Institut Universitaire de Technologie
Aix-Marseille Université
Département Informatique, Aix

M3202C Modélisation

Morgan Morancey

413 Avenue Gaston Berger 13625

Aix-en-Provence

Sylvie.seropian@univ-amu.fr

04 42 93 90 00

Iut.univ-amu.fr

Projet Modélisation – Le RSA

Année universitaire 2020-2021

Année 2 Groupe 1

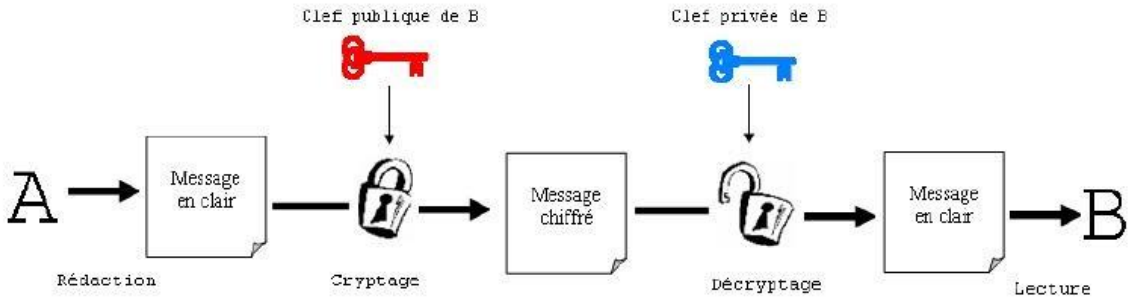
16 Novembre 2020

Guillaume RISCH – MOUMDJIAN THEO – LOURGUI Oussama – FERTOUT Haïtam

Sommaire

Le Chiffrement RSA

Le chiffrement RSA a été inventé par 3 personnes qui ont donné le sigle : Ronald Rivest (Cryptologue américain), Adi Shamir (Expert en cryptanalyse israélien) et Leonard Adlema (Chercheur en informatique et biologie moléculaire américain). Il a été écrit en 1977 et breveté par le Massachusetts Institute of Technology en 1983, le brevet a expiré le 20 septembre 2000.
Le chiffrement RSA est asymétrique : il utilise une paire de clés (des nombres entiers) composée d'une clé publique pour chiffrer et d'une clé privée pour déchiffrer des données confidentielles. Une condition indispensable est qu'il soit « calculatoirement impossible » de déchiffrer à l'aide de la seule clé publique, en particulier de reconstituer la clé privée à partir de la clé publique, c'est-à-dire que les moyens de calcul disponibles et les méthodes connues au moment de l'échange (et le temps que le secret doit être conservé) ne le permettent pas.
Le chiffrement RSA est souvent utilisé dans dans les navigateurs Internet comme Firefox et Internet Explorer.



Cryptographie à clef publique : le message est chiffré avec la clef publique du destintaire et décrypté avec sa clef privée.

Le principe général est qu'un utilisateur A souhaite envoyer un message M à un destinataire B. Ce destinataire B possède deux clés que l'on nomme clé publique et clé privée. L'utilisateur A s'il souhaite communiquer avec B doit chiffrer son message avec la clé publique de B, donc accessible par tous, ce message va transiter jusqu'à arriver à B.
Une fois le message reçu par B, B va se servir de sa clé privée connue uniquement de lui seul pour déchiffrer le message envoyé par A.

Si un intrus souhaiterait intercepter le message, il doit alors se servir de la clé publique de b et du message chiffré. A partir de la clé publique il doit récupérer un nombre nommé ϕ(n), le problème étant qu'il ne connait pas ce nombre. Il doit alors factoriser n qui est le produit de deux nombres premiers. Si il arrive à factoriser n assez vite pour que le message soit toujours utilisable, il pourra alors refabriquer la clé privée composé d'un nombre secret d, qui se calcule à partir d'un nombre e faisant partie de la clé publique et de ce nombre ϕ(n).
d permet à B de déchiffrer le message chiffré avec sa clé publique, l'intrus n'aurait donc qu'à déchiffrer le message grâce à ce nombre d et ainsi lire le contenu du message.

C'est donc sur la factorisation d'un très grand nombre n que repose la sécurité du RSA, en effet ce nombre n peut être codé sur un très grand nombre de bits et alors le factoriser peut-être très long.

Les calculs de chiffrage et de déchiffrage des informations à envoyer se font en grande partie grâce au modulo.
Le chiffrement RSA est basé en grande partie sur plusieurs théorèmes, le théorème d'Euclide qui sert à calculer le Plus Grand Commun Diviseur entre deux nombres premiers, le théorème de Bézout et le petit théorème de Fermat.
Le théorème de Bézout permet de calculer deux coefficients (nombres entiers) qui serviront à la création de la clé privée, tout en se servant de l'algorithme d'Euclide, la combinaison de ces deux théorèmes est nommée l'algorithme d'Euclide étendu qui une fois le pgcd calculé, remonte (en partant de la dernière équation où le reste de la division euclidienne est égale à 1) les étapes effectuées lors du calcul du pgcd.

Une fois que les clés publiques et privées sont formées, il ne reste plus qu'à chiffrer et déchiffrer le message. Le chiffrement RSA étant basé sur le petit théorème de Fermat, il assure le fait qu'un message M avant chiffrement est bien identique après le déchiffrement.

Logigramme RSA

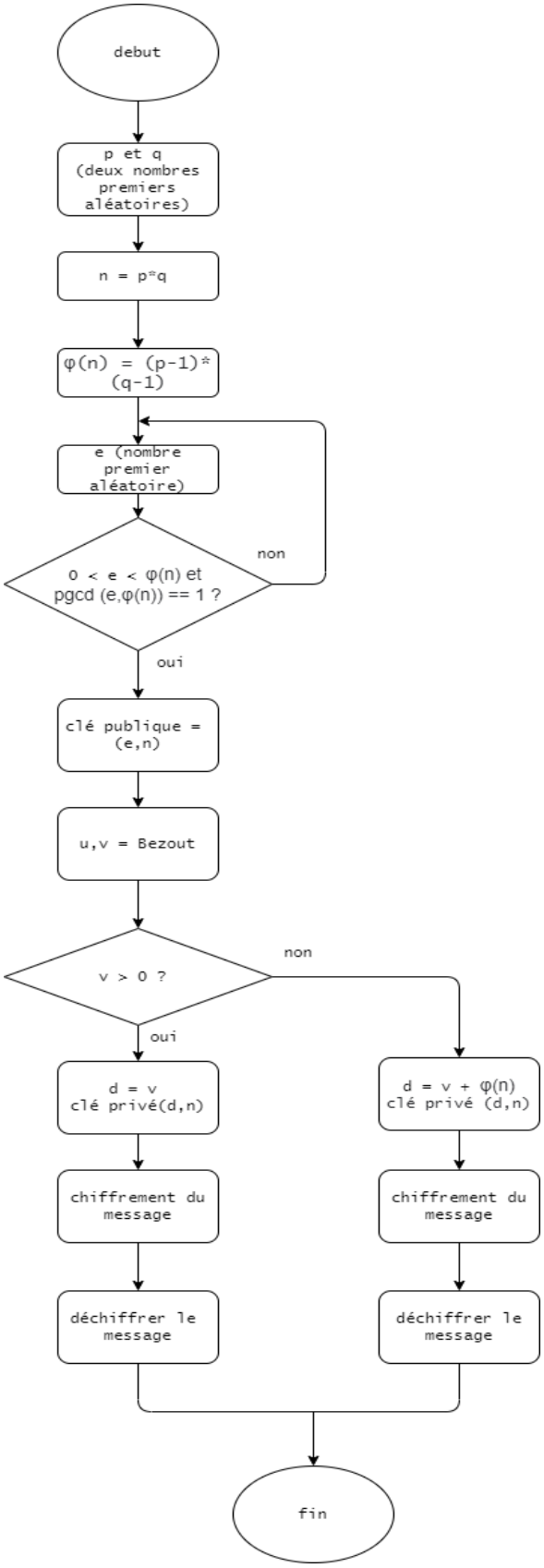


Figure 1 - Logigramme du RSA

Algorithme d'Euclide

L'algorithme d'Euclide nous sert pour calculer le PGCD (le Plus Grand Diviseur Commun), mais il nous permet aussi de calculer les coefficients de Bézout.
Pour calculer le PGCD de A et B, on utilise la division Euclidienne de A par B, tel que :
A = B * Quotient + reste

Exemple PGCD :

24 et 36

Les diviseurs de 24 sont : 1, 2, 3, 4, 6, 8, 12 et 24

Les diviseurs de 36 sont : 1, 2, 3, 4, 6, 9, 12, 18 et 36

24 et 36 ont pour diviseurs communs : 1, 2, 3, 4, 6 et 12

Le plus grand d'entre eux est 12

12 est donc le pgcd de 24 et de 36.

Dans le cas qui nous intéresse le chiffrement RSA fonctionne avec des nombres premiers ici on cherche deux nombres tels que leur PGCD est égal à 1, exemple avec 43 et 5 :

1. 43 = 8 * 5 + 3
2. 5 = 3*1 + 2
3. 3 = 2*1 + 1

Théorème de Bézout

Le théorème de Bézout est un résultat d'arithmétique élémentaire, il est nommé théorème de Bézout-Bachet ou encore identité de Bézout. Ce théorème a permis de prouver l'existence de deux solutions à une équation diophantienne linéaire de type : a*x + b*y = pgcd(a,b).

Le théorème affirme que deux nombres sont premiers entre eux si et seulement si a*x + b*y = 1 admet des solutions. Ce théorème est donc très utile dans le chiffrement RSA, car il utilise des nombres premiers pour rendre plus difficile la factorisation des clés privées.
On appelle alors les coefficients u et v, coefficients de Bézout, les solutions x et y de l'équation a*x + b*y = 1.

Exemple avec 43 et 5 où a = 43 et b = 5 :

1. 43 = 8 * 5 + 3
2. 5 = 3*1 + 2
3. 3 = 2*1 + 1

Calcul du PGCD précédemment réalisé

Application du théorème de Bézout :

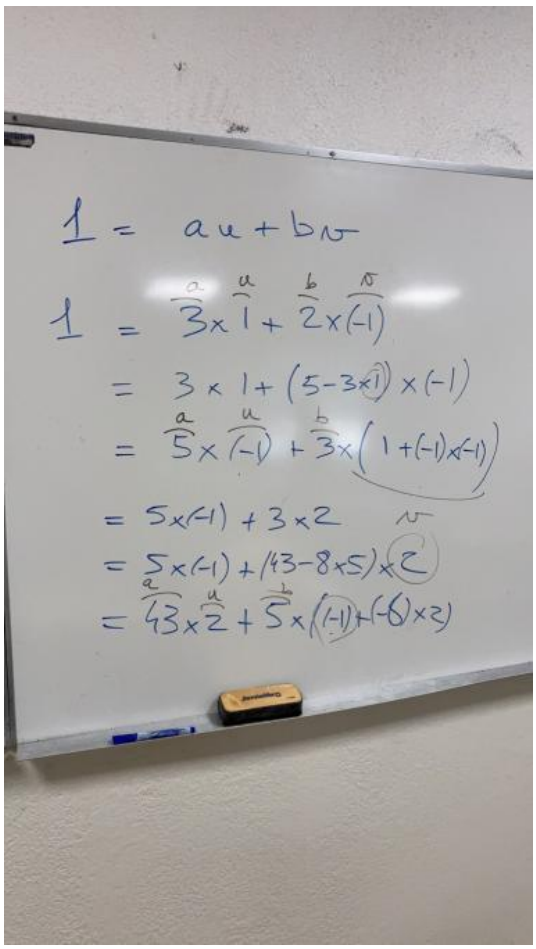
4. 1 = 3*1 + 2*(-1)
5. 1 = 5* (-1) + 3 * 2
6. 1 = 43*2 + 5 * (-17)

u ; **v** ; **a** ; **b**

A chaque étape les valeurs u, v, a, b changent :

1. a prend chaque valeur prises par les diviseurs durant la division euclidienne ici a prend 3 puis 5 puis 43
2. u prend la valeur de v à l'étape précédente ici u prend -1 puis 2
3. b prend les valeurs de a ici 3 puis 5
4. v prend la valeur : u + v*(-quotient) où quotient sont les quotients de chaque division euclidienne lors du calcul du pgcd (ici 1,8)

Pour arriver à cette récurrence à chaque itération nous avons identifié ce que devenait chaque valeur à chaque étape (cf photo ci-dessous)



Petit théorème de Fermat

Le petit théorème de Fermat est fondamental au fonctionnement du chiffrement RSA, c'est sur lui que repose le chiffrement et le déchiffrement il assure qu'en envoyant un message M en le chiffrant puis déchiffrant on obtient effectivement ce message M.

Dans le cas du RSA ce n'est pas directement le petit théorème de Fermat mais sa généralisation nommée Euler-Fermat.

Le chiffrement et le déchiffrement sont réalisés de cette manière :

1. $c = M^e$ modulo n // où c, est le message chiffré, M à la puissance e modulo n, e et n étant les deux nombres composant la clé publique accessible à tous. Rappel : n est la multiplication de deux nombres premiers et e est un nombre premier aléatoirement choisi de tel manière que le pgcd de e et $\varphi(n) = 1$

2. $m = c^d$ modulo n // où m, est le message originel M déchiffré, c étant le message chiffré et d une partie de la clé privée supposée connue que du destinataire. Ce d est calculé grâce aux coefficients de Bézout u et v où arbitrairement nous avons choisi v pour devenir d (cf logigramme).

```
Saisir un mot : MANGE
Le mot chiffré est :  6383210955816150333 le mot déchiffré est :  MANGE
```

Principales fonctions du codage du RSA

genereAlphabet

```
def genereAlphabet( ):
    alphabet = [ #table ASCII
        [' ',40], ['a',97],  ['0',60],
        ['A',65], ['b',98],  ['1',61],
        ['B',66], ['c',99],  ['2',62],
        ['C',67], ['d',100], ['3',63],
        ['D',68], ['e',101], ['4',64],
        ['E',69], ['f',102], ['5',65],
        ['F',70], ['g',103], ['6',66],
        ['G',71], ['h',104], ['7',67],
        ['H',72], ['i',105], ['8',68],
        ['I',73], ['j',106], ['9',69],
        ['J',74], ['k',107],
        ['K',75], ['l',108],
        ['L',76], ['m',109],
        ['M',77], ['n',110],
        ['N',78], ['o',111],
        ['O',79], ['p',112],
        ['P',80], ['q',113],
        ['Q',81], ['r',114],
        ['R',82], ['s',115],
        ['S',83], ['t',116],
        ['T',84], ['u',117],
        ['U',85], ['v',118],
        ['V',86], ['w',119],
        ['W',87], ['x',120],
        ['X',88], ['y',121],
        ['Y',89], ['z',122],
        ['Z',90],
        ['.',56], ['!',41],['\'',47],[',',54]]
    return alphabet
```

Le but de cette fonction est de créer une base, pour pouvoir chiffrer et déchiffrer un message. Concrètement la fonction créée une matrice nommée alphabet, dans cette matrice chaque sous-tableau est composé de deux éléments, le premier est un est caractère et le deuxième un nombre, les nombres ne sont pas choisis aléatoirement ils sont ceux attribués à chaque caractère dans la table ASCII.

Initialisation

```
def initialisation(tabNombrePremier,tabReste,tabDiviseur,tabDividende,tabQuotient):
    p : int = random.choice(tabNombrePremier)
    q : int = random.choice(tabNombrePremier)
    if p == q:
        q = random.choice(tabNombrePremier)
    n : int = p*q
    phi : int = (p-1)*(q-1)
    e : int = random.choice(tabNombrePremier)
    while e > phi:
        e = random.choice(tabNombrePremier)
    pgcd(e,phi,tabReste,tabDiviseur,tabDividende,tabQuotient)

    return (n,phi,e,p,q)
```

Cette étape du codage du RSA est la première, elle permet de choisir aléatoirement deux nombres premiers p et q parmi un tableau des nombres premiers généré automatiquement en fonction du nombre maximum passé en paramètre. p et q permettent par la suite de créer n, n = p*q, et
ϕ(n) = (p-1)*(q-1). e nombre premier aléatoirement choisi doit être strictement inférieur à ϕ(n) son pgcd doit être aussi égal à 1 avec ϕ(n).

PGCD

```
def pgcd(a,b,tabReste,tabDiviseur,tabDividende,tabQuotient):
    if a !=0 and b != 0:
        if b > a :
            return pgcd(b,a,tabReste,tabDiviseur,tabDividende,tabQuotient)
        if a > b :

            reste : int = a%b
            tabReste.append(int(a%b))
            tabDiviseur.append(int(b))
            tabDividende.append(int(a))
            tabQuotient.append(int(a//b))
            if tabReste[len(tabReste)-1] == 1:

                return (tabReste,tabDiviseur,tabDividende,tabQuotient, tabReste[len(tabReste)-1])
            else:
                return pgcd(b,reste,tabReste,tabDiviseur,tabDividende,tabQuotient)
```

Comme dit précédemment e et ϕ(n) doivent être premiers entre eux c'est pour cela qu'on crée une fonction pgcd qui calcule le pgcd de ces deux nombres et qui vérifie que le reste à la fin des divisions euclidiennes est bien égal à 1.
La stratégie que nous avons utilisé dans le codage du RSA est de stocker chaque valeur utilisée dans le calcul du pgcd dans des tableaux où chacun correspond à une partie d'une division (reste,dividende,diviseur,quotient). Ces tableaux vont servir dans le calcul des coefficients de Bezout.

Bezout

```
def Bezout(a,b,u,v,i,tabReste,tabDiviseur,tabDividende,tabQuotient):
    if tabDividende[0] * u + tabDiviseur[0] * v == 1:
        print("1 = ", a,"*",u,"+", b,"*",v)
        return (u,v)
    elif tabDividende[0] * u  + tabDiviseur[0] * v == 1 and i == 0:
        return (1,-1)
    else:
        print("1 = ", a,"*",u,"+", b,"*",v)
        return Bezout(tabDividende[i-1],a,v,u+v*(-tabQuotient[i-1]),i-1,tabReste,tabDiviseur,tabDividende,tabQuotient)
```

La fonction Bézout permet de calculer les coefficients de Bézout de manière récurrente en utilisant les tableaux remplis par la fonction pgcd. Les itérations à chaque étape de la récurrence sont décrites dans la partie dédiée au théorème de Bézout. Pour parcourir nos tableaux nous initialisons une variable i qui prend comme valeur la taille du tableau Dividende (choisi arbitrairement car les tableaux ont tous la même taille), on parcourt les tableaux, stockant les valeurs qui nous intéressent, en sens inverse pour pouvoir remonter le calcul du pgcd et ainsi une fois que i vaut 0 cela veut dire que le programme est arrivé à la dernière étape et alors on renvoie les coefficients u et v.

IsKeyNegative

```
def isKeyNegative(v,phi):
    d : int
    if v < 0:
        d = v + phi
    else:
        d = v
    return int(d)
```

Si le coefficient de Bézout utilisé pour la création de la clé privée est inférieur à 0 le chiffrement ne peut pas fonctionner. De ce fait nous avons réglé ce problème en s'inspirant de cette page.

crackage

```
def crackage (n,tabNombrePremier):
    indice : int = findLastIndexNbrPrime(n,tabNombrePremier)

    for i in range(indice, -1, -1):
        if n%tabNombrePremier[i] == 0:
            return tabNombrePremier[i]
```

La fonction crackage permet de trouver soit p soit q, la méthode pour trouver un des deux facteurs a été inspirée par cette page, l'idée de cette méthode est de trouver la position du premier nombre premier sous la racine carrée de n de telle sorte que n%tabNombrePremier[i] = 0, c'est-à-dire que si n/tabNombrePremier[i] est un nombre entier alors c'est un des deux facteurs qui a servi à la création de n. Le deuxième se trouve donc aisément en faisant :
n/premierfacteur = deuxiemefacteur.

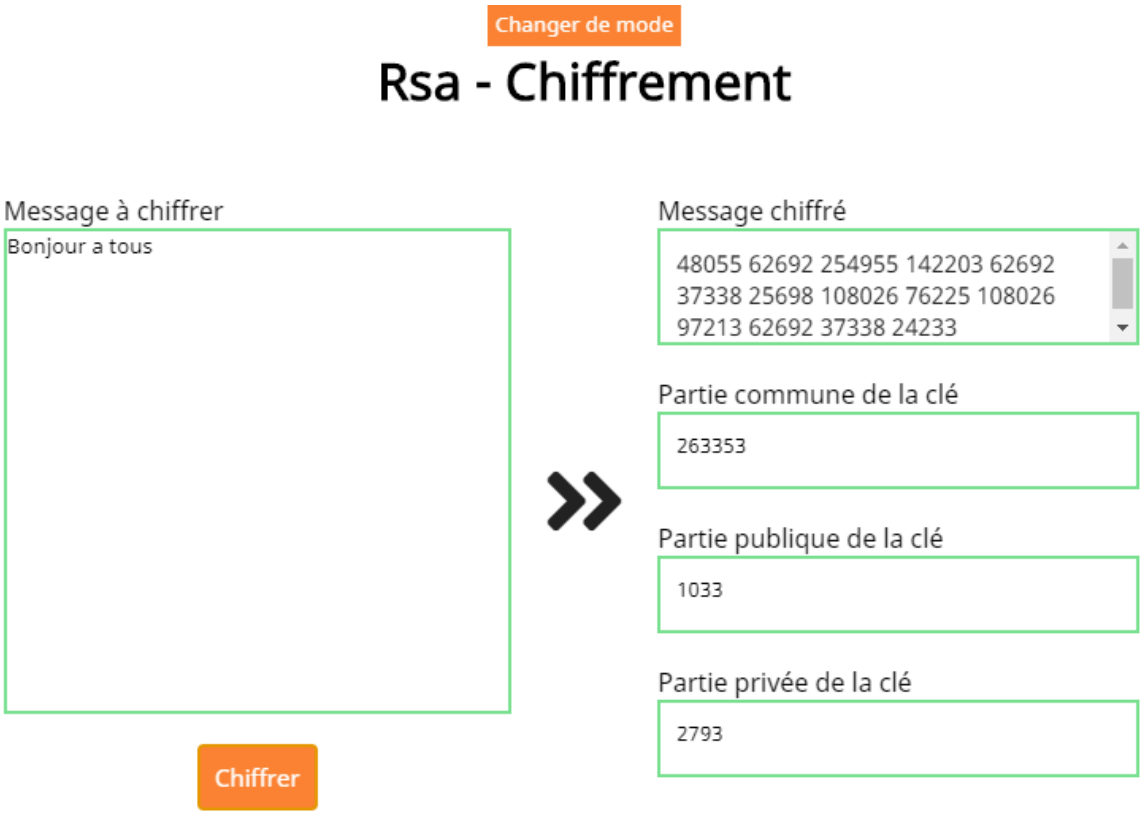
Interface Homme-Machine

Accueil


```
document.getElementById('lockButton').addEventListener('click', function() {
    this.classList.add('big');

    setTimeout(function() {
        document.getElementById('popupStart').classList.add('noDisplay')
        document.getElementById('formContainer').classList.remove('noDisplay')
      }, 1100);
})
```

Ecran d'accueil avec une animation de fermeture du cadenas après avoir cliqué dessus

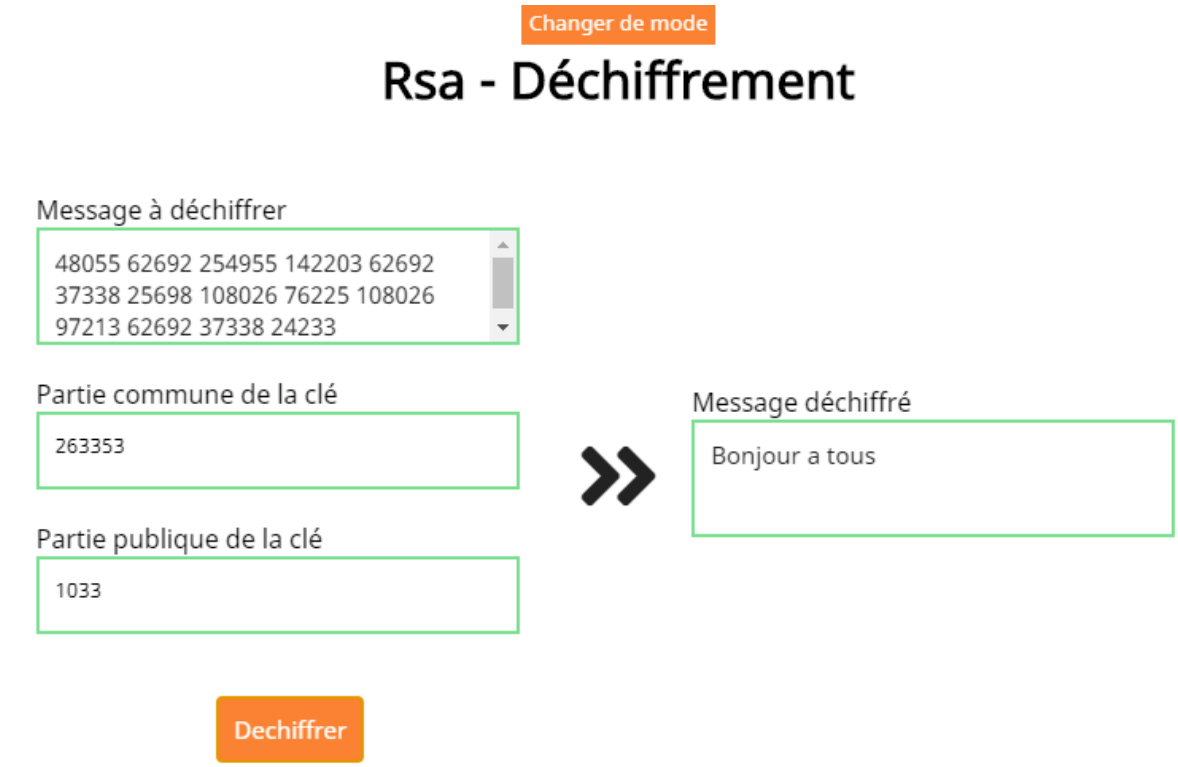
Chiffrement




```
document.getElementById('Chiffrement').addEventListener('click', function(){
    window.api.send("toCrypt", document.getElementById('source').value);
    window.api.receive("fromCrypt", (data) => {
        var clecommune = data[1].toString();
        document.getElementById('clecommuneC').value = clecommune;

        var clepublique = data[2].toString()
        document.getElementById('clepubliqueC').value = clepublique

        var cleprivee = data[3].toString()
        document.getElementById('clepriveeC').value = cleprivee

        var str = data[0];
        var result = ''
        for(let i = 0; i < str.length; ++i) {
            result += str[i]+' '
        }
        document.getElementById('receptor').value = result;
    });
});
```

Ici on récupère les valeurs du programmes python pour la partie chiffrement pour les afficher.

Déchiffrement




```
document.getElementById('Dechiffrement').addEventListener('click', function() {
    window.api.send("toDecrypt", [document.getElementById('sourceDecrypt').value, document.getElementById('clecommuneD').value
        , document.getElementById('clepubliqueD').value]);
    window.api.receive("fromDecrypt", (data) => {
        document.getElementById('receptorDecrypt').value = data[0];
    });
```

Ici on récupère la partie commune de la clé (n), la partie publique (e) et le message chiffrer pour que le programme python puisse les déchiffrer.

Sources

Algorithme d'Euclide :

https://fr.wikipedia.org/wiki/Algorithme_d%27Euclide#:~:text=En%20mathématiques%2C%20l%27algorithme%20d,factorisation%20de%20ces%20deux%20nombres.

Théorème de Bachet-Bézout :

https://fr.wikipedia.org/wiki/Théorème_de_Bachet-Bézout#:~:text=Le%20théorème%20de%20Bézout%20affirme,by%20%3D%201%20admet%20des%20solutions.

Petit théorème de Fermat :

https://fr.wikipedia.org/wiki/Petit_théorème_de_Fermat

Gérer l'exception où la clé privée est inférieure à 0 :

https://crypto.stackexchange.com/questions/10805/how-does-one-deal-with-a-negative-d-in-rsa

Trouver les facteurs p et q ayant servi à créer la clé publique :

https://stackoverflow.com/questions/4078902/cracking-short-rsa-keys

Comment Marche Electron

https://www.electronjs.org/

Logigramme : Nous