Université de Technologie
Belfort-Montbéliard

En vue de l'obtention de l'unité de valeur

AC20

Cryptographie RSA

Auteur :
Aurore CHAPPUIS

Enseignant encadrant :
Frédéric HOLWECK

17 Juin 2020 - P2020

Remerciements

Je souhaite remercier M. Holweck pour m'avoir supervisée et pour l'aide qu'il m'a apporté tout au long de cette étude et de ce semestre.

Je tiens également à remercier M. Tachikart de m'avoir autorisée et encouragée à suivre cette UV de recherche puis à traiter ce sujet ainsi que le juré qui va consacrer de son temps en tant qu'évaluateur.

Avant-propos

Pourquoi avoir choisi la cryptographie RSA ?

Mon projet étant d'intégrer la branche informatique, ce sujet fait partie des domaines ayant des corrélations étroites avec l'arithmétique et l'informatique ce qui en fait un sujet propice à la recherche scientifique.

Cette UV me permettra de débuter et d'approfondir mes connaissances en cryptographie, thème pour lequel j'éprouve de la curiosité suite à toutes les avancées scientifiques connues (des chiffrements anciens à Enigma, cybersécurité...). Il est un bon sujet pour s'initier à des notions d'informatique théorique telle que l'étude de la complexité algorithmique. Il me donne également la possibilité de pratiquer la programmation de fonctions utiles pour le domaine d'étude.

J'espère que la lecture de ce rapport vous plaira.

Sommaire

Introduction

La science du secret

La cryptologie regroupe deux disciplines qui sont vouées à servir à la sécurité de nos données : la cryptographie, le codage secret d'une information et la cryptanalyse, le décodage d'une information codée. Cette science remonte à l'Antiquité puisque les plus anciennes formes de codage comme celui du célèbre Jules César qui était utilisée par les Romains. En conséquence, cela montre que nous avons toujours eu la volonté de communiquer de manière confidentielle et de garder nos informations en sûreté. Mais cette nécessité perdure et augmente. La transmission des informations confidentielles de manière sécurisée est devenue un besoin capital pour maints secteurs (militaire, politique, bancaire, particulier...). D'autant plus que les multiples systèmes de communication et les progrès technologiques depuis les années 1970 pour le grand public contribuent à la complexité de garder en lieu sûr les informations ou de les envoyer confidentiellement. Le problème est d'autant plus actuel depuis que nous sommes tous reliés par Internet. Depuis son apparition, la cryptologie est le centre de nombreuses études académiques qui tendent à se multiplier au fur et à mesure des découvertes et progrès dans les domaines scientifiques respectifs à ceux-ci.

La cryptographie vise justement à sécuriser des données en employant des méthodes de chiffrement permettant de cacher ces données en les transformant sous une autre forme telle qu'elle soit incompréhensible aux intercepteurs indésirables. Une protection qui est réalisable sous plusieurs méthodes.

Symétrique ou Asymétrique

Ce sont les grandes catégories de cryptographie à ce jour. Même si elles sont bien distinctes, elles reposent toutes deux sur une utilisation de clés qui servent à la mise en sécurité des données. En opposition avec le codage de César mentionnée précédemment, qui reposait en grande partie sur le secret de la méthode, Auguste Kerckhoffs[1] en 1883 a mis en avant un principe fondateur de la cryptographie moderne[8] :

> "Il faut que le système n'exige pas le secret, et qu'il puisse sans inconvénient tomber entre les mains de l'ennemi."

Cette approche est un peu déconcertante étant donné que l'on pense aisément que, plus les informations sur la méthode de codage/décodage sont confidentielles, mieux les données à protéger le seront. Or, Kerckhoffs avait derrière la tête que les mécanismes connus de tous seraient pleinement décortiqués, si bien même que les meilleurs d'entre eux, assurément robustes et fiables, seraient employés et ne reposeraient en fin de compte que sur cette clé gardée à l'abri.

Ainsi, lorsque nous employons une méthode symétrique, sa clé est dite secrète ou partagée. Il s'agit de la (seule) donnée qui permet de coder et de décoder un message. De ce fait, le fameux codage de Jules César dispose d'une clé secrète qui est constituée d'un nombre entier représentant le décalage constant de lettres que l'on effectue en codant le message. Si nous connaissons la clé secrète de décalage, soit parce que l'on est le receveur légitime du message soit parce que nous avons trouvé la clé secrète par un quelconque moyen, il est donc aisé de le décoder. Enigma, la machine employée par les Nazis pendant la Seconde Guerre Mondiale utilisait ce même procédé mais en extrêmement plus complexe. On peut cependant mentionner que Alan Turing, le mathématicien et cryptologue britannique, a réussi à décoder la machine grâce à une approche très novatrice des mathématiques. Néanmoins, plusieurs problèmes se posent quand au partage de cette clé secrète qui peut vite devenir la faille de ce système. En fait, il faut envoyer autant de clés que "combinaisons de droit" ce qui peut compliquer grandement leur gestion (il faut créer autant de canaux de partage que de transmetteurs), sans parler du fait qu'il est primordial de transmettre la clé de manière sûre puisqu'elle sert aussi à décoder, c'est-à-dire que l'assurance d'envoyer la clé au bon récepteur doit être à 100% positive.

1. Auguste Kerckhoffs von Nieuwenhoff (1835 - 1903) est un cryptologue militaire néerlandais.

Quand à la cryptologie asymétrique, elle répond à ces problèmes liés au partage des clés sensées rester secrètes y compris lors de leur transmission. A vrai dire, le système ne dépend pas d'une clé mais de deux : la clé publique à laquelle tout le monde à accès et la clé privée que seul le récepteur du message détient. La première permet de coder le message tandis que l'autre effectue l'opération inverse à ce codage. De ce fait, il n'est plus nécéssaire de transmettre la clé qui rend possible le décodage. Une bonne analogie sera d'imaginer un coffre fort d'abord ouvert (= la clé publique) que le receveur rend disponible à quiconque ayant le besoin de communiquer secrètement avec lui. Il le fait en prenant soin de ne pas perdre la clé du coffre (= la clé privée). La personne souhaitant lui faire parvenir un message l'insère dans le coffre et le referme comme il pourrait le faire en codant avec la clé publique. Ainsi, seul le receveur est capable de l'ouvrir avec sa clé qu'elle soit réelle pour le coffre fort ou privée pour le message codé. Issu du travail publié en 1976 par Whitfield Diffie et Martin Hellman [2], ce système asymétrique est défini comme robuste, ayant fait ses preuves jusqu'à maintenant même si, toutefois, son efficacité en terme de rapidité et de simplicité est au dessous de la cryptographie symétrique.

De surcroît, c'est la cryptographie asymétrique qui va constituer en partie notre sujet.

Naissance de la cryptographie RSA

En 1977, un rédacteur d'une revue scientifique appelée *Scientific American*, Martin Gardner, a mis au défi ses lecteurs de déchiffrer un nombre de 129 chiffres avec une bonne récompense à la clé. Un indice laissé avec ce défi indiquait qu'il fallait décomposer un nombre en deux nombre premiers. C'est seulement en 1994, soit 17 ans plus tard, que la réponse exacte fut trouvée grâce au travail de plus de 600 personnes. C'est à l'occasion de ce défi que fut présenté à la communauté scientifique et au monde entier le système RSA, qui vient de l'acronyme des noms de leur créateurs du MIT : Ronald Rivest, Adi Shamir et Leonard Adleman. Ces derniers cherchaient à l'origine un moyen de prouver que les systèmes cryptographiques à clé publique possédaient une faille. En échouant, ils ont découvert cette méthode de codage qui les a, en fin de compte, rendus célèbres.

Aujourd'hui, leur travail est un pilier de la confidentialité des données dans de nombreuses applications, la plupart des transactions bancaires que l'on effectue seraient sécurisées par ce système par exemple. Pour comprendre

2. Whitfield Diffie et Martin Hellman sont deux cryptologues américains du XXème siècle.

combien il a impacté notre manière de vivre, ajoutons à cela le fait que plus de 300 millions de programmes installés l'emploient selon les estimations et qu'il permet de garder secret des codes de l'armée américaine.

Répondant à ce besoin si important de conserver des données qui doivent rester cachées, la cryptographie RSA a su faire ses preuves... mais pour combien de temps ?

Chapitre 1

Notions de base

Ce premier chapitre servira essentiellement à mettre en place toutes les bases utiles pour comprendre la suite.

I Vocabulaire en cryptographie

1 Chiffrer, Coder ou Crypter ?

Pour commencer, nous allons aborder quelques mots de vocabulaire afin d'être rigoureux sur les termes propres à la cryptographie.
Nous avons déjà parlé de la cryptologie et ses dérivées, mais quand est-il des termes propres à l'art de concevoir des algorithmes de chiffrement, analyser leurs forces et leurs faiblesses ?

Un **code** sert à transmettre des informations. Il s'agit d'un protocole qui n'est pas forcément secret comme le morse, un code qui met en relation des lettres de l'alphabet et des séquences de sons courts et longs. Coder un message consiste surtout à le transformer en une autre forme. -C'est donc avec le système de (dé)codage que repose le secret du message codé-.

Lorsque l'on a une succession de transformations pour chiffrer un message, on peut dire qu'il s'agit d'un **algorithme**.

Le **chiffrement**, quand à lui est, une procédure qui permet de rendre une donnée incompréhensible à toute personne qui ne possède pas une **clé de chiffrement**. Cette dernière laisse la possibilité de connaitre le message codé et il faut que les acteurs de cette transmission s'accordent sur cette clé qui peut-être, rappelons-le, publique, privée ou secrète.

Le **déchiffrement** est l'action de retrouver le message initialement chiffré avec cette clé. Il est important donc de souligner la nuance avec le **décryptage** qui permet aussi d'obtenir le message chiffré mais sans avoir la clé de déchiffrement. Il n'existe pas d'antonyme pour décrypter, sinon cela signifierait qu'il serait impossible de chiffrer une donnée sans clé.
Quand au **cryptage**, il s'agit d'un abus de language qui est pourtant fréquemment employé en cryptographie pour désigner le chiffrement d'un code. Il est donc préférable de parler de chiffrement car crypter n'est reconnu ni par Référentiel Général de Sécurité de l'ANSSI Référentiel Général ni par le dictionnaire de l'Académie française. Il en est de même pour "encrypter" qui est un anglicisme ou "chiffrage" qui n'a rien à voir avec la cryptographie.

2 Fonctions et clés

Il est important de bien choisir sa clé de chiffrement, pour que la sécurité d'un système soit assurée. Elle doit être comprise dans un très large champ de possibilités afin de rendre la tâche plus complexe à celui qui souhaite la trouver illégalement. On appelle **espace des clés** l'ensemble des éléments que contient $\mathbb{Z}/n\mathbb{Z}$, ce qui signifie qu'une clé est comprise entre 0 et n. Dans le cas du chiffrement de César où la clé est un décalage constant de lettres pour coder un message, la clé est comprise entre 0 et 26 donc dans $\mathbb{Z}/26\mathbb{Z}$ qui est l'ensemble de tous les éléments de $\mathbb{Z}$ modulo 26. Pour renforcer ce système, nous pouvons utiliser des clés composées d'une certaine longueur, que l'on nomme **longueur des clés**.

II Arithmétique pour le système RSA

Introduisons à présent les notions d'arithmétiques utiles pour comprendre le système RSA.

1 Congruences dans $\mathbb{Z}$

Nombres premiers et Modulo

Soit un entier $n \leq 2$.

Définition 1

On dit que a et b, des entiers relatifs non nuls, sont premiers entre eux s'ils n'ont pour diviseurs communs que 1 et -1.

Exemple : Montrons que la fraction $\frac{n}{2n+1}$ avec $n \in \mathbb{N}$ est irréductible. Il faut pour cela montrer que n et $2n+1$ sont premiers entre eux.
Soit $d \in \mathbb{Z}$ un diviseur commun de n et $2n+1$. On a alors $d|n$ et $d|2n+1$

$$\Rightarrow d|2n+1-2n \Rightarrow d|1$$

$$\Rightarrow d \in \{-1, 1\}$$

Conclusion : Comme leurs seuls diviseurs communs sont 1 et -1, n et $2n+1$ sont premiers entre eux et $\frac{n}{2n+1}$ est irréductible.

Définition 2

On dit que **a est congru à b modulo n**, si $n|b-a$, $(a;b) \in \mathbb{Z}^2$. On note alors :

$$a \equiv b \pmod{n}$$

Remarque : Cela signifie que $b = a + k \times n$ avec $k \in \mathbb{Z}$.

PGCD

Définition 3

On définit le **pgcd(a,b)** par le Plus Grand Diviseur Commun de a et b, $(a; b) \in \mathbb{Z}^2$.

Le *pgcd*[19] permet également de montrer que deux nombres sont premiers entre eux. Il est évident que si 1 est le seul plus grand diviseur commun, c'est également qu'il est le seul diviseur de ces deux nombres. D'où :

$$pgcd(a, b) = 1 \Longleftrightarrow a \text{ et } b \text{ sont premiers entre eux}$$

2 Indicatrice d'Euler

Définition 4

Soient n un entier naturel et $\phi(n)$ la **fonction indicatrice d'Euler** qui donne la quantité de nombres premiers avec n et qui sont inférieurs à n. Autrement dit,

$$\phi(n) = card\{1 \leq k \leq n; k \text{ est premier avec } n\}$$

Aussi, $\phi(1) = 1$

Exemple : $\phi(10) = 4$ car 10 est premier avec 1, 3, 7 et 9.

Nous pouvons remarquer que plus n est grand, plus $\phi(n)$ sera difficile à calculer puisque cela signifie qu'il faut compter chaque nombre premier avec n en allant de 1 à n pour obtenir le résultat... Ce qui est compliqué sauf pour des nombres premiers.

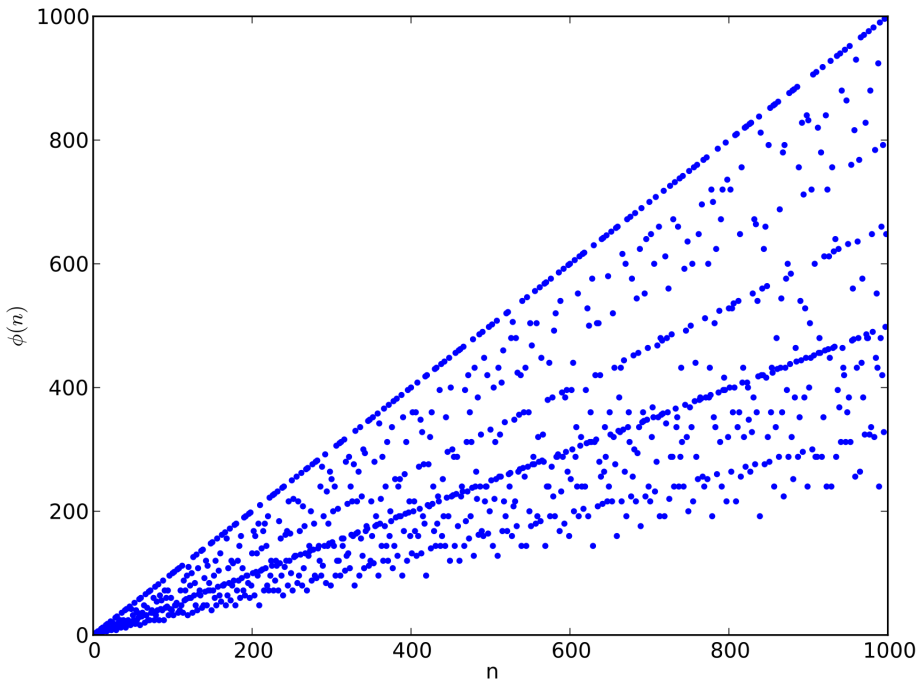

FIGURE 1.1 – Les mille premières valeurs de $\phi(n)$ par Pietro Battiston

Lorsque l'on regarde la représentation graphique de $\phi(n)$, on constate que la suite de points formant quasiment des droites sont des nombres premiers. En effet, pour tout p premier, tout $k \in [\![1; p]\!]$ est premier avec lui donc

$$\phi(p) = p - 1$$

Il s'agit d'une condition nécessaire et suffisante pour qu'un nombre soit premier.

3 Théorème de Bézout et théorème de Gauss

Théorème 5 (Identité de Bézout)

Soient a et b deux nombres entiers distincts non nuls et soit $pgdc(a, b) = c$. Alors, $\exists (u, v) \in \mathbb{Z}^2$ tels que :

$$au + bv = c$$

Démonstration. Soit $c = pgcd(a, b)$, l'existence de u et v est justifiée par la définition du *pgcd*. De plus, si $c|a$ et $c|b$ et vérifie $au + bv = c$, tout diviseur commun de a et b divise c. Ce qui signifie que $c = pgcd(a, b)$ d'après les caractéristiques du *pgcd*. □

Théorème 6 (Théorème de Bézout)

Deux nombres entiers distincts a et b sont premiers entre eux si et seulement si il existe $(u, v) \in \mathbb{Z}^2$ tels que $au+bv = 1$. Autrement dit :

$$pgcd(a, b) = 1 \Longleftrightarrow \exists (u, v) \in \mathbb{Z}^2 / au + bv = 1$$

Démonstration. $\Rightarrow$: Immédiat avec l'utilisation de l'identité de Bézout

$\Leftarrow$: Supposons qu'il existe bien deux entiers u et v tels que $au + bv = 1$. Comme $pgcd(a, b) = c$ alors $c|a$ et $c|b$ donc $c|au + bv$ (combinaison linéaire). Donc c divise 1 ce qui implique que $c = 1$. □

Note : Il est possible que certains ouvrages inversent la dénomination "d'identité" de Bezout avec son théorème qui sont liés entre eux.

Théorème 7 (Théorème de Gauss)

Soient a, b et c trois entiers relatifs non nuls.
Si **a|bc** et si **a et b sont premiers entre eux** alors **a|c**.

Démonstration. On a a|bc, alors $\exists k \in \mathbb{Z}$ tel que : $bc = ka$

Comme a et b sont premiers entre eux alors, d'après le théorème de Bézout, il existe deux entiers u et v tels que : $au + bv = 1$. Si on multiplie par c, on a :

$$\begin{aligned} acu + bcv &= c \text{ où } bc = ka \\ acu + (ka)v &= c \\ a(cu + kv) &= c \end{aligned}$$

D'où $a|c$.

□

4 Algorithme d'Euclide étendu

Commençons par présenter la méthode des soustractions successives :

Propriété 8

Si $(a, b) \in \mathbb{Z}^2$ avec $a > b$. Alors :

$$pgcd(a, b) = pgcd(a - b, b)$$

L'algorithme d'Euclide permet de raccourcir la méthode précédente :

Propriété 9 (Algorithme d'Euclide)

Si $(a, b) \in \mathbb{Z}^2$ non nuls avec $a > b$. Alors :

$$pgcd(a, b) = pgcd(b, a \pmod b)$$

Remarque : Il s'agit de faire la division euclidienne du plus grand nombre, a, par le second nombre, b, puis de faire la division euclidienne de b avec $a \pmod b$, c'est à dire, le reste. On continue ainsi jusqu'à trouver un reste nul. Le dernier reste non nul est le pgcd.

La méthode de calcul avec un exemple : $pgcd(234, 642)$

$$\begin{aligned}
\mathbf{642} &= \mathbf{243} \times 2 + \underline{156} \\
\mathbf{243} &= \underline{156} \times 1 + 87 \\
156 &= 87 \times 1 + 69 \\
87 &= 69 \times 1 + 18 \\
69 &= 18 \times 3 + 15 \\
18 &= 15 \times 1 + \underbrace{\mathbf{3}}_{\text{pgcd}} \\
15 &= 5 \times 3 + 0
\end{aligned}$$

On arrive au reste nul donc :

$$pgcd(234, 642) = 3$$

L'algorithme d'Euclide étendu, quant à lui, est une variante de cette algorithme. On remonte l'algorithme d'Euclide pour obtenir le couple de coefficients u et v du théorème de Bézout. La propriété suivante découle donc de ce théorème.

Propriété 10 (Algorithme d'Euclide étendu)

Si $(a, b) \in \mathbb{Z}^2$ non nuls. Alors :

$$\exists (u, v) \in \mathbb{Z}^2 \text{ tels que } pgcd(a, b) = au + bv$$

Reprenons notre exemple du $pgcd(234, 642) = 3$. Il faut remonter les calculs faits précèdement pour trouver ce $pgcd$. On part de $18 = 15 \times 1 + 3 \Leftrightarrow 3 = 18 - 15 \times 1$. On prendra soin de toujours avoir une égalité avec le reste 3 d'une part et on replace le plus petit reste par son équation équivalente de l'algorithme dans l'autre part. Ainsi :

$$\begin{aligned}
3 &= 18 - \mathbf{15} \times 1 \\
3 &= 18 - \underbrace{(69 - 18 \times 3)}_{=15} \times 1 \\
3 &= \mathbf{18} \times 4 - 69 \\
3 &= \underbrace{(87 - 69 \times 1)}_{=18} \times 4 - 69 \\
3 &= 87 \times 4 - \mathbf{69} \times 5 \\
3 &= 87 \times 4 - \underbrace{(156 - 87 \times 1)}_{=69} \times \\
3 &= \mathbf{87} \times 9 - 156 \times 5 \\
3 &= \underbrace{(243 - 156 \times 1)}_{=87} \times 9 - 156 \times 5 \\
3 &= 243 \times 9 - \mathbf{156} \times 14 \\
3 &= 243 \times 9 - \underbrace{(642 - 243 \times 2)}_{=156} \times 14 \\
3 &= 243 \times \underbrace{\mathbf{37}}_{u} - 642 \times \underbrace{\mathbf{14}}_{v}
\end{aligned}$$

Ainsi, $u = 37$ et $v = -14$.

5 Le petit théorème de Fermat (amélioré)

Il s'agit d'un théorème de Pierre de Fermat [1], essentiel pour comprendre comment fonctionne la cryptographie RSA.

Théorème 11 (Petit Théorème de Fermat)

Si p est un nombre premier et $a \in \mathbb{Z}$ alors :

$$a^p \equiv a \pmod{p}$$

Démonstration. Voyons dans un premier temps[13] que les coefficients binomiaux [2] $\binom{p}{k}$ pour un nombre premier p sont divisibles par p, hormis $\binom{p}{1} = 1$. Pour cela, montrons que $p|\binom{p}{k}$ avec $0 < k < p$. Vérifions que $p\binom{p-1}{k-1} = k\binom{p}{k}$:

$$\begin{aligned} p\binom{p-1}{k-1} &= \frac{p(p-1)!}{(k-1)!((p-1)-(k-1))!} \\ &= \frac{p!}{(k-1)!(p-k)!} \\ &= k\frac{p!}{k!(p-k)!} \text{ en multipliant par facteur } k \\ p\binom{p-1}{k-1} &= k\binom{p}{k} \end{aligned}$$

Comme p est premier et $k < p$ alors p et k sont premiers entre eux donc d'après le théorème de Gauss, on en déduit que $p|\binom{p}{k}$

Ensuite, rappelons que la multiplication est commutative pour les modulos. De plus, on vient de vérifier que $p|\binom{p}{k}$, ce qui implique $\binom{p}{k} \equiv 0 \pmod{p}$ pour $k < p$. Le binôme de Newton est alors applicable d'où la formule :

$$(a+b)^p \equiv a^p + b^p \pmod{p}$$

Enfin, par récurrence, montrons que la propriété $\mathscr{P}_a : a^p \equiv a \pmod{p}$ est vraie pour tout $a \in \mathbb{N}$.

— Initialisation : pour $\mathscr{P}_0$, c'est évident.

1. Pierre de Fermat (XVIIème siècle) est un magistrat et surtout un mathématicien français aussi appelé "le prince des amateurs"
2. Rappel : $\binom{p}{k} = \frac{p!}{k!(p-k)!}$

— Hérédité : pour $\mathscr{P}_a$, on suppose que $a^p \equiv a \pmod{p}$. Alors, au rang $a+1$, on aura :

$$\begin{aligned}(a+1)^p &= a^p + 1 \pmod{p} \\ &= a+1 \pmod{p}\end{aligned}$$

$\mathscr{P}_{a+1}$ est hériditaire pour tout $a \in \mathbb{N}$

— Conclusion : $a^p \equiv a \pmod{p}$ est vraie pour tout $a \in \mathbb{N}$ et par extension pour tout $a \in \mathbb{Z}$.

On a donc $a^p - a \equiv 0 \pmod{p} \iff$ p divise $a^p - a$. Or, comme a et p sont premiers entre eux, d'après le théorème de Gauss, p divise $a^p - a$ ce qui nous donne le petit théorème de Fermat :

$$a^p \equiv a \pmod{p}$$

□

De ce premier théorème on obtient le corollaire suivant :

Corrolaire 12

Si p ne divise pas a alors :

$$a^{p-1} \equiv 1 \pmod{p}$$

Théorème 13 (Petit Théorème de Fermat amélioré)

Soient p et q deux nombres premiers distincts et soit $n = pq$. $\forall a \in \mathbb{Z}$ tel que $pgcd(a, n) = 1$ alors :

$$a^{(p-1)(q-1)} \equiv 1 \pmod{n}$$

Démonstration. Notons $c = a^{(p-1)(q-1)}$. Calculons c modulo p.

$c \equiv a^{(p-1)(q-1)} \equiv (a^{(p-1)})^{(q-1)} \pmod{p}$ avec $a^{(p-1)} \in \mathbb{Z}$
ce qui fait donc que $c \equiv 1^{(q-1)} \pmod{p}$ (Petit théorème de Fermat)

En procédant de la même manière pour c modulo q, on obtient aussi

$$c \equiv 1^{(q-1)} \pmod{q}$$

Conclusion partielle : $c \equiv 1^{(q-1)} \pmod{p}$ et $c \equiv 1^{(p-1)} \pmod{q}$

Comme $c \equiv 1 \pmod{p}$ alors $\exists \alpha \in \mathbb{Z}$ tel que $c = \alpha p + 1$ et comme $c \equiv 1 \pmod{q}$ alors $\exists \beta \in \mathbb{Z}$ tel que $c = \beta q + 1$. On a alors l'égalité $c - 1 = \alpha p = \beta q$.

De $\alpha p = \beta q$, on en déduit que $p | \beta q$. Or, p et q sont premiers entre eux d'où $p | \beta$ (Théorème de Gauss) donc $\exists \beta' \in \mathbb{Z}$ tel que $\beta = \beta' p$.
Ainsi

$$c \equiv 1 \pmod{q} \iff c = \beta q + 1 = \beta' pq + 1$$

Ce qui fait que $c \equiv 1 \pmod{n}$ c'est-à-dire $a^{(p-1)(q-1)} \equiv 1 \pmod{n}$ □

6 Fonction à sens unique

Il s'agit d'une fonction qui, grâce à son antécédent, permet de calculer aisément son image mais dont il est compliqué ou impossible de faire le calcul inverse pour retrouver cet antécédent. Néanmoins, en cryptographie asymétrique, il faut pouvoir déchiffrer les messages chiffrés. Alors, utiliser une fonction à sens unique comme clé de chiffrement sera problématique. On utilisera alors une fonction à sens unique avec trappe secrète, cette dernière sera la "clé" de déchiffrement.

Exemple :[3] Soit $f : x \mapsto x^3 \pmod{100}$
Des opérations simples suffisent pour trouver y = f(x) mais pour trouver x(y = f(x)), ce n'est pas aussi facile. Par exemple, pour trouver x tel que $x^3 \equiv 11 \pmod{100}$, il existe deux solutions :

- La première qui peut être longue et fastidieuse est la recherche complète de chaque solution de 1 à 99 : on trouve 71.

$$71^3 \equiv 357911 \equiv 11 \pmod{100}$$

- Le second est d'utiliser la trappe secrète qui donne très facilement la réponse : $y \mapsto y^7$:

$$11^7 \equiv 19487171 \equiv 71 \pmod{100}$$

Cette trappe est secrète car il est difficile de la trouver sans la connaître. Ainsi, pour l'obtenir, il existe une astuce ou méthode particulière que nous aborderons plus tard. Les fonctions à sens unique ont la particularité de compliquer le calcul de l'inverse d'une de leur opérations, ce qui est utile en cryptographie asymétrique.

3. Dans cet exemple, la fonction f employée n'est pas bijective !

III Introduction à la complexité

1 Complexité temporelle

Lorsque l'on cherche à résoudre un problème $\mathscr{P}$ avec l'aide d'un algorithme $\mathscr{A}$, on cherche à ce que ce dernier soit le plus rapide et le moins gourmand en ressources. Nous allons pour cela nous pencher sur la notion de complexité temporelle[14] de $\mathscr{A}$.

Définition 14 (Complexité temporelle)

On appelle fonction coût temporel de $\mathscr{A}$ toute fonction qui, ayant comme donnée d'entrée $n \in \mathbb{N}^*$, résoud le problème $\mathscr{P}$ après avoir effectué $f(n)$ opérations élémentaires *dans le pire des cas*. Les opérations élémentaires considérées sont l'affectation, l'addition, la multiplication et le modulo.
Ainsi, la **complexité temporelle** est donc la classe de la fonction coût temporel, ce qui signifie qu'elle représente l'ordre de grandeur de cette fonction.

Remarque importante : Surtout en cryptographie, on s'intéresse à la **taille** d'entrée de n.

Si l'on prend l'exemple de l'addition, calculer la somme de deux chiffres requiert d'effectuer une seule addition. La complexité de ce calcul est d'ordre 1. Mais si l'on calcule la somme de deux entiers à n chiffres, le calcul devient une addition chiffres à chiffres et demande donc d'effectuer n opérations d'additions soit une complexité d'ordre n.

Remarquons dans la définition de la complexité temporelle que la fonction coût temporel est évaluée selon le **pire des cas**. Cela signifie que la fonction coût temporel a mesuré le nombre d'opérations élémentaires pour la donnée d'entrée qui a effectué le nombre maximal d'opération possibles.

Or, il est aussi possible d'évaluer le **meilleur des cas**, c'est-à-dire, celui qui possède une donnée d'entrée qui minimise le nombre l'opérations, et le **cas générique**, qui fait la moyenne des opérations effectuées avec plusieurs données d'entrée différentes. On préférera en règle générale, utiliser celui qui évalue le pire des cas pour se prémunir des pires situations lorsque l'on emploie un algorithme mais il est tout à fait envisageable de pondérer l'usage des cas selon la probabilité d'apparition des pires ou meilleures données d'entrées d'un algorithme.

2 Notations asymptotiques

Définition 15 (fonctions asymptotiquement dominées)

Soit $\mathscr{F} = \{f : \mathbb{N} \to \mathbb{R}^+\}$, avec f, la fonction coût temporelle. On définit l'ordre :

$$\mathscr{O}(f) = \{g \in \mathscr{F} / \exists c \in \mathbb{R}^{+*}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, g(n) \leq cf(n)\}$$

Voici des ordres de grandeurs classiques sur les classes de complexité :

Classe	$\mathscr{O}(C)$	temps d'exécution arbitraire
constant	1	très rapide
logarithmique	$log(n)$	très rapide
linéaire	n	très rapide
quasilinéaire	$nlog(n)$	rapide
polynômiale	$n^\alpha (\alpha > 1)$	réalisable avec un ordinateur
exponentielle	$e^{\alpha * n} = k^n (\alpha > 0, k > 1)$	trop long
factorielle	$n!$	trop long

Exemple avec un algorithme de recherche dans un tableau (en python)

Soit $\mathscr{A}_1$ un algorithme[12] qui indique si une valeur x est contenue dans un tableau T de taille prédéfinie $n > 0$. Il renvoie Vrai si x est présent, Faux à contrario. Voici son implémentation en python :

```
1 def contains1(x,T):
2         i = 1
3         while(i > len(T)):
4                 if(x == T[i]):
5                         return true
6                 else:
7                         i=i+1
8         return false
```

Si l'on compte l'affectation ligne 2, puis les deux opérations (if($x == T[i]$) et $i = i + 1$) et la fonction $len()$ qui donne la taille n du tableau T en comptant chaque élément du tableau, ce qui donne $n+2$ opérations, que l'on répète n fois avec la boucle *while* dans le pire des cas, c'est-à-dire, que x n'est pas dans le tableau. Le coût temporel C_1 de $\mathscr{A}_1$ est alors $C_1 = n^2 + 2n + 1$ et donc $\mathscr{O}(C_1) = n^2$

On prend maintenant $\mathscr{A}_2$ qui possède la même fonction coût temporel que $\mathscr{A}_1$ mais en étant plus efficace. Voyons cela :

```
1 def contains2(x,T):
2         t = len(T)
3         for i in range(t):
4                 if(x == T[i]):
5                         return true
6         return false
```

Ici, on a une affectation ligne 2 qui vaut $n+1$ opérations à cause de la fonction $len()$ et une boucle for et le test $if(x == T[i])$ qui comptent au pire des cas $2n$ opérations. La complexité C_2 de $\mathscr{A}_2$ est alors $C_2 = 3n+1$ et donc $\mathscr{O}(C_2) = n$.

La différence est significative puisque l'on baisse d'une classe de complexité. Si l'on recherche x dans un tableau contenant un nombre de valeurs conséquent, la différence de durée de calcul entre les deux algorithmes sera alors très importante à cause du nombre d'opérations élémentaires effectuées. Par exemple, pour $n = 1000$, cela représente au moins 1000000 opérations élémentaires pour $\mathscr{A}_1$ contre 1000 pour $\mathscr{A}_2$. Nous verrons un peu plus tard, l'importance de gagner en rapidité.

Chapitre 2

Déchiffrons le système RSA

Maintenant que nous sommes armés de presque toutes les connaissances nécessaires pour comprendre le système RSA, voyons ensemble son fonctionnement.

I Comment ça marche

Posons le problème suivant : Alice et Bob souhaitent communiquer secrètement un message mais Eve, une espionne les écoute. La cryptographie RSA étant asymétrique, le processus emploie de ce fait une clé publique connue par tous et créée par Alice avec de très grands nombres, ainsi qu'une clé privée qu'elle a pris soin de cacher aux yeux de Bob et Eve. Bob va alors envoyer son message secret à Alice par le biais de sa clé publique. Alice va alors déchiffrer son message avec sa clé privée à la réception de celui-ci. Ainsi, Eve n'a aucun moyen de décrypter le message même si elle connait la clé publique et le message chiffré.

1 Explication

Le chiffrement RSA se déroule en 4 étapes, les trois premières vont aboutir au couple (n, e) la clé publique et la dernière, à la clé privée d. On utilisera e comme exposant pour chiffrer le message à transmettre secrètement (en modulo n). De ce fait, on utilise une fonction à sens unique où la trappe secrète sera d, l'exposant "inverse" de e qui permettra de retrouver le message codé. Plus précisément, voici les 4 étapes qui permettent de mettre en place ce système RSA :

— Etape 1 : Choix de deux nombres premiers.
Soient **p et q deux nombres premiers distincts** tels que $p \times q = n$, un entier naturel non nul. n constitue la première clé publique de ce système.
— Etape 2 : Calcul de la fonction indicatrice d'Euler de n.
On recherche avec p et q la **fonction indicatrice d'Euler de** n, facilement calculable car n est un produit de nombres premiers d'où $\phi(n) = \phi(p \times q) = \phi(p) \times \phi(q) = (p-1)(q-1)$. On détruit alors p et q par sécurité.
— Etape 3 : Choix de l'exposant.
On choisit un exposant e tel que e et $\phi(n)$ soient premiers entre eux. e est la seconde clé publique. On a donc crée le couple clé publique (e, n).
— Etape 4 : Calcul de l'inverse de l'exposant.
On crée la trappe secrète de la fonction à sens unique : **d l'inverse de e modulo** $\phi(n)$. Autrement dit, $\exists d \in \mathbb{Z} | d \times e \equiv 1 \pmod{\phi(n)}$. Ainsi, on obtient la clé privée de déchiffrement.

2 Mise en situation

Prenons un exemple pour illustrer :
— Alice prend $p = 5$ et $q = 17$ ce qui donne $n = 5 \times 17 = 85$ qu'elle donne à Bob (et que Eve peut connaître donc).
— $\phi(n) = (p-1)(q-1) = 64$ reste secret.
— Elle choisit ensuite $e = 5$. On pourra vérifier que $pgcd(e, \phi(n)) = pgcd(5, 64) = 1$ pour que 5 et 64 soient premiers entre eux.
— Alice calcule l'inverse de 5 modulo 64 pour trouver d en effectuant l'algorithme d'Euclide étendu. En effet, l'un des coefficients de Bézout correspond à cette inverse. On a alors $5 \times 13 + 64 \times (-1) = 1$. Comme $5 \times 13 \equiv 1 \pmod{64}$, $d = 13$ et cette clé restera secrète pour Bob et Eve. Elle détruit également $\phi(n)$ pour garantir la sécurité de sa clé secrète.

Les nombres pris ici sont volontairement petits pour un souci de compréhension et lisibilité mais, dans la réalité, les clés utilisées et recommandées pour une sécurité optimale sont de 2048 bits, voire 4096 bits pour un usage sensible.

3 Principe mathématique du chiffrement

Comme dit précédemment, le théorème de Fermat amélioré est la base du fonctionnement de ce système de (dé)chiffrement :

Théorème 16

Soit $d \in \mathbb{Z}$, l'inverse de e modulo $\phi(n)$, la fonction indicatrice d'Euler de $n = p \times q$ deux nombres premiers distincts. Alors pour m, un message secret, on aura :

$$\text{Si } x \equiv m^e \pmod{n} \text{ alors } m \equiv x^d \pmod{n}$$

x étant le message m chiffré

Démonstration. Soit d l'inverse de e modulo $\phi(n)$. Cela signifie que $de \equiv 1 \pmod{\phi(n)}$ donc que $\exists k \in \mathbb{Z}$ tel que $de = 1 + k\phi(n)$. Distinguons deux cas pour utiliser le fait que lorsque m et n sont premiers entre eux, on peut appliquer le petit théorème de Fermat amélioré.

Soit $x \in \mathbb{N}^*$ le message entier m chiffré avec la clé publique e.

— Premier cas : $pgdc(m, n) = 1$
On a alors $x \equiv m^e \pmod{n}$. Calculons x^d :

$$\begin{aligned}
x^d &\equiv (m^e)^d \pmod{n} \\
x^d &\equiv m^{1+k\phi(n)} \pmod{n} \\
x^d &\equiv m \times m^{k\phi(n)} \pmod{n} \\
x^d &\equiv m(\underbrace{m^{\phi(n)}}_{\in\mathbb{Z}})^k \pmod{n} \text{ (Application du petit théorème de Fermat)} \\
x^d &\equiv m \times 1^k \pmod{n} \\
x^d &\equiv m \pmod{n}
\end{aligned}$$

— Deuxième cas : $pgcd(m, n) \neq 1$
Rappel : on a $n = p \times q$, p et q premiers et $m < n$. Si $pgcd(m, n) \neq 1 \Rightarrow p|m$ ou $q|m$. Supposons que $pgcd(m, n) = p$ et $pgcd(m, q) = 1$. De même pour $pgcd(m, n) = q$ et $pgcd(m, p) = 1$.
Calculons d'abord $(m^e)^d$ modulo p :

$$\begin{aligned}
m &\equiv 0 \pmod{p} \\
\text{et } (m^e)^d &\equiv 0 \pmod{p} \\
\text{alors } m &\equiv (m^e)^d \pmod{p}
\end{aligned}$$

puis $(m^e)^d$ modulo q :

$$\begin{aligned}
(m^e)^d &\equiv m(\underbrace{m^{(q-1)}}_{\in\mathbb{Z}})^{(p-1)k} \pmod{q} \text{ (Application du Corrolaire du th. de Fermat)} \\
(m^e)^d &\equiv m \times 1^{(p-1)k} \pmod{q} \\
(m^e)^d &\equiv m \pmod{q}
\end{aligned}$$

Comme p et q sont premiers, cela implique qu'ils sont premiers entre eux. De la même manière qu'avec le petit théorème de Fermat amélioré, on peut conclure que :

$$(m^e)^d \equiv m \pmod{n}$$

□

4 En résumé

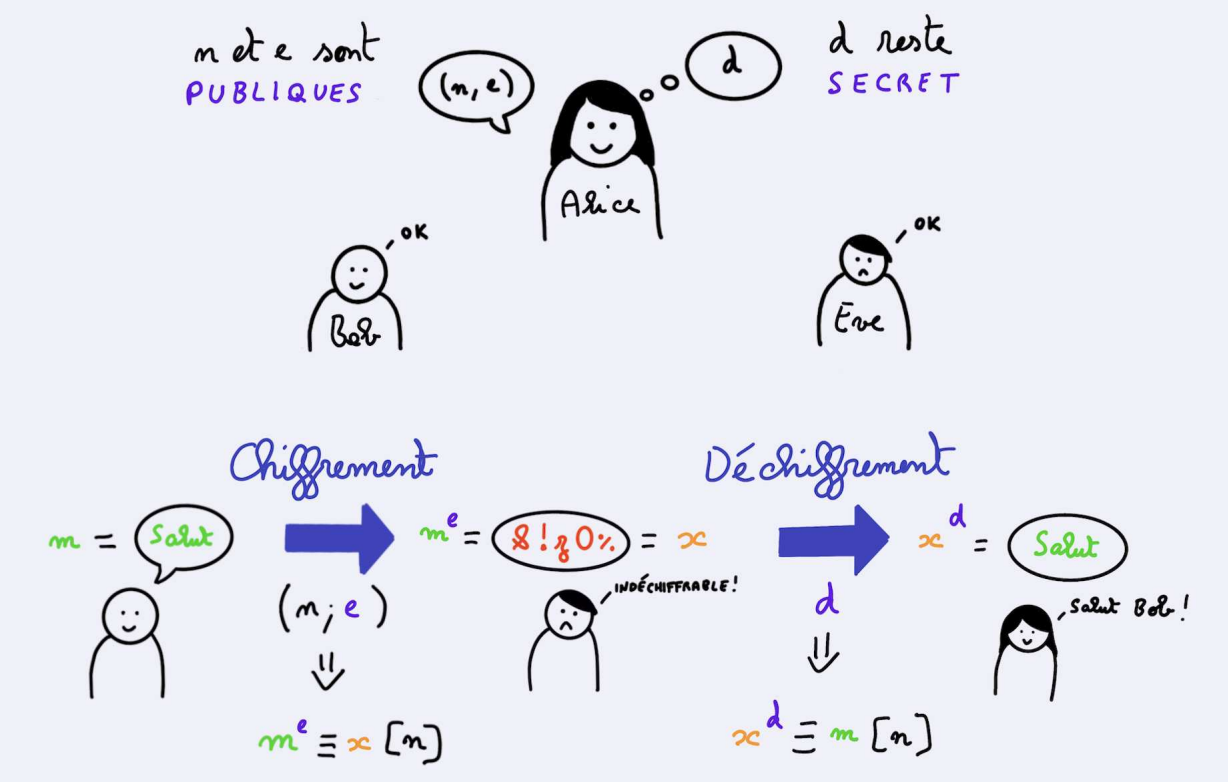

FIGURE 2.1 – Schéma synthétique du chiffrement RSA

5 Programme de chiffrement/décriffrement

Une programme de simulation en python est disponible en **annexe A**. Il permet à l'utilisateur de créer une clé publique et une clé privée grâce à la méthode évoquée ci-dessus puis de chiffrer un message et le déchiffrer avec ces mêmes clés.

```
1  # L'utilisateur entre p
2  p = int(input("Entrez un grand nombre premier p : "))
3  # L'utilisateur entre q
4  q = int(input("Entrez un grand nombre premier q : "))
5
6  # On calcule phi(n)
7  phiN = (p-1)*(q-1)
8  print("\nphi(n)= ",phiN)
9
10 # On determine les cles publiques n et e
11 n,e = clePublique(p,q)
12 print("\nCle publique (n,e) = (",n,",",e,")")
13
14 # On determine la cle privee d
15 d = clePrivee(phiN,e)
16 print("\nCle privee d =",d)
17
18 # On demande d'entrer le texte a chiffrer
19 m = str(input('\nEntrez le mot ou la phrase a chiffrer : '))
20
21 # chiffrement
22 x = chiffrementRSA(m,e,n)
23 afficherMessage(x)
24
25 # dechiffrement
26 messageSecret = dechiffrementRSA(x,d,n)
27 afficherMessage(messageSecret)
```

II Pourquoi ce système est-il robuste aujourd'hui ?

Toute la force de la cryptographie RSA repose sur l'absence d'une technique efficace de factorisation de très grands nombres entiers : le record est la factorisation d'un entier de 768 bits et la meilleure méthode de factorisation est d'ordre $e^{4n^{1/3}}$. Seule la trappe secrète d permet de trouver instantanément le message codé et celui qui veut déchiffrer le message doit réussir à factoriser n en deux nombres premiers... ce qui est techniquement trop long à l'heure actuelle, voyons cela :

1 Exponentiation rapide

Comme le système RSA utilise la fonction puissance (modulaire mais on va parler de la puissance simple) pour chiffrer et déchiffrer le message, il faut que l'algorithme d'exponentiation soit efficace. Or, l'**exponentiation naïve** x^k consiste à multiplier x avec lui même k fois, ce qui n'est pas efficace. Autant dire que l'ordre de la complexité d'une telle opération est relativement élevée pour ce type de calcul ($\mathcal{O}(k)$). Heureusement, il existe une méthode bien plus rapide d'exponentiation que nous allons présenter :

On utilise développement de l'exposant k en base 2 : $(k_l, ..., k_2, k_1, k_0)$ avec $k_i \in \{0, 1\}$ tel que :

$$k = \sum_{i=0}^{l} k_i 2^i$$

Le calcul de x^k est alors de

$$x^{\sum_{i=0}^{l} k_i 2^i} = \prod_{i=0}^{l} (x^{2^i})^{k_i}$$

On appelle cette méthode l'**exponentiation rapide**. Comparons avec un exemple simple les deux méthodes pour observer leur différence. Pour a^{10}, l'exponentiation de la naïve sera $a \times \cdots \times a$ alors que pour l'exponentiation rapide cela sera $((a^2)^2)^2 \times a^2$. On voit aisément que ce n'est pas la même opération bien que le résultat sera le même.

On va s'intéresser expérimentalement à ces méthodes avant de regarder la complexité algorithmique de chacune de ces deux méthodes.

Algorithme d'exponentiation naïve de a puissance k :

```
1 def exponentiationNaive(a,k):
2     puissance = 1
3     for i in range(k):
4         puissance = (puissance*a)
5     return puissance
```

Algorithme d'exponentiation rapide de a puissance k :

```
1 def exponentiationRapide(a,k):
2     puissance = 1
3     while(k>0):
4         if(k\%2 == 0):
5             a = (a*a)
6             k = k//2
7         else:
```

```
8                 puissance = (puissance*a)
9                 k = k-1
10      return puissance
```

Nous allons exécuter ces deux programmes python pour 5^k et récupérer leurs temps de calculs en fonction de la longueur en chiffres de l'exposant k puis de la valeur de k.

Première expérience :

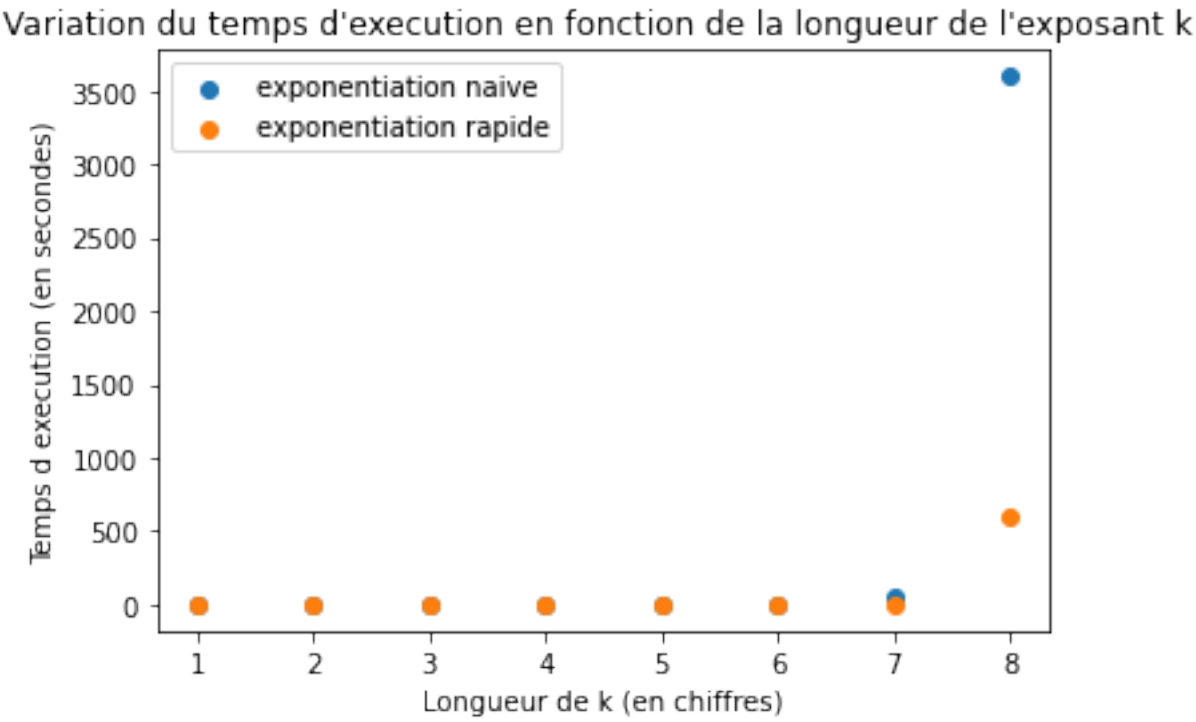

FIGURE 2.2 – Première expérience

On peut voir que les deux algorithmes ont une durée d'exécution assez similaire lorsque la longueur de k est inférieure à 6 chiffres ($k < 100000$), l'écart commence à beaucoup se creuser à partir de $k = 7$. Il est donc intéressant de chercher plus de valeurs à partir de $k = 100000$. Les calculs au delà de 8 chiffres sont trop longs pour être relevés.

Deuxième expérience :

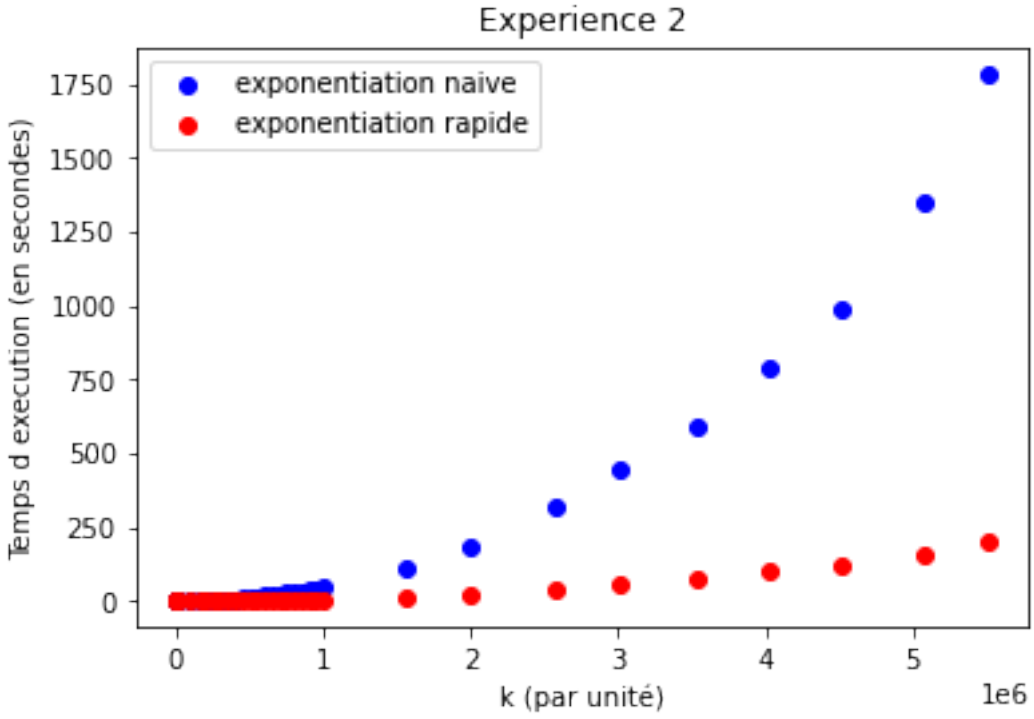

FIGURE 2.3 – Deuxième expérience

A partir de $k = 1000000$, l'écart devient de plus en plus flagrant entre les deux algorithmes. Par exemple, toujours pour $k = 1000000$, l'exponentiation naïve met quasiment 8 fois plus de temps que l'exponentiation rapide. D'ailleurs, nous pouvons voir que la courbe de points de l'exponentiation rapide peut être assimilée à une droite alors que l'exponentiation naïve ressemble à une courbe de fonction exponentielle.

Analysons maintenant la complexité de l'algorithme d'exponentiation rapide. On va s'intéresser à la taille binaire de k. Soit $l+1$ avec $l \in \mathbb{N}$, la taille binaire de k l'exposant. On peut dire que :

$$2^{l+1} \leq k < 2^{l+2}$$

donc que

$$(l+1)ln(2) \leq ln(k) < (l+2)ln(2)$$

En passant au logarithme binaire, on a donc :

$$(l+1) \leq ln_2(k) < (l+2)$$

Pour résumer, la boucle *while* (ligne 3) de l'algorithme d'exponentiation rapide effectue dans le pire des cas (à savoir un k en binaire composé exclusivement de bits égaux à 1) $2l + 1$ opérations élémentaires dans cette boucle qui serait répétée 4 fois. On a donc une fonction coût temporel $C = f(2l + 1) = 3 + (2l + 1) \times 4 = 8l + 7$ d'où la complexité en $\mathcal{O}(l)$. Une complexité linéaire donc, par rapport à la taille de l et quasi linéaire si l'on raisonne en fonction de la valeur de k $\mathcal{O}(ln_2(k))$.

En revanche, pour l'algorithme d'exponentiation naïve, on a une complexité en $\mathcal{O}(2^l)$ où l est toujours la taille de la clé. De manière équivalente, elle est en $\mathcal{O}(k)$ (où k est l'exposant), si l'on raisonne en fonction de la valeur de la clé.

2 Factorisation naïve

Si l'on arrivait à factoriser n, on pourrait casser le système RSA puisque l'on connait e. En effet, en ayant p et q qui forment $\phi(n)$, on pourra trouver d en calculant l'inverse de d modulo $\phi(n)$. Seulement, ce n'est pas si simple. Regardons comment procède une fonction de factorisation naïve.

On teste tous les nombres entiers jusqu'à la racine carrée de n afin de trouver un diviseur de n. Illustration :

```
1  def factorisationNaive(n):
2          i = 2
3          # On teste chaque entier inferieur a racine de n
4          while(i < sqrt(n)):
5                  if(n%i == 0):
6                          a = n//i
7                          b = i
8                          return (a,b)
9                  i = i + 1
10         # Si l'on ne trouve aucun diviseur: n est premier
11         return (1,n)
```

Parlons complexité. Sachant que pour la valeur de n, on a $\mathcal{O}(\sqrt{n})$ en regardant l'algorithme précédent. Ce qui fait, par rapport à la taille de n, une complexité de $\mathcal{O}(e^{\frac{1}{2}l})$: les factorisations de nombres très grands deviennent alors quasi-impossible à l'heure actuelle ! D'autant plus qu'il faut que la factorisation soit composée uniquement de deux nombres premiers distincts et l'on se retrouve avec un temps de calcul incommensurable pour factoriser n.

Pourrait on améliorer cette méthode comme cela a été fait pour l'exponentiation ? En réalité, il n'existe à ce jour aucune méthode de factorisation qui possède une complexité meilleure qu'exponentielle. On se contentera de

méthodes dotées d'une complexité dite "sous-exponentielle" qui sont plus efficaces que la factorisation que nous venons de voir.

Cette partie nous montre bien que la fiabilité du système RSA repose de nos jours essentiellement sur cette difficulté à factoriser des grands nombres.

Chapitre 3

Factorisation des grands nombres

Factoriser la clé publique n est le rêve de celui qui souhaite casser le système RSA. Cela signifie qu'il a obtenu les facteurs p et q, essentiels pour former $\phi(n)$ et ainsi retrouver d, la clé privée. En effet, on rappelle que, connaissant e, on a :

$$\exists d \in \mathbb{Z} | d \times e \equiv 1 \pmod{\phi(n)} \text{ où } \phi(n) = (\mathbf{p}-1)(\mathbf{q}-1) \text{ serait connu}$$

Toutefois, nous avons vu qu'il n'est pas si facile de factoriser de très grands nombres comme ceux utilisés actuellement pour chiffrer (à savoir des nombres composés de minimum 2048 bits) et qu'une factorisation dite naïve est de complexité $\mathcal{O}(e^{\frac{1}{2}l})$ par rapport à la taille de d'entrée. Voyons à présent quelques méthodes de factorisation, dont une sous exponentielle, qui peuvent *potentiellement* constituer des attaques en RSA.

I La factorisation selon Fermat

Cette méthode est surprenante par sa simplicité mais elle n'est pas évidente pour autant. On cherche à factoriser un nombre n composé, strictement positif et impair. La chose importante à souligner est que les nombres premiers sont tous impairs hormis 2, que l'on va exclure car trop petit dans notre étude (on factorise de très grands nombres !). La méthode qui suit comme les suivantes convient parfaitement à n car elles s'appliquement uniquement à ces nombres.

En 1643, le Père Mersenne met au défi Fermat de factoriser 100 895 598 169[15]. Quelle ne fut pas sa surprise de recevoir sa réponse quelques jours plus tard soit 898423×112303, s'avérant être la bonne réponse. Il indique

cependant dans une lettre ultérieure une méthode générale suivante : On a $n = d_1 d_2$ avec $1 < d_1 < d_2$ tel que $a = \frac{d_1+d_2}{2}$ et $b = \frac{d_1-d_2}{2}$ tel que $n = a^2 - b^2$.

Ce qui nous intéresse c'est d'obtenir a et b. On procède ainsi : posons $a = \lfloor \sqrt{n} \rfloor + 1$. Si $a^2 - n$ est un carré parfait, on note $a^2 - n = b^2$ et on trouve a et b. Sinon, $a^2 - n$ n'est pas un carré parfait donc on reprend $a = \lfloor \sqrt{n} \rfloor + \mathbf{2}$ et ainsi de suite. Alors la formule générale sera

$$a = \lfloor \sqrt{n} \rfloor + k$$

avec $k \in \mathbb{Z}$ et on est sûr que l'opération se terminera car on a déjà $b = \frac{d_1-d_2}{2}$.

Exemple : Fermat dans sa lettre explicative a décomposé $n = 2027651281$ pour illustrer sa méthode. Ainsi :

$$\lfloor \sqrt{n} \rfloor = 45029 \text{ et donc } a = 45029 + 1 = 45030$$

Or, $45030^2 - 2027651281 = 49619$ n'est pas un carré parfait. On continue avec $\lfloor \sqrt{n} \rfloor + 2 = 45031$, etc... La bonne étape est pour $k = 12$ car $a = 45029 + 12 = 45041$ et $45041^2 - 2027651281 = 1040400$. Comme $\sqrt{1040400} = 1020 = b$, on a notre un carré parfait.

Ce qui montre que

$$\begin{aligned} n &= 2027651281 = 45041^2 - 1020^2 \\ &= (45041 - 1020)(45041 + 1020) \\ &= 44021 \times 46061 \end{aligned}$$

Le programme écrit en Python pour tester :

```
1 import math
2
3 n = 20276512810
4 a = math.floor(math.sqrt(n))+1
5 b = math.sqrt(a**2-n)
6 while b !=int(b) :
7         a += 1
8         b = math.sqrt(a**2-n)
9 print("a= %d et b= %d : n= %d x %d"%(int(a),int(b),int(a- b),int(a+b)))
```

Seulement, cette technique n'est efficace que dans certains cas à cause des diviseurs milieux de n. Plus ils sont éloignés de $\sqrt{n}$, moins la méthode est efficace. Dans ce cas, on vérifie au préalable la divisibilité de n par les petits nombres premiers, pour ensuite appliquer la méthode de Fermat.

La complexité dépend de k puisque l'algorithme s'effectue en k étapes : $k = \frac{d_1+d_2}{2} - \lfloor \sqrt{d_1 d_2} \rfloor$ avec d_1 plus grand diviseur inférieur à $\sqrt{n}$ et $d_2 = n/d_1$.

Ainsi, cette méthode reste d'ordre exponentiel mais sert de base pour celles qui vont réussir à factorier des n de taille actuellement obsolète.

II Méthode du crible quadratique

Ceci est la méthode de factorisation de Dixon qui est améliorée par Carl Pomerance [1] et connue sous le nom de l'algorithme du crible quadratique[17].

Décomposée en deux parties, elle part sur la même base que la méthode de Fermat vue précédement : on cherche deux entiers premiers x et y tels que $n = x^2 - y^2 = (x - y)(x + y)$ ce qui équivaut à chercher des x et y tels que

$$x^2 \equiv y^2 \pmod{n}$$

— Phase 1 :

La première partie aboutit à l'obtention de relations qui sont constituées de factorisation de $y(x) = x^2 - n$ sur une base $\mathscr{F}$ de petits nombres premiers inférieurs ou égaux à un entier B. On dit alors que ces nombres sont B-friables et on pose la base $\mathscr{F} = \{p_0, ..., p_n\}$ avec $p_0 = -1$ pour factoriser des entiers négatifs. Si $y(x)$ se factorise sur cette base, il est B-régulier et constitue une relation.

Ainsi, en prenant x proche de la racine carrée de n et en décomposant son carré modulo n, on trouve cette décomposition de facteurs premiers B-friables, on obtient un $y(x)$ dont on vérifie la régularité efficacement grâce au crible quadatique.

$$y(x) = x^2 - n = (\lfloor\sqrt{n}\rfloor + a)^2 - n \approx 2a\lfloor\sqrt{n}\rfloor \text{ (a un entier très petit)}$$

Ainsi, les relations seront de la forme :

$$x^2 \equiv p_0^{\alpha_0} \times ... \times p_n^{\alpha_n} \pmod{n}$$

avec $\{\alpha_0, ..., \alpha_n\}$ entiers positifs.
En réalité, il serait trop long de tester la régularité à chaque fois que l'on choisi un x même si le criblage ne donne pas l'information détaillée de la factorisation des y réguliers. Or, comme ils sont composés de petits facteurs premiers, ils sont calculables rapidement ce qui contitue

1. Carl Pomerance est américain, théoricien des nombres très productif depuis le XXème siècle. On lui doit la méthode du crible quadratique mais également un algorithme déterministe testant la primarité d'un entier positif

une très légère perte de temps.

— Phase 2 :

L'objectif de cette partie est d'obtenir à l'aide des relations calculées précèdemment un carré à droite de l'équation $x^2 \equiv p_0^{\alpha_0} \times ... \times p_n^{\alpha_n} \pmod{n}$ ce qui va aboutir une congruence de carrés, c'est-à-dire, $x^2 \equiv y^2 \pmod{n}$.
On va donc (si besoin) multiplier les relations entre elles pour produire ce carré parfait. Pour ce faire, il faut sélectionner celles dont la somme des exposants $\{\alpha_0, ..., \alpha_n\}$ est paire une fois ces relations multipliées. Le tout est de choisir efficacement ces relations pour atteindre le résultat escompté du premier coup, même si malheurement il est possible que l'on trouve un facteur trivial et donc recommencer la recherche des bonnes relations.
La recherche efficace et systématique de ces relations s'effectue à l'aide de l'algèbre linéaire en trouvant une famille liée dans l'espace vectoriel $(\mathbb{Z}/2\mathbb{Z})^k$.
On contruit alors la matrice $A_{n,i}$ avec une relation par colonne et dont les coefficients sont les exposants des facteurs composant ces i relations (modulo 2 puisqu'on s'intérresse à la parité).

$$A_{n,i} = \begin{pmatrix} (\alpha_0)_1 & \cdots & (\alpha_0)_i \\ \vdots & \ddots & \vdots \\ (\alpha_n)_1 & \cdots & (\alpha_n)_i \end{pmatrix} \pmod{2}$$

Grâce au pivot de Gauss, on trouve une combinaison linéaire dont les solutions indiquent quelles sont les relations à multiplier (plusieurs combinaisons de relations sont possibles). On obtient alors y.

— Aboutissement :
Comme nous connaissons x et y grâce l'équation $x^2 \equiv y^2 \pmod{n}$ fraîchement trouvé, donc on calcule $p = pgcd(x+y, n)$ et $q = pgcd(x-y, n)$ et l'on obtient la factorisation de n.

Exemple : On souhaite factoriser $n = 2041$. On pose $B = 10$, ce qui donne une base $\mathscr{F} = \{2, 3, 5, 7\}$ [2].

On recherche simultanément toutes les valeurs B-friables de $y(x)$ pour x dans un tableau $[x_0, x_1, ..., x_{t-1}]$ de taille t. Si l'on connait un x tel que $y(x) \equiv 0 \pmod{p}$, on en déduit toute une suite de y qui sont divisibles par

2. ici, on ne prendra pas en compte $p_0 = -1$ pour facilité la démarche

p. On utilise un second tableau pour stocker les y : $[y_0, ..., y_{t-1}]$ initialement rempli de 1. Ensuite, pour un p donné par la base, on calcule les deux racines de $y(x)$ modulo p (une seule pour l'exception $p = 2$). On obtient alors deux indices i_0 et i_1 tels que $p|y(x_{i_0})$ et $p|y(x_{i_1})$. On multiplie tous les y_{i_1+kp} et y_{i_2+kp} en parcourant de p en p le tableau des y pour cribler. Un fois tous les p de la base criblés, on repère les éléments du tableau des y_i qui sont friables : ce sont ceux qui se présentent sous la forme $y_i = q(x_i)$. On crible également avec des puissances de nombres premiers pour ne pas oublier des B-friables. Or, multiplier les y_i par des p est trop fastidieux. On utilise à la place les logarithmes binaires et on additionne des approximations de $log_2(p)$.

Soient $\lfloor\sqrt{n}\rfloor = 46$ et l'intervalle $[x_1; t] = [10; 20]$. Commencons par $p = 2$. x et x_0 sont pairs et n impair, l'équation $(x + 46)^2 - 2041 \equiv 0 \pmod 2$ on a une seule solution qui est $x \equiv 1 \pmod 2$ Ainsi, nous avons $i_0 = 1$, ce qui donne le premier tableau ci dessous (celui des y) :

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

Pour $p = 3$, les deux solutions de $(x + 46)^2 - 2041 \equiv 0 \pmod p$ sont $x = 0$ et $x = 1$ alors on obtient $i_0 = 0$ et $i_1 = 2$. En prenant $log_2(3) \approx 2$ et en avançant de 2 en 2, on a :

2 1 2 3 0 3 2 1 2 3 0 3 2 1 2 3 0 3 2 1

Pour $p = 5$, les deux solutions à $(x+46)^2 - 2041 = 0 \pmod 5$ sont $x = 0$ et $x = 3$ d'où $i_0 = 0$ et $i_1 = 3$. Alors avec $log_2(5) \approx 2$, on obtient :

4 1 2 5 0 5 2 1 4 3 2 3 2 3 2 5 0 3 4 1

pour $p = 7$, $log_2(7) \equiv 3$:

4 1 5 5 0 8 2 1 4 6 2 3 5 3 2 5 3 3 4 4

La valeur 8 est intéressante car correspond à une relation $512 = 24 \times 5 \times 7 \pmod{2041}$, etc...

On trouve donc à l'aide du crible les congruences suivantes :

$$
\begin{array}{rcl}
46^2 & \equiv & 075 \equiv \quad 3 \times 5^2 \quad \pmod{2041} \\
47^2 & \equiv & 168 \equiv 2^3 \times 3 \quad \times 7 \pmod{2041} \\
49^2 & \equiv & 360 \equiv 2^3 \times 3^2 \times 5 \quad \pmod{2041} \\
51^2 & \equiv & 560 \equiv 2^4 \times \quad 5 \times 7 \pmod{2041} \\
53^2 & \equiv & 768 \equiv 2^8 \times 3 \quad \pmod{2041}
\end{array}
$$

Afin de choisir quelles relations utiliser, on crée la matrice A des exposants modulo 2 :

$$
A = \begin{pmatrix} 0 & 3 & 3 & 4 & 8 \\ 1 & 1 & 2 & 0 & 1 \\ 2 & 0 & 5 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \end{pmatrix} \equiv \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \end{pmatrix} \pmod{2}
$$

On échellonne A :

$$
\iff \begin{pmatrix} 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \end{pmatrix} \quad L_1 \leftrightarrow L_2
$$

$$
\iff \begin{pmatrix} 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix} \quad L_4 \leftarrow L_4 + L_2
$$

$$
\iff \begin{pmatrix} 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \pmod{2} \quad L_4 \leftarrow L_4 + L_3
$$

Et on trouve $B = (\lambda_1, \lambda_2, \lambda_3, \lambda_4, \lambda_5)^t$ avec $\lambda_i \in \mathbb{R}$ tel que $A \times B = 0$

$$
\begin{array}{rcl}
A \times B = 0 & \iff & \left\{ \begin{array}{rcl} \lambda_1 + \lambda_2 + \lambda_5 & \equiv & 0 \pmod{2} \\ \lambda_2 + \lambda_3 + \lambda_4 & \equiv & 0 \pmod{2} \\ \lambda_3 + \lambda_4 & \equiv & 0 \pmod{2} \end{array} \right. \\
& \iff & \left\{ \begin{array}{rcl} \lambda_1 + \lambda_5 & \equiv & 0 \pmod{2} \\ \lambda_2 & \equiv & 0 \pmod{2} \\ \lambda_3 + \lambda_4 & \equiv & 0 \pmod{2} \end{array} \right.
\end{array}
$$

On trouve $B = (1, 1, 1, 1, 0)^t$ ou $B = (1, 0, 0, 0, 1)^t$. La première solution signifie qu'il faut prendre toutes les relations sauf la dernière pour avoir un carré parfait ce qui donne :

$$\begin{aligned} 46 \times 47 \times 49 \times 51 &\equiv 311 \pmod{2041} \\ 311^2 (\equiv 794) &\equiv 2^{10} \times 3^4 \times 5^4 \times 7^2 \pmod{2041} \\ &\equiv (2^5 \times 3^2 \times 5^2 \times 7)^2 \pmod{2041} \\ 311^2 &\equiv 1416^2 \pmod{2041} \end{aligned}$$

Ainsi, $x = 311$ et $y = 1416$.

Calculons $pgcd(1416 + 311, 2041) = 157$ et $pgcd(1416 - 311, 2041) = 13$. Et on peut constater que nous avons bien $13 \times 157 = 2041$!

Pour la complexité, on admettra qu'elle est de $\mathcal{O}(n) = e^{2\sqrt{log(n)log(log(n))}}$ Le dernier plus grand nombre factorisé par cette méthode était composé de 135 chiffres décimaux. Depuis 1994, c'est le crible sur les corps de nombres qui permet les records de factorisation de tous les nombres RSA. Il est ainsi considéré l'algorithme classique le plus rapide (ce qui exclu l'algorithme de Shor qui est quantique).

Chapitre 4

Génération de grands nombres premiers

Comme nous l'avons vu précèdement, le système RSA possède un critère essentiel qui repose sur l'utilisation des nombres premiers. Seulement, sa plus grande force à l'heure actuelle est l'utilisation de très grands nombres premiers que les algorithmes de factorisation ne peuvent pas encore décomposer rapidement puisqu'ils sont de complexité sous exponentielle.

Le problème est alors d'engendrer un nombre entier impair aléatoire suffisament grand et de tester très rapidement sa primalité, s'il n'est pas premier, pouvoir recommencer un nombre de fois raisonnable. Nous allons donc nous pencher sur le test de primalité le plus utilisé pour notre sujet.

I Test de Fermat

Soit n un entier impair de primalité inconnue. On prend a un entier inférieur à n et on effectue un test avec le petit théorème de Fermat du Chapitre 1.

Si $a^{n-1} \equiv 1 \pmod{n}$ alors n est probablement premier, n est composé dans le cas contraire.

Ce simple test probabiliste de primalité est la base de celui employé par la suite, à savoir le test de Miller-Rabin (ou encore celui de Solovay-Strassen).

II Test de Miller-Rabin

Le test de primalité[20] par excellence du chiffrement RSA est celui de Miller-Rabin, très rapide et probabiliste de type Monte-Carlo. Cela implique pour un entier impair donné, l'affirmation de sa primalité est *probable* tandis

que celle de sa composition est sûre et certaine. Heureusement, la probabilité d'erreur peut être rendue faible selon les paramètres rentrés dans les faits.

Lemme 17 (de Miller-Rabin)

Soient p un nombre premier impair, $(s, d) \in \mathbb{N}^*$ avec d impair tels que p vérifie : $p = 2^s d + 1$.
Alors, $\forall a \in [\![1; p-1]\!]$ non divisible par p, on a :

$$a^d \equiv 1 \pmod{n} \text{ ou } \exists i \in \{0, 1, ..., p-1\} \text{ tel que } a^{2^i d} \equiv -1 \pmod{p}$$

L'algorithme qui permet la génération d'un nombre premier pour le chiffrement RSA emploie ce test très efficace malgrè sa dimension probabiliste. Voici son déroulement :

Soit un nombre entier impair n, de grande taille dont on ignore s'il est premier. On le décompose de la manière suivante : $n = 2^s d + 1$, avec s et d entiers non nuls et d impair.

On utilise le lemme de Miller-Rabin pour affirmer que $\forall a \in [\![1; n-1]\!]$ on a :

ou bien $a^d \equiv 1 \pmod{n}$ ou bien $\exists i \in \{0, 1, ..., p-1\}$ tel que $a^{2^i d} \equiv -1 \pmod{p}$

Si l'une des deux conditions est validée alors n **est probablement premier** et a est potentiellement un menteur fort si n n'est pas premier.

Par contraposition, si on a $a^{d2^s} \not\equiv 1 \pmod{n}$ et :

$$a^d \not\equiv 1 \pmod{n} \text{ et } \forall i \in \{0, 1, ..., p-1\} \text{ tel que } a^{2^i d} \not\equiv -1 \pmod{p}$$

Alors n **est forcément composé** et a est appelé témoin de Miller.

Démonstration. Posons $p > 2$, un nombre premier, décomposé ainsi $p = 2^s d + 1$. Soit (b_i) la suite définie par $b_i = a^{d2^i}$, $a \in \{1, ..., p-1\}$

Premièrement, dans $(\mathbb{Z}/p\mathbb{Z})$, l'équation $x^2 = 1 \iff (x-1)(x+1) = 0$ possède deux solutions : $x = 1$ ou $x = -1$.

Ensuite, comme $p \nmid a$ alors d'après le Corrolaire (12) du petit théorème de Fermat,

$$b_s = a^{2^s d} = a^{p-1} \text{ alors } a^{2^s d} \equiv 1 \pmod{p}$$

Comme $b_0 \not\equiv 1 \pmod{p}$ et $\forall j \geq s,$ on a $b_j \equiv 1 \pmod{p}$ (les b_j sont majorés par $s-1$), on peut prendre

$$i = sup\{j \geq 0, b_j \not\equiv 1 \pmod{p}\} = \{0, 1, ..., s-1\}$$

Aussi on a l'équation $(b_i)^2 \equiv (a^{2^i d})^2 \equiv a^{2^{i+1}d} \equiv b_{i+1} \equiv 1 \pmod{p}$.

Or, comme vu dans le premièrement, $(b_i)^2 \equiv 1 \pmod{p}$ donne $(b_i) \equiv -1 \pmod{p}$ ou $(b_i) \equiv -1 \pmod{p}$. Seulement, la première solution est impossible selon i d'où $(b_i) \equiv -1 \pmod{p}$

Donc si $b_0 \neq 1$ et $\forall i \in \{1, ..., s-1\}$, $b_i \not\equiv -1 \pmod{p}$, alors p n'est pas premier par contraposition.

□

La probabilité d'un "faux positif" est de $1/4$. En effet, le théorème de Rabin (ici admis) dit que dans notre cas de $n = d2^i d$ impair, il existe -au plus- $\phi(n)/4$ a menteurs forts. $\phi(n)$ étant l'indicatrice d'Euler. On estime qu'une vingtaine de répétitions du test permet d'être confiant quand à la primalité d'un nombre. Donc, si l'on réitère le test k fois avec des a aléatoires à chaque répétition, on borne à $(1/4)^k$ la probabilité d'identifier à tort un nombre premier qui ne l'est pas. Ainsi, si k augmente la probabilité d'erreur diminue au dépend du temps de calcul. Cependant, la complexité du test est polynomiale : $\mathcal{O}((log(n))^3)$ puisque les calculs les plus fastidieux à répéter k fois sont $a^d \pmod{n}$ et $(a^d)^2 \pmod{n}$. Cela signifie que ce temps de calcul restera raisonnable pour générer des nombres premiers très sûrement.

Est-ce vraiment grave si l'on obtient un nombre composé pour chiffrer ? Avec des nombres de très grande taille, la seule conséquence serait des problèmes de chiffrement/déchiffrement mais qui ne se savent pas dans l'absolu. Ajoutons à cela qu'il est possible qu'un nombre composé demande moins d'itérations, ce qui minimise les dommages d'une erreur commise par le test.

Nous pourrons alors compléter l'algorithme en annexe A avec un générateur de nombres premiers utilisant ce test de primalité pour générer p et q si l'on ne souhaite pas les rentrer manuellement.

```
1
2 import random
3
4 def generateurNombrePremier(borneInf,borneSup,nbIteration):
5     p = 6
6     while(not(estPremier(p,nbIteration))):
7         p = random.randint(borneInf,borneSup)
8     return p
```

III Tests sur les nombres de Mersenne

Pour illustrer l'algorithme de Miller-Rabin, nous allons tester une famille de nombres, ceux de Mersenne :

Définition 18 (les nombres de Mersenne)

Soit $n > 1$, les nombres de Mersenne s'écrivent sous la forme :

$$M_n = 2^n - 1$$

Pour que M_n soit premier, n doit être nécessairement premier mais cela n'est pas une condition suffisante (le contre-exemple le plus petit est $n = 11$).

Le test de primalité le plus efficace pour ces nombres est celui de Lucas-Lehmer mais nous allons employer Miller-Rabin pour l'observer avec un programme en python détaillé dans l'Annexe B. Il détecte les nombres premiers formés par les nombres de Mersenne.

Après un test pour $k = 11$ du programme de l'annexe B, on peut construire le tableau suivant :

n	M(n)	Premier
2	3	premier
3	7	premier
5	31	premier
7	127	premier
11	2047	composé
13	8191	premier
17	131071	premier
19	524287	premier
23	8388607	composé
29	536870911	composé
31	2147483647	premier
37	137438953471	composé
41	2199023255551	composé
43	8796093022207	composé
47	140737488355327	composé
53	9007199254740991	composé
59	576460752303423487	composé
61	2305843009213693951	premier
67	147573952589676412927	composé
71	2361183241434822606847	composé
73	9444732965739290427391	composé
79	604462909807314587353087	composé
83	967140655691703339764940 7	composé
89	618970019642690137449562111	premier
97	158456325028528675187087900671	composé

TABLE 4.1 – Primalité des nombres de Mersenne avec n premier pour $k = 70$

Ainsi, on remarque que pour $n = 11$ par exemple, on obtient un nombre de Mersenne composé, ce qui montre que la formule de nombres premiers est un échec.
Cependant, si on le test avec $k = 1$, on peut être surpris de voir que le test de primalité à lui aussi échoué pour M_{11} :

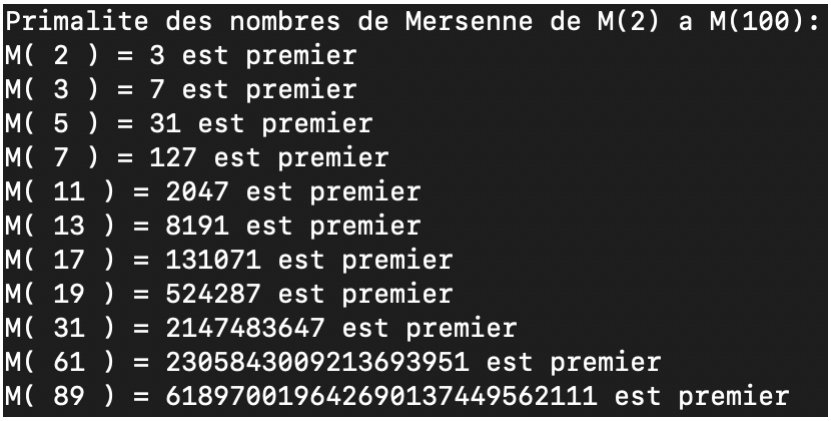

FIGURE 4.1 – Résultat du programme avec $k = 1$ en paramètre

Ainsi, le test de Miller-Rabin n'est pas infaillible, surtout avec une petite valeur de k.

Conclusion

Le chiffrement RSA est protégé par la difficulté de factoriser des grands nombres. A l'heure actuelle, tous les algorithmes classiques sont de complexité sous exponentielle. La grande majorité des ordinateurs ne peuvent casser le code RSA. Il faut donc se tourner vers l'informatique quantique qui possède un algorithme dont la complexité est polynomiale : c'est l'algorithme de Shor. C'est dans cette perspective qu'il faut regarder si l'on souhaite faire progresser la sécurité de nos données, quitte à rendre obsolète un système pourtant bien mené.

Arrivée à la fin de ce projet, je le quitte comme si j'avais voyagé avec lui. Il m'a permis de découvrir le système RSA en approfondissant le sujet et de l'enrichir avec des notions de cryptographie, de théorie de la complexité ou encore d'arithmétique. Mes connaissances scientifiques autant sur les domaines d'informatiques théoriques que mathématiques ont augmenté. Cela m'a donné la possibilité de me responsabiliser davantage dans mon rythme de travail, faire des recherches documentées et bibliographiées et d'apprendre à travailler accompagnée par un enseignant.
J'ai particulièrement apprécié d'utiliser de nouveaux outils tels que LaTeX pour rédiger ce rapport ou encore python pour effectuer mes expériences, langage inconnu pour moi jusqu'alors... Cette UV m'a fait prendre conscience de l'importance de la cryptographie, des sciences informatiques et mathématiques dans lesquelles je me projette.

Annexe A

Programme de simulation RSA

```
1  # RSA.py
2  # P2020
3
4  import math
5
6  # ====================
7  # Fonctions importantes
8  # ====================
9
10 # Retourne u et v tels que au + bv = pgcd(a, b)
11 def euclideEtendu(a,b):
12         (u0,v0,u1,v1) = (1,0,0,1)
13         while(b != 0):
14                 (q,r) = divmod(a,b)
15                 (a,b) = (b,r)
16                 (u0,v0,u1,v1) = (u1,v1,u0-q*u1,v0-q*v1)
17
18         return (u0,v0)
19
20 # Donne l'inverse de a modulo n
21 def inverseModulo(a,n):
22     u,v = euclideEtendu(a,n)
23     return u%n
24
25 def clePublique(p,q):
26     # Premiere cle
27     n = p*q
28     phiN = (p-1)*(q-1)
29
30     # Deuxieme cle
31     e = 3
32     # Variable de la prochaine boucle
33     compteur = 0
```

```
34      PGCD1 = 0
35      # On cherche le PGCD de e et phi(n) tel qu il soit egal a 1
36      while((math.gcd(e,phiN) != 1) or (q*2 >= e) or (p*2 >= e)):
37          e = e + 1
38      return (n,e)
39
40  def clePrivee(phiN,e):
41      u,v = euclideEtendu(e,phiN)
42          return u%phiN
43
44  # Chiffre le message m
45  def chiffrementRSA(m,e,n):
46          tailleM = len(m)
47          messageChif = []
48
49          # Chiffre caractere par caractere le message
50          for i in range(tailleM):
51              # Comme i s'incremente jusqu'a egalite avec la taille du message
52              # a chaque passage dans la fonction, chaque caractere sera converti
53              # grace a la fonction ord() qui revoit le code ASCII de son parametre
54              ascii = ord(m[i])
55
56                  # Erreur si le code ASCII est superieur a n
57              if(ascii > n):
58                  print("Les nombres p et q sont trop petits veuillez recommencer")
59
60              # Chiffre le caractere represente par son code ASCII
61              # et on ajoute le caratere au message chiffre
62              messageChif.append(pow(ascii,e,n))
63          return messageChif
64
65  # Dechiffre le message x
66  def dechiffrementRSA(x,d,n):
67          tailleX = len(x)
68          messageDechif = []
69
70          # Dechiffre bloc par bloc le message
71          for i in range(tailleX):
72                  lettreDechif = pow(x[i],d,n)
73                  messageDechif.append(chr(lettreDechif))
74          return messageDechif
75
76  # Affiche le message code ou non
77  def afficherMessage(m):
78          message = ""
79          for i in range(len(m)):
80                  message = str(message + str(m[i]))
81          print("Le message est :",message)
82
```

```
83 #DEBUT DU PROGRAMME
84
85 # L'utilisateur entre p
86 p = int(input("Entrez un grand nombre premier p : "))
87 # L'utilisateur entre q
88 q = int(input("Entrez un grand nombre premier q : "))
89
90 # On calcule phi(n)
91 phiN = (p-1)*(q-1)
92 print("phi(n)= ",phiN)
93
94 # On determine les cles publiques n et e
95 n,e = clePublique(p,q)
96 print("Cle publique (n,e) = (",n,",",e,")")
97
98 # On determine la cle privee d
99 d = clePrivee(phiN,e)
100 print("Cle privee d =",d)
101
102 # On demande d'entrer le texte a chiffrer
103 m = str(input("Entrez le mot ou la phrase a chiffrer : "))
104
105 # chiffrement
106 x = chiffrementRSA(m,e,n)
107 afficherMessage(x)
108
109 # dechiffrement
110 messageSecret = dechiffrementRSA(x,d,n)
111 afficherMessage(messageSecret)
```

Annexe B

Programme de test de primalité des nombres de Mersenne

```
1  #test-Miller-Rabin.py
2  #P2020
3
4  import random
5  import math
6
7  # la fonction est appele k fois lors du test de primalite
8  # Elle retourne false si n est compose, true si il est premier
9  # d doit etre un nombre impair obligatoirement decomposable
10 # de cette maniere : n = d*2^i - 1  pour i > 0
11 def testMiller(d, n):
12
13     # On prend un a aleatoire compris entre 2 et n-2
14     # avec n > 4 pour eviter des cas triviaux
15     a = 2+random.randint(1, n-4);
16
17     # On calcule son exponentiation modulo n
18     x = pow(a, d, n);
19
20         # Premiere condition :
21     if (x == 1 or x == n - 1):
22         return True;
23
24     # Deuxieme condition :
25     # On eleve au carre x tant que
26     # l'un des if n'est pas satisfait
27     # ou que d est different de n-1
28     while (d != n-1):
29         x = (x*x)%n;
30         d = d*2;
31
```

```
32          if (x == 1):
33              return False;
34          if (x == n-1):
35              return True;
36
37      # n est compose :
38      return False;
39
40 # k est le nomber de fois ou
41 # le test de Miler-Rabin est effectue
42 # plus k est eleve, plus la reponse
43 # de ce test est fiable
44 def estPremier(n,k):
45
46      # cas triviaux
47      if (n <= 1 or n == 4):
48          return False;
49      if (n <= 3):
50          return True;
51
52      # Trouve un i > 0 tel que
53      # n = 2^d * i + 1
54      d = n - 1;
55      while (d % 2 == 0):
56          d //= 2;
57
58          # on effectue le test de Miler-Rabin k fois
59      for i in range(k):
60          if (testMiller(d, n) == False):
61              return False;
62
63      return True;
64
65
66 #DEBUT DU PROGRAMME
67
68 k = 1
69 print("Primalite des nombres de Mersenne de M(2) a M(100):")
70
71 for n in range(2,100):
72
73      mersenne = pow(2,n) - 1
74      if(estPremier(mersenne,k)):
75          print("M(",n,") =",mersenne,"est premier")
```

Table des figures

Liste des tableaux

Bibliographie

[1] Crible quadratique. [En ligne] https ://fr.wikipedia.org/wiki/Crible$_q uadratique$.

[2] Nombre de mersenne premier. [En ligne] https ://fr.wikipedia.org/wiki/Nombre$_d e_M ersenne_p remier$.

[3] Savez vous factoriser a la mode de fermat. En ligne : https ://blogdemaths.wordpress.com/2014/05/18/savez-vous-factoriser-a-la-mode-de-fermat/ - Publié le 18 mai 2014.

[4] Test de primalité de miller-rabin. https ://fr.wikipedia.org/wiki/Test$_d e_p rimalit_d e_M iller-Rabin$.

[5] Test de primalité de miller-rabin. [En ligne] http ://www.bibmath.net/dico/index.php ?action=affichequoi=./m/millerrabin.html.

[6] Test de primalité de miller-rabin. [En ligne] http ://www.bibmath.net/ressources/justeunexo.php ?id=721.

[7] www.bibmath.net. http ://www.bibmath.net/crypto/index.php ?action=affichequoi=lexique/definitions.

[8] A. Bodin and F. Recher. Cryptographie. http ://exo7.emath.fr/cours/chcrypto.pdf.

[9] B. Cruise. Modern cryptography. https ://www.khanacademy.org/computing/computer-science/cryptography/modern-crypt/v/the-fundamental-theorem-of-arithmetic-1.

[10] G. Dubertret. Initiation à la cryptographie. Vuibert, 2018.

[11] L. D. Feo. Dm3 - test de miller-rabin. [En ligne] http ://defeo.lu/in420/DM3

[12] R. Guerraoui. Complexité d'un algorithme : définition et exemple. https ://www.youtube.com/watch ?v=exaHKrP6RsA.

[13] F. Holweck. Cryptographie, le système rsa. Sujet d'étude MT11, A2007, non disponible.

[14] F. Holweck. Fondements théoriques de l'informatique. cours de MT42 - UTBM, non disponible.

[15] J.-M. D. Koninck. L'héritage de fermat pour la factorisation des grands nombres. Accromath, pages Volume 14.2 été–automne 2019, 2019.

[16] H. Lehning. Tangente Hors-série n 26, Crytographie et codes secrets. POLE, 2006.

[17] L. Maranget. Factorisation par la méthode du crible quadratique. http ://-pauillac.inria.fr/ maranget/X/IF/PI/maranget/sujet.html.

[18] S. Marie-Aude. Etude de la Primalite motivee par le besoin de Nombres Premiers dans le Chiffrement RSA. PhD thesis, 2006.

[19] P. Milan. Pgcd - ppcm théorèmes de bézout et de gauss. pdf.

[20] mits. Primality test set 3 (miller-rabin). [En ligne] https ://www.geeksforgeeks.org/primality-test-set-3-miller-rabin/.

[21] P. Wassef. Arithmétique pour l'informatique. Vuibert, 2014.

Résumé

Cette Acquisition de Connaissances traite de l'étude de la cryptographie RSA, un système de chiffrement asymétrique. Elle aborde donc son fonctionnement : de la robustesse du système à l'épreuve de ceux qui veulent le casser. Cela passe par l'acquisition de notions d'arithmétiques, d'informatique théorique et bien entendu de cryptographie.

Ainsi, découvrez comment ce célèbre système de chiffrement de données permet de les sécuriser à l'heure actuelle.

Mots clés : Chiffrement asymétrique - Complexité - Algorithme - Factorisation d'entiers - Pierre de Fermat - Test de primalité - Sécurité - Arithmétiques - Nombres premiers