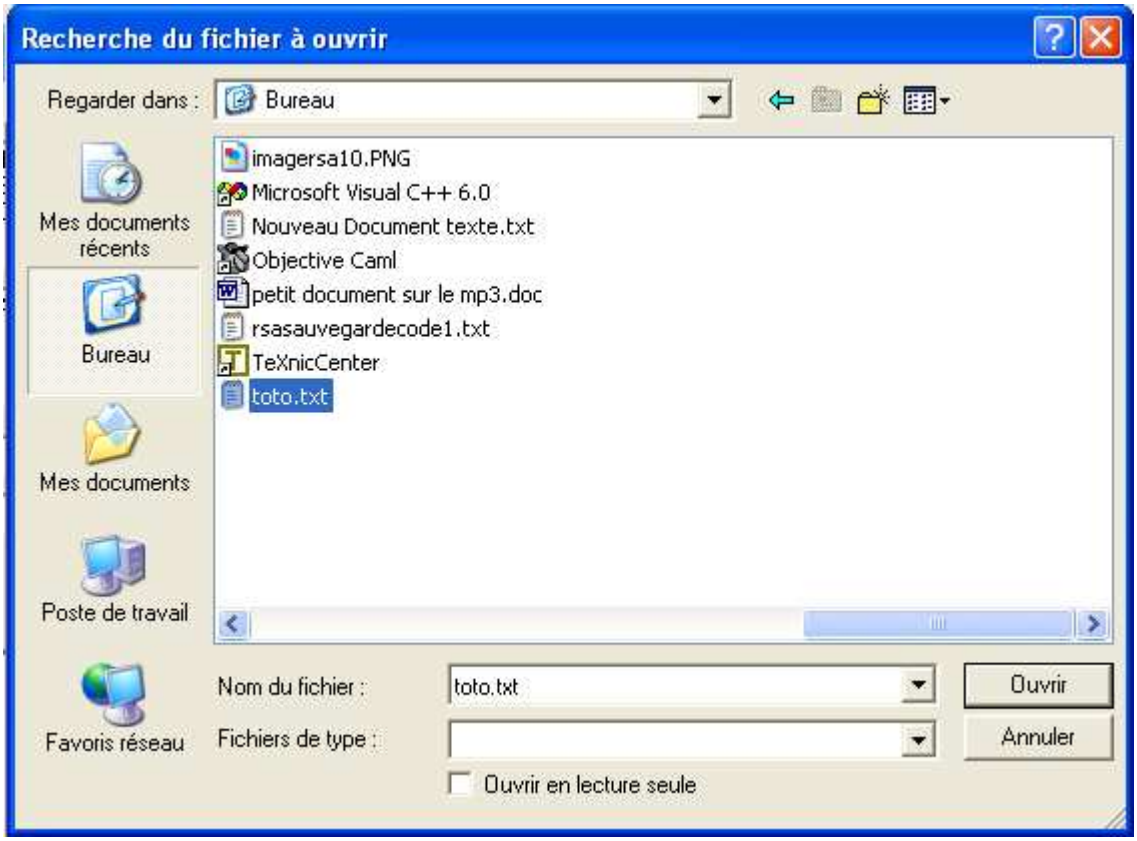
2. Cliquer sur le bouton "Charger une clé", ce qui permettra à l'utilisateur de charger une clé précédemment enregistrée grâce à la boite de dialogue suivante :


RSA window FINAL
Recherche du fichier à ouvrir
Regarder dans :
Nouveau dossier (3)
cle_rsa_512.public
Mes documents récents
Bureau
Mes documents
Poste de travail
Favoris réseau
Nom du fichier :
Fichiers de type :
public
Ouvrir
Annuler
Ouvrir en lecture seule
Charger une clé

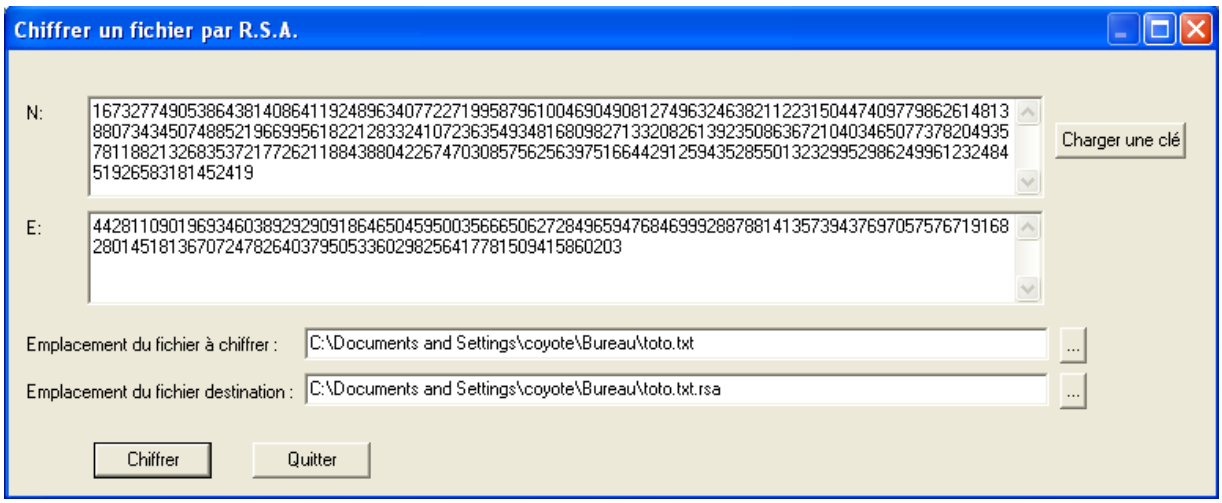
Ce qui aura pour effet de remplir automatiquement les champs N et E :



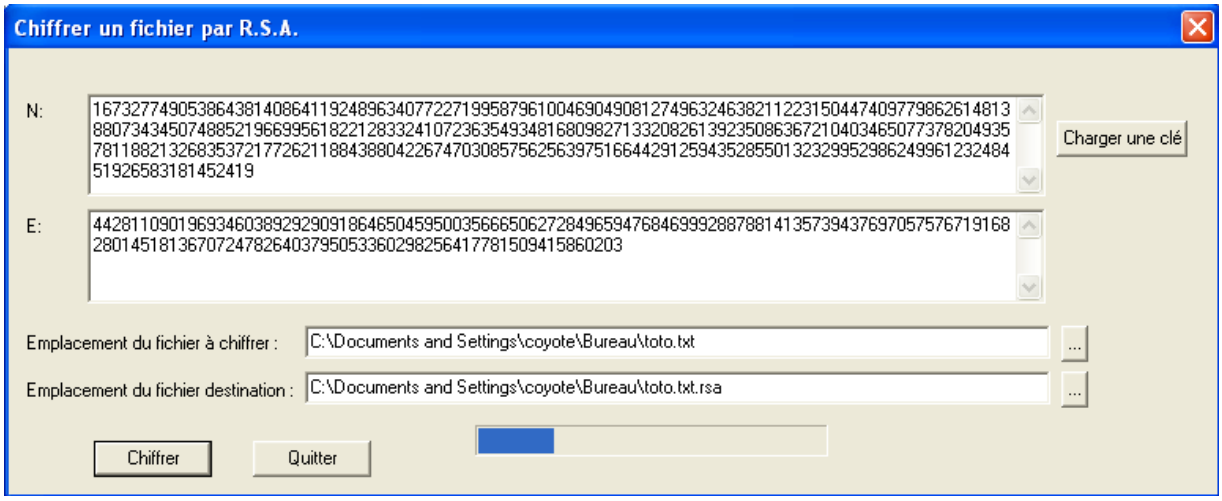
Il ne reste plus qu'à entrer le chemin du fichier que l'utilisateur souhaite chiffrer dans l'emplacement prévu à cet effet. Il est possible d'ouvrir une boite d'exploration en cliquant sur le bouton "..." à droite de l'emplacement :


Recherche du fichier à ouvrir
Regarder dans :
Bureau
imagersa10.PNG
Microsoft Visual C++ 6.0
Nouveau Document texte.txt
Objective Caml
petit document sur le mp3.doc
rsasauvegardecode1.txt
TeXnicCenter
toto.txt
Mes documents récents
Bureau
Mes documents
Poste de travail
Favoris réseau
Nom du fichier :
toto.txt
Fichiers de type :
Ouvrir
Annuler
Ouvrir en lecture seule

Il faut ensuite, de la même façon, remplir le chemin du fichier destination dans lequel sera enregistré le résultat du chiffrement :

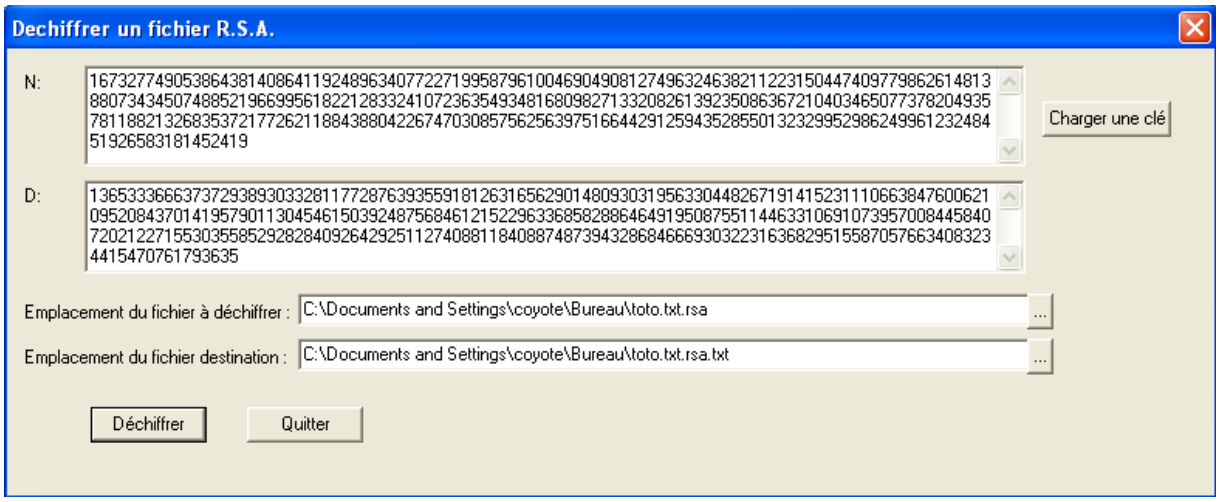


Pour lancer l'opération de chiffrement, l'utilisateur doit cliquer sur le bouton "Chiffrer". Une barre de progression sera visible pour suivre l'état d'avancement du chiffrement :

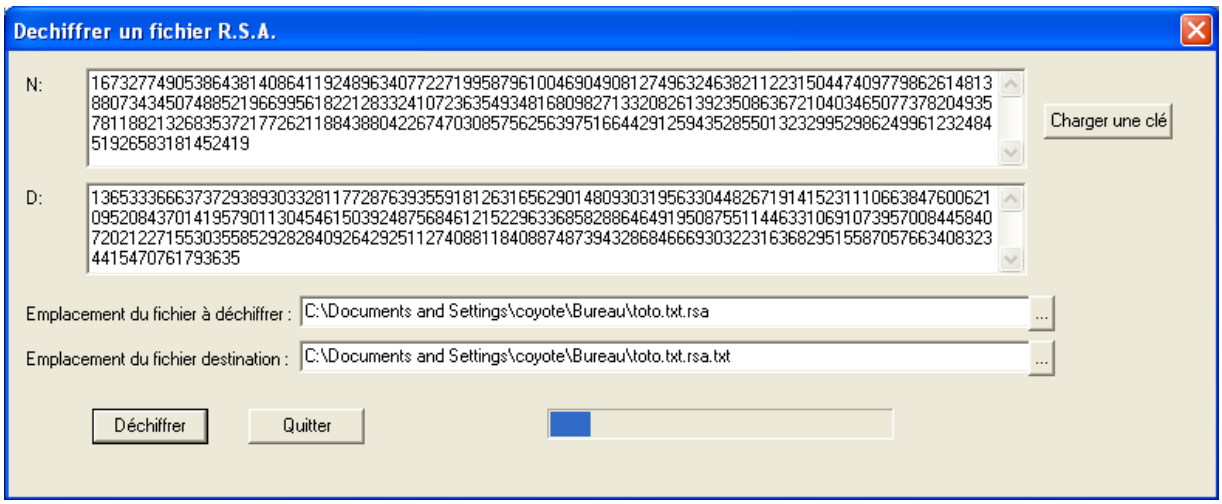


2.3.2 Déchiffrer un document

Pour déchiffrer un document, il suffit de cliquer sur le menu "R.S.A.", puis de sélectionner "déchiffrer un document". La fenêtre se remplit comme celle de chiffrement, à l'exception de la clé qui doit être une clé privée. Ainsi, lorsque l'utilisateur va cliquer sur le bouton "charger une clé", seuls les fichiers de clés privées (.private) seront disponibles.

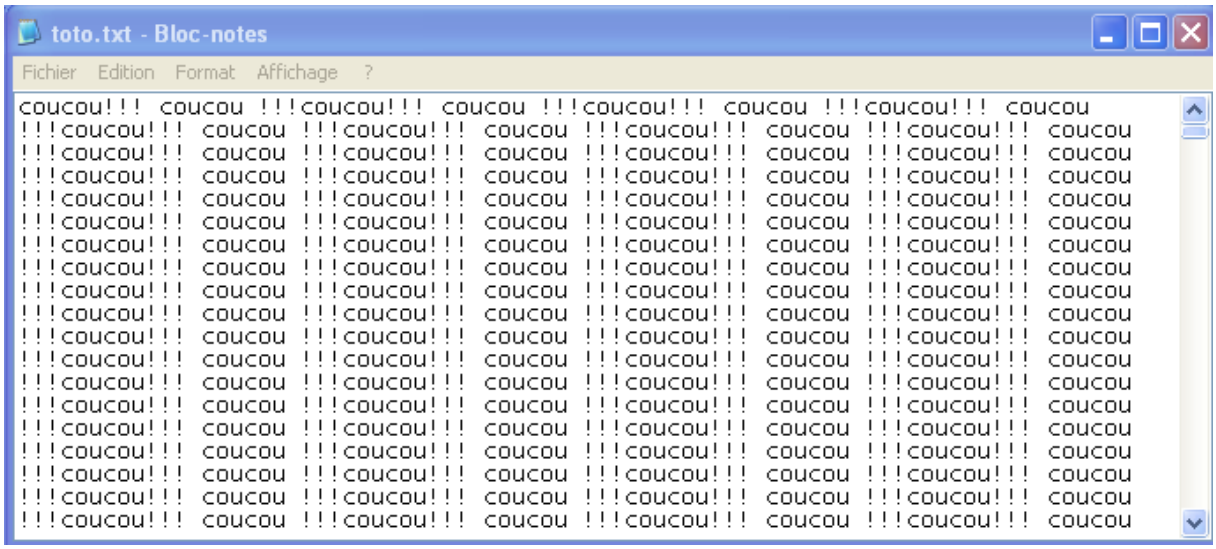


Pour lancer l'opération de déchiffrement, il suffit de cliquer sur le bouton "Déchiffrer". Une barre de progression informera l'utilisateur sur l'état d'avancement de l'opération.

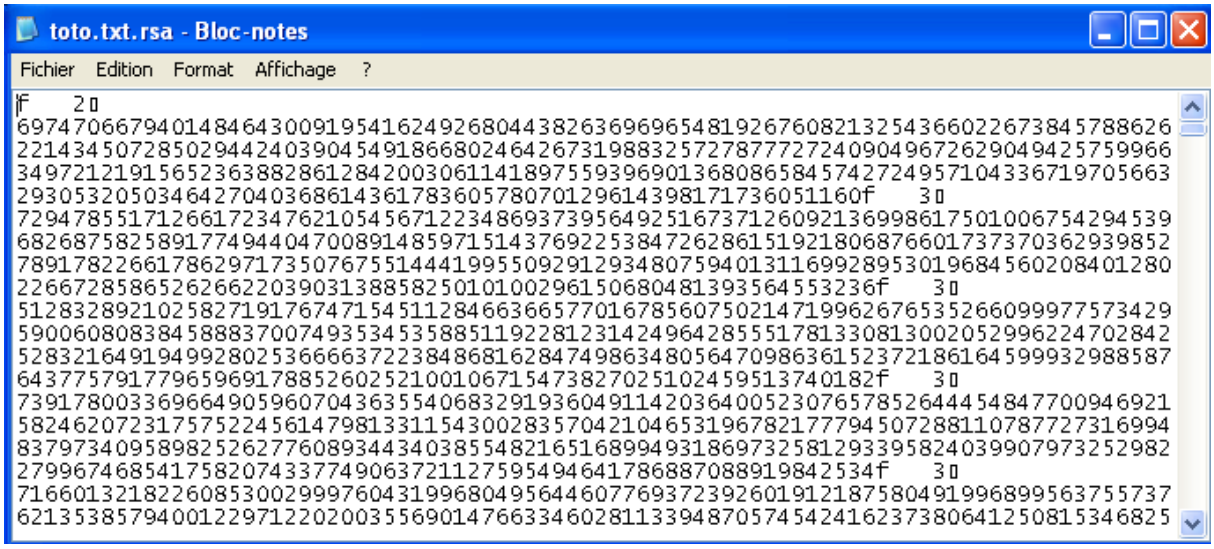


2.3.3 Exemple de chiffrement et de déchiffrement

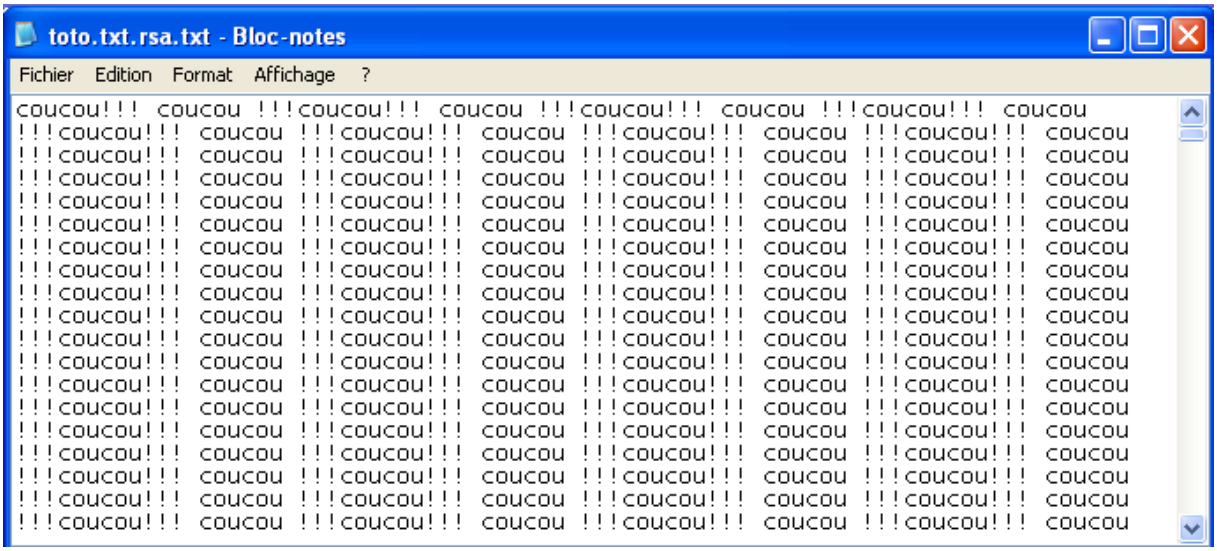
Voici un exemple de chiffrement. Nous prenons le fichier "toto.txt" :



Voici le fichier "toto.txt.rsa" qui contient le résultat du chiffrement du fichier "toto.txt" par notre application avec une clé de 512 bits :

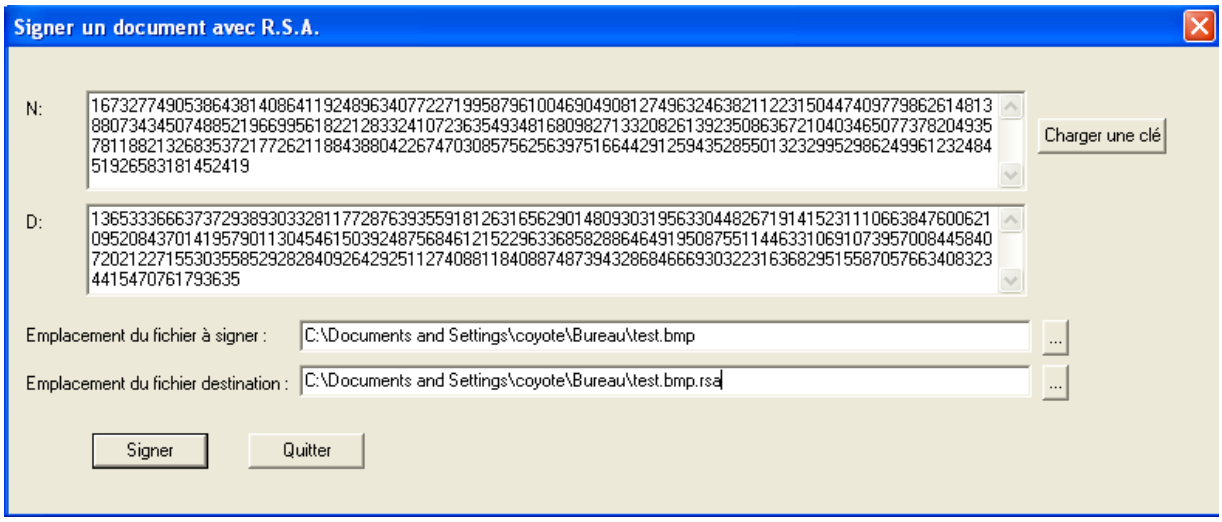


Et voici le fichier "toto.txt.rsa.txt" qui est le résultat du déchiffrement du fichier "toto.txt.rsa" :



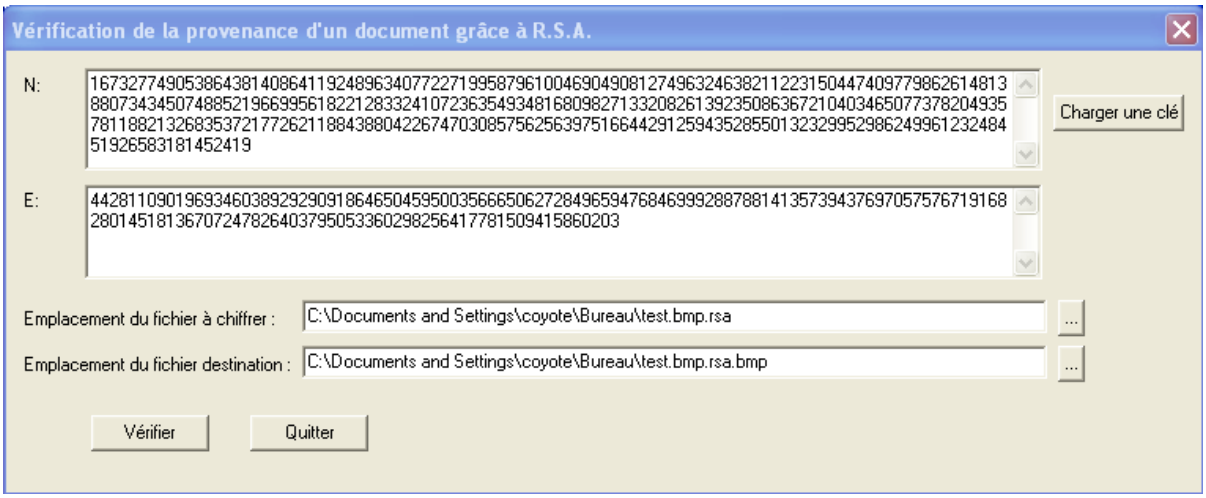
2.3.4 Signer un document

La méthode pour signer un document est exactement la même que pour l'opération de chiffrement, si ce n'est qu'au lieu d'utiliser une clé publique, il faut utiliser une clé privée :



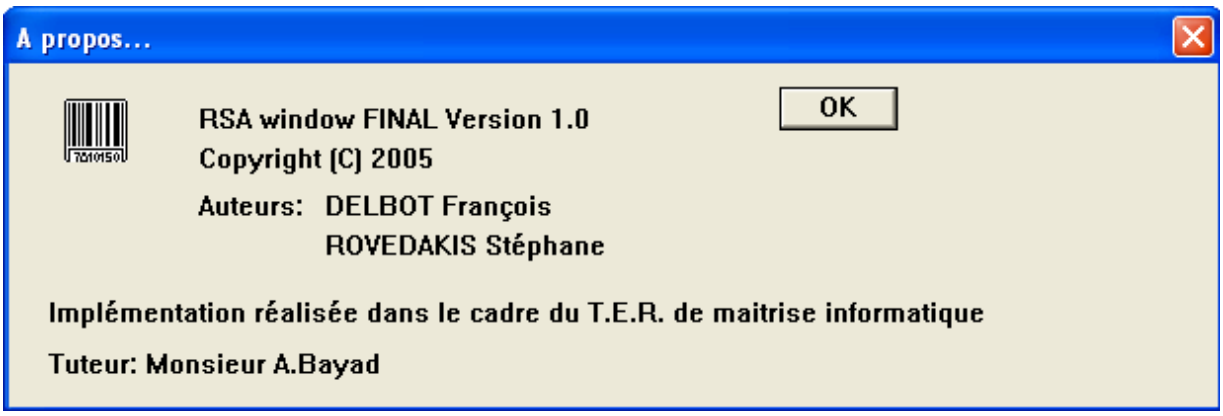
2.3.5 Vérifier la provenance d'un document

La méthode pour vérifier la provenance d'un document est exactement la même que pour l'opération de déchiffrement, si ce n'est qu'au lieu d'utiliser une clé privée, il faut utiliser une clé publique :



2.4 A propos...

Cette application a été réalisée dans le cadre du T.E.R de maîtrise informatique.



Chapitre 3

Exemples d'utilisation de la distance de Hamming

3.1 Avant propos

Pour économiser de l'espace mémoire ainsi qu'un temps de calcul relativement important, nous avons choisi de générer l'alphabet de notre code correcteur en fonction du nombre de blocs différents présents dans le message original. Cette utilisation n'est pas réaliste dans le cadre d'une implantation réelle car il faudrait générer un alphabet de tous les blocs pouvant être rencontrés.

3.2 Choix des exemples

Le premier exemple montre la transmission d'un message en utilisant une distance de Hamming suffisamment grande pour corriger toutes les erreurs générées par le canal. Dans le deuxième exemple, au contraire, le canal génère plus d'erreurs que le code correcteur peut en corriger, ce qui implique que le message est transmis avec des erreurs. Le troisième exemple nous permet de montrer un alphabet complet, c'est à dire que cette solution est véritablement implantable (mais la taille du message transmis est importante).

3.2.1 Définitions

Définition 1 (Message) *Un message est une chaîne de caractères à transmettre via un canal bruité.*

Définition 2 (Bloc) *Un bloc est une sous-partie d'un message. Les blocs sont de longueur fixe.*

Définition 3 (Mot) *Un mot est un bloc qui a été encodé en utilisant un code correcteur.*

3.3 Exemple 1

3.3.1 Caractéristiques du code correcteur utilisé

Taille de bloc

La taille de bloc utilisée est de 8 bits.

Taille de mot

La taille d'un mot associé à un bloc est de 10 bits.

Nombre d'erreurs détectées

Ce code correcteur peut détecter 3 erreurs par mot transmis.

Nombre d'erreurs corrigées

Ce code correcteur peut corriger 1 erreurs par mot transmis.

Alphabet

bloc du message	Code correspondant au bloc
01010110	0100010101
01101111	1011000110
01101001	0010010000
01100011	0001110111
00100000	0101110001
01110101	1000110110
01101110	1100011010
01100101	1000000101
01111000	0110111111
01101101	1011111100
01110000	1110110001
01101100	0111100000
01100100	0010010111
01110011	0110001011
01100001	0100000010
01100111	0101000100
00100001	0111000111

3.3.2 Message original

Chaîne de caractères originale

Pour illustrer le fonctionnement de ce code correcteur, nous avons choisis d'utiliser le message suivant :

Voici un exemple de message ! ! !

Message original traduit en binaire

La représentation binaire du message original est la suivante :

01010110 01101111 01101001 01100011 01101001 00100000 01110101 01101110 00100000 01100101
01111000 01100101 01101101 01110000 01101100 01100101 00100000 01100100 01100101 00100000
01101101 01100101 01110011 01110011 01100001 01100111 01100101 00100000 00100001 00100001
00100001

3.3.3 Message encodé

Pour encoder le message, nous remplaçons chaque bloc par le mot qui lui est associé dans l'alphabet.

0100010101 1011000110 0010010000 0001110111 0010010000 0101110001 1000110110 1100011010 0101110001
1000000101 0110111111 1000000101 1011111100 1110110001 0111100000 1000000101 0101110001 0010010111
1000000101 0101110001 1011111100 1000000101 0110001011 0110001011 0100000010 0101000100 1000000101
0101110001 0111000111 0111000111 0111000111

3.3.4 Simulation du passage d'un canal bruité

Caractéristiques du canal

Le canal utilisé est capable de générer des erreurs de façon aléatoire. Le canal peut générer au plus 1 erreurs par mot transmis.

Message reçu

0100011101 1011000010 0010010000 0001110111 0010010000 0101110001 1010110110 1100011010
0101100001 1000000101 0110111111 1000100101 1011111100 1110110001 0111100010 1000000101 0111110001
0010010111 1000000111 0101110001 1011111101 1000010101 1110001011 0110001011 0100000010 0101000100
1010000101 0001110001 0111100111 0111010111 0111000111

Les erreurs générées par le canal apparaissent en rouge.

3.3.5 Détection et correction des erreurs

N° de bloc	bloc	bloc corrigé	nombre d'erreurs détectées	bilan
0	0100011101	0100010101	1 erreurs détectées	Le bloc est corrigé.
1	1011000010	1011000110	1 erreurs détectées	Le bloc est corrigé.
2	0010010000	0010010000	0 erreurs détectées	Il n'y a pas d'erreurs.
3	0001110111	0001110111	0 erreurs détectées	Il n'y a pas d'erreurs.
4	0010010000	0010010000	0 erreurs détectées	Il n'y a pas d'erreurs.
5	0101110001	0101110001	0 erreurs détectées	Il n'y a pas d'erreurs.
6	1010110110	1000110110	1 erreurs détectées	Le bloc est corrigé.
7	1100011010	1100011010	0 erreurs détectées	Il n'y a pas d'erreurs.
8	0101100001	0101110001	1 erreurs détectées	Le bloc est corrigé.
9	1000000101	1000000101	0 erreurs détectées	Il n'y a pas d'erreurs.
10	0110111111	0110111111	0 erreurs détectées	Il n'y a pas d'erreurs.
11	1000100101	1000000101	1 erreurs détectées	Le bloc est corrigé.
12	1011111100	1011111100	0 erreurs détectées	Il n'y a pas d'erreurs.
13	1110110001	1110110001	0 erreurs détectées	Il n'y a pas d'erreurs.
14	0111100010	0111100000	1 erreurs détectées	Le bloc est corrigé.
15	1000000101	1000000101	0 erreurs détectées	Il n'y a pas d'erreurs.
16	0111110001	0101110001	1 erreurs détectées	Le bloc est corrigé.
17	0010010111	0010010111	0 erreurs détectées	Il n'y a pas d'erreurs.
18	1000000111	1000000101	1 erreurs détectées	Le bloc est corrigé.
19	0101110001	0101110001	0 erreurs détectées	Il n'y a pas d'erreurs.
20	1011111101	1011111100	1 erreurs détectées	Le bloc est corrigé.
21	1000010101	1000000101	1 erreurs détectées	Le bloc est corrigé.
22	1110001011	0110001011	1 erreurs détectées	Le bloc est corrigé.
23	0110001011	0110001011	0 erreurs détectées	Il n'y a pas d'erreurs.
24	0100000010	0100000010	0 erreurs détectées	Il n'y a pas d'erreurs.
25	0101000100	0101000100	0 erreurs détectées	Il n'y a pas d'erreurs.
26	1010000101	1000000101	1 erreurs détectées	Le bloc est corrigé.
27	0001110001	0101110001	1 erreurs détectées	Le bloc est corrigé.
28	0111100111	0111000111	1 erreurs détectées	Le bloc est corrigé.
29	0111010111	0111000111	1 erreurs détectées	Le bloc est corrigé.
30	0111000111	0111000111	0 erreurs détectées	Il n'y a pas d'erreurs.

Message corrigé

0100010101 1011000110 0010010000 0001110111 0010010000 0101110001 1000110110 1100011010 0101110001 1000000101 0110111111 1000000101 1011111100 1110110001 0111100000 1000000101 0101110001 0010010111 1000000101 0101110001 1011111100 1000000101 0110001011 0110001011 0100000010 0101000100 1000000101 0101110001 0111000111 0111000111 0111000111

3.3.6 Décodage du message

Message binaire décodé

0101011001 1011110110 1001011000 1101101001 0010000001 1101010110 1110001000 0001100101 0111100001 1001010110 1101011100 0001101100 0110010100 1000000110 0100011001 0100100000 0110110101 1001010111 0011011100 1101100001 0110011101 1001010010 0000001000 0100100001 00100001

Message final

Voici un exemple de message ! ! !

3.4 Exemple 2

3.4.1 Caractéristiques du code correcteur utilisé

Taille de bloc

La taille de bloc utilisée est de 8 bits.

Taille de mot

La taille d'un mot associé à un bloc est de 10 bits.

Nombre d'erreurs détectées

Ce code correcteur peut détecter 3 erreurs par mot transmis.

Nombre d'erreurs corrigées

Ce code correcteur peut corriger 1 erreurs par mot transmis.

Alphabet

bloc du message	Code correspondant au bloc
01010110	0100010101
01101111	1011000110
01101001	0010010000
01100011	0001110111
00100000	0101110001
01110101	1000110110
01101110	1100011010
01100101	1000000101
01111000	0110111111
01101101	1011111100
01110000	1110110001
01101100	0111100000
01100100	0010010111
01110011	0110001011
01100001	0100000010
01100111	0101000100
00100001	0111000111

3.4.2 Message original

Chaîne de caractères originale

Pour illustrer le fonctionnement de ce code correcteur, nous avons choisis d'utiliser le message suivant :

Voici un exemple de message ! ! !

Message original traduit en binaire

La représentation binaire du message original est la suivante :

01010110 01101111 01101001 01100011 01101001 00100000 01110101 01101110 00100000 01100101 01111000 01100101 01101101 01110000 01101100 01100101 00100000 01100100 01100101 00100000 01101101 01100101 01110011 01110011 01100001 01100111 01100101 00100000 00100001 00100001 00100001

3.4.3 Message encodé

Pour encoder le message, nous remplaçons chaque bloc par le mot qui lui est associé dans l'alphabet.

0100010101 1011000110 0010010000 0001110111 0010010000 0101110001 1000110110 1100011010 0101110001 1000000101 0110111111 1000000101 1011111100 1110110001 0111100000 1000000101 0101110001 0010010111

1000000101 0101110001 1011111100 1000000101 0110001011 0110001011 0100000010 0101000100 1000000101 0101110001 0111000111 0111000111 0111000111

3.4.4 Simulation du passage d'un canal bruité

Caractéristiques du canal

Le canal utilisé est capable de générer des erreurs de façon aléatoire. Le canal peut générer au plus 2 erreurs par mot transmis.

Message reçu

0100011101 1011001100 0110110000 0011111001 0010010000 0111110001 1000111010 1100011110 1101110001 1100010101 0110100111 1100000101 1011101100 1110100001 0111100100 1000101101 0101110001 0010010011 1000000101 0111110001 1111110100 1000000101 0110001111 0010001111 0100000010 0101000100 1001000111 0101110001 0111000111 1111000111 1111000111

Les erreurs générées par le canal apparaissent en rouge.

3.4.5 Détection et correction des erreurs

N° de bloc	bloc	bloc corrigé	nombre d'erreurs détectées	bilan
0	0100011101	0100010101	1 erreurs détectées	Le bloc est corrigé.
1	1011001100	1011000110	2 erreurs détectées	Trop d'erreurs.Bloc non corrigé.
2	0110110000	0010010000	2 erreurs détectées	Trop d'erreurs.Bloc non corrigé.
3	0011110101	0001110111	2 erreurs détectées	Trop d'erreurs.Bloc non corrigé.
4	0010010000	0010010000	0 erreurs détectées	Il n'y a pas d'erreurs.
5	0111110001	0101110001	1 erreurs détectées	Le bloc est corrigé.
6	1000111010	1000110110	2 erreurs détectées	Trop d'erreurs.Bloc non corrigé.
7	1100011110	1100011010	1 erreurs détectées	Le bloc est corrigé.
8	1101110001	0101110001	1 erreurs détectées	Le bloc est corrigé.
9	1100010101	0100010101	1 erreurs détectées	Le bloc est corrigé.
10	0110100111	0110111111	2 erreurs détectées	Trop d'erreurs.Bloc non corrigé.
11	1100000101	1000000101	1 erreurs détectées	Le bloc est corrigé.
12	1011101100	1011111100	1 erreurs détectées	Le bloc est corrigé.
13	1110100001	1110110001	1 erreurs détectées	Le bloc est corrigé.
14	0111100100	0111100000	1 erreurs détectées	Le bloc est corrigé.
15	1000101101	1000000101	2 erreurs détectées	Trop d'erreurs.Bloc non corrigé.
16	0101110001	0101110001	0 erreurs détectées	Il n'y a pas d'erreurs.
17	0010010011	0010010111	1 erreurs détectées	Le bloc est corrigé.
18	1000000101	1000000101	0 erreurs détectées	Il n'y a pas d'erreurs.
19	0111110001	0101110001	1 erreurs détectées	Le bloc est corrigé.
20	1111110100	1011111100	2 erreurs détectées	Trop d'erreurs.Bloc non corrigé.
21	1000000101	1000000101	0 erreurs détectées	Il n'y a pas d'erreurs.
22	0110001111	0110001011	1 erreurs détectées	Le bloc est corrigé.
23	0010001111	0010010111	2 erreurs détectées	Trop d'erreurs.Bloc non corrigé.
24	0100000010	0100000010	0 erreurs détectées	Il n'y a pas d'erreurs.
25	0101000100	0101000100	0 erreurs détectées	Il n'y a pas d'erreurs.
26	1001000111	1011000110	2 erreurs détectées	Trop d'erreurs.Bloc non corrigé.
27	0101110001	0101110001	0 erreurs détectées	Il n'y a pas d'erreurs.
28	0111000111	0111000111	0 erreurs détectées	Il n'y a pas d'erreurs.
29	1111000111	0111000111	1 erreurs détectées	Le bloc est corrigé.
30	1111000111	0111000111	1 erreurs détectées	Le bloc est corrigé.

Message corrigé

0100010101 1011000110 0010010000 0001110111 0010010000 0101110001 1000110110 1100011010 0101110001 0100010101 0110111111 1000000101 1011111100 1110110001 0111100000 1000000101 0101110001 0010010111 1000000101 0101110001 1011111100 1000000101 0110001011 0010010111 0100000010 0101000100 1011000110 0101110001 0111000111 0111000111 0111000111

3.4.6 Décodage du message

Message binaire décodé

0101011001 1011110110 1001011000 1101101001 0010000001 1101010110 1110001000 0001**01**01**10** 0111100001 1001010110 1101011100 0001101100 0110010100 1000000110 0100011001 0100100000 0110110101 1001010111 0011011**0**0**1 00**01100001 0110011101 10**1**1**1**10010 0000001000 0100100001 00100001

Message final

Voici un Vxemple de mesdago ! ! !

3.5 Exemple 3

3.5.1 Caractéristiques du code correcteur utilisé

Taille de bloc

La taille de bloc utilisée est de 1 bits.

Taille de mot

La taille d'un mot associé à un bloc est de 13 bits.

Nombre d'erreurs détectées

Ce code correcteur peut détecter 5 erreurs par mot transmis.

Nombre d'erreurs corrigées

Ce code correcteur peut corriger 2 erreurs par mot transmis.

Alphabet

bloc du message	Code correspondant au bloc
0	1010100010101
1	0111011000110

3.5.2 Message original

Chaîne de caractères originale

Pour illustrer le fonctionnement de ce code correcteur, nous avons choisis d'utiliser le message suivant :

coucou

Message original traduit en binaire

La représentation binaire du message original est la suivante :

0 1 1 0 0 0 1 1 0 1 1 0 1 1 1 1 0 1 1 1 0 1 0 1 0 1 1 0 0 0 1 1 0 1 1 0 1 1 1 1 0 1 1 1 0 1 0 1

3.5.3 Message encodé

Pour encoder le message, nous remplaçons chaque bloc par le mot qui lui est associé dans l'alphabet.

1010100010101 0111011000110 0111011000110 1010100010101 1010100010101 1010100010101 0111011000110
0111011000110 1010100010101 0111011000110 0111011000110 1010100010101 0111011000110 0111011000110
0111011000110 0111011000110 1010100010101 0111011000110 0111011000110 0111011000110 1010100010101
0111011000110 1010100010101 0111011000110 1010100010101 0111011000110 0111011000110 1010100010101
1010100010101 1010100010101 0111011000110 0111011000110 1010100010101 0111011000110 0111011000110
1010100010101 0111011000110 0111011000110 0111011000110 0111011000110 1010100010101 0111011000110
0111011000110 0111011000110 1010100010101 0111011000110 1010100010101 0111011000110

3.5.4 Simulation du passage d'un canal bruité

Caractéristiques du canal

Le canal utilisé est capable de générer des erreurs de façon aléatoire. Le canal peut générer au plus 2 erreurs par mot transmis.

Message reçu

1010100011101 0111001000110 1111011000110 1010100110100 1010100010101 1110100010101 0111001000110
0111011100110 1010100010101 0111011000110 0111011010110 1010100110101 0110011010110 0111011000111
0111111000110 0111010100110 1010101010101 0011111000110 0110011100110 0111011000110 1010100010001
0111001000110 1010000011101 0111010001110 1010100010101 0111011100110 0111011001110 1010100000101
1010100010101 1010100010101 0111001000110 0111011100110 1010100011101 0111011000110 0111011000111
1110100010111 0111011000011 0111011000110 0111011000110 0111011000110 1010100010101 0111011000110
0111011000110 0111011010110 1010100010101 0111011100110 1010100010101 0111001000110

Les erreurs générées par le canal apparaissent en rouge.

3.5.5 Détection et correction des erreurs

N° de bloc	bloc	bloc corrigé	nombre d'erreurs détectées	bilan
0	10101000**1**1101	10101000**0**0101	1 erreurs détectées	Le bloc est corrigé.
1	01110**0**1000110	01110**1**1000110	1 erreurs détectées	Le bloc est corrigé.
2	**1**111011000110	**0**111011000110	1 erreurs détectées	Le bloc est corrigé.
3	10101001101**0**0	10101000101**1**	2 erreurs détectées	Le bloc est corrigé.
4	1010100010101	1010100010101	0 erreurs détectées	Il n'y a pas d'erreurs.
5	1**1**10100010101	1**0**10100010101	1 erreurs détectées	Le bloc est corrigé.
6	01110**0**1000110	01110**1**1000110	1 erreurs détectées	Le bloc est corrigé.
7	0111011**1**00110	0111011**0**00110	1 erreurs détectées	Le bloc est corrigé.
8	1010100010101	1010100010101	0 erreurs détectées	Il n'y a pas d'erreurs.
9	0111011000110	0111011000110	0 erreurs détectées	Il n'y a pas d'erreurs.
10	0111011**1**0110	0111011**0**0110	1 erreurs détectées	Le bloc est corrigé.
11	10101001**1**0101	10101000**0**0101	1 erreurs détectées	Le bloc est corrigé.
12	011**0**011**1**0110	011**1**011**0**0110	2 erreurs détectées	Le bloc est corrigé.
13	0111011000111	0111011000110	1 erreurs détectées	Le bloc est corrigé.
14	0111**1**11000110	0111**0**11000110	1 erreurs détectées	Le bloc est corrigé.
15	011101**0**100110	011101**1**000110	2 erreurs détectées	Le bloc est corrigé.
16	101010**1**010101	101010**0**010101	1 erreurs détectées	Le bloc est corrigé.
17	0**0**111**1**11000110	0**1**11**0**11000110	2 erreurs détectées	Le bloc est corrigé.
18	011**0**011**1**00110	011**1**011**0**00110	2 erreurs détectées	Le bloc est corrigé.
19	0111011000110	0111011000110	0 erreurs détectées	Il n'y a pas d'erreurs.
20	10101000100**0**1	10101000101**1**	1 erreurs détectées	Le bloc est corrigé.
21	01110**0**1000110	01110**1**1000110	1 erreurs détectées	Le bloc est corrigé.
22	1010**0**0001**1**101	1010**1**0001**0**101	2 erreurs détectées	Le bloc est corrigé.
23	0111010**0**01**1**110	0111011**1**00**0**110	2 erreurs détectées	Le bloc est corrigé.
24	1010100010101	1010100010101	0 erreurs détectées	Il n'y a pas d'erreurs.
25	0111011**1**00110	0111011**0**00110	1 erreurs détectées	Le bloc est corrigé.
26	0111011001**1**10	0111011000**0**10	1 erreurs détectées	Le bloc est corrigé.
27	10101000**0**0101	10101000**1**0101	1 erreurs détectées	Le bloc est corrigé.
28	1010100010101	1010100010101	0 erreurs détectées	Il n'y a pas d'erreurs.
29	1010100010101	1010100010101	0 erreurs détectées	Il n'y a pas d'erreurs.
30	01110**0**1000110	01110**1**1000110	1 erreurs détectées	Le bloc est corrigé.
31	0111011**1**00110	0111011**0**00110	1 erreurs détectées	Le bloc est corrigé.
32	10101000**1**1101	10101000**0**0101	1 erreurs détectées	Le bloc est corrigé.
33	0111011000110	0111011000110	0 erreurs détectées	Il n'y a pas d'erreurs.
34	0111011000111	0111011000110	1 erreurs détectées	Le bloc est corrigé.
35	1**1**10100010**1**1	1**0**10100010**0**1	2 erreurs détectées	Le bloc est corrigé.
36	0111011000**0**1**1**	0111011000**1**1**0**	2 erreurs détectées	Le bloc est corrigé.
37	0111011000110	0111011000110	0 erreurs détectées	Il n'y a pas d'erreurs.
38	0111011000110	0111011000110	0 erreurs détectées	Il n'y a pas d'erreurs.
39	0111011000110	0111011000110	0 erreurs détectées	Il n'y a pas d'erreurs.
40	1010100010101	1010100010101	0 erreurs détectées	Il n'y a pas d'erreurs.
41	0111011000110	0111011000110	0 erreurs détectées	Il n'y a pas d'erreurs.
42	0111011000110	0111011000110	0 erreurs détectées	Il n'y a pas d'erreurs.
43	0111011**1**0110	0111011**0**0110	1 erreurs détectées	Le bloc est corrigé.
44	1010100010101	1010100010101	0 erreurs détectées	Il n'y a pas d'erreurs.
45	0111011**1**00110	0111011**0**00110	1 erreurs détectées	Le bloc est corrigé.
46	1010100010101	1010100010101	0 erreurs détectées	Il n'y a pas d'erreurs.
47	01110**0**1000110	01110**1**1000110	1 erreurs détectées	Le bloc est corrigé.

Message corrigé

1010100010101 0111011000110 0111011000110 1010100010101 1010100010101 1010100010101 0111011000110
0111011000110 1010100010101 0111011000110 0111011000110 1010100010101 0111011000110 0111011000110
0111011000110 0111011000110 1010100010101 0111011000110 0111011000110 0111011000110 1010100010101
0111011000110 1010100010101 0111011000110 1010100010101 0111011000110 0111011000110 1010100010101
1010100010101 1010100010101 0111011000110 0111011000110 1010100010101 0111011000110 0111011000110
1010100010101 0111011000110 0111011000110 0111011000110 0111011000110 1010100010101 0111011000110
0111011000110 0111011000110 1010100010101 0111011000110 1010100010101 0111011000110

3.5.6 Décodage du message

Message binaire décodé

0110001101101 1110111010101 1000110110111 101110101

Message final

coucou

Chapitre 4

Exemples d'utilisation du code de Hamming

4.1 Avant propos

Les codes de Hamming ne permettent pas de corriger beaucoup d'erreurs, cela dit ils sont utilisés dans les cas où les erreurs ne se succèdent pas. C'est à dire que les erreurs doivent être relativement dispersées dans le message reçu. Pour des canaux très parasités, il faudra plutôt utiliser d'autres codes correcteurs d'erreurs qui permettent de corriger plus d'erreurs.

4.2 Choix des exemples

Le premier exemple montre la transmission d'un message contenant assez d'erreurs pour que le code de Hamming utilisé puisse les corriger. Dans le deuxième exemple, au contraire, le canal génère plus d'erreurs que le code correcteur peut en corriger.

4.2.1 Définitions

Définition 4 (Message) *Un message est une chaîne de caractères à transmettre via un canal bruité.*

Définition 5 (Bloc) *Un bloc est une sous-partie d'un message. Les blocs sont de longueur fixe.*

Définition 6 (Mot) *Un mot est un bloc qui a été encodé en utilisant un code correcteur d'erreurs.*

4.3 Exemple 1

4.3.1 Caractéristiques du code correcteur utilisé

Taille de bloc

La taille de bloc utilisé est de 4 bits.

Taille de mot

La taille d'un mot associé à un bloc est de 7 bits.

Nombre d'erreurs détectées

Les codes de Hamming ne peuvent détecter que 2 erreurs.

Nombre d'erreurs corrigées

Les codes de Hamming ne peuvent corriger qu'une seule erreur.

Matrice de contrôle

Voici ci-dessous la matrice de contrôle du code, qui permet de vérifier si le mot reçu contient des erreurs :

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Matrice de contrôle sous forme systématique

Voici ci-dessous la matrice précédente mais sous une forme plus pratique à utiliser :

$$\begin{pmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

Matrice de génératrice sous forme systématique

Enfin, voici la matrice permettant d'encoder un bloc du message, pour obtenir un mot à transmettre :

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

4.3.2 Message original

Chaîne de caractères originale

Afin illustrer le fonctionnement de ce code correcteur d'erreurs, nous avons choisis d'utiliser le message suivant :

Bonjour le monde!!

Message original traduit en binaire

La représentation du message original est la suivante :

01000010 01101111 01101110 01101010 01101111 01110101 01110010 00100000 01101100 01100101
00100000 01101101 01101111 01101110 01100100 01100101 00100001 00100001

4.3.3 Message encodé

Pour encoder le message, nous associons chaque bloc du message à un mot du code.

0100011 0010110 0110101 1111111 0110101 1110000 0110101 1010011 0110101 1111111 0111010
0101100 0111010 0010110 0010110 0000000 0110101 1100110 0110101 0101100 0010110 0000000
0110101 1101001 0110101 1111111 0110101 1110000 0110101 0100011 0110101 0101100 0010110
0001111 0010110 0001111

4.3.4 Simulation du passage dans un canal bruité

4.3.5 Caractéristiques du canal

Le canal utilisé est capable de générer des erreurs de façon aléatoire. Le canal peut générer au plus 1 erreur(s) par mot transmis.

Message reçu

0101011 0110110 0110101 1111111 0110101 1110100 0110101 1010011 0110111 1111101
0111010 0101100 0111010 0010100 0010100 0000000 1110101 1100110 0110101 0101100
0011110 0000000 0110101 1101011 0110111 1111111 0111101 0110000 0110100 0100011
0110101 0111100 0110110 0001111 0110110 0001111

Les erreurs générées par le canal apparaissent en rouge.

4.3.6 Détection et correction des erreurs

Message corrigé

0100011 0010110 0110101 1111111 0110101 1110000 0110101 1010011 0110101 1111111
0111010 0101100 0111010 0010110 0010110 0000000 0110101 1100110 0110101 0101100
0010110 0000000 0110101 1101001 0110101 1111111 0110101 1110000 0110101 0100011
0110101 0101100 0010110 0001111 0010110 0001111

4.3.7 Décodage du message

Message binaire décodé

01000010 01101111 01101110 01101010 01101111 01110101 01110010 00100000 01101100 01100101 00100000 01101101 01101111 01101110 01100100 01100101 00100001 00100001

Message final

```
Bonjour le monde!!
```

4.4 Exemple 2

4.4.1 Caractéristiques du code correcteur utilisé

Taille de bloc

La taille de bloc utilisé est de 4 bits.

Taille de mot

La taille d'un mot associé à un bloc est de 7 bits.

Nombre d'erreurs détectées

Les codes de Hamming ne peuvent détecter que 2 erreurs.

Nombre d'erreurs corrigées

Les codes de Hamming ne peuvent corriger qu'une seule erreur.

Matrice de contrôle

Voici ci-dessous la matrice de contrôle du code, qui permet de vérifier si le mot reçu contient des erreurs :

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Matrice de contrôle sous forme systématique

Voici ci-dessous la matrice précédente mais sous une forme plus pratique à utiliser :

$$\begin{pmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

Matrice de génératrice sous forme systématique

Enfin, voici la matrice permettant d'encoder un bloc du message, pour obtenir un mot à transmettre :

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

4.4.2 Message original

Chaîne de caractères originale

Afin illustrer le fonctionnement de ce code correcteur d'erreurs, nous avons choisis d'utiliser le message suivant :

Bonjour le monde!!

Message original traduit en binaire

La représentation du message original est la suivante :

01000010 01101111 01101110 01101010 01101111 01110101 01110010 00100000 01101100 01100101 00100000 01101101 01101111 01101110 01100100 01100101 00100001 00100001

4.4.3 Message encodé

Pour encoder le message, nous associons chaque bloc du message à un mot du code.

0100011 0010110 0110101 1111111 0110101 1110000 0110101 1010011 0110101 1111111 0111010 0101100 0111010 0010110 0010110 0000000 0110101 1100110 0110101 0101100 0010110 0000000 0110101 1101001 0110101 1111111 0110101 1110000 0110101 0100011 0110101 0101100 0010110 0001111 0010110 0001111

4.4.4 Simulation du passage dans un canal bruité

4.4.5 Caractéristiques du canal

Le canal utilisé est capable de générer des erreurs de façon aléatoire. Le canal peut générer au plus 2 erreur(s) par mot transmis.

Message reçu

0100011 0010111 0111101 0111111 0110001 0110000 0110101 0010011 0111101 1111111
0101011 1101100 0111010 0010110 0010110 1001000 0110101 1000110 0100101 0101110
0100110 0000000 0110111 1101101 0110111 0111011 0110101 1010001 0010101 0101010

0110100 0101110 1010110 1001110 1010110 1001110

Les erreurs générées par le canal apparaissent en rouge.

4.4.6 Détection et correction des erreurs

Message corrigé

0100011 0010110 0110101 1111111 0110101 1110000 0110101 1010011 0110101 1111111
0100011 0101100 0111010 0010110 0010110 1001010 0110101 1100110 0110101 0101100
1100110 0000000 0110101 1101001 0110101 0111010 0110101 1010011 0110101 0111010
0110101 0101100 0010110 1001010 0010110 1001010

4.4.7 Décodage du message

Message binaire décodé

01000010 01101111 01101110 01101010 01101111 01000101 01110010 00101001 01101100 01100101
11000000 01101101 01100111 01101010 01100111 01100101 00101001 00101001

Message final

```
BonjoEr)leÀmgjge))
```

4.5 Exemple 3

4.5.1 Caractéristiques du code correcteur utilisé

Taille de bloc

La taille de bloc utilisé est de 26 bits.

Taille de mot

La taille d'un mot associé à un bloc est de 31 bits.

Nombre d'erreurs détectées

Les codes de Hamming ne peuvent détecter que 2 erreurs.

Nombre d'erreurs corrigées

Les codes de Hamming ne peuvent corriger qu'une seule erreur.

Matrice de contrôle

Voici ci-dessous la matrice de contrôle du code, qui permet de vérifier si le mot reçu contient des erreurs :

$$\begin{pmatrix}
1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1
\end{pmatrix}$$

Matrice de contrôle sous forme systématique

Voici ci-dessous la matrice précédente mais sous une forme plus pratique à utiliser :

$$\begin{pmatrix}
1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}$$

Matrice de génératrice sous forme systématique

Enfin, voici la matrice permettant d'encoder un bloc du message, pour obtenir un mot à transmettre :

$$\left(\begin{array}{cccccccccccccccccccccccccccccccc}
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&0&1&1\\
0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1\\
0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&0&0&0\\
0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&1&1&1\\
0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&1&0&0\\
0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&0&0\\
0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&0&0\\
0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1\\
0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&1&0\\
0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&1&0\\
0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&0&1&0\\
0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&0\\
0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&1&1&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&1&1&1&1&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&1&0&0&0&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&1&0&0&0&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&1&1&0&0&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&1&0&1&0&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&1&0&1&0&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&1&1&0&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&1&1&1&0&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&1&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&1&0&0&1&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&1&0&1&1
\end{array}\right)$$

4.5.2 Message original

Chaîne de caractères originale

Afin illustrer le fonctionnement de ce code correcteur d'erreurs, nous avons choisis d'utiliser le message suivant :

Bonjour le monde!!

Message original traduit en binaire

La représentation du message original est la suivante :

01000010 01101111 01101110 01101010 01101111 01110101 01110010 00100000 01101100 01100101 00100000 01101101 01101111 01101110 01100100 01100101 00100001 00100001

4.5.3 Message encodé

Pour encoder le message, nous associons chaque bloc du message à un mot du code.

0100001001101111011011100101100 1010100110111101110101011111010 0010001000000110110001100110111
0100100000011011010110111111101 0110111001100100011001010011011 1000010010000100000000000001011

4.5.4 Simulation du passage dans un canal bruité

4.5.5 Caractéristiques du canal

Le canal utilisé est capable de générer des erreurs de façon aléatoire. Le canal peut générer au plus 1 erreur(s) par mot transmis.

Message reçu

0100001001101111011011100101100 1010100110111101110101010111010 0010001001000110110001100110111
0100100000011011010110110111101 0110111001100100011001010011011 1000010010000100000000000000011

Les erreurs générées par le canal apparaissent en rouge.

4.5.6 Détection et correction des erreurs

Message corrigé

0100001001101111011011100101100 1010100110111101110101011111010 0010001000000110110001100110111
0100100000011011010110111111101 0110111001100100011001010011011 1000010010000100000000000001011

4.5.7 Décodage du message

Message binaire décodé

01000010 01101111 01101110 01101010 01101111 01110101 01110010 00100000 01101100 01100101
00100000 01101101 01101111 01101110 01100100 01100101 00100001 00100001

Message final

```
Bonjour le monde!!
```

Chapitre 5

Cahier de programmation de l'application RSA

5.1 Liste complète des fonctions de l'application

5.1.1 About

Prototype de la fonction

LRESULT CALLBACK About (HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)

Paramètres

en entrée

- HWND hDlg : L'identifiant de la boite de dialogue.
- UINT message : Message événementiel.
- WPARAM wParam : 1^{er} Paramètre du message.
- LPARAM lParam : 2^{eme} Paramètre du message.

en sortie

- LRESULT : Réponse spécifique au message reçu en entrée.

Description

Cette fonction gère la boucle des messages associés à la boîte de dialogue About. La boîte de dialogue About contient des informations sur la version de l'application, ses auteurs...

5.1.2 affiche_barre_progression

Prototype de la fonction

void affiche_barre_progression (HWND barre, BOOL bVisible)

Paramètres

en entrée
- HWND barre : L'identifiant de la barre de progression.
- BOOL bVisible : Booléen indiquant si la barre de progression doit être masquée (False) ou affichée (True).

en sortie
- void : Rien.

Description

Cette fonction permet d'afficher ou de masquer une barre de progression. Cela permet de ne l'afficher que lorsqu'une opération est en cours d'exécution.

5.1.3 chiffrer

Prototype de la fonction

void chiffrer (HWND hWnd, long identifiant, char *source, char *destination, mpz_t n, mpz_t e)

Paramètres

en entrée
- HWND hWnd : L'identifiant de la boite de dialogue appelant la fonction de chiffrement.
- long identifiant : Identifiant de la barre de progression.
- char *source : Le chemin du fichier que l'on souhaite chiffrer.
- char *destination : Le chemin du fichier résultant du chiffrement du fichier source.
- mpz_t n : Entier n de la clé publique (n,e).
- mpz_t e : Entier e de la clé publique (n,e).

en sortie
- void : Rien.

Description

Cette fonction permet de chiffrer par RSA un fichier source et d'enregistrer le résultat dans un autre fichier en utilisant une clé publique (n,e).

5.1.4 chiffrerdlg

Prototype de la fonction

LRESULT CALLBACK chiffrerdlg (HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)

Paramètres

en entrée
- HWND hWnd : L'identifiant de la boite de dialogue.
- UINT message : Message événementiel.
- WPARAM wParam : 1^{er} Paramètre du message.
- LPARAM lParam : 2^{eme} Paramètre du message.

en sortie
- LRESULT : Réponse spécifique au message reçu en entrée.

Description

Cette fonction gère la boucle des messages associés à la boîte de dialogue chiffrerdlg. La boîte de dialogue chiffrerdlg permet de chiffrer un document, charger un fichier de clé publique. Cette boîte de dialogue possède une barre de progression pour suivre l'état d'avancement du chiffrement.

5.1.5 clersa

Prototype de la fonction

LRESULT CALLBACK clersa (HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)

Paramètres

en entrée
- HWND hWnd : L'identifiant de la boite de dialogue.
- UINT message : Message événementiel.
- WPARAM wParam : 1^{er} Paramètre du message.
- LPARAM lParam : 2^{eme} Paramètre du message.

en sortie
- LRESULT : Réponse spécifique au message reçu en entrée.

Description

Cette fonction gère la boucle des messages associés à la boîte de dialogue clersa. La boîte de dialogue clersa permet la création et l'enregistrement de clés RSA.

5.1.6 compter_nombre_dechiffrer

Prototype de la fonction

long compter_nombre_dechiffrer (char *source)

Paramètres

en entrée
- char *source : Chemin d'un fichier chiffré par RSA.

en sortie
- long : Le nombre de blocs (d'entiers) du fichier.

Description

Cette fonction permet de connaître le nombre de blocs chiffrés par RSA. Cela permet de connaitre le nombre d'étapes nécessaires pour l'opération de déchiffrement, et ainsi de mettre la barre de progression du déchiffrement à jour.

5.1.7 dechiffrer

Prototype de la fonction

void dechiffrer (HWND hWnd, long identifiant, char *source, char *destination, mpz_t n, mpz_t d)

Paramètres

en entrée
- HWND hWnd : L'identifiant de la boite de dialogue appelant la fonction de déchiffrement.
- long identifiant : Identifiant de la barre de progression.
- char *source : Le chemin du fichier que l'on souhaite déchiffrer.
- char *destination : Le chemin du fichier résultant du déchiffrement du fichier source.
- mpz_t n : Entier n de la clé privée (n,d).
- mpz_t e : Entier e de la clé privée (n,d).

en sortie
- void : Rien.

Description

Cette fonction permet de déchiffrer un fichier précédemment crypté par RSA et d'enregistrer le résultat dans un autre fichier en utilisant une clé privée (n,d).

5.1.8 dechiffrerdlg

Prototype de la fonction

LRESULT CALLBACK dechiffrerdlg (HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)

Paramètres

en entrée
- HWND hWnd : L'identifiant de la boite de dialogue.
- UINT message : Message événementiel.
- WPARAM wParam : 1^{er} Paramètre du message.
- LPARAM lParam : 2^{eme} Paramètre du message.

en sortie
- LRESULT : Réponse spécifique au message reçu en entrée.

Description

Cette fonction gère la boucle des messages associés à la boîte de dialogue dechiffrerdlg. La boîte de dialogue dechiffrerdlg permet de déchiffrer un document, charger un fichier de clé privée. Cette boîte de dialogue possède une barre de progression pour suivre l'état d'avancement du déchiffrement.

5.1.9 ecrire_mpz

Prototype de la fonction

void ecrire_mpz (FILE *fp, mpz_t b, long nbcl)

Paramètres

en entrée
- FILE *fp : Pointeur vers le fichier de destination.
- mpz_t b : Un entier b que l'on souhaite écrire dans le fichier de destination.
- long nbcl : Taille de l'entier b avant chiffrement par RSA.

en sortie
- void : Rien.

Description

Cette fonction permet d'écrire un entier dans un fichier. La structure d'enregistrement choisie est la suivante :
- nombre de digits de l'entier b avant chiffrement.
- nombre de digits de l'entier b après chiffrement.
- chaîne de caractères représentant l'entier b après chiffrement.

5.1.10 finalisation_barre_progression

Prototype de la fonction

void finalisation_barre_progression (HWND hWnd, long identifiant)

Paramètres

en entrée
- HWND hWnd : L'identifiant de la fenêtre parente de la barre de progression.
- long identifiant : Identifiant de la barre de progression.

en sortie
- void : Rien.

Description

Cette fonction permet de masquer et de réinitialiser une barre de progression à la fin d'une opération.

5.1.11 generer_d

Prototype de la fonction

void generer_d (mpz_t d, mpz_t e, mpz_t psi_n)

Paramètres

en entrée
- mpz_t d : Pointeur vers un entier d.
- mpz_t e : Pointeur vers un entier e.
- mpz_t psi_n : Pointeur vers un entier psi_n.

en sortie
- void : Rien.

Description

Cette fonction calcule la valeur de l'entier d en fonction de e et de psi_n, grâce à la formule suivante : $d = e^{-1} (\text{mod } \varphi(\text{n}))$.

5.1.12 generer_e

Prototype de la fonction

void generer_e (mpz_t e, mpz_t psi_n)

Paramètres

en entrée
- mpz_t e : Pointeur vers un entier e.
- mpz_t psi_n : Pointeur vers un entier psi_n.

en sortie
- void : Rien.

Description

Cette fonction génère un entier et l'enregistre dans e tel que PGCD(e,φ(n))=1 avec $1 < e < \varphi(n)$.

5.1.13 generer_une_cle_RSA_aleatoire

Prototype de la fonction

void generer_une_cle_RSA_aleatoire (mpz_t p, mpz_t q, mpz_t e, mpz_t d, mpz_t n, mpz_t psi_n)

Paramètres

en entrée
- mpz_t p : Pointeur vers un entier p.
- mpz_t q : Pointeur vers un entier q.
- mpz_t e : Pointeur vers un entier e.
- mpz_t d : Pointeur vers un entier d.
- mpz_t n : Pointeur vers un entier n.
- mpz_t psi_n : Pointeur vers un entier psi_n.

en sortie
- void : Rien.

Description

Cette fonction génère une clé RSA valide :

1. Elle génère deux nombres premiers qu'elle stocke dans p et q.
2. Elle calcule pq et stocke le résultat dans n.
3. Elle calcule (p-1)(q-1) et stocke le résultat dans psi_n.
4. Elle génère un entier et l'enregistre dans e tel que PGCD(e,φ(n))=1 avec $1 < e < \varphi(n)$.
5. Elle calcule un entier en fonction de e et de psi_n et le stocke dans d, grâce à la formule suivante : $d = e^{-1} (\text{mod } \varphi(\text{n}))$.

5.1.14 incremente_barre_progression

Prototype de la fonction

void incremente_barre_progression (HWND hWnd, long identifiant)

Paramètres

en entrée
- HWND hWnd : L'identifiant de la fenêtre parente de la barre de progression.
- long identifiant : Identifiant de la barre de progression.

en sortie
- void : Rien.

Description

Cette fonction fait avancer la barre de progression d'un cran.

5.1.15 initialisation_barre_progression

Prototype de la fonction

void initialisation_barre_progression (HWND hWnd, long max, long identifiant)

Paramètres

en entrée
- HWND hWnd : L'identifiant de la fenêtre parente de la barre de progression.
- long max : Le nombre d'étapes maximum lors de la progression de la barre.
- long identifiant : Identifiant de la barre de progression.

en sortie
- void : Rien.

Description

Cette fonction initialise une barre de progression en lui affectant une borne max et en mettant sa progression initiale à 0.

5.1.16 initrandom

Prototype de la fonction

void initrandom (void)

Paramètres

en entrée
- void : Rien

en sortie
- void : Rien.

Description

Cette fonction initialise la génération de grands nombres aléatoires.

5.1.17 lire_mpz

Prototype de la fonction

long lire_mpz (FILE *fp, mpz_t b)

Paramètres

en entrée

- FILE *fp : Pointeur de fichier à partir duquel on souhaite lire un entier
- mpz_t b : Pointeur vers l'entier dans lequel sera stocké l'entier lu.

en sortie

- long : Taille (en nombre de digits) de l'entier avant chiffrement par RSA.

Description

Cette fonction permet de lire un entier à partir d'un fichier source en respectant la structure :
– nombre de digits de l'entier b avant chiffrement.
– nombre de digits de l'entier b après chiffrement.
– chaîne de caractères représentant l'entier b après chiffrement.

5.1.18 trouver_un_nombre_premier

Prototype de la fonction

void trouver_un_nombre_premier (mpz_t a)

Paramètres

en entrée

- mpz_t a : Pointeur vers l'entier dans lequel sera stocké le résultat.

en sortie

- void : Rien.

Description

Cette fonction génère un nombre premier. Cette génération utilise le test de Miller-Rabin itéré 10 fois.

5.1.19 transformer_en_chaine

Prototype de la fonction

char * transformer_en_chaine (mpz_t b, long nbcl)

Paramètres

en entrée
- mpz_t b : Pointeur vers l'entier à transformer en chaîne de caractères.
- long nbcl : Taille finale de la chaîne de caractères.

en sortie
- char * : Pointeur vers la chaîne de caractères résultat.

Description

Cette fonction transforme un entier en chaîne de caractères. Le paramètre nbcl nous indique la taille originale de l'entier(en nombre de digits). Si cette taille n'est pas atteinte, on rajoute des 0 en tête de la chaîne pour la compléter. Ces 0 sont nécessaires, car chaque groupe de 3 caractères représente le code ASCII d'une lettre.

5.1.20 transformer_en_mpz

Prototype de la fonction

void transformer_en_mpz (mpz_t b, unsigned char *chaine, long taille)

Paramètres

en entrée
- mpz_t b : Pointeur vers l'entier dans lequel sera stocké le résultat.
- unsigned char *chaine : Chaîne de caractères que l'on souhaite convertir en entier.
- long taille : taille de la chaîne de caractères.

en sortie
- void : Rien.

Description

Cette fonction traduit une chaîne de caractères en un entier de la façon suivante : chaque caractère est convertit en son code ASCII concaténé à la suite. Par exemple la chaîne "abc" nous donne l'entier "097098099".

5.1.21 verification_validite_cle_RSA

Prototype de la fonction

int verification_validite_cle_RSA (mpz_t p, mpz_t q, mpz_t e, mpz_t d, mpz_t n, mpz_t psi_n)

Paramètres

en entrée
- mpz_t p : Pointeur vers un entier p.
- mpz_t q : Pointeur vers un entier q.
- mpz_t e : Pointeur vers un entier e.
- mpz_t d : Pointeur vers un entier d.
- mpz_t n : Pointeur vers un entier n.
- mpz_t psi_n : Pointeur vers un entier psi_n.

en sortie
- int : 0 si la clé est valide, 1 sinon.

Description

Cette fonction vérifie si une clé RSA est valide ou non.

5.1.22 WndProc

Prototype de la fonction

LRESULT CALLBACK WndProc (HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)

Paramètres

en entrée
- HWND hWnd : L'identifiant de la boite de dialogue.
- UINT message : Message événementiel.
- WPARAM wParam : 1^{er} Paramètre du message.
- LPARAM lParam : 2^{eme} Paramètre du message.

en sortie
- LRESULT : Réponse spécifique au message reçu en entrée.

Description

Cette fonction gère la boucle des messages associés à la Fenêtre principale. La fenêtre principale permet d'accéder aux différentes boites de dialogues de l'application (chiffrement d'un document, déchiffrement d'un document, signature et vérification de la provenance d'un document, génération de clés RSA...). La fenêtre principale affiche en permanence l'aide de l'application.

5.1.23 designerdlg

Prototype de la fonction

LRESULT CALLBACK designerdlg (HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)

Paramètres

en entrée
- HWND hWnd : L'identifiant de la boite de dialogue.
- UINT message : Message événementiel.
- WPARAM wParam : 1^{er} Paramètre du message.
- LPARAM lParam : 2^{eme} Paramètre du message.

en sortie
- LRESULT : Réponse spécifique au message reçu en entrée.

Description

Cette fonction gère la boucle des messages associés à la boîte de dialogue designerdlg. La boîte de dialogue designerdlg permet de vérifier qu'un document provient bien de la personne attendue. Cette boîte de dialogue possède une barre de progression pour suivre l'état d'avancement de cette vérification.

5.1.24 signerdlg

Prototype de la fonction

LRESULT CALLBACK signerdlg (HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)

Paramètres

en entrée
- HWND hWnd : L'identifiant de la boite de dialogue.
- UINT message : Message événementiel.
- WPARAM wParam : 1^{er} Paramètre du message.
- LPARAM lParam : 2^{eme} Paramètre du message.

en sortie
- LRESULT : Réponse spécifique au message reçu en entrée.

Description

Cette fonction gère la boucle des messages associés à la boîte de dialogue signerdlg. La boîte de dialogue signerdlg permet de signer un document, ainsi, tout le monde sera capable de lire ce document en étant sûr de sa provenance. Cette boîte de dialogue possède une barre de progression pour suivre l'état d'avancement de la signature du document.

5.1.25 mise_a_jour_TAILLEBLOC

Prototype de la fonction

void mise_a_jour_TAILLEBLOC (mpz_t n)

Paramètres

en entrée

- mpz_t n : Pointeur vers l'entier n d'une clé RSA.

en sortie

- void : Rien.

Description

Cette fonction calcule la taille maximum des blocs que nous pourrons chiffrer par RSA.

5.2 Les choix de programmation

5.2.1 Gestion des clés

Durant nos tests, nous nous sommes rendu compte qu'il était très gênant de faire des copier/coller pour pouvoir réutiliser une clé RSA. Nous avons donc décider de créer un format de fichier permettant la sauvegarde des clés et leur chargement.

Génération de clés

Lorsque l'on génère une clé grâce à notre générateur de clé, il est possible de la sauvegarder. Cette sauvegarde créée deux fichiers :

- Un fichier .public, qui contiendra la clé publique, celle qui doit être publiée (n,e).
- Un fichier .private, qui contiendra la clé privée, qui doit rester secrète (n,d).

Les fichiers de clé privée

Les fichiers de clé privée sont organisés de la façon suivante :
"N :" chaîne de caractères représentant le module n en base 10.
"D :" chaîne de caractères représentant l'exposant d en base 10.

Les fichiers de clé publique

Les fichiers de clé publique sont organisés de la façon suivante :
"N :" chaîne de caractères représentant le module n en base 10.
"E :" chaîne de caractères représentant l'exposant e en base 10.

chargement de clé

Dans notre application, il est possible de charger les clés directement à partir d'un bouton. Ce bouton ouvre une boite de dialogue qui permet de sélectionner le fichier de clé.

Pour le chiffrement Dans la boite de dialogue de chiffrement, le bouton permet uniquement de charger les fichiers de clé publique (*.public).

Pour le déchiffrement Dans la boite de dialogue de déchiffrement, le bouton permet uniquement de charger les fichiers de clé privée (*.private).

Pour la signature Dans la boite de dialogue de signature, le bouton permet uniquement de charger les fichiers de clé privée (*.private).

Pour la vérification de provenance Dans la boite de dialogue de vérification de la provenance d'un message, le bouton permet uniquement de charger les fichiers de clé publique (*.public).

5.3 Les jeux de tests

5.3.1 Temps d'exécution

Dans cette partie, nous allons montrer l'influence de la taille des différents paramètres du chiffrement RSA en les modifiants tour à tour.

En changeant la taille de e

Soit n=428633664584670270469430393764530778544629485710250294386931332817473626
750920109593674642701863924599415714694491197074112441730674738060912875160982461 3.

Nous utiliserons une taille de bloc de 20 caractères. Nous allons modifier la taille de e :

Valeur de e	Nombre de blocs chiffrés par seconde
924794307772639867324626413446633783315 56908320669127609413004055903089501831	301
65537	4602
3	17490

Conclusion Plus la taille de e augmente, moins le nombre de blocs chiffrés par seconde est élevé (on peut même dire que les performances s'effondrent). Cette augmentation de la durée du chiffrement est dûe à l'augmentation de la complexité de calcul de l'exponentiation modulaire. Attention ! On peut être tenté d'utiliser une taille de e relativement faible pour accélérer le temps de calcul. C'est une mauvaise idée car il existe des attaques permettant de casser les exposants (e ou d) de faible valeur (voir attaque de l'exposant e=3 dans le chapitre Sécurité).

En changeant la taille de n

Soit e=3. Nous utiliserons une taille de bloc de 3 caractères. Nous allons modifier la taille de n :

Valeur de n	Nombre de blocs chiffrés par seconde
342083343795753056870927614526375330044065880202645075605165660012034515910 9	63889
181498788542106317	233196
821088689	310928

Conclusion La taille de n borne les valeurs des exposants e et d, ce qui permet d'accélérer les calculs lorsque l'on diminue la valeur de n (et donc de e et d), mais cela fait baisser la sécurité du chiffrement (car la taille des clés dépends de n). De plus, il faut faire attention, car avec RSA on ne peut pas chiffrer de bloc>n, ainsi la taille de bloc va diminuer si on diminue la valeur de n. Attention ! Une taille de bloc faible (<4) permet des attaques basées sur la fréquence d'apparition des caractères.

En changeant la taille de bloc

Soit n=428633664584670270469430393764530778544629485710250294386931332817473626
75092010959367464270 1 8639245994157146944911970741124417306747380609128751609824613.

Soit e=3. Nous allons modifier la taille de bloc :

Taille de bloc utilisée	Nombre de blocs chiffrés par seconde	Temps d'exécution en secondes
50	13991	10
25	16827	17
5	20576	68

Conclusion Diminuer la taille de bloc permet d'accélérer la vitesse du chiffrement d'un bloc. Cette constatation est trompeuse. En effet, on peut penser qu'une petite taille de bloc augmente la vitesse de chiffrement d'un fichier, or c'est faux. Si on regarde les temps d'exécution, on se rend compte qu'on a allongé la durée totale du chiffrement. Cet allongement du temps d'exécution vient de l'augmentation du nombre de blocs total.

Exemple : Avec un message de 9 caractères et une taille de bloc de 9 caractères on mettra 1 seconde pour chiffrer ce bloc. Donc le temps total d'exécution sera d'une seconde. Si on prend une taille de bloc de 3 caractères, on améliorera le temps de chiffrement d'un bloc à 0,5 secondes. Cependant, le temps total d'exécution sera de 1,5 secondes.

En pratique, il faudra trouver un bon compromis entre nombre de blocs et le temps de chiffrement d'un bloc, afin d'obtenir les meilleurs performances.

Chapitre 6

Cahier de programmation de l'application Distance de Hamming

6.1 Liste complète des fonctions

6.1.1 affiche_alphabet

Prototype de la fonction

void affiche_alphabet(FILE *fp,ALPHABET *code)

Paramètres

en entrée
- FILE *fp : Le fichier dans lequel on souhaite écrire le code latex résultant.
- ALPHABET *code : L'alphabet que l'on souhaite utiliser.

en sortie
- void : rien

Description

Cette fonction écrit dans un fichier le code latex correspondant au tableau suivant : première colonne : Bloc du message, deuxième colonne : Code correspondant au bloc. La première colonne correspond aux différents blocs de l'alphabet. La deuxième colonne donne le mot associé au bloc correspondant.

6.1.2 afficher_bilan_bloc

Prototype de la fonction

void afficher_bilan_bloc(FILE *fp,char *message,ALPHABET *code,int nberreurs) ;

Paramètres

en entrée
- FILE *fp : : Le fichier dans lequel on souhaite écrire le code latex résultant.
- char *message : Le message bruité que l'on souhaite corriger.
- ALPHABET *code : L'alphabet utilisé pour encoder le message.
- int nberreurs : Le nombre maximum d'erreurs pouvant êtres corrigées.

en sortie
- void : rien

Description

Le résultat de cette fonction est produit sous forme de code Latex.
Cette fonction décrit sous forme de tableau, mot par mot, la détection des erreurs d'un message bruité. Pour chaque mot du message bruité, cette fonction donne le mot de l'alphabet étant le plus proche (par la distance de Hamming), et indique si il contenait des erreurs ? Si oui, alors un message indique si le mot à été corrigé, ou non.

6.1.3 afficher_les_erreurs

Prototype de la fonction

void afficher_les_erreurs(FILE *fp,char *message_recu,char *message_original,int taille,int couleur) ;

Paramètres

en entrée
- FILE *fp : Le fichier dans lequel on souhaite écrire le code latex résultant.
- char *message_recu : Chaîne de caractères représentant le message bruité
- char *message_original : Chaîne de caractères représentant le message original
- int taille : le nombre de caractères à vérifier.
- int couleur : couleur avec laquelle on souhaite afficher les différences entre les deux chaînes.

en sortie
- void : rien

Description

Le résultat de cette fonction est produit sous forme de code Latex.
Cette fonction prends deux chaînes de caractères en entrée. Ces deux chaînes sont supposées de même longueur. La fonction affiche la première chaîne de caractère. Chaque fois qu'un caractère de même indice est différent sur les deux chaînes, alors il est mit en gras et est affiché en couleur. Si le paramètre couleur vaut 0, alors la couleur utilisée sera le rouge, le bleu sinon.

6.1.4 binary_to_chaine

Prototype de la fonction

char *binary_to_chaine(char *message_binaire) ;

Paramètres

en entrée
- char *message_binaire : La chaîne 'binaire' que l'on souhaite traduire en chaîne de caractères.

en sortie
- char * : La chaîne traduite.

Description

Cette fonction traduit une chaîne de caractères composée uniquement de '0' et de '1' en chaîne de caractères normale. Par exemple, la chaîne "01000001" sera traduite en "a".

6.1.5 calcul_distance

Prototype de la fonction

int calcul_distance(char ** alphabet,int index,mpz_t cpt) ;

Paramètres

en entrée
- char ** alphabet : Un alphabet en cours de construction.
- int index : L'indice du dernier élément de l'alphabet.
- mpz_t cpt : L'entier que l'on teste.

en sortie
- int : La distance minimale entre l'entier testé et les autres éléments de l'alphabet.

Description

Cette fonction nous permet de vérifier qu'un entier est bien à une certaine distance de tous les blocs de notre alphabet. Si la distance retournée est suffisamment grande, alors on pourra insérer l'entier, sinon on ne l'utilisera pas.

6.1.6 chaine_to_binary

Prototype de la fonction

char *chaine_to_binary(char *message) ;

Paramètres

en entrée
- char *message : La chaîne de caractères à traduire.

en sortie
- char * : La chaîne traduite.

Description

Cette fonction traduit une chaîne de caractères normale en suite de '0' et de '1'. Par exemple, la chaîne "aa" sera traduite en "0100000101000001".

6.1.7 coder_le_message

Prototype de la fonction

char *coder_le_message(char *message,ALPHABET *code) ;

Paramètres

en entrée
- char *message : Le message 'binaire' que l'on souhaite encoder.
- ALPHABET *code : L'alphabet que l'on souhaite utiliser pour encoder le message.

en sortie
- char * : La chaîne de caractères encodée.

Description

Cette fonction attend une chaîne de caractères 'binaire' et la découpe en blocs. Chaque bloc est ensuite remplacé par le mot qui lui est associé dans l'alphabet de notre code correcteur. La chaîne de caractères retournée est donc le message encodé.

6.1.8 compter_nombre_de_blocs_differents

Prototype de la fonction

int compter_nombre_de_blocs_differents(char *message_binaire,int taille_de_bloc) ;

Paramètres

en entrée
- char *message_binaire : La chaîne de caractères 'binaire' dont on souhaite connaître le nombre de blocs différents.
- int taille_de_bloc : La taille de bloc utilisée.

en sortie

- int : Le nombres de blocs différents.

Description

Cette fonction compte le nombre de bloc différents de taille "taille_de_bloc" dans une chaîne de caractères 'binaire'.

6.1.9 corriger_le_message

Prototype de la fonction

char * corriger_le_message(char *message_binaire,ALPHABET *code) ;

Paramètres

en entrée

- char *message_binaire : Le message "binaire" bruité que l'on souhaite corriger.
- ALPHABET *code : L'alphabet de notre code correcteur.

en sortie

- char * : Le message "binaire" corrigé.

Description

Cette fonction utilise le code correcteur afin de retrouver un message non bruité.

6.1.10 corriger_le_mot

Prototype de la fonction

char *corriger_le_mot(char *mot,ALPHABET *code) ;

Paramètres

en entrée

- char *mot : Le mot que le souhaite corriger.
- ALPHABET *code : L'alphabet de notre code correcteur.

en sortie

- char * : Le mot corrigé.

Description

Cette fonction attend un mot à corriger et renvoie le mot le plus proche (en utilisant la distance de Hamming) de l'alphabet.

6.1.11 decoder_le_message

Prototype de la fonction

char * decoder_le_message(char *message_binaire,ALPHABET *code) ;

Paramètres

en entrée
- char *message_binaire : Le message "binaire" encodé.
- ALPHABET *code : L'alphabet que l'on utilise pour coder/décoder nos messages.

en sortie
- char * : Le message décodé.

Description

Cette fonction attend une chaîne de caractères 'binaire' et la découpe en mots. Chaque mot est ensuite remplacé par le bloc qui lui est associé dans l'alphabet de notre code correcteur. La chaîne de caractères retournée est donc le message décodé. Cette fonction fait le travail inverse de la fonction encoder_le_message.

6.1.12 ecrire_fin

Prototype de la fonction

void ecrire_fin(FILE *fp) ;

Paramètres

en entrée
- FILE *fp :

en sortie
- void : rien

Description

Cette fonction ferme le fichier latex.

6.1.13 ecrire_message_binaire

Prototype de la fonction

void ecrire_message_binaire(FILE *fp,char *message,int taille) ;

Paramètres

en entrée
- FILE *fp : Le fichier dans lequel on souhaite écrire le code latex résultant.
- char *message
- int taille : La taille de bloc.

en sortie
- void : rien

Description

Le résultat de cette fonction est produit sous forme de code Latex.
Cette fonction écrit de façon formatée une chaîne 'binaire'. En effet, entre chaque bloc est inséré un espace. Cela permet de mieux visualiser cette chaîne.

6.1.14 genere_debut_latex

Prototype de la fonction

FILE * genere_debut_latex(char *destination) ;

Paramètres

en entrée
- char *destination : L'emplacement où l'on souhaite créer le fichier.

en sortie
- FILE * : Le descripteur de notre fichier latex.

Description

Cette fonction ouvre un fichier .tex en écriture, génère l'entête du fichier latex que l'on complétera par la suite et renvoie le descripteur de fichier.

6.1.15 generer_ALPHABET

Prototype de la fonction

ALPHABET *generer_ALPHABET(char *message,int taille_de_bloc,int taille_de_mot,int nombre_erreurs) ;

Paramètres

en entrée
- char *message : Le message à partir duquel sera généré l'alphabet.
- int taille_de_bloc : La taille de bloc utilisée.
- int taille_de_mot : La taille des mots générés.

- int nombre_erreurs : Le nombre d'erreurs détectées par notre code correcteur.

en sortie
- ALPHABET * : L'alphabet de notre code correcteur.

Description

Cette fonction génère un alphabet en fonction d'un message. On répertorie d'abord les différents blocs du message, puis on associe à chaque bloc un mot qui permettra d'encoder ce bloc. Les mots sont générés en fonction du nombre d'erreurs que le code correcteur est sensé détecter.

6.1.16 generer_alphabet

Prototype de la fonction

char ** generer_alphabet(int nombre_de_blocs,int nombre_erreurs,int taille_de_mot) ;

Paramètres

en entrée
- int nombre_de_blocs : Le nombre de mots à générer.
- int nombre_erreurs : La distance minimale des mots.
- int taille_de_mot : La taille d'un mot.

en sortie
- char ** : Le tableau des mots.

Description

Cette fonction génère un tableau de "nombre_de_blocs" chaînes de caractères. Chaque chaîne représente un mot. Tous les mots ont une distance de Hamming au moins de int nombre_erreurs les uns des autres. Tous les mots ont la même taille pour faciliter leur manipulation.

6.1.17 generer_erreurs

Prototype de la fonction

char *generer_erreurs(char *message,int nb_par_bloc,int taille) ;

Paramètres

en entrée
- char *message : Le message "binaire" que l'on souhaite bruiter.
- int nb_par_bloc : Le nombre d'erreurs maximum par bloc.
- int taille : La taille d'un bloc.

en sortie

- char * : La chaîne de caractères bruitée.

Description

Cette fonction simule la transition du message par un canal bruite. Ainsi, certains bits sont modifiés. Le nombre d'erreurs générées par bloc est limité à "nb_par_bloc".

6.1.18 get_bloc

Prototype de la fonction

char *get_bloc(int num_bloc,char *message_binaire,int taille_de_bloc) ;

Paramètres

en entrée

- int num_bloc : L'indice du bloc que l'on veut extraire.
- char *message_binaire : La chaîne de caractères "binaire" dont on veut extraire un bloc.
- int taille_de_bloc : La taille de bloc utilisée. Ce paramètre permet de connaître la position du bloc d'indice "num_bloc".

en sortie

- char * : Le bloc d'indice "num_bloc".

Description

Cette fonction retourne une chaîne de caractères contenant le bloc d'indice souhaité.

6.1.19 mettre_a_la_bonne_taille

Prototype de la fonction

char *mettre_a_la_bonne_taille(char *buffer,int taille) ;

Paramètres

en entrée

- char *buffer : La chaîne que l'on souhaite normaliser.
- int taille : La taille que l'on souhaite donner à la chaîne.

en sortie

- char * : La chaîne contenant le bon nombre de caractères.

Description

Cette fonction rajoute des '0' au début d'une chaîne de caractères jusqu'a ce qu'elle fasse la taille voulue.

6.2 Mode d'emploi de l'application

Notre programme simule le passage d'une chaîne de caractères au travers d'un canal bruité. Pour sécuriser la transmission, nous utilisons la distance de Hamming. Le programme génère un fichier latex.

6.2.1 Paramètres

Usage

dhamming.exe 1 2 3 4 5 6

Détail des arguments

1. nom du fichier contenant le message à transmettre.
2. taille de bloc.
3. taille de mot.
4. nombre d'erreurs maximum par mot.
5. distance de Hamming.
6. nom du fichier résultat.

Exemple d'appel du programme

dhamming toto.txt 5 10 1 3 resultat.tex

Il ne reste plus qu'a compiler le fichier généré (dans notre exemple resultat.tex) comme un fichier latex ordinaire. Attention, pour visionner le résultat, il faut s'assurer que le visionneur permet d'utiliser les couleurs.

6.3 Choix de programmation

6.3.1 Interface graphique

Nous avons eu du mal pour trouver une méthode acceptable afin d'afficher toutes les informations. En effet, les codes correcteurs génères beaucoup d'information :

- le message original
- le message original convertit en suite de nombres
- le message codé grâce au code correcteur
- les erreurs générée lors du passage du canal

– les erreurs détectées par le code correcteur
– les erreurs corrigées par le code correcteur
– les erreurs persistantes après la correction du code correcteur
– le message reçu décodé
– le message reçu (en chaîne de caractères)

Nous avons essayé d'utiliser une interface graphique conventionnelle (avec des fenêtres) mais le résultat était tout simplement illisible. Nous avons donc choisi de générer un fichier latex. Ce fichier contient toutes les étapes de la transition d'un message.

6.3.2 Simulation du passage via un canal bruité

Pour simuler le passage d'un message au travers d'un canal bruité, nous avons créé une fonction qui génère au maximum t erreurs par bloc (la taille d'un bloc étant précisée). Ces erreurs sont générées de manière aléatoire. Les erreurs sont donc réparties de façon homogène sur le message. Nous ne fabriquons pas de paquets d'erreurs.

6.3.3 Génération de l'alphabet

Nous avons choisi de généré un alphabet en fonction du message original, ce qui nous évite de générer un alphabet complet, qui possède une taille exponentielle. En effet, si on prend une taille de bloc de 8, on aura un nombre de blocs de $2^8 = 256$, et la majorité de ces blocs ne nous serons pas utiles. Nous économisons donc du temps (pour générer l'alphabet) et de la mémoire (pour stocker l'alphabet) puisque seuls les blocs présents dans le message seront stockés.

6.3.4 La structure ALPHABET

struct ALPHABET{ char ** alphabet ; char **blocs ; int cardinalite ; int taille_de_bloc ; int taille_de_mot ;} ;

Cette structure contient la liste des différents blocs pouvant êtres rencontré dans un message, ainsi que la liste contenant les mots associés à chaque bloc. Tous les blocs sont de la même taille. Tous les mots sont de la même taille. Le champs cardinalité renseigne sur la taille des deux tableaux.

Chapitre 7

Cahier de programmation de l'application Codes de Hamming

7.1 Liste complète des fonctions

7.1.1 affiche_erreur

Prototype de la fonction

void affiche_erreur(int *m1,int *m2,int taille,int couleur,FILE *fp)

Paramètres

en entrée

- int *m1 : Première matrice d'entiers
- int *m2 : Deuxième matrice d'entiers
- int taille : Taille des deux matrices d'entiers
- int couleur : Type de couleur d'affichage, 0 en rouge, 1 en bleu
- FILE *fp : Fichier latex de destination

en sortie

- void : Rien

Description

Cette fonction permet d'écrire dans le fichier destination les éléments du premier tableau, et met en couleur les éléments du premier tableau qui diffèrent avec ceux du deuxième tableau.

7.1.2 alloc_tab

Prototype de la fonction

int ** alloc_tab(int l,int c)

Paramètres

en entrée
- int l : Nombre de lignes
- int c : Nombre de colonnes

en sortie
- int ** : Tableau d'entiers à deux dimensions

Description

Cette fonction alloue une matrice bidimensionnelle d'entiers.

7.1.3 egale

Prototype de la fonction

int egale(int *a,int *b,int taille)

Paramètres

en entrée
- int *a : Tableau d'entiers
- int *b : Tableau d'entiers
- int taille : Taille des deux tableaux

en sortie
- int : Entier indiquant si les deux tableaux sont identiques, 0 les deux tableaux sont identiques, 1 si non

Description

Cette fonction compare deux tableaux d'entiers, et retourne un entier indiquant s'ils sont identiques.

7.1.4 permutation

Prototype de la fonction

void permutation(int **H,int n,int k,int *v,int rang)

Paramètres

en entrée
- int **H : Matrice de contrôle bidimensionnelle d'entiers
- int n : Taille d'un mot
- int k : Taille d'un bloc

- int *v : Tableau d'entiers à permuter dans la matrice de contrôle
- int rang : Numéro de la colonne où placer le tableau v dans la matrice de contrôle

en sortie
- void : Rien

Description

Cette fonction effectue une permutation entre deux colonne de la matrice de contrôle.

7.1.5 systematic_H

Prototype de la fonction

void systematic_H(int **H,int n,int k)

Paramètres

en entrée
- int **H : Matrice de contrôle bidimensionnelle d'entiers
- int n : Taille d'un mot
- int k : Taille d'un bloc

en sortie
- void : Rien

Description

Cette fonction transforme la matrice de contrôle pour la mettre sous systématique, en réalisant une série de permutations de colonnes.

7.1.6 dec_to_bin

Prototype de la fonction

void dec_to_bin(int *H,int dec,int taille)

Paramètres

en entrée
- int *H : Une colonne de la matrice de contrôle
- int dec : Un nombre en base dix
- int taille : Taille de la colonne H

en sortie
- void : Rien

Description

Cette fonction transforme un entier de base dix en base deux et le range dans une colonne de la matrice de contrôle passée en paramètre.

7.1.7 remplir_H

Prototype de la fonction

void remplir_H(int **H,int n,int k)

Paramètres

en entrée
- int **H : Matrice de contrôle bidimensionnelle d'entiers
- int n : Taille d'un mot
- int k : Taille d'un bloc

en sortie
- void : Rien

Description

Cette fonction permet de construire la matrice de contrôle. La matrice ainsi obtenue n'est pas sous forme systématique.

7.1.8 remplir_G

Prototype de la fonction

void remplir_G(int **G,int **H,int n,int k)

Paramètres

en entrée
- int **G : Matrice génératrice bidimensionnelle d'entiers
- int **H : Matrice de contrôle bidimensionnelle d'entiers
- int n : Taille d'un mot
- int k : Taille d'un bloc

en sortie
- void : Rien

Description

Cette fonction permet de construire la matrice génératrice à partir de la matrice de contrôle, mise sous forme systématique. La matrice génératrice obtenue est aussi sous forme systématique.

7.1.9 codage

Prototype de la fonction

int * codage(int **G,int *m,int l,int c)

Paramètres

en entrée
- int **G : Matrice génératrice bidimensionnelle d'entiers
- int *m : Tableau d'entiers représentant un bloc du message
- int l : Nombre de lignes
- int c : Nombre de colonnes

en sortie
- int * : Tableau d'entiers qui est le bloc m encodé (un mot)

Description

Cette fonction permet d'encoder un bloc du message grâce aux codes de Hamming.

7.1.10 decodage

Prototype de la fonction

int * decodage(int **H,int *m,int n,int k,FILE *f)

Paramètres

en entrée
- int **H : Matrice de contrôle bidimensionnelle d'entiers
- int *m : Tableau d'entiers représentant un mot
- int n : Taille d'un mot
- int k : Taille d'un bloc
- FILE *f : Fichier latex de destination

en sortie
- int * : Tableau d'entiers représentant le mot décodé

Description

Cette fonction permet de décoder, grâce aux codes de Hamming, un mot transmit.

7.1.11 genere_erreurs

Prototype de la fonction

int * genere_erreurs(int *m,int taille,int nb_max)

Paramètres

en entrée
- int *m : Tableau d'entiers représentant un mot
- int taille : Taille du mot m
- int nb_max : Nombre maximum d'erreurs à générer sur le mot m

en sortie
- int * : Tableau d'entiers représentant le mot m contenant des erreurs

Description

Cette fonction simule le bruit généré par le canal de transmission, en générant des erreurs dans le mot m passé en paramètre.

7.1.12 genere_message

Prototype de la fonction

void genere_message(int *m,int taille)

Paramètres

en entrée
- int *m : Tableau d'entiers d'un bloc
- int taille : Taille du tableau m

en sortie
- void : Rien

Description

Cette fonction génère un bloc binaire et le range dans le tableau m.

7.1.13 affiche

Prototype de la fonction

void affiche(int **tab,int l,int c)

Paramètres

en entrée
- int **tab : Tableau bidimensionnelle d'entiers à afficher
- int l : Nombre de lignes de tab
- int c : Nombre de colonnes de tab

en sortie
- void : Rien

Description

Cette fonction permet d'afficher une matrice d'entiers bidimensionnelle.

7.1.14 decoupe_message

Prototype de la fonction

int * decoupe_message(char *message,int k,int *nb_m)

Paramètres

en entrée
- char *message : La chaîne de caractère du message
- int k : Taille d'un bloc du message
- int *nb_m : Nombre de mots

en sortie
- int * : Tableau d'entiers représentant le message en binaire

Description

Cette fonction transforme le message en un tableau binaire, et calcul le nombre de mot nécessaire pour encoder le message.

7.1.15 reconstruit_message

Prototype de la fonction

char * reconstruit_message(int **m_decode,int k,int nb_m,FILE *f)

Paramètres

en entrée
- int **m_decode : Matrice bidimensionnelle contenant tout les mots du message reçu
- int k : Taille d'un bloc
- int nb_m : Nombre de mots
- FILE *f : Fichier latex de destination

en sortie
- char * : Chaîne de caractère correspondant au message reçu.

Description

Cette fonction reconstruit le message original à partir des bloc reçus.

7.1.16 Hamming

Prototype de la fonction

void Hamming(char *message,int nb_parite,int nb_erreur,FILE *f)

Paramètres

en entrée
- char *message : Chaîne de caractère correspondant au message à envoyer
- int nb_parite : Le nombre de bits de parité pour chaque mot envoyé
- int nb_erreur : Le nombre d'erreur à générer pour simuler le canal
- FILE *f : Fichier latex de destination

en sortie
- void : Rien

Description

Cette fonction implémente les codes de Hamming, c'est dire construit les matrices nécessaire à l'encodage et au décodage, encode le message à transmettre, simule le canal de transmission, et décode le message reçu.

7.1.17 entete

Prototype de la fonction

void entete(FILE *f,int nb_parite)

Paramètres

en entrée
- FILE *f : Fichier latex de destination
- int nb_parite : Le nombre de bits de parité pour chaque mot envoyé

en sortie
- void : Rien

Description

Cette fonction écrit l'entête du document latex dans le fichier de destination.

7.1.18 fin

Prototype de la fonction

void fin(FILE *f)

Paramètres

en entrée
- FILE *f : Fichier latex de destination

en sortie
- void : Rien

Description

Cette fonction écrit la fin du document latex dans le fichier de destination.

7.2 Mode d'emploi de l'application

Notre programme simule le passage d'une chaîne de caractères au travers d'un canal bruité. Pour sécuriser la transmission, nous utilisons cette fois les codes de Hamming. Le programme génère un fichier latex.

7.2.1 Paramètres

Usage

dhamming 1 2 3 4

Détail des arguments

1. nom du fichier contenant le message à transmettre.
2. nombre de bits de parité par mot.
3. nombre d'erreurs maximum par mot.
4. nom du fichier résultat.

Exemple d'appel du programme

dhamming message.txt 3 1 resultat.tex

Il ne reste plus qu'a compiler le fichier généré (dans notre exemple resultat.tex) comme un fichier latex ordinaire. Attention, pour visionner le résultat, il faut s'assurer que le visionneur permet d'utiliser les couleurs.

7.3 Choix de programmation

7.3.1 Interface graphique

Nous avons eu du mal pour trouver une méthode acceptable afin d'afficher toutes les informations. En effet, les codes correcteurs génères beaucoup d'information :

- le message original
- le message original convertit en suite de nombres
- le message codé grâce au code correcteur
- les erreurs générée lors du passage du canal
- les erreurs détectées par le code correcteur
- les erreurs corrigées par le code correcteur
- les erreurs persistantes après la correction du code correcteur
- le message reçu décodé
- le message reçu (en chaîne de caractères)

Nous avons essayé d'utiliser une interface graphique conventionnelle (avec des fenêtres) mais le résultat était tout simplement illisible. Nous avons donc choisi de générer un fichier latex. Ce fichier contient toutes les étapes de la transition d'un message.

7.3.2 Simulation du passage via un canal bruité

Pour simuler le passage d'un message au travers d'un canal bruité, nous avons créé une fonction qui génère au maximum t erreurs par bloc. Ces erreurs sont générées de manière aléatoire. Les erreurs sont donc réparties de façon homogène sur le message. Nous ne fabriquons pas de paquets d'erreurs.

7.3.3 Structure de données

Les codes de Hamming n'utilisent que des matrices bidimensionnelles ou des vecteurs, ainsi nous avons utilisé des structures de tableaux d'entiers. Ce choix peut paraître inapproprié au premier abord, puisque l'on stocke un bit à l'aide d'un entier en mémoire. Mais malheureusement, il n'existe pas de structure de données permettant de manipuler des bits dans le langage C. Il est vrai que nous aurions pu utiliser des entiers courts (*short* en C) pour économiser un peu de place en mémoire, mais pour des raisons de commodité nous avons préféré utiliser les entiers classiques (*int* en C).

Chapitre 8

Présentation de quelques algorithmes de la bibliothèque GMP

Voici différents algorithmes de multiplication. Chaque algorithme possède des avantages et des inconvénients. Lorsque l'on effectue un appel à la fonction de multiplication

mpz_mul(mpz_t res,mpz_t nb1,mpz_t nb2)

la fonction appelle l'algorithme le plus approprié, en fonction de la taille des opérandes.

8.1 Multiplication de base

Il s'agit de la multiplication que l'on effectue à la main. Cet algorithme est en $O(NM)$, avec N la taille de la première opérande et M la taille de la deuxième opérande.

8.2 Multiplication de Karatsuba

Cet algorithme travaille uniquement avec deux opérandes de même taille. Sa complexité est en $O(N^{1.585})$, avec N la taille des deux opérandes.

8.3 Méthode Toom-3

Cet algorithme travaille uniquement avec deux opérandes de même taille. Sa complexité est en $O(N^{1.465})$, avec N la taille des deux opérandes. Cet algorithme est donc plus performant que celui de Karatsuba. Cependant, cet algorithme effectue plus de travail sur les petits nombres, ce qui, dans ce cas là, le rend moins performant que Karatsuba.

8.4 Méthode FFT

Cet algorithme ne devient performant qu'en manipulant des entiers de plus de 300 chiffres. Sa complexité diminue lorsque la taille des nombres augmente. Sa complexité est en $O(N^{\frac{k}{k-1}})$, avec N la

taille de l'opérande la plus grande. k est un paramètre qui varie en fonction de la taille des opérande.