Repères – Comprendre RSA

nico34 – buffer

13 avril 2011

Table des matières

Chapitre 1

Introduction

RSA est un algorithme cryptographique qui fait partie de la catégorie des algortihmes à clés publiques ou systèmes asymétriques. L'acronyme RSA, créé en 1977, vient du nom de ses trois inventeurs (Rivest, Shamir et Adleman).
Ce chiffrement est le plus connu en termes d'algorithmes cryptographiques de type asymétrique. Nous le retrouvons beaucoup dans le e-commerce pour effectuer le paiement en ligne. Mais RSA est aussi utilisé dans les systèmes bancaires (cartes bleues). Ce chiffrement est également présent dans les signatures numériques de documents (clé PGP[1]) et dans les réseaux avec les connexions ssh (Security Shell) via des logiciels comme OpenSSL très connu dans les systèmes de type Unix.

Tout d'abord, il est nécessaire de préciser quelques points pour les personnes qui ne sont pas adeptes de cryptographie. Nous avons parlé de système asymétrique, c'est-à-dire de sytème mettant en oeuvre une clé publique et une clé privé qui servent respectivement à chiffrer et déchiffrer. A l'opposé, les systèmes symétriques ou à clé secrète utilisent une seule clé servant à chiffrer et à déchiffrer, ce qui, à taille de clé comparable, rend l'information plus vulnérable.

1.1 L'évolution de la cryptographie d'un point de vue non-technique

Jusqu'en 1999 la loi française interdisait toute utilisation de logiciels de chiffrement de données, ceux-ci étant considérés comme des armes de guerre de 2$^{\text{ème}}$ catégorie. Plus tard, la loi s'est assouplie grâce à Lionel Jospin en autorisant l'utilisation de la cryptographie aux particuliers notamment pour favoriser le commerce électronique[2]. Il faut savoir que depuis le 11 septembre 2001, l'utilisation de la cryptographie a fait débat après que les médias ont déclaré que certains terroristes utilisaient la cryptographie pour communiquer et organiser des attentats.

Suite à l'augmentation de l'utilisation, ou plutôt la démocratisation des technique de surveillance, les particuliers s'intéressent de plus en plus à la sécurité de

[1] Pretty Good Privacy
[2] SSL : `http://www.signelec.com/news/1030778870/index_html`

leurs données. Néanmoins, le fait que les particuliers protègent leurs données via des techniques de cryptographie n'arrange pas les gouvernements qui se doivent de contrôler toutes les informations transitant entre les pays.
Par contre, suite aux échos de l'affaire Echelon[3] et à la sensibilisation faite auprès des entreprises françaises, les industriels se sont mis à chiffrer leurs données considérées comme confidentielles pour éviter l'espionnage industriel[4].
De plus, depuis la sortie de la loi Hadopi, les Etats-Unis reprochent à la France de parer à la cybersurveillance des internautes en les incitants à chiffrer leurs données de communication[5].

En revanche le gouvernement s'autorise d'après l'article 434-15-2 du code pénal[6], de demander la clé ayant servi à chiffrer des messages susceptibles d'avoir été en relation avec un crime ou un délit.
"Est puni de trois ans d'emprisonnement et de 45 000 euros d'amende le fait, pour quiconque ayant connaissance de la convention secrète de déchiffrement d'un moyen de cryptologie susceptible d'avoir été utilisé pour préparer, faciliter ou commettre un crime ou un délit, de refuser de remettre ladite convention aux autorités judiciaires ou de la mettre en oeuvre, sur les réquisitions de ces autorités délivrées en application des titres II et III du livre Ier du code de procédure pénale.

FIG. 1.1 – Pas de Libertés sans Vie Privée

Si le refus est opposé alors que la remise ou la mise en oeuvre de la convention aurait permis d'éviter la commission d'un crime ou d'un délit ou d'en limiter les effets, la peine est portée à cinq ans d'emprisonnement et à 75 000 euros d'amende."

Par ailleurs, un organisme gouvernemental connu sous le nom de Centre Technique d'Assistance (CTA), peut, dans le cadre d'une enquète et si le mot de passe de l'expéditeur n'est pas divulgé, recourir à une cryptanalyse des données du disque dur de cette même personne. Cette organisation est protégée par le secret défense[7] ; c'est-à-dire à partir du moment où une communication en

[3] http://www.projet22.com/histoire-29/les-services-de-renseignements/article/echelon-un-reseau-d-ecoute-mondial
[4] http://video.google.fr/videoplay?docid=1319842311215805304
[5] http://bugbrother.blog.lemonde.fr/2010/10/02/frenchelon-la-dgse-est-en-1ere-division/
[6] http://www.legifrance.gouv.fr/affichCode.do?idArticle=LEGIARTI000006418637&idSectionTA=LEGISCTA000006165379&cidTexte=LEGITEXT000006070719&dateTexte=vig
[7] http://www.zdnet.fr/actualites/cryptographie-le-droit-au-secret-chamboule-par-les-decrets-de-la-lsq-212257.htm

rapport avec l'enquête a été interceptée puis rapportée au CTA, et que celle-ci présente un délit ou un crime, il ne peut pas y avoir de contre-expertise.

1.2 Notions à propos du système RSA

La première clé est la clé publique, elle est visible par tout le monde et est composée de deux nombres distincts : un module – lui-même le produit de deux nombres premiers – et un exposant public, que l'on appelera, respectivement, dans cet article, n et e. La clé publique sert à chiffrer le message.

La deuxième clé de la paire est appelée clé privée, elle sert à déchiffrer le message. Celle si est connue par le créateur de la paire de clé seulement. Elle est composée du même module n que la clé publique et d'un exposant privé que l'on nommera d. La clé privée sert à déchiffrer le message.

1.3 Où réside la force du cryptosystème RSA ?

Le chiffrement RSA est très prisé par les grandes organisations ayant besoin de chiffrer des informations de haute importance. La sécurité de ce chiffrement intéresse les mathématiciens et chercheurs depuis son invention, en 1977.

Le succès de l'algorithme RSA vient du fait qu'il n'y a aucun transfert de clé secrète entre l'expéditeur et le destinataire. Par conséquent personne d'autre, mis à part ce dernier, ne peut déchiffrer les messages sans attaque.

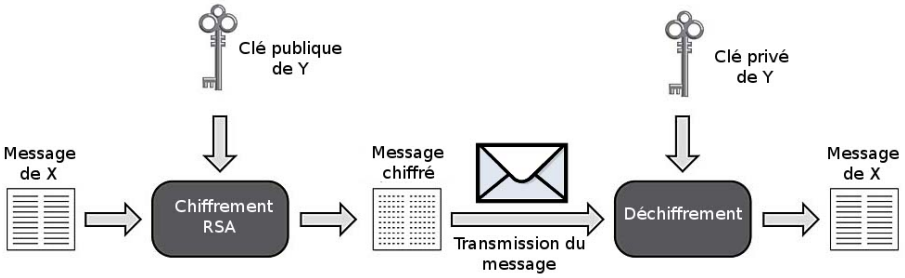

Fig. 1.2 – Principe de chiffrement RSA

RSA assure sa sécurité grâce à la difficulté mathématique de factorisation du module, c'est-à-dire la réécriture de ce module sous la forme d'un produit de deux nombres premiers p et q. Plus la clé est grande, plus les attaquants auront du mal à factoriser. Par ailleurs, pour faire face à certaines attaques, différents schéma de chiffrement existent dont RSAES-OAEP décrit dans la directive PKCS#1 v2.1 rédigée par RSA-labs.

1.4 Un peu d'histoire...

Depuis 1991, la factorisation de nombres à plus de 100 chiffres (en base 10) est désormais possible, c'est donc depuis cette année que les chercheurs s'y atèlent avec détermination, ce qui a par conséquent relancé l'intérêt pour de tels challenges.

La dernière factorisation réussie date de décembre 2009 par une clé de 768 bits. Une clé de 768 bits est un nombre à 768 chiffres en base 2, soit 232 chiffres en base 10. Lorsqu'une clé d'une certaine taille a pu être cassée, il est légitime de penser que toutes les informations ayant été chiffrées avec cette longueur de clé deviennent vulnérables. Néanmoins il faut savoir que les technologies évoluent et que certaines organisations disposent des machines, conçues pour ces calculs, de plus en plus sophistiquées.

Si l'on s'attache aux prédictions de Moore[1] qui stipulent que la quantité de transistors sur une surface donnée de silicium double tous les deux ans, augmentant ainsi la puissance de calcul, on estime que l'on sera capable de factoriser des clés de 2048 bits en un temps raisonnable aux alentours de 2020. Néanmoins, cette loi n'est plus vérifiable[8] depuis quelque temps, notamment depuis que le *parallel computing*[9] prend une place de plus en plus importante en informatique.

[8] `http://www.forbes.com/2010/04/29/moores-law-computing-processing-opinions-contributors-bill-dally.html`

[9] `https://computing.llnl.gov/tutorials/parallel_comp/`

Chapitre 2

Comprendre les fondements mathématiques de RSA

Comprendre les fondements mathématiques de RSA nécessite quelques notions de structures algébriques, d'arithmétique dans $\mathbb{Z}$ et en particulier d'arithmétique modulaire.

2.1 Notion de nombres premiers

2.1.1 Définition

On notera $\mathbb{N}$ l'ensemble des entiers naturels, c'est-à-dire $\mathbb{N} = \{0, 1, 2, ...\}$. Un nombre $a \in \mathbb{N}$ est dit premier si ses seuls diviseurs dans $\mathbb{N}$ sont 1, et lui même. Par exemple, 2, 3, 5, 7, et 11 sont premiers

2.1.2 Décomposition en facteurs premiers

Tous les éléments $n \in \mathbb{N}$, $n \geq 2$ peuvent être écrit comme un produit de nombres premiers. L'ensemble de ces nombres, appelé spectre de n et noté $\text{spec}(n)$ est fini. On peut alors écrire pour tout n de $\mathbb{N}$ tel que $n \geq 2$, $n = \prod_{p \in \text{spec}(n)} p^{v_p(n)}$.

Par exemple, $20 = 2^2 \times 5$, et le spectre $\text{spec}(20) = \{2, 5\}$.

2.2 Calcul modulaire et division euclidienne

La division euclidienne est la division dans $\mathbb{N}$ la plus intuitive qui soit – c'est d'ailleurs celle qui nous est apprise à l'école primaire. Elle consiste à décomposer un nombre en la somme du reste, et du facteur du diviseur du quotient. Mathématiquement, pour la division euclidienne de a par b – le diviseur –, on peut démontrer l'existence et l'unicité du quotient q et du reste r :

$$\exists!(q, r) \in \mathbb{Z}^2 \text{ avec } 0 \leq r < |b|,\ a = qb + r$$

La notion de reste r permet d'introduire la notion de calcul modulaire : on dit que a est **congru** à r modulo b, que l'on notera dans le reste de l'article :

$$a \equiv r \mod (b)$$

2.3 Notion de Plus Grand Commun Diviseur

La notion de Plus Grand Commun Diviseur fera ici appel à l'intuition plutôt qu'à la définition mathématique qui introduit des notions de structures algébriques (générateur d'anneau), qui sortiraient du cadre de l'article.

Le Plus Grand Commun Diviseur – noté dans la suite de l'article pgcd – du couple $(a, b) \in \mathbb{N}^2$ est le plus grand entier naturel $d \in \mathbb{N}$ qui divise à la fois a et b.

On peut trouver ce pgcd "manuellement" en décomposant en facteur premiers successivement a et b et en calculant le produit des facteurs premiers communs. Par exemple, pour $a = 12 = 3*2*2$ et $b = 30 = 3*5*2$, le Plus Grand Commun Diviseur de a et b est égal à $\text{pgcd}(a, b) = 3 * 2 = 6$.
Cet algorithme n'est pas optimal et on préférera pour ce calcul utiliser l'algorithme d'Euclide :

```
def pgcd(a, b):
    """
    Recursively calculates pgcd(a, b)
    """
    if a < b:
        return pgcd(b, a)
    else:
        if b == 0:
            return a
        else:
            return pgcd(b, (a % b))
```

2.4 Primalité de deux nombres

Deux entiers $(a, b) \in \mathbb{N}^2$ sont dits *premiers entre eux* si leur plus grand commun diviseur est 1. On écrira dans la suite de l'article $\text{pgcd}(a, b) = 1$.

2.5 Identité de Bézout

Une des conséquences de la définition mathématique (structures algébriques et algèbre dans $\mathbb{Z}$, non détaillée ici) du plus grand commun diviseur est l'identité de Bézout. En effet, $\forall (a, b) \in \mathbb{Z}^2$, $\exists (u, v) \in \mathbb{Z}^2$ tels que

$$au + bv = \text{pgcd}(a, b)$$

et on peut calculer les coefficients de Bézout et le Plus Grand Commun Diviseur en utilisant l'algorithme d'Euclide étendu :

```
def extendedEuclide(a, b):
    """
    Returns (x, y) such that Bezout's identity holds, i.e.
        a * x + b * y = GCD(a, b).
    """
    if a < b:
        return extendedEuclide(b, a)
    else:
        if a % b == 0:
            return (0, 1)
        else:
            (x, y) = extendedEuclide(b, a % b)
            return (y, x - (y * int(a / b)))
```

2.6 Indicatrice d'Euler

On note $\mathbb{Z}_n^*$ l'ensemble des entiers positifs strictement plus petits que n et premiers avec n :

$$\mathbb{Z}_n^* = \{k \in \mathbb{N} \ / \ 0 < k < n \text{ avec } \text{pgcd}(k, n) = 1\}.$$

L'**indicatrice d'Euler**, notée $\varphi(n)$, représente le **cardinal**[1] de l'ensemble $\mathbb{Z}_n^*$. On peut démontrer les propriétés suivantes :

- Soit $p \in \mathbb{N}$ un nombre premiers, alors $\varphi(p) = p - 1$
- Soient $(m, n) \in \mathbb{N}^2$ tels que m et n sont premiers entre eux, alors $\varphi(mn) = \varphi(m)\varphi(n)$.

2.7 Théorèmes divers

Les trois théorèmes qui suivent sont énoncés pour référence. Ils sont utilisés pour la démonstration du théorème RSA.

2.7.1 Théorème d'Euler

Soit $a \in {\mathbb{Z}_n}^*$, alors on a $a^{\varphi(n)} \equiv 1 \mod n$

2.7.2 Théorème de Fermat

A partir du théorème precedent, on peut montrer que :
Si p est premier, alors $\forall a \in \mathbb{Z}_p$, où $\mathbb{Z}_p = \mathbb{Z}_p^* \bigcup \{0\}$, alors

$$a^p - 1 \equiv 1 \mod p$$

[1] Le cardinal d'un ensemble fini est le nombre d'éléments que contient cet ensemble

2.7.3 Théorème des restes chinois

Soient $n_1, ..., n_k$ des entiers premiers entre eux deux à deux. Il existe x, unique modulo $N = \prod_{i=1}^{k} n_i$, solution au système d'équations modulaires :

$$x \equiv a_1 \mod n_1$$

$$...$$

$$x \equiv a_k \mod n_k$$

En posant $N_i = \frac{N}{n_i} = n_1...n_{i-1}n_{i+1}...n_k$, cette unique solution s'écrit :

$$x = \sum_{i=1}^{k} a_i N_i {N_i}^{-1} \mod n_i$$

où ${N_i}^{-1} \mod n_i$ est l'inverse modulo n_i de N_i, et dont l'existence est donnée par l'identité de Bézout :

$$\forall N_i \in {\mathbb{Z}_n}^*, \exists {N_i}^{-1} \in {\mathbb{Z}_n}^* \text{ tel que } N_i \times {N_i}^{-1} \equiv 1 \mod n$$
$$\iff \exists k \in \mathbb{Z} \text{ tel que } N_i \times {N_i}^{-1} = kn + 1$$

Chapitre 3

RSA

3.1 Choix du module et des exposants ; éléments de sécurisation

Dans cette partie, nous décrirons les méthodes permettant de choisir le module et les exposants, en prenant en considération quelques éléments de sécurisation. Ces éléments seront détaillés dans un futur article dédié aux attaques sur RSA.

3.1.1 Choix du module

Le module n est un grand nombre, produit de deux grands nombres premiers p et q, tels que $n = pq$. De la taille de ce module va dépendre la difficulté de factorisation, donc la solidité de RSA. Aujourd'hui, on recommande une taille minimale pour n de 1024 bits. Par conséquent, on choisira les deux facteurs premiers p et q de taille 512 bits. Le choix de ces nombres peut prendre énormément de temps si l'on procède itérativement en testant la primalité de chaque nombre. L'algorithme de Miller-Rabin est alors utilisé pour vérifier la primalité probabiliste d'un nombre.

Voici un exemple d'implémentation en Python de l'algorithme de Miller-Rabin :

```
import random
import sys

_mrpt_num_trials = 50 # number of bases to test

def is_probable_prime(n):
    assert n >= 2
    # special case 2
    if n == 2:
        return True
    # ensure n is odd
    if n % 2 == 0:
        return False
    # write n-1 as 2**s * d
```

```
    # repeatedly try to divide n-1 by 2
    s = 0
    d = n-1
    while True:
        quotient, remainder = divmod(d, 2)
        if remainder == 1:
            break
        s += 1
        d = quotient
    assert(2**s * d == n-1)

    # test the base a to see whether it is a witness for
        the compositeness of n
    def try_composite(a):
        if pow(a, d, n) == 1:
            return False
        for i in range(s):
            if pow(a, 2**i * d, n) == n-1:
                return False
        return True # n is definitely composite

    for i in range(_mrpt_num_trials):
        a = random.randrange(2, n)
        if try_composite(a):
            return False

    return True # no base tested showed n as composite


if __name__ == '__main__':
    print "::: Test probabiliste de primalite :::"
    print is_probable_prime(int(sys.argv[1]))
```

Après avoir généré ces deux grands nombres premiers, par une simple multiplication nous obtenons notre module n.

3.1.2 Choix de l'exposant public

L'exposant public e est choisi pour être premier avec $\varphi(n)$. Afin d'accélérer le test de primalité entre ces deux nombres, l'utilisation de l'algorithme d'euclide s'avérera judicieuse.
Néanmoins certaines personnes préfèrent utiliser une autre méthode pour choisir l'exposant public e. Cette méthode consiste à choisir e de manière à ce qu'il ne soit divisible ni par $p-1$, ni par $q-1$. Celle ci n'est pas conseillée dans la mesure où une attaque par "broadcast"[1] peut être effectuée.

Enfin, e sera choisi avec un poids de Hamming (nombre de bits à 1) relativement faible afin d'accélerer les calculs où e apparaît (chiffrement et vérification

[1] Håstad : `http://www.jscoron.fr/thesis/node6.html#SECTION03122000000000000000`

de signature). En particulier, sur les cartes bleues, e est égal à 3.

3.1.3 Choix de l'exposant privé

Enfin, l'exposant privé d sera choisi tel que $d < n$ et $ed \equiv 1 \mod \varphi(n)$.
La dernière identité nous indique qu'il existe $v \in \mathbb{Z}$ tel que $ed = v\varphi(n) + 1$, soit $ed + (-v)\varphi(n) = 1$. Nous reconnaissons l'identité de Bézout $au + bv = \text{pgcd}(a, b)$, car e et $\varphi(n)$ étant premiers entre eux, $\text{pgcd}(e, \varphi(n)) = 1$. On utilise alors l'algorithme d'Euclide étendu pour trouver les deux coefficients de Bézout, un de ces deux coefficients étant bien entendu l'exposant privé d.

Note sur certains challenges RSA

Certains challenges[2] proposent de décrypter[3] un message chiffré avec RSA en connaissant uniquement la clé publique. A partir de la clé publique, il est possible de retrouver le module n et l'exposant public e. Le module n est suffisament petit pour être factorisable ; on retrouve donc p et q et on en déduit l'indicatrice d'Euler $\varphi(n)$. Le processus pour retrouver l'exposant privé et permettre le déchiffrement du message est alors identique à celui décrit pour générer la clé privée ; l'attaquant reconnaitra l'identité de Bézout et retrouvera les coefficients - dont d - en appliquant l'algorithme d'Euclide étendu.

3.2 Applications RSA

3.2.1 Chiffrement

On notera M le message à chiffrer. Ce message doit être inférieur au module n. La partie 4 détaillera la façon dont le message est agencé pour obtenir cette condition.
Le message chiffré est noté C et est calculé de la façon suivante :

$$C \equiv M^e \mod n$$

De manière brutale – en élevant M à la puissance e – ce calcul peut être excessivement long. Dans la pratique, l'algorithme *Square And Multiply*[4] est utilisé pour améliorer la rapidité de calcul. Un exemple d'implémentation en Python de cet algorithme est décrit ci-dessous :

```
def squareAndMultiply(m, e, n):
    """
    Calculates m^e mod n
    """
    # rslt contains calculation result
    rslt = 1
```

[2] http://www.scrt.ch/insomnihack/2010/epreuves

[3] le décryptage est l'opération qui consiste à retrouver le message clair sans connaître la clé. Lorsqu'on posséde cette clé, on parle de déchiffrement.

[4] http://conferenze.dei.polimi.it/FDTC08/fdtc08-schmidt.pdf

```
    # convert exponent e to bitstream
    ebits = []
    while( e != 0 ):
        ebits.insert(0, e & 1)
        e >>= 1

    # for each bit of e (MSB to LSB)
    for ebit in ebits:
        # square...
        rslt = rslt**2 % n
        # ... and multiply (always)
        rslt = rslt * (m**ebit) % n

    return rslt
```

En particulier, on voit que dans le cadre du chiffrement avec un exposant e qui a un poids de Hamming faible (cf. 3.1.2), les calculs sont considérablement simplifiés ici.

3.2.2 Déchiffrement

On notera C le message chiffré reçu et $\hat{M}$ le message déchiffré et celui-ci est calculé de la façon suivante :

$$\hat{M} \equiv C^d \mod n$$

Pour effectuer ce calcul, l'algorithme Square And Multiply peut être utilisé ; mais comme la plupart des implémentations, telles que openssl, stockent la factorisation de n dans la clé privée, utilise le théorème des restes chinois pour optimiser le calcul.

3.2.3 Pourquoi ça marche ? – Théorème RSA

Dans la section précédente, le message déchiffré est noté $\hat{M}$ et non M pour mettre en exergue que nous n'avons pas encore démontré que l'on peut effectivement retrouver le message d'origine. Nous proposons au lecteur de faire le calcul suivant :

$$\hat{M} \equiv C^d \mod n$$

En fonction du message original :

$$\hat{M} \equiv M^{ed} \mod n$$

L'idée est donc de démontrer que $\hat{M}$ est congru à M modulo n.
Comme on a choisi d tel que le produit $ed \equiv 1 \mod \varphi(n)$, il existe un entier $k \in \mathbb{Z}$ tel que $ed = k\varphi(n) + 1$. De plus, $n = pq$ avec p et q premiers, alors $\varphi(n) = (p-1)(q-1)$; on peut alors écrire que :

$$\hat{M} \equiv M^{k(p-1)(q-1)+1} \mod n$$

M, par construction, est un entier naturel strictement plus petit que le module n. $n = pq$, avec p et q premiers. Le raisonnement suivant est effectué avec p, et doit être effectué de manière symétrique avec q :

– M et p sont premiers entre eux, alors d'après le théorème de Fermat, $M^{(p-1)} \equiv 1 \mod p$, donc en élevant à la puissance $k(q-1)$ et en multipliant par M, on obtient :

$$M^{k(p-1)(q-1)+1} \equiv M \mod p$$

En raisonnant de manière symétrique pour q :

$$M^{k(p-1)(q-1)+1} \equiv M \mod q$$

En appliquant le théorème des restes chinois, p et q étant premiers entre eux car premiers, et avec $n = pq$, il vient que :

$$M^{k(p-1)(q-1)+1} \equiv M \mod n$$

– Si M et p ne sont pas premiers entre eux, alors M est un multiple de p alors $M \equiv 0 \mod p$, et en élevant à la puissance $k(p-1)(q-1)+1$,

$$M^{k(p-1)(q-1)+1} \equiv 0 \mod p \equiv M \mod p$$

En raisonnant de manière symétrique avec q et en appliquant le théorème des restes chinois, on obtient :

$$M^{k(p-1)(q-1)+1} \equiv M \mod n$$

On a donc bien $\hat{M} \equiv M \mod n$. Comme on a pris M inférieur à n, alors $M \mod n = M$, donc le message déchiffré est bien identique au message d'origine.

Le théorème RSA s'écrit alors :

Soit $n = pq$ le produit de deux nombres premiers p et q, soit $a \in \mathbb{Z}_n$, alors

$$\forall k \in \mathbb{N}, \text{ on a } a^{k\varphi(n)+1} \equiv a \mod n$$

3.2.4 Signature électronique et RSA

RSA peut être utilisé pour définir une signature digitale. Un hash du message en clair est chiffré avec RSA "à l'envers", c'est-à-dire chiffré avec la clé privée de l'expéditeur. Ce hash chiffré est la signature du message et sera délivré avec le message :

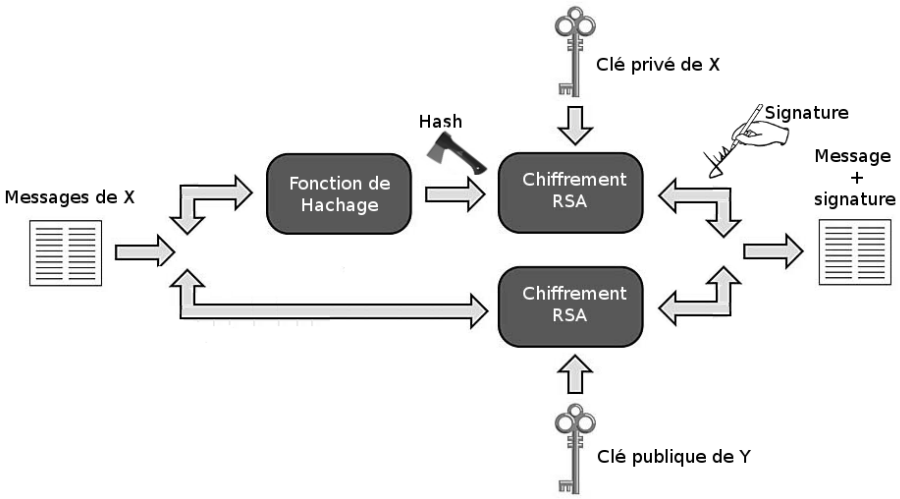

FIG. 3.1 – Emission d'un message et de sa signature

Le destinataire calcule l'empreinte du message déchiffré et "déchiffre" la signature avec la clé publique de l'émetteur. Les deux valeurs doivent être égales pour valider l'intégrité du message :

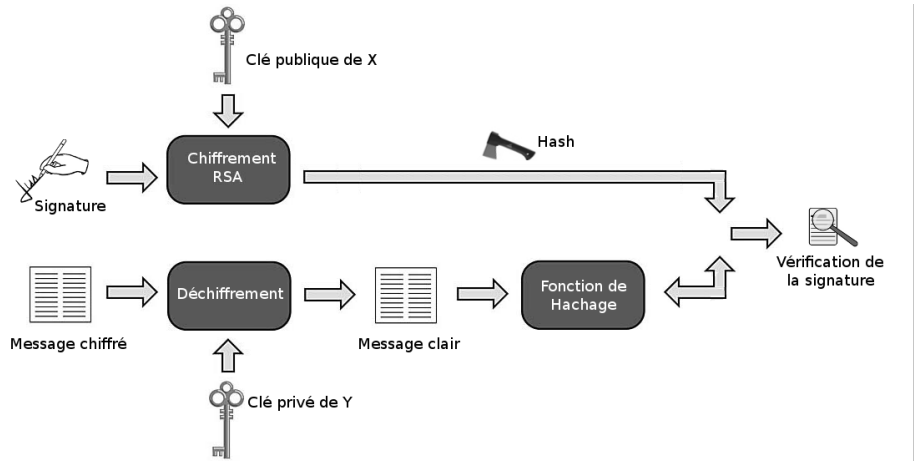

FIG. 3.2 – Réception d'un message et vérification de la signature

En particulier, ceci fonctionne car on peut démontrer mathématiquement que

$$H^{ed} \equiv H^{ed} \equiv H^{d^e} \mod n$$

où H est le résultat de la fonction de hachage.

3.3 Propriétés applicatives remarquables et comparatives de RSA

Un algorithme asymétrique tel que RSA est conforme aux quatres règles fondamentales de la cryptographie. Ces règles, que l'on dénommera par la suite comme les règles CAIN, sont :

- Confidentialité des données,
- Authentification des émetteurs et récepteurs du message, c'est-à-dire la garantie que l'interlocuteur est bien celui qu'il prétend être,
- Intégrité des infomations, c'est-à-dire la garantie que le message n'a pas été altéré de manière volontaire ou involontaire lors du transit dans le canal de communication.
- Non-répudiation des informations, afin de protéger l'échange entre les interlocuteurs et non plus vis-à-vis d'un tiers. La non-répudiation garantie que l'un des interlocuteurs ne peut pas prétendre qu'il n'a pas envoyé le message ou que le message a été mal compris, voire altéré.

La confidentialité, but premier de tout algorithme de chiffrement, est ici renforcée par l'asymétrie de RSA. En effet, dans le cas d'un algorithme dit symétrique, l'émetteur et le récepteur doivent partager une même clé secréte pour être capable respectivement de chiffrer et déchiffrer le message. Se pose alors le probleme de l'échange initial de cette clé, car il faut trouver un vecteur d'échange garantissant lui aussi la confidentialité. Les algorithmes asymétriques dont fait partie RSA, résolvent cette problématique. La clé utilisée pour le chiffrement est publique ; n'importe qui peut obtenir cette clé pour chiffrer et transmettre une information au destinataire. Le déchiffrement se fait par une clé secrète connue seulement du destinataire.

Dans le cas d'un algorithme symétrique, l'intégrité et l'authentification sont généralement gérées par une fonction de hachage à sens unique paramétrée avec une clé secrète : on parle de Message Authentification Code (MAC). L'intégritée est garantie par la fonction de hachage tandis que l'authentification est garantie par le chiffrement de l'empreinte par la clé secrète. En effet, dans le cas d'un algorithme symétrique, les interlocuteurs, seuls à connaître la clé, sont aussi les seuls à pouvoir déchiffrer l'empreinte. Ceci est une condition suffisante pour l'authentification des deux parties.
Cependant, comme le secret de la clé est partagé entre les interlocuteurs, la non-répudiation n'est pas garantie. Un des deux interlocuteurs peut nier avoir émis le message car le doute sur l'identité de l'émetteur demeure, chacun des interlocuteurs ayant pu émettre le message.
A l'inverse, RSA permet de garantir l'intégrité, l'authentification ET la non-répudiation grâce au système de signature décrit en 3.2.4. L'intégrité du message est garantie encore une fois grâce à la fonction de hachage et la non-répudiation est induite par le fait que seul l'émetteur connaît la clé secrète, donc est le seul à pouvoir signer le document.

RSA souffre cependant d'un défaut : le chiffrement et le déchiffrement sont relativement longs. Le protocole SSL (Secure Socket Layer), développé à l'origine par NetScape pour assurer la sécurité des communications sur Internet, implémente une solution originale qui utilise les propriétés CAIN de RSA et la

rapidité relative des algorithmes symétriques. Prenons l'exemple (simplifié) d'un site Web communicant par protocole HTTPS. Un client émet une requête vers ce serveur, lequel répond en envoyant un 'certificat'. Le client va alors authentifier le serveur en vérifiant la validité de ce certificat, puis va récupérer la clé publique dudit serveur. Le client va alors génerer une master key, qu'il va chiffrer avec la clé publique et envoyer au serveur. Le serveur va alors déchiffrer la master key et renvoyer un message authentifié avec cette dernière. Les communications suivantes vont alors toutes être chiffrées avec cette master key, en utilisant donc un algorithme symétrique tel que RC2, RC4, IDEA, DES ou DES3.

3.4 Exemple d'application

Afin d'illustrer toutes ces notions mathématiques et théoriques, nous allons, sous forme d'expemple, chiffrer un message avec RSA en utilisant le mode ECB (Electronic Code Book). Ce mode est cependant déconseillé car vulnérable à des attaques par analyse fréquentielle, mais donné ici pour des raisons de clarté. Pour résumer très rapidement la situation nous devons choisir deux grands nombres premiers p et q et tester leur primalité avec l'algorithme de Miller-Rabin (cf. 3.1.1). Prenons $p = 572417$ et $q = 774853$. Pour trouver n il suffit simplement du multiplier les deux.

$$p \times q = n$$

$$572417 \times 774853 = 443539029701$$

Calculons maintenant l'indicatrice d'Euler $\varphi(n)$:

$$\varphi(n) = (p-1) \times (q-1)$$

$$572416 \times 774852 = 443537682432$$

Nous avons presque terminé de créer la clé publique, il ne nous reste plus qu'à trouver l'exposant public e tel que $\text{pgcd}(e, \varphi(n)) = 1$. En testant plusieurs valeurs avec l'algorithme d'Euclide, on trouve très rapidement une valeur possible $e = 5$ car en effet $\text{pgcd}(5, \varphi(n)) = 1$

Nous venons de créer le couple (e, n) de la clé publique : $(5, 443539029701)$.

Afin de finaliser la génération, il nous reste à créer l'exposant privé d. On sait que $ed \equiv 1 \mod \varphi(n)$, donc $ed + k\varphi(n) = 1$. On reconnaît l'identite de Bézout et on peut calculer d et k en utilisant utilisant l'algorithme d'Euclide étendu présenté en 2.5 :

$$e \times d + phi(n) \times b = \text{pgcd}(e, \varphi(n))$$

$$5 \times 177415072973 + 443537682432 \times -2 = 1$$

Notre clé privé (d, n) sera donc : $(177415072973, 443539029701)$

Maintenant que nous avons généré nos clés, tentons de chiffré ce message "p0n3y" ☺. Le chiffrement s'opère ainsi : $\text{ValeurChiffrée} = \text{ValeurAscii}^e \mod n$

Tout d'abord il va nous falloir une bonne mémoire ou une table ascii(ref) sous la main. C'est parti : $112^5 \mod 443539029701 = 17623416832$ $48^5 \mod 443539029701 = 254803968$ $110^5 \mod 443539029701 = 16105100000$ $51^5 \mod 443539029701 = 345025251$ $121^5 \mod 443539029701 = 25937424601$

Le déchiffrement s'opère ainsi : ValeurDéchiffrée = ValeurChiffrée^d mod n.
Si le calcul à la puissance 5 s'effectue aisément, il n'en est pas de même lorsque l'exposant est important. Dans notre exemple, la clé privée est égale à 177415072973 ; dans ce cas, afin d'optimiser les calculs, on utilise l'algorithme "Square and Multiply" décrit en 3.2.1 : squareAndMultiply(17623416832, 177415072973, 443539029701) = 112L C'est correct, on retrouve notre message original.

Chapitre 4

Chiffrement

En supposant qu'un message soit chiffré de la façon la plus simple qui soit, c'est-à-dire en mode ECB (Electronic Code Book), où chaque bloc de caractères est codé indépendamment, alors ce message est vulnérable à une attaque par analyse statistique du texte chiffré. L'attaquant devra déterminer les fréquences d'apparition des symboles pour ensuite les comparer aux fréquences types des différentes langues.

Après avoir généré ses clés, il faut choisir un schéma de chiffrement sûr pour éviter le décryptage[1]. Il a été démontré que RSA couplé au schéma de chiffrement OAEP de Bellare et Rogaway est sémantiquement sûr contre les attaques CCA1 et CCA2[2].

4.1 Fonction du schéma de chiffrement OAEP

RSAES-OEAP est un schéma de chiffrement faisant appel à un nombre généré aléatoirement r, deux fonctions de hachage G (fonction `pkcs_mgf1()`) et H.

$$\text{OAEP}(M) = [(M \oplus G(r))||(r \oplus H(M \oplus G(r)))]$$

Soient $s = M \oplus G(r)$ et $t = r \oplus H(s)$. Ce sera le couple (t, s), sous forme de nombre, qui servira d'entrée au chiffrement RSA.
Afin de retrouver notre message M, calculons successivement $H(s)$, $r = H(s) \oplus t$, $G(r)$, et on obtient $M = s \oplus G(r)$.

[1] le décryptage est l'opération qui consiste à retrouver le message clair sans connaître la clé. Lorsqu'on possède cette clé, on parle de déchiffrement.

[2] PKCS 1 v2.1 RSA-labs

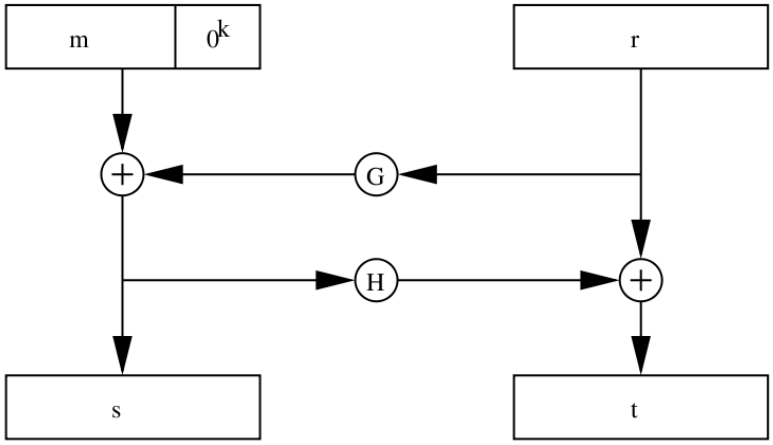

Fig. 4.1 – RSAES-OAEP

Afin de vous présenter le système RSAES-OAEP, nous nous servirons du célèbre logiciel libre Scapy[3] écrit en Python qui permet de manipuler des paquets réseau.

4.2 Implémentation de OAEP dans Scapy

Scapy a très bien implémenté les fonctions de la PKCS rédigée par RSA-labs. Dans le dossier `scapy/crypto`, vous y trouverez un fichier nommé `cert.py` d'environ 2500 lignes et comportant plusieurs classes selon vos besoins.
Dans cette partie nous utiliserons la classe `Key` pour chiffrer mais aussi déchiffrer. Il faut savoir que si vous voulez utiliser cette classe, il vous faudra fournir en entrée un fichier structuré de la même façon que la sortie du logiciel openssl :

```
$ openssl genrsa
```

Cette structure est visible grâce à la commande :

```
$ openssl rsa -text -in key.PEM
```

et se présente comme ceci :

```
-----BEGIN RSA PRIVATE KEY-----
base64(modulus,
       publicExponent,
       privateExponent,
       prime1,
       prime2,
       exponent1,
```

[3]Ref

```
        exponent2,
        coefficient)
-----END RSA PRIVATE KEY-----
```

Les différents éléments se caractérisent par :
- `modulus`, module n
- `privateExponent`, exposant privé d

Les suivants sont utilisés afin d'augmenter la rapidité des calculs :
- `prime1`, nombre premier p (premier facteur de n)
- `prime2`, nombre premier q (deuxième facteur de n)
- `exponent1`, $d \mod (p-1)$
- `exponent2`, $d \mod (q-1)$
- `coefficient`, coefficient des restes chinois $q^{-1} \mod (p)$

La classe `Key` utilise d'autres classes qui sont `OSSLHelper`, `_DecryptAndSignMethods`, `_EncryptAndVerify`. Pour chiffrer notre message, nous utiliserons la fonction `encrypt` de la classe `_EncryptAndVerify`. Dans cette partie nous utiliserons un chiffrement de bloc de type OAEP. En spécifiant l'argument t (`t="oaep"`), notre fonction encrypt() fera appel à une nouvelle fonction : `_rsaes_oaep_encrypt` qui reprendra les arguments de la précédente.

```
def encrypt(self, m, t=None, h=None, mgf=None, L=None)
```

Ses arguments sont :
- `m`, le message à chiffrer
- `h`, le type de hash à utiliser (md2, md4, md5, tls, sha1, sha224, sha256, sha384),
- `mgf`, une fonction de génération de masque,
- et `L` qui est un message associatif qui devra être spécifié pour vérifier et autoriser le déchiffrement.

Voici un exemple très simple de code python faisant appel aux fonctions de chiffrement.

```
## This script use crypto's scapy class
## Paper RSA

from cert import * # Import du fichier cert.py

# define message to cipher
m = "Exemple de message "

# Instanciate the Key class with PEM from a file
key = Key('/home/nico/Bureau/key.PEM')

# Cipher message
cipher = key.encrypt(m, t="oaep", h="sha1", mgf=None,
    L="secur")
print "Message chiffre : %s" % (cipher)

# Decipher message
```

```
decipher = key.decrypt(cipher, t="oaep", h="sha1",
    mgf=None, L="secur")
print "Message dechiffre : %s" % (decipher)
```

Chapitre 5

Conclusion

Pas d'affolement, RSA, algorithme utilisé exhaustivement dans de nombreuses applications, reste un algorihtme sémantiquement sûr, sous réserve de bien choisir le module et les exposants lors de la génération des clés et être couplé à un bon schéma de chiffrement.
Après ce tour d'horizon des fondements de RSA - premier article d'une série de deux - nous espérons que vous aurez pu affiner vos connaisssances théoriques et techniques sur cet algorithme. Cette présentation des aspects historiques, légaux, mathématiques, de choix et d'implémentation du module, des exposants, ainsi que des méthodes de chiffrement, permettra d'appréhender par la suite les différentes attaques possibles.
Nous resterons à votre écoute pour toutes suggestions, questions et même insultes ☺à propos de cet article.