LA CRYPTOGRAPHIE ASYMÉTRIQUE AVEC RSA

Vayel, Dominus Carnufex

31 octobre 2015

LA CRYPTOGRAPHIE ASYMÉTRIQUE AVEC RSA

Introduction

Parmi tous les systèmes cryptographiques disponibles à l'heure actuelle, RSA est considéré comme un des plus solides, si ce n'est *le* plus solide. Après presque trente ans de vie et d'attaques nombreuses mettant en jeu ce que la technologie fait de meilleur, il est encore jugé assez robuste pour protéger les échanges bancaires et autres données critiques du monde civil. Alors, qu'est-ce qui fait sa force ?

Avec vous, je vais aborder les théories mathématiques et cryptographiques qui constituent les fondements de ce système cryptographique et l'ont rendu possible. Il me sera alors possible de vous présenter l'algorithme en lui-même, comment on chiffre et comment on déchiffre. Mais cela ne suffit pas : aussi chaud que soit un manteau, vous mourrez de froid si vous le posez sur votre tête. C'est pourquoi je vous expliquerai comment il faut s'y prendre pour *bien* utiliser le système cryptographique RSA et bénéficier de toute sa robustesse. Enfin, pour nos amis programmeurs, un dernier chapitre permettra de voir, sans entrer dans les détails, comment on peut implémenter RSA pour l'utiliser dans un logiciel.

Pour suivre ce tutoriel, il vous faudra les outils suivants :

- des connaissances mathématiques de Seconde environ ; toutes les notions plus complexes seront expliquées de manière aussi détaillée que possible ;
- un papier et d'un crayon pour réaliser les quelques exercices présentés au long du tuto ; une calculette peut aider les plus flemmards ;
- absolument aucune connaissance en informatique, si vous ne souhaitez pas lire le dernier chapitre ; sinon, de simples connaissances théoriques sont suffisantes, puisque l'on parle d'algorithmique et non d'un langage en particulier ;
- un cerveau qui vous appartient, pas de zombi ici, ouste !

Un peu d’histoire et beaucoup de mathématiques

Ce que l’on appelle RSA est un algorithme de chiffrement et déchiffrement. Et la cryptographie, c’est avant tout des maths, beaucoup de maths. Dans cette partie, nous allons voir les fondements théoriques et mathématiques qui ont permis la création de l’algorithme RSA. N’ayez crainte : si nous allons être bien obligés d’utiliser des termes barbares, comme « chiffrement asymétrique » ou « PGCD », nous ferons en sorte de les expliquer le plus simplement possible.

Rendons à César ce qui est à César

Le système cryptographique RSA a été inventé en 1977, et publié en 1978, par trois mathématiciens qui travaillaient alors au MIT : Ronald Rivest, Adi Shamir et Leonard Adleman. Comme vous pouvez le constater, le nom de l’algorithme est simplement tiré des initiales de leurs noms de famille. La grande réussite de leur travail en commun est sans doute due au fait que leurs compétences se complètent.

Rivest est un grand spécialiste de la cryptographie, la discipline qui cherche à **protéger** les messages d’une lecture inopportune : il est notamment à l’origine de l’algorithme MD5, dont vous avez sans doute entendu parler.

Shamir est au contraire un spécialiste de la cryptanalyse, dont le but est de **casser** les protections mises en place par la cryptographie : il est notamment connu pour avoir cassé en 1982 le cryptosystème de Merkle-Hellman, un algorithme proche de RSA mais reposant sur une base mathématique un peu différente.

Adleman, quant à lui, représentait le grain de folie du groupe : il travaillait à l’époque sur la bio-informatique, en gros, des ordinateurs utilisant de l’ADN pour faire leurs calculs. Ça vous place le personnage.

Figure : *Voici une photo des trois mathématiciens en question.*

Le système cryptographique auquel ils ont abouti n'est pas sorti de leur cerveau à partir de rien, telle une Athéna moderne : il repose sur les dernières avancées de l'époque en matière de cryptographie, auxquelles ils ont adjoint un outil mathématique déjà fort ancien. C'est ce que nous allons voir à présent.

Faire du neuf…

Si je vous parle de codes secrets et de messages chiffrés, vous allez penser immédiatement « cryptographie symétrique ».

[[q]] | Non. Jamais entendu ce mot-là.

La formulation vous est, en effet, sans doute inconnue, mais vous connaissez parfaitement le concept. Un système cryptographique est dit **symétrique** si toute la solidité du chiffrement repose sur un secret — on l'appelle généralement « clé » — qui doit être connu *à la fois* de l'envoyeur et du récipiendaire.

Par exemple, le chiffre dit « de César » est un système symétrique. Il s'agit de décaler chaque lettre du message d'un certain nombre de rangs dans l'alphabet : avec un décalage de trois rangs, *A* devient *D*, *B* devient *E*, *Z* devient *C*, etc. Ainsi, le message « trololo » devient « wururor ». Il suffit au récipiendaire de décaler les lettres du message chiffré de trois rangs dans l'autre sens pour retrouver le message d'origine. Et dans l'affaire, la clé n'est autre que le nombre de rangs dont il faut décaler les lettres : cette information doit être connue des seuls interlocuteurs, mais les deux interlocuteurs doivent la connaître.

Petite pause vocabulaire

Il y a des concepts qui reviennent souvent quand on parle de cryptographie, et on a fini par se mettre d'accord sur un vocabulaire commun pour en parler. Afin de pouvoir l'utiliser dans la suite du tutoriel, il me faut vous le présenter.

Un **algorithme de chiffrement** est un ensemble d'opérations, généralement mathématiques, à effectuer sur le message que l'on veut protéger, afin de le rendre en théorie incompréhensible par ceux auxquels il n'est pas destiné. Il vient toujours avec un **algorithme de déchiffrement**, qui permet au récipiendaire du message de retrouver le message d'origine. Parfois, ces deux algorithmes sont exactement identiques, comme dans le cas du ROT-13, un cas particulier du chiffre de César.

Tant qu'on y est, le message d'origine, avant toute forme de chiffrement, est appelé **message en clair**. Une fois qu'un algorithme de chiffrement lui a été appliqué, on l'appelle **message chiffré**.

Un algorithme de chiffrement/déchiffrement décrit une suite d'opérations qui est la même pour tout le monde. Or, si tout le monde utilise exactement la même méthode pour protéger ses données, tout le monde peut lire les messages des autres, puisque l'algorithme de déchiffrement est le même que pour ses propres messages. C'est pourquoi les algorithmes laissent toujours un tout petit espace pour que l'utilisateur puisse les personnaliser : en gardant secret cet élément de personnalisation, que l'on appelle une (ou des) **clé**, la protection est assurée. Il s'agit généralement d'un mot ou d'une phrase de passe.

Pour désigner les personnes impliquées dans une opération de chiffrement, on pourrait parler de personne A, personne B, personne C, etc. Mais cette solution serait assez déshumanisée. C'est pourquoi, dans les exemples, on a pris l'habitude de parler d'un couple d'amoureux appelés Alice et Bob, qui tentent de s'écrire sans être dérangés par la vilaine Ève, qui leur veut du mal sans qu'on sache trop pourquoi. Pour ma part, je ferai preuve d'un peu d'originalité, en vous parlant du sympathique **Adalbéron**, qui désire envoyer quelque missive enflammée à sa douce **Brunehaut**, sans que le perfide **Ébroïn** ne vienne y mettre son groin.

Adalbéron, Brunehaut et Ébroïn

Le contexte

Revenons à nos moutons électriques

Pour faire une analogie avec le monde physique, Adalbéron utilise un système symétrique lorsqu'il enferme sa missive dans un coffre solide, fermé par un lourd cadenas. Il ne lui reste plus qu'à envoyer le coffre à Brunehaut, qui l'ouvrira avec son propre exemplaire de la clef, et Ébroïn se trouvera bien embêté.

Il y a juste une faille. Énorme. C'est qu'il faut faire parvenir la clé à Brunehaut. Évidemment, dans le cas d'un vrai couple d'amoureux, les deux tourtereaux se remettent la clé au cours d'un quelconque rendez-vous où ils peuvent se voir physiquement. Mais dans la vraie vie, une telle solution n'est pas toujours possible : connaissez-vous personnellement le directeur du service informatique de votre site de vente en ligne préféré ? Non, bien sûr. Alors comment peut-il vous faire parvenir la clé à utiliser pour chiffrer votre numéro de carte bancaire, sans qu'un individu mal intentionné ne l'intercepte ?

C'est tout un problème.

Et une solution a finalement été trouvée en 1976 par Whitfield Diffie et Martin Hellman : c'est ce que l'on appelle la **cryptographie asymétrique**. Le principe est double.

Tout d'abord, il n'existe pas une clé, mais deux. La première est appelée **clé privée**, parce qu'elle ne doit être diffusée à personne, pas même à son interlocuteur. La seconde est la **clé publique**, parce que tout le monde peut la connaître : ce n'est pas gênant et même plutôt recommandé. Toute la subtilité réside dans le fait que la clé publique peut être facilement générée à partir de la clé privée, mais que la clé privée est presque impossible à retrouver à partir de la clé publique.

Pour reprendre l'analogie du coffre-fort, dans un système asymétrique, Brunehaut ne donne pas une clé à Adalbéron, elle lui donne un cadenas ouvert dont elle seule a la clé. Adalbéron met sa lettre dans un coffre, le ferme à l'aide du cadenas fourni par Brunehaut et, à ce stade, même lui ne peut plus ouvrir le coffre pour accéder à la lettre : seule Brunehaut le peut. Ainsi, plus besoin de transmettre une clé, Brunehaut la garde près d'elle. Et Ébroïn peut bien intercepter le coffre ouvert : la lettre ne s'y trouve pas encore.

Ensuite, l'algorithme de chiffrement lui-même consiste à appliquer une fonction mathématique d'un genre particulier, appelée **fonction à sens unique et porte dérobée**. Il s'agit d'une fonction dont il est facile de calculer l'image d'un antécédent donné, mais très difficile de calculer l'antécédent d'une image donnée, sauf si l'on possède une certaine information qui permet d'appliquer une nouvelle fonction à l'image, dont le résultat sera l'antécédent.

[[information]] | Si ces notions mathématiques vous sont inconnues, faites donc un tour ici. ^^

[[q]] | Ça pique un peu les yeux, ton explication, là…

Je sais. Désolé. C'était le seul moyen de rester général dans mon explication. Car, vous vous en doutez sûrement, une fonction qui remplit ces conditions, cela ne se trouve pas sous le sabot d'un cheval. D'ailleurs, lorsque Diffie et Hellman ont présenté pour la première fois le concept de cryptographie asymétrique, il ne savaient pas encore comment le mettre en pratique.

C’est là qu’interviennent Rivest, Shamir et Adleman. Ils sont les premiers à avoir proposé un cas particulier viable de la théorie générale exposée ci-dessus, qui utilise des concepts mathématiques bien connus et maîtrisés.

…avec du vieux

Nombres premiers

Le premier de ces concepts qu’il nous faut aborder est celui de **primalité**. Il est très ancien, puisqu’on en trouve les premières traces certaines dans les *Éléments* d’Euclide, un traité de mathématiques grecques du IV^{e} siècle avant notre ère.

Euclide ne travaillait guère qu’avec des entiers (dont l’ensemble est noté $\mathbb Z$). Aussi, la **division euclidienne** ne donne-t-elle pas de résultat à virgule, mais un quotient entier et éventuellement un reste, entier aussi. Par exemple, 5 divisé par 2, ne donne par 2,5 mais un quotient 2 et un reste 1. Formellement, la division euclidienne se représente ainsi, avec *q* représentant le quotient et *r* le reste (*r* < *b*, sinon ça n’a aucun intérêt).

$$\frac {a}{b} \Rightarrow a = b \cdot q + r$$

Lorsque le reste est nul, on dit que le *a* est **divisible** par *b*, ce qui se note *a* | *b*. L’ensemble des nombres entiers, positifs ou négatifs, par lesquels un entier *a* est divisible est appelé **l’ensemble des diviseurs de *a***, et il se note $\mathcal D(a)$. Par exemple, $\mathcal D(10) = \{-10, -5, -2, -1, 1, 2, 5, 10\}$.

Euclide, en son temps, avait déjà remarqué que certains nombres entiers ne faisaient rien comme tout le monde : en effet, ils n’ont que deux diviseurs positifs, 1 et eux-mêmes. On les appelle des **nombres premiers**. Par exemple, 2, 3, 5, 7, 11, ou encore 53 et 97 sont des nombres premiers. Mais pas 1. Ben non, il n’a pas deux diviseurs positifs, il n’en a qu’un. Et ces nombres premiers sont un des plus grands mystères des mathématiques, qui n’en finissent plus de donner du travail aux mathématiciens depuis plus de deux mille ans qu’on connaît leur existence : la conjecture d’Erdős-Straus, énoncée en 1948, n’a toujours pas été démontrée ni invalidée parce que certains cas particuliers impliquant des nombres premiers résistent encore et toujours. Et ce n’est qu’un exemple parmi tant d’autres : il existe une infinité de nombres premiers, et chacun d’entre eux a sa propre manière de nous casser les pieds.

Cependant, ces nombres premiers sont également à l’origine d’outils mathématiques très intéressants. L’exemple le plus simple, déjà connu d’Euclide, est la **décomposition en facteurs premiers** : tout nombre entier, s’il n’est pas lui-même premier, est le produit de la multiplication de deux ou plus nombres premiers. Par exemple, $10 = 2 \times 5$, ou encore, $199121 = 13 \times 17^2 \times 53$. Et ce qui est encore mieux, c’est que cette décomposition est unique : il n’existe qu’une seule liste de **facteurs premiers** qui permette, en les multipliant, d’obtenir un entier donné.

Congruences

Soient trois entiers *a*, *b* et *n*, ce dernier étant différent de 0. On dit que *a* est **congru** à *b* **modulo** *n*, ce qui s’écrit $a \equiv b\ [n]$, lorsque le reste de la division euclidienne de *a* par *n* est le même que celui de la

division euclidienne de b par n : $17 \equiv 13\ [4]$, puisque $17 = 4 \times 4 + 1$ et $13 = 3 \times 4 + 1$.

Souvent, on s'intéresse surtout au plus petit b avec lequel a est congru modulo n : dans notre exemple, il s'agissait de 1. Par abus de langage, on dira que a modulo n (noté $a[n]$) est égal à ce plus petit b possible : $17[4] = 1$.

Il existe quelques propriétés intéressantes avec les congruences, dont trois nous seront utiles par la suite.

$$\forall k \in \mathbb N, a \equiv b \ [n] \Rightarrow k \cdot a \equiv k \cdot b \ [n]$$

$$\forall k \in \mathbb N, a \equiv b \ [n] \Rightarrow a^k \equiv b^k \ [n]$$

$a \equiv b\ [n] \Leftrightarrow n \mid a - b$

Nombres premiers entre eux

[[a]] | Prenez garde à ne pas confondre ce concept avec celui assez proche de nombres premiers *tout court*.

Prenons deux entiers, a et b. Pour chacun d'entre eux, il est possible d'établir l'ensemble de leurs diviseurs $\mathcal D(a)$ et $\mathcal D(b)$. Ces deux ensembles peuvent avoir des éléments en commun : c'est l'ensemble des **diviseurs communs** à a et à b, que l'on note $\mathcal D(a, b)$. Par exemple, $\mathcal D(10) = \{-10, -5, -2, -1, 1, 2, 5, 10\}$ et $\mathcal D(15) = \{-15, -5, -3, -1, 1, 3, 5, 15\}$, donc $\mathcal D(10, 15) = \{-5, -1, 1, 5\}$.

On appelle **plus grand diviseur commun à *a* et à *b*** ou $pgcd(a, b)$ le plus grand des éléments de $\mathcal D(a, b)$. Et lorsque le PGCD de deux nombres vaut 1, on dit que ces deux nombres sont **premiers entre eux**.

Il existe un moyen simple de déterminer le PGCD de deux nombres : on le nomme algorithme d'Euclide, parce que là encore, c'est le mathématicien grec qui l'a exposé le premier. Il repose sur le résultat suivant.

$$pgcd(a, b) = pgcd(b, a[b])$$

L'algorithme consiste à appliquer cette formule récursivement jusqu'à ce que $a[b] = 0$. Par exemple, pour 42 et 30, les itérations successives de l'algorithme sont les suivantes.

$pgcd(42, 30) = pgcd(30, 12)$puisque $42 = 1 \times 30 + 12$

$pgcd(30, 12) = pgcd(12, 6)$puisque $30 = 2 \times 12 + 6$

$pgcd(12, 6) = pgcd(6, 0)$puisque $12 = 2 \times 6$

Donc $pgcd(42, 30) = 6$

C'est aussi simple que cela.

Petit théorème de Fermat

On termine avec ce petit théorème, indispensable pour comprendre pourquoi l'algorithme RSA fonctionne bel et bien. Il a été exposé par le mathématicien français Pierre de Fermat en 1640 mais ne sera réellement démontré qu'en 1743 par Leonhard Euler, puis à nouveau d'une manière plus simple par Friedrich Gauss en 1801. Il est relativement simple.

Soient un nombre premier p et un entier a non divisible par p.

$a^{p-1} \equiv 1 \ [p]$

En outre, les trois formulations suivantes sont parfaitement équivalentes et parfois plus pratiques d'utilisation.

$p \mid a^{p-1} - 1$

$p \mid a^p - a$

$a^p \equiv a \ [p]$

Si vous n'avez pas bien compris l'un des concepts exposés dans ce chapitre, retournez lire : ils seront indispensables pour comprendre l'algorithme lui-même, que nous allons vous présenter dans un instant.

L’algorithme proprement dit

Maintenant que vous possédez les bases, vous allez pouvoir comprendre comment fonctionne l’algorithme RSA. Celui-ci se divise en trois parties. Il faut d’abord générer une paire de clés, avec une clé privée et la clé publique qui lui correspond. Ensuite, on étudiera l’algorithme permettant de chiffrer des données à l’aide de la clé publique, suivi de l’algorithme permettant de les déchiffrer à l’aide de la clé privée. Enfin, nous vous présenterons la démonstration mathématique qui explique pourquoi le message déchiffré est bien le message en clair de départ.

Une grosse paire de clés

Dans le premier chapitre, nous avons vu que dans un système asymétrique, la clé publique est générée à partir de la clé privée à l’aide d’une fonction mathématique simple mais difficilement réversible. Nos trois larrons ont utilisé la plus simple de toutes : la multiplication. En effet, il est très facile de multiplier deux nombres. En revanche, si je vous donne un nombre entier de grande taille, par exemple 332403452489, vous serez bien en peine de retrouver qu’il est le produit de 738653 et de 450013. Avec des nombres suffisamment grands, la factorisation demande tellement de temps que tous les ordinateurs du monde réunis n’y parviendraient pas même en un million d’années.

Cela étant, on ne peut pas choisir n’importe quelle paire de nombres pour notre affaire. En effet, un nombre peut avoir plusieurs factorisations valides : $20 = 4 \times 5 = 2 \times 10$. Le seul moyen de garantir qu’il n’existe qu’une seule factorisation valide, c’est de multiplier deux nombres premiers. Nous y voilà.

[[information]] | Rappelez-vous : la décomposition d’un entier strictement supérieur à 1 en nombres premiers est unique. En en multipliant deux, on a directement la décomposition et on peut être sûr qu’il n’y en a pas d’autres.

Soient deux nombres premiers P et Q *différents*, par exemple 53 et 97. On définit les nombres $N = P \times Q$ et $M = (P - 1)(Q - 1)$, dans notre exemple $N = 5141$ et $M = 4992$.

[[i]] | Pour votre culture personnelle, sachez que M s’appelle l’indicatrice d’Euler. Ne me remerciez pas pour cette occasion de briller en soirée.

Pour terminer la génération de nos clés, il nous faut choisir un nombre E qui soit *à la fois* inférieur à M et premier avec ce même nombre. Ici, nous pouvons choisir 7.

Voilà, c’est terminé pour la génération des clés ! La clé publique est composée du couple (N, E) et la clé privée est composée du triplet (P, Q, E).

Vous allez me dire que dans notre exemple, il est relativement rapide de retrouver P et Q à partir de N. C’est parfaitement exact. C’est pourquoi, en situation réelle, on utilisera plutôt le produit suivant :

3 107 418 240 490 043 721 350 750 035 888 567 930 037 346 022 842 727 545 720 161 948 823 206 440 518 081 504 556 346 829 671 723 286 782 437 916 272 838 033 415 471 073 108 501 919 548 529

007 337 724 822 783 525 742 386 454 014 691 736 602 477 652 346 609

= 1 634 733 645 809 253 848 443 133 883 865 090 859 841 783 670 033 092 312 181 110 852 389 333 100 104 508 151 212 118 167 511 579

× 1 900 871 281 664 822 113 126 851 573 935 413 975 471 896 789 968 515 493 666 638 539 088 027 103 802 104 498 957 191 261 465 571

Mais pour donner un exemple trivial, ce n'est pas hyper-simple à utiliser.

Ainchi ffrons, ffrons, ffrons

L'algorithme RSA travaille exclusivement avec des nombres. Aussi, avant de chiffrer un message composé de texte, il faut transformer ce texte en nombres de manière non ambiguë. Cette étape ne pose pas de difficulté : tous les ordinateurs stockent le texte sous forme de nombres et on pourra utiliser le standard Unicode, qui nous offre une base bien pratique. Supposons que notre brave Adalbéron veuille envoyer le message « Douce Brunehaut, souffrez de recevoir mon ardent amour. » à sa bien aimée Brunehaut. Une fois converti en Unicode, on obtient la représentation suivante.

68 111 117 99 101 32 66 114 117 110 101 104 97 117 116 44 32 115 111 117 102 102 114 101 122 32 100 101 32 114 101 99 101 118 111 105 114 32 109 111 110 32 97 114 100 101 110 116 32 97 109 111 117 114 46

La fonction de chiffrement à l'aide d'une clé (N, E) est la suivante.

$f : x \mapsto x^E\ [N]$

Cela signifie que pour obtenir votre code chiffré, vous devez élever votre code en clair à la puissance E et lui appliquer le modulo N : on dit que **à tout antécédent** x, la fonction f **associe l'image** $x^E\ [N]$.

[[information]] | Si vous n'êtes pas à l'aise avec les fonctions, faites donc un tour ici !

D'emblée, on comprend qu'il n'est possible de chiffrer que des nombres strictement inférieurs à N. En effet, $(N + 1)[N] = 1$: si l'on n'y prend pas garde, les nombres supérieurs à N donneront le même résultat que certains nombres inférieurs, et il sera impossible de les distinguer lors du déchiffrement. Dans l'exemple qui nous intéresse, nous allons réutiliser la clé $(5141, 7)$, donc ce problème ne se posera pas. Voyons ce que donne la fonction appliquée à notre message.

254 1858 1774 4105 1640 3675 386 737 1774 2127 1640 4960 970 1774 3935 4362 3675 3853 1858 1774 1689 1689 737 1640 4607 3675 4515 1640 3675 737 1640 4105 1640 204 1858 2437 737 3675 1657 1858 2127 3675 970 737 4515 1640 2127 3935 3675 970 1657 1858 1774 737 3522

Je vous épargne la reconversion en caractères, le résultat fait boguer mon navigateur. Cependant, vous pouvez voir que le code en clair ne présage en rien du code chiffré : 111 et 110 sont très proches et le premier supérieur au second, mais leurs équivalents chiffrés, 1858 et 2127, sont très différents et le premier inférieur au second.

[[i]] | Vous vous demandez sans doute, maintenant que le message est chiffré, sous quelle forme on le transmet à son destinataire. La reconversion en caractères n'est pas une bonne idée, puisque le mélange de caractères orientés de droite à gauche et de gauche à droite donne des résultats assez imprévisibles. Cette question, la description de l'algorithme RSA n'y répond pas : c'est à vous de vous débrouiller. ^^ Comme je ne suis pas chien, je vous donnerai la solution la plus usuelle au chapitre 4.

Eh bien, déchiffrez, maintenant !

Longue introduction

Le déchiffrement va nous demander un petit peu plus de calcul. En effet, nous allons avoir besoin d'un entier D tel que $2 < D < M$ et que $D \cdot E \equiv 1\ [M]$. Une formulation équivalente à cette congruence est qu'il existe un entier k tel que $D \cdot E + k \cdot M = 1$.

Il est possible de trouver D grâce à l'algorithme d'Euclide **étendu**. Il commence comme l'algorithme d'Euclide tout court qui permet de trouver le PGCD de deux entiers, par exemple 43 et 12. Souvenez-vous.

$$\begin{align} 43 = 12 \times 3 + 7 & \Leftrightarrow 7 = 43 - 12 \times 3 \\ 12 = 7 \times 1 + 5 & \Leftrightarrow 5 = 12 - 7 \times 1 \\ 7 = 5 \times 1 + 2 & \Leftrightarrow 2 = 7 - 5 \times 1 \\ 5 = 2 \times 2 + 1 & \Leftrightarrow 1 = 5 - 2 \times 2 \\ 2 = 1 \times 2 + 0 & \ \end{align}$$

Les équivalences données à côté de chaque égalité ne sont pas innocentes. En effet, nous allons prendre la dernière, celle qui commence par 1 = , et remplacer dans le membre droit de l'égalité tous les nombres à l'exception des quotients par une formulation alternative telle que déterminée plus haut. Voyez plutôt.

$$\begin{align} \ & 1 = 5 - 2 \times 2 \\ \Leftrightarrow \ & 1 = 5 - (7 - 5 \times 1) \times 2 \\ \Leftrightarrow \ & 1 = (12 - 7 \times 1) - (7 - (12 - 7 \times 1) \times 1) \times 2 \\ \Leftrightarrow \ & 1 = (12 - (43 - 12 \times 3) \times 1) - ((43 - 12 \times 3) - (12 - (43 - 12 \times 3) \times 1) \times 1) \times 2 \\ \Leftrightarrow \ & 1 = (12 - (43 - 12 \times 3)) - ((43 - 12 \times 3) - (12 - (43 - 12 \times 3))) \times 2 \\ \Leftrightarrow \ & 1 = 12 - 43 + 12 \times 3 - (43 - 12 \times 3 - 12 + 43 - 12 \times 3) \times 2 \\ \Leftrightarrow \ & 1 = 12 - 43 + 12 \times 3 - 43 \times 2 + 12 \times 6 + 12 \times 2 - 43 \times 2 + 12 \times 6 \\ \Leftrightarrow \ & 1 = 12 \times 18 + 43 \times (-5) \end{align}$$

[[q]] | Pourquoi ne pas avoir pris 7 et 4992 comme exemple, ça nous aurait fait gagner du temps !

Pour une raison très simple : $4992 = 713 \times 7 + 1 \Leftrightarrow 1 = 4992 \times 1 + 7 \times (-713)$. Comme vous le voyez, il n'y a qu'une seule opération à faire pour trouver le résultat, ce qui ne permet pas de mettre en lumière l'algorithme.

Cependant, vous aurez remarqué que $D = -713$, ce qui ne colle pas avec l'obligation d'être supérieur à 2 que nous avons vue plus haut. Il est possible de contourner la difficulté. On va chercher un nombre entier a tel que $2 < D + a \cdot M < M$; dans la plupart des cas, $a = 1$ fonctionne, sinon on

essaye avec 2, 3, etc. Revenons à notre égalité de départ $D \cdot E + k \cdot M = 1$.

$$\begin{align} \ & D \cdot E + k \cdot M = 1 \\ \Leftrightarrow \ & D \cdot E + k \cdot M + E \cdot M \cdot a - E \cdot M \cdot a = 1 \\ \Leftrightarrow \ & E \times (D + a \cdot M) + M \times (k - a \cdot E) = 1 \end{align}$$

On voit bien que $D + a \cdot M$ remplit les deux conditions énoncées au début de cette section. Dans le cas qui nous intéresse, $-713 + 1 \cdot 4992 = 4279$: nous avons trouvé notre D !

[[i]] | En outre, pour une clé privée (P, Q, E) donnée, et donc pour une clé publique (N, E) donnée, il n'existe qu'un seul D valide : quand on chiffre un message avec une clé, il n'existe qu'une seule clé pour le déchiffrer. Vous verrez au chapitre 3 en quoi c'est important.

Courte explication

Une fois que ce D est déterminé, la fonction permettant de déchiffrer un message x est ridiculement simple.

$f : x \mapsto x^D \ [N]$

[[q]] | Mais cela ressemble énormément à la formule de chiffrement !

Oui. C'est ça qui est bien. Voyons à présent ce que cela donne sur un message qu'Adalbéron vient de recevoir de la part de Brunehaut. Il ne s'agit pas du message de tout à l'heure, sinon ce serait trop simple.

Code chiffré : 3852 1858 2127 3675 2799 737 1640 1774 3712 3675 787 4515 970 4527 2814 4964 737 1858 2127 4632 3675 737 1640 3935 737 1858 1774 204 1640 1489 1657 1858 2437 3675 1640 2127 3675 4527 970 3675 298 1858 1774 737 3675 3852 970 4650 2127 1640 3675 3543 3675 4527 970 3675 1657 2437 2127 1774 2437 3935 3522

Code en clair : 77 111 110 32 112 114 101 117 120 32 65 100 97 108 98 233 114 111 110 44 32 114 101 116 114 111 117 118 101 45 109 111 105 32 101 110 32 108 97 32 84 111 117 114 32 77 97 103 110 101 32 224 32 108 97 32 109 105 110 117 105 116 46

Message en clair : « Mon preux Adalbéron, retrouve-moi en la Tour Magne à la minuit. »

Et voilà ! C'est on ne peut plus simple ! Dans cet exemple, on a utilisé la même clé pour Adalbéron et Brunehaut, pour simplifier, mais elle doivent bien sûr impérativement être différentes.

La démonstration

Il nous reste à présent à montrer pourquoi l'algorithme fonctionne. C'est-à-dire pourquoi, si l'on applique successivement la fonction de chiffrement et de déchiffrement, on obtient le message de départ, quel qu'il soit. On l'a vu, si le message en clair est un entier positif x, le message chiffré est x^E

$[N]$ et le message déchiffré est $(x^E\ [N])^D\ [N] = x^{ED}\ [N]$. Il faut donc montrer que $x^{ED} \equiv x\ [N]$.

Cela va se faire en plusieurs étapes. Tout d'abord, il faut introduire une règle que l'on appelle couramment le [lemme de Gauss](https://fr.wikipedia.org/wiki/Lemme_d'Euclide) (voyez le lien pour la démonstration de ce lemme), et qui appliqué à notre cas dit que si $P \mid bQ$ et P et Q sont premiers entre eux (ce qui est nécessairement le cas puisqu'ils sont premiers tout court et différents) alors $P \mid b$, c'est à dire qu'il existe un entier c tel que $b = cP$.

À présent, prenons u et v deux entiers tels que $u \equiv v\ [P]$ **et** $u \equiv v\ [Q]$.
Alors, il existe un couple d'entiers (a, b) tel que $(u - v) = aP = bQ$.
Donc $P \mid bQ$.
D'après le lemme de Gauss, $P \mid b$ *i.e.* il existe un entier c tel que $b = cP$.
Alors, $(u - v) = bQ = cPQ$ ce qui équivaut à $u \equiv v\ [PQ]$.
Comme $PQ = N$, cela signifie que si nous pouvons démontrer que $x^{ED} \equiv x\ [P]$ et $x^{ED} \equiv x\ [Q]$, alors nous aurons démontré que $x^{ED} \equiv x\ [N]$.

Deux cas se présentent. Soit P divise x, soit P ne divise pas x. Le raisonnement est strictement le même avec Q.

Cas 1 : *P* divise *x*.
On a : $P \mid x \Leftrightarrow x \equiv 0\ [P] \Leftrightarrow x^{DE} \equiv 0\ [P]$.
Si ce résultat vous étonne, retournez lire le paragraphe sur les congruences dans le chapitre précédent.
Puis, x et x^{DE} étant congrus au même nombre modulo P, par définition, $x^{ED} \equiv x\ [P]$.

Cas 2 : *P* ne divise pas *x*.
Il faut se souvenir que $DE + kM = 1$. Donc $DE = 1 + (-k)M = 1 + (-k)(P - 1)(Q - 1)$.
Or, d'après le petit théorème de Fermat, comme P est premier et P ne divise pas x, on a le développement suivant :

$$\begin{align} \ & x^{P-1} \equiv 1 \ [P] \\ \Leftrightarrow \ & (x^{P-1})^{(-k)(Q-1)} \equiv 1 \ [P] \\ \Leftrightarrow \ & x \cdot (x^{P-1})^{(-k)(Q-1)} \equiv x \ [P] \\ \Leftrightarrow \ & x^{1 + (-k)(P-1)(Q-1)} \equiv x \ [P] \\ \Leftrightarrow \ & x^{ED} \equiv x \ [P] \end{align}$$

En somme, $x^{ED} \equiv x\ [P]$. De manière analogue, $x^{ED} \equiv x\ [Q]$, donc $x^{ED} \equiv x\ [N]$.

CQFD. Le raisonnement est un peu abscons, mais on s'est efforcé de le rendre le plus évident possible.

Malgré la présence de quelques exemples, cet exposé reste fondamentalement théorique. Dans le chapitre suivant, nous verrons comment le système cryptographique RSA est utilisé dans la vraie vie.

Un algorithme ne s’utilise pas n’importe comment

Dans ce chapitre, nous allons voir que RSA n’est pas dénué de faiblesses. Cependant, jusqu’à présent, il a toujours été possible de les contourner. Le système cryptographique RSA reste donc une protection robuste, à condition de l’utiliser correctement. Et nous allons voir comment on s’y prend.

La taille, ça compte

Avant toute chose, je vais vous demander de retourner au chapitre précédent et d’analyser attentivement le résultat produit par le chiffrement sur les deux messages échangés par nos amoureux. N’y a-t-il pas quelque chose qui vous interpelle ?

[[q]] | Le « o » est toujours chiffré de la même manière ?

Exactement ! Le « o » est systématiquement chiffré 1858, de même que les deux « M » donnent 3852 et que les « r » deviennent tous 737. Et en chiffrant plus de messages, vous arriveriez toujours au même résultat : un caractère donné est toujours chiffré de la même manière, à condition bien sûr d’utiliser la même clé publique. En d’autres termes, bien qu’il soit extrêmement complexe de retrouver quel caractère se cache derrière chaque code, nous avons affaire à un simple chiffrement par substitution.

[[q]] | Et c’est grave ?

Très ! Combien y a-t-il de lettres dans l’alphabet français ? Vingt-six. Ou plus exactement, trente-neuf en comptant les lettres avec diacritique (cédille, accent, tréma, etc.), soit soixante-dix-huit avec les majuscules. Ajoutez-y dix chiffres, l’espace et quelques caractères de ponctuation, vous avez grosso modo une centaine de caractères qui représentent la quasi-totalité de tout ce que vous pourrez être amenés à écrire en français.

Or une clé publique, comme son nom l’indique, est publique. Ce qui signifie qu’Ébroïn n’a pas besoin de « casser » la clé de Brunehaut pour lire ses messages : il lui suffit de se faire un tableau de cette centaine de caractères et du résultat que l’on obtient en les passant à la moulinette de la clé publique de Brunehaut pour pouvoir déchiffrer tous les messages que la damoiselle pourra recevoir.

Tenez, vous allez le faire vous-mêmes ! Les minuscules non-accentuées de l’alphabet latin sont regroupées entre le code 97 (pour « a ») et le code 122 (pour « z »). À l’aide de la clé publique de Brunehaut, chiffrez chacun de ces caractères (ainsi que l’espace, de code 32) et faites-vous un tableau de correspondance caractère → code chiffré. C’est fait ? Normalement, vous aboutissez à ceci.

-> a	b	c	d	e	f	g	h	i	j	k	l	m
970	2814	4105	4515	1640	1689	4650	4960	2437	2809	2598	4527	1657 <-

-> n	o	p	q	r	s	t	u	v	w	x	y	z	espace
2127	1858	2799	4640	737	3853	3935	1774	204	4149	3712	1897	4607	3675 <-

À présent, supposons que vous ayez intercepté le message suivant, chiffré avec cette même clé : « 2571 970 3675 4960 970 3675 244 3675 459 1640 3675 204 1858 1774 3853 3675 970 2437 3675 2814 2437 1640 2127 3675 2127 2437 4640 1774 4964 3853 3675 244 3675 3574 2437 4650 2127 4964 3675 3766 3675 375 970 2127 4650 4515 970 737 ». Essayez de déchiffrer ce que vous pouvez à l'aide de votre tableau : même si vous n'avez pas pu déchiffrer tous les caractères, cela ne vous empêche en rien de comprendre le message.

Ah ben non, je vous donnerai pas la solution, sinon ce n'est pas drôle. :P Soyez pas faignants, hé !

Vous voyez ! Vingt-six caractères à chiffrer par avance et vous comprenez déjà l'essentiel d'un message qui ne vous était pas destiné ! Vous pourriez même créer un programme qui fasse les maths à votre place et simplement savourer la lecture.

Autrement dit, utilisé comme au chapitre précédent, l'algorithme RSA ne résiste pas plus de quelques minutes à la cryptanalyse !

Damnation !

[[q]] | Alors en fait, c'est un gros étron nauséabond ton système cryptographique, là…

Pas du tout, il faut juste ne pas l'utiliser comme un amateur.

Vous vous souvenez qu'il n'est pas possible de chiffrer un nombre supérieur à la clé publique N utilisée ? Cela signifie par corollaire que n'importe quel nombre inférieur *peut* être chiffré avec cette clé-là. Dans la pratique, cependant, ce n'est pas tout à fait vrai. Sur un ordinateur, les données sont stockées sous forme d'octets, c'est-à-dire d'un nombre constitué de 8 bits : ce nombre peut donc aller de 0 à $2^8 - 1$. Et ces 8 bits sont indissociables. Cela a pour conséquence que, pour un N donné, un programme informatique ne pourra pas chiffrer de nombre supérieur à la plus grande puissance de 256 ($= 2^8$) qui soit inférieure à N.

Prenons un exemple : la clé que je vous ai donnée au début du chapitre précédent et qui a presque deux cents chiffres. Cette clé a une valeur N comprise entre 2^{639} et 2^{640}. Avec cette clé, il ne sera possible de chiffrer que 79 octets au maximum, car $(2^8)^{79} < N < (2^8)^{80}$. Pourquoi je vous raconte ça ? Nous allons y revenir.

Dans la mémoire de l'ordinateur, un texte est stocké sous forme d'octets, comme n'importe quelle donnée. Le standard le plus pratique pour les représenter est l'Unicode. Dans ce standard, les caractères les plus usuels des alphabets occidentaux sont stockés dans un seul octet, les caractères latins plus originaux, la quasi-totalité des alphabets et syllabaires du monde, ainsi que les caractères chinois, sont stockés dans deux octets, et quelques systèmes d'écriture plus originaux (essentiellement des systèmes utilisés dans l'Antiquité et qui ont depuis disparu) sont stockés dans trois octets.

Un texte complet n'est donc, pour l'ordinateur, qu'une suite d'octets, qu'on lui a dit d'interpréter comme du texte. Mais fondamentalement, un octet est un nombre, et c'est nous humains qui établissons (ou plutôt, faisons établir par l'ordinateur) une correspondance entre un nombre et le caractère qu'il est censé représenter. Et il est tout à fait possible de demander à l'ordinateur d'interpréter la suite d'octets autrement que comme du texte : on va, par exemple, lui dire qu'il faut considérer chaque groupe de quatre octets successifs comme la représentation d'un grand nombre

stocké sur quatre octets (donc compris entre 0 et $(2^8)^4 - 1$). De cette manière, le texte « Papa », au lieu d'être interprété comme la suite de nombres stockés sur un seul octet 50 - 61 - 70 - 61 (en notation hexadécimale), sera interprété comme le nombre stocké sur quatre octets 50617061 (ce qui équivaut à $1348550753 = 80 \cdot (2^8)^3 + 97 \cdot (2^8)^2 + 112 \cdot (2^8)^1 + 97 \cdot (2^8)^0$ en notation décimale).

Ce nombre stocké sur quatre octets reste largement inférieur à la limite de 79 octets imposée par la clé publique dont nous parlions tantôt. Il est donc possible de le chiffrer à l'aide de cette même clé, ce qui revient à chiffrer le texte « Papa » d'un seul bloc et non plus caractère par caractère. De cette manière et avec cette clé, on peut par conséquent chiffrer d'un bloc n'importe quelle suite de caractères stockée en mémoire dans 79 octets ou moins.

L'attaque consistant à chiffrer par avance les caractères les plus courants du français et à les rechercher dans le message chiffré n'est plus envisageable. En effet, il faudrait chiffrer par avance n'importe quelle combinaison de 79 ou moins des caractères les plus courants… et prier pour qu'un caractère plus rare comme « ñ » ou « * » ne se soit pas glissé dans le lot ! Comme il y a une centaine de caractères usuels, il y a environ $100^{79} = 10^{158}$ combinaisons de 79 caractères usuels. Vous comprenez d'ores et déjà que la tâche est trop titanesque pour être sérieusement entreprise.

[[q]] | Et si mon message tient sur plus de 79 octets ?

Qu'à cela ne tienne ! Il suffit de découper le message en tronçons de 79 octets et de chiffrer chaque tronçon. Le message chiffré sera composé de chacun de ces tronçons chiffrés mis les uns à la suite des autres. Techniquement, c'est déjà ce que nous faisions en chiffrant les caractères un par un : on découpait le message en tronçons d'un seul octet (parfois deux), que l'on chiffrait individuellement. On applique ici le même processus, mais à une plus grande échelle !

En outre, cette technique peut être appliquée à autre chose qu'à du texte. Une image aussi est stockée dans la mémoire de l'ordinateur sous forme d'une suite d'octets. Elle aussi, on peut la découper en tronçons de 79 octets pour chiffrer chaque tronçon l'un après l'autre. Vous voulez envoyer une photo coquine de vous à votre compagne(on) sans que Google ou votre fournisseur d'accès à Internet ne puisse la visionner ? Découpez-là en tronçons et chiffrez-les avec la clé publique de votre amant/maîtresse ! Cela marche bien évidemment aussi pour la musique, les films, les archives, les logiciels, ou n'importe quoi d'autre qui peut être stocké sur un ordinateur. Désolé pour ceux qui voulaient chiffrer leur grand-mère et s'apprêtaient déjà à la découper en tronçons…

Pour ne pas finir sur le Net comme une vulgaire JLaw, chiffrez vos photos.

[[i]] | **Ça va sans dire, mais ça va mieux en le disant** : depuis tout à l'heure, je parle de faire des tronçons en 79 octets parce que la clé donnée en exemple ne permet pas d'en chiffrer plus. Si votre clé à vous est comprise entre 2^{40} et 2^{41}, vous ne pourrez chiffrer que 5 octets d'un coup et vous devrez alors découper vos données en tronçons de 5 octets, alias 40 bits.

Qui es-tu, belle inconnue ?

Au stade où l'on en est, Ébroïn est en train d'avaler goulûment son chapeau. Mais ce faisant, il lui vient une idée lumineuse. Il prend son plus beau clavier, écrit une lettre d'insultes qu'il conclut par « Je ne veux plus jamais te revoir, grosse vache ! », la signe « Adalbéron », et l'envoie à Brunehaut en faisant bon usage de sa clé publique. Brunehaut, qui n'est pas bête, et qui rentre à nouveau dans du 36 depuis quelques semaines, se doute qu'il y a anguille sous roche à la lecture du « grosse vache ».

Si les deux amoureux restent en bons termes, cela met cependant en lumière un véritable problème : à partir du moment où la clé publique d'un destinataire est connue de tous, comment être certain que l'expéditeur est bien celui qu'il prétend ? Surtout quand le message dit « Coucou, veuillez utiliser ma carte bleue pour envoyer douze lingots d'or à une adresse au Nigéria que je n'ai encore jamais utilisée, bisoux ! » ?

La solution est relativement simple : il faut signer le message à l'aide d'une information que seul l'expéditeur connaît, mais sans la divulguer pour autant !

[[q]] | Ha ! Ha ! Très drôle… Et je fais ça comment, moi ?

C'est pourtant simple ! Il suffit de chiffrer tout d'abord le message à l'aide de votre clé privée, comme si vous étiez en train de le déchiffrer, puis de la clé publique de votre correspondant. De cette manière, lorsque votre correspondant reçoit le message chiffré, il commence par le déchiffrer avec sa clé privée, puis vérifie que vous êtes bien l'envoyeur en utilisant dessus votre clé publique, comme s'il voulait le chiffrer pour vous l'envoyer. Si le message en clair apparaît, il ne fait alors aucun doute que vous avez bien envoyé ce message : vous seul connaissez votre clé privée, vous seul pouvez l'avoir utilisée pour authentifier le message.

[[i]] | Retournez jeter un œil à la démonstration de la validité de l'algorithme. Nous avons vu que lorsque l'on applique la fonction de chiffrement, puis la fonction de déchiffrement, à un message x, on obtient pour résultat $(x^E \ [N])^D \ [N] = x^{ED} \ [N]$, ce qui est strictement égal à x, le message de départ. | | Si l'on applique *d'abord* la fonction de déchiffrement, *puis* la fonction de chiffrement, on obtient pour résultat $(x^D \ [N])^E \ [N] = x^{DE} \ [N]$, c'est-à-dire exactement la même chose ! Les deux fonctions peuvent être utilisées à la suite dans n'importe quel ordre, on obtient toujours le message d'origine, mais seul le propriétaire de la clé privée peut appliquer l'ordre déchiffrement puis chiffrement.

Accessoirement, cette authentification vous engage : à moins de déclarer que l'on vous a volé votre clé privée, et d'annuler ainsi tout ce que vous avez pu faire avec depuis le vol supposé, vous ne pouvez pas nier être l'expéditeur d'un message utilisant ce type de signature.

Cependant, le gros défaut de la cryptographie asymétrique, c'est que les fonctions mathématiques

utilisées consomment énormément d'énergie car elles demandent beaucoup de calculs : le chiffrement et le déchiffrement sont longs, parfois *très* longs. C'est pourquoi, dans la vraie vie, on signe rarement un message en le chiffrant en entier avec sa clé privée.

On va plutôt signer un « résumé » du message. Mais ce résumé doit avoir un certain nombre de caractéristiques pour être valide.

1. Il doit être plus petit que le message d'origine, sinon, cela n'a pas grand intérêt.
2. Il doit être absolument impossible de retrouver le message d'origine à partir du résumé. En effet, le résumé est chiffré à l'aide de la clé privée de l'expéditeur, et déchiffré à l'aide de sa clé publique, qui est largement diffusée. Cela signifie que n'importe qui peut déchiffrer la signature et donc lire le résumé. S'il existait un moyen de remonter jusqu'au message à partir du résumé, toutes les mesures de chiffrement appliquées au message seraient vaines, un attaquant n'ayant qu'à le reconstituer à partir du résumé facile à obtenir.
3. Deux messages même similaires doivent avoir des résumés différents. De cette manière, il est possible de savoir qu'un message a été altéré lors de son transfert si le résumé réalisé au départ et signé est différent d'un résumé réalisé à la réception. Il est important de garantir l'intégrité du message lors de son transfert : quand vous payez 100€ par carte bancaire, vous ne tenez pas à ce que, du fait des aléas de l'Internet ou de l'action malveillante de quelqu'un, votre banque reçoive un message lui demandant de payer 10000€.

La tâche semble ardue, n'est-ce pas ? Il existe pourtant un type de fonction qui remplit exactement ces objectifs : on les appelle des **fonctions de hachage**. Les plus célèbres sont SHA-1, MD5 ou Tiger. On ne rentrera pas dans le détail de leur fonctionnement, cela mériterait un tuto à part entière. Il faut seulement retenir que ces fonctions ont trois caractéristiques essentielles.

1. À partir d'une entrée de n'importe quelle taille, elles produisent un résultat de taille fixe, souvent 128, 160 ou 256 bits.
2. Il est strictement impossible de savoir quelle entrée a produit un résultat donné : contrairement à RSA, il n'y a pas de porte dérobée, la fonction est vraiment à sens unique.
3. Deux entrées extrêmement similaires, même si elles ne diffèrent que d'un bit, donneront des résultats radicalement différents. Par corollaire, il est très difficile de produire volontairement un message qui donnera le même résultat qu'un autre message donné, ce que l'on appelle une **collision**.

Il n'existe pas de moyen plus simple de définir ce qu'est une fonction de hachage : si une fonction respecte ces trois impératifs, alors c'est une fonction de hachage, point final. Le « résumé » d'un message produit par une fonction de hachage est appelé un **condensat**.

Le principe de la **signature par clé privée** ou *signature numérique* est de réaliser un condensat du message que l'on veut envoyer, de le chiffrer à l'aide de sa clé privée, et de joindre cette signature au message afin de l'authentifier. Quand Brunehaut reçoit un message ainsi signé, elle déchiffre le condensat avec la clé publique d'Adalbéron, déchiffre le message avec sa clé privée, réalise un condensat du message déchiffré (à l'aide de la même fonction de hachage, bien entendu) et compare les deux condensats : ils doivent être identiques, sans quoi le message peut être un faux ou avoir été corrompu.

Seul le condensat étant chiffré par la clé privée, l'ensemble du processus est beaucoup plus rapide que

si l'on chiffrait le message entier avec la clé privée de l'expéditeur puis la clé publique du destinataire.

[[i]] | Cette méthode permet de remplir les quatre objectifs d'un système cryptographique complet : | | - authenticité ; | - intégrité ; | - non répudiation ; | - secret.

À présent, voyons un exemple concret. Pour pouvoir manipuler des messages faisant un peu plus qu'un seul caractère, sans pour autant utiliser des nombres à 120 chiffres, nous allons utiliser la clé (N = 8889895013, E = 23), avec P = 72421 et Q = 122753. L'exposant de déchiffrement est D = 3865086887, comme vous pouvez le calculer vous-mêmes. Quant à Adalbéron, sa clé sera N = 8042009699, E = 4001, D = 110547521. Ces deux clés sont légèrement supérieures à 2^{32} : comme nous l'avons vu dans la section précédente, elles nous permettront de chiffrer au maximum quatre octets (soit 32 bits) à la fois. Il faudra donc découper tout message plus long en tronçons de quatre octets.

En guise de fonction de hachage, nous allons utiliser Adler-32. Ce n'est pas une très bonne fonction de hachage, parce qu'il est facile d'obtenir des collisions, mais elle produit des condensats de 32 bits, ce qui reste facilement manipulable et peut être chiffré en une seul fois à l'aide des clés fournies ci-dessus.

Adalbéron désire envoyer le message « Douce amie ! ». Il commence par le découper en tronçon de quatre octets, ce qui nous donne 1148155235, 1696620909 et 1768235041. Chacun de ces tronçons est chiffré à l'aide de la clé publique ci-dessus : 1338172656, 923473543 et 8857666592. Ensuite, Adalbéron génère le condensat de son message par Adler-32, ce qui donne 1bb103ee (en hexadécimal), c'est-à-dire 464585710 (en décimal). Il applique alors sa **propre** fonction de déchiffrement sur ce condensat : $464585710^{110547521}$ [8042009699] = 2226882204. Il ne lui reste plus qu'à envoyer le message « 1338172656 923473543 8857666592 2226882204 » à Brunehaut.

À la réception du message, celle-ci procède en plusieurs étapes. D'abord, elle déchiffre les trois premiers nombres avec sa clé privée, ce qui lui permet de retrouver le message : je ne vous montre pas, vous savez comment faire. Ensuite, elle applique la fonction de chiffrement **d'Adalbéron** au dernier nombre (la signature) : 2226882204^{4001} [8042009699] = 464585710. Enfin, elle génère le condensat par Adler-32 du message tel qu'elle l'a déchiffré, ce qui est censé donner 464585710 (en décimal) aussi. Elle a alors la preuve que le message a bien été envoyé par Adalbéron. S'il s'avérait que les deux condensats ne correspondent pas, cela signifierait que, soit le message n'a pas été envoyé par Adalbéron, soit le message a été altéré en chemin. Dans les deux cas, elle peut le mettre à la poubelle.

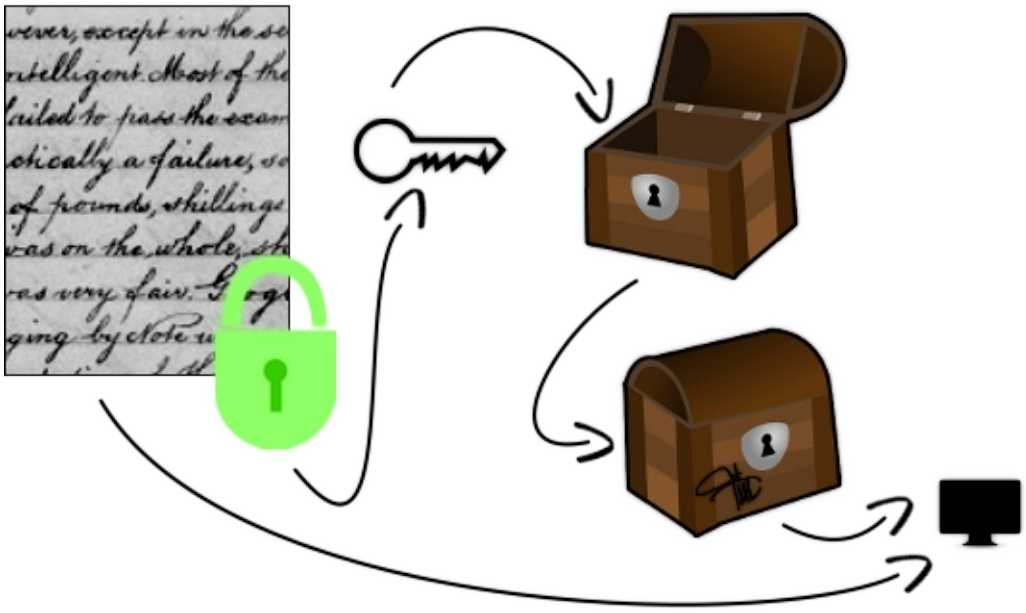
[[a]] | **La signature par clé privée n'est pas spécifique à RSA.** D'une part, tous les systèmes cryptographiques asymétriques (El-Gamal, par exemple) ont des clés privées, lesquelles peuvent servir à signer un condensat. D'autre part, cette technique peut être utilisée pour authentifier un message chiffré avec un autre algorithme que RSA, voire un message en clair ! Eh oui ! De nombreux sites, en particulier des blogs, permettent de laisser des commentaires sans pour autant s'inscrire au site. Mais qui garantit que derrière un pseudo donné se cache toujours la même personne ? Vous pouvez prouver que vous êtes bien l'auteur d'un message public (donc lisible par tout le monde) : il suffit d'en faire un condensat et de le signer à l'aide de votre clé privée, RSA ou autre.

Parce que Snowden n’était pas le premier

Mais il y a mieux encore ! Appliquer la fonction de chiffrement de votre destinataire à l’ensemble du message est également une opération très énergivore : il faudrait pouvoir en chiffrer le moins possible. Vous pensez sans doute à une fonction de compression pour réduire la taille du message, mais cela ne suffirait pas. La solution ultime s’appelle la **cryptographie hybride**.

Celle-ci fut inventée en 1991 par un dénommé Phil Zimmerman lorsqu’il créa et mit à disposition du public sur Internet le logiciel PGP, dont vous avez peut-être entendu parler. Le principe est vraiment simple, mais il fallait y penser.

Un système cryptographique symétrique, quel qu’il soit, peut être extrêmement solide, presque autant qu’un système asymétrique. Sa seule réelle faiblesse réside dans le moment où il faut faire parvenir la clé à son destinataire. Le principe de la cryptographie hybride, c’est de chiffrer le message à l’aide d’un algorithme symétrique le plus solide possible (de nos jours, on utiliserait AES), parce que c’est beaucoup plus rapide qu’avec un système asymétrique, puis de chiffrer *la clé* et seulement la clé à l’aide du système asymétrique. Il ne reste plus qu’à envoyer le message chiffré, la clé chiffrée aussi et la signature réalisée comme on l’a vu plus haut.



Le principe de la cryptographie hybride

[[i]] | Pour la petite histoire, le premier système de cryptographie hybride est en réalité l’**échange de clés de Diffie–Hellman**, inventé en 1976. Cependant, celui-ci a été peu utilisé et ses fondements mathématiques le rendent moins souple qu’un système à la PGP.

De cette manière, le système est tout aussi sécurisé, mais on réduit comme peau de chagrin la quantité de données à chiffrer de manière asymétrique : les clés de systèmes symétriques dépassent rarement 256 bits de longueur, et les fonctions de hachage actuellement sur le marché ne produisent pas d’empreinte de plus de 512 bits. Au maximum, les fonctions symétriques n’auront à travailler que sur un millier de bits.

À quelques détails près, vous avez là le système cryptographique le plus robuste possible avec les connaissances actuelles. Quelques années après sa sortie, il a été dit de PGP que c’était ce qui

s'approchait le plus d'une protection de niveau militaire donnée aux particuliers. Ce n'est pas pour rien que ce logiciel, ou son équivalent libre GnuPG, est le plus utilisé au monde pour protéger ses mails.

Bon, je vous entends demander à cor et à cri un exemple. D'accord, d'accord. Adalbéron doit envoyer le message « RDV au moulin ce soir » à Brunehaut. Pour cet exemple, nous utiliserons les dernières clés en date qui, je le rappelle, ne permettent pas de chiffrer plus de quatre octets donc 32 bits.

1. Il chiffre son message à l'aide de l'algorithme symétrique AES et de la clé « 4891e674ff312b9519df3245b7d1c3ff ». Pour la petite histoire, il s'agit du condensat MD5 de « Brunehaut, je te kiffe ! ». Le résultat (découpé en tronçons de 32 bits et en notation hexadécimale, pour plus de clarté) est le suivant : « 7bd19a87 3c0f39a5 d7656446 d1d5007a 21757f1d 3a36fd25 97c527cf 5f03f75d ».
2. Il réalise un condensat par Adler-32 de son message (4d23075c), qu'il signe à l'aide de sa fonction de déchiffrement : « c8c43b7e » (en hexadécimal, encore).
3. Il découpe la clé qu'il a utilisée pour AES en tronçons de 32 bits et les chiffre à l'aide de la clé publique de Brunehaut : « 1bc83cd40 1bcb41456 1c1936f3f fa66823f » (en hexadécimal, toujours).
4. Il envoie à Brunehaut le message suivant.

[[i]] | CLÉ : 1bc83cd40 1bcb41456 1c1936f3f fa66823f | | MSG : 7bd19a87 3c0f39a5 d7656446 d1d5007a 21757f1d 3a36fd25 97c527cf 5f03f75d | | SIG : c8c43b7e

À la réception, Brunehaut réalise un déchiffrement en 6 étapes.

1. Elle déchiffre la clé AES à l'aide de sa propre clé privée.
2. Elle déchiffre le message à l'aide de la clé AES qu'elle vient de trouver.
3. Elle génère par Adler-32 un condensat de ce message fraîchement déchiffré.
4. Elle déchiffre la signature à l'aide de la clé publique d'Adalbéron.
5. Elle compare les deux condensats ainsi obtenus.
6. S'ils sont identiques, elle se plonge goulûment dans la lecture du message qu'elle a déchiffré.

[[q]] | Mais pourquoi tu nous as parlé de Snowden, sale rascal, à part pour récolter des vues sur Google ?

Parce que ce sympathique M. Zimmerman a eu des déboires importants avec diverses instances de son pays. Comme un certain Edward S. dont nous ne parlerons plus.

En premier lieu, il faut savoir que la loi américaine permet de breveter le logiciel, contrairement à la loi française. En second lieu, il faut savoir que nos amis Rivest, Shamir et Adleman, pas fous, avaient breveté le RSA et même créé une société privée pour l'exploiter. Ce sont eux qui ont attiré le regard de la justice sur Zimmerman, au motif qu'il était un vilain pirate qui mettait à disposition de n'importe qui des moyens de protéger sa vie privée, certes, mais gratuitement, ce qui est bien plus grave.

Là où ça devient rigolo, c'est que lorsque les autorités américaines s'emparent du dossier en 1993 et ouvrent une enquête criminelle au sujet de ce brave homme, ce n'est pas au sujet d'une éventuelle contrefaçon. Non. C'est sous des accusations de complicité de terrorisme. Oui, déjà.

Car il faut savoir, en troisième lieu, que jusqu'à la fin des années 1990, la loi américaine considérait qu'exporter hors du territoire national un outil cryptographique utilisant des clés de plus de 40 bits (le terme technique pour des clés de moins de 40 bits est « une protection de merde ») équivalait à exporter des munitions d'armes de guerre. À des terroristes, puisque c'est à l'étranger. Et puisque le logiciel était trouvable sur Internet, il y avait bien exportation hors du territoire. Pour contourner cela, Zimmerman a été obligé d'imprimer le code de PGP sur des livres et d'exporter les livres, qui sont eux protégés par le premier amendement de la constitution américaine.

L'enquête a duré trois ans, avant d'être abandonnée sans raison officielle en 1996. La raison officieuse est que Zimmerman a reçu de très nombreux mails et lettres de soutien venues du monde entier, ce qui aurait fait plier les services d'espionnage américains. Il y a peut-être une autre raison, encore plus spéculative, que nous verrons dans la section suivante.

Comment l'autorité publique reprit la main

Il reste une grosse faille dans notre système cryptographique. La seule qui soit à l'heure actuelle encore réellement exploitable, au moyen de ce que l'on appelle **l'attaque de l'homme du milieu**.

Revenons à Ébroïn. Faute de meilleure solution, il décide de placer une troupe de ses fidèles sur la route qui mène de chez Adalbéron à chez Brunehaut, pour intercepter tout messager qui irait de l'un à l'autre. Lorsqu'Adalbéron demande à Brunehaut de lui envoyer sa clé publique, Ébroïn lui envoie en vérité sa propre clé publique et demande à Brunehaut la sienne. Il fait de même dans l'autre sens. À ce stade, chacun des amoureux a la clé publique d'Ébroïn en pensant que c'est celle de l'autre, et Ébroïn possède la clé publique des deux. De cette manière, lorsque l'un d'entre eux envoie un message, il l'intercepte, le déchiffre avec sa clé privée, éventuellement le modifie, et le chiffre à nouveau avec la vraie clé du destinataire, avant de faire suivre.

Une personne malveillante s'est glissée *au milieu* de la communication au moment de la transmission des clés publiques et met tout le système en l'air. Patatras ! On en revient au même point qu'avec les chiffrements symétriques : comment transmettre la clé de chiffrement, certes publique, à son destinataire sans que quelqu'un l'intercepte à des fins douteuses ?

Petite parenthèse avant l'explication. Vous vous dites peut-être qu'une telle attaque a peu de chance d'arriver, qu'elle nécessite quand même de gros moyens. Elle est beaucoup plus facile à mettre en œuvre que vous pourriez le penser. Quand vous vous connectez à un WiFi public, qui vous garantit que votre connexion va directement au routeur et ne transite pas par l'ordinateur d'un autre des clients affairés ? Si vous allez voir vos comptes bancaires au boulot, comment être certains que le responsable du service informatique ne détourne pas ce genre de requêtes avant qu'elles ne sortent du bâtiment ? Quand vous achetez en ligne à l'aide de votre numéro de carte bleue, qui vous dit que votre fournisseur d'accès à Internet ne le note pas dans un coin en profitant de sa position obligatoire d'homme du milieu ?

La seule solution connue est de faire certifier la clé publique par un intermédiaire de confiance. Par exemple, au lieu de s'envoyer mutuellement leur clé publique par la poste, Adalbéron et Brunehaut la confient chacun à leur ami Warnachaire en qui ils ont toute confiance et qu'ils peuvent rencontrer physiquement. Et c'est lui et lui seul qui distribue les clés publiques quand on les lui demande, et en main propre.

L’idée n’est vraiment pas mauvaise. Warnachaire étant ami un peu avec tout le monde, bien vite, les clés de Chimnechilde, de Drogon de Frédégonde et de Genséric viennent rejoindre son trousseau. Il est devenu une **autorité de certification** ou AC pour les intimes. Seulement, quand il ne peut pas se déplacer en personne, Warnachaire est obligé d’envoyer la clé voulue authentifiée par sa propre clé privée. Comment certifier que la clé publique qui va avec est bien la sienne et pas celle d’Ébroïn ?

Eh bien sa clé publique est elle-même confiée à une AC de niveau supérieur, qui confie sa propre clé à une autorité supérieure, et ainsi de suite, créant ainsi une pyramide de certifications.

Ce système de certificats distribués par des autorités hiérarchisées est celui qu’utilise le système TLS, plus connu sous le nom de SSL. Oui, oui, celui-là même qui se cache derrière les adresses en HTTPS de votre navigateur.

[[q]] | Mais alors, ~~je faisais de la prose~~ j’utilisais RSA sans même le savoir ?

Tout à fait ! C’est encore l’utilisation de RSA la plus courante. Et si nous vivions dans un monde parfait, elle serait excellente. Mais il y a un hic. Les autorités de certification étant organisées de manière pyramidale, tout en haut ne se trouvent que quelques dizaines d’autorités « ultimes ». Or on sait de source sûre depuis 2010 que la NSA fait pression sur les AC américaines de plus haut niveau pour obtenir des certificats « de complaisance » et sans doute sur les AC des autres pays aussi. Et nul doute que les services d’espionnage des autres pays font de même avec leurs AC nationales.

La NSA fait pression sur les AC américaines

[[q]] | Mais on ne peut pas faire autrement, non ? Et je préfère me faire espionner par la NSA que par mon voisin de pallier à l’air louche, ils ont moins de raisons d’en vouloir à ma carte bleue…

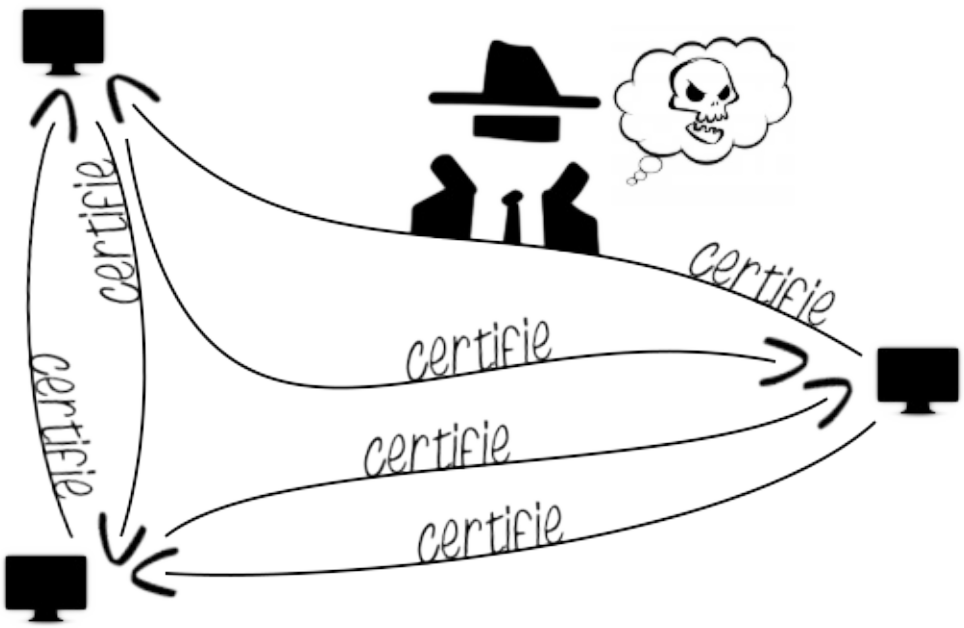
Si, on peut faire autrement. Des logiciels comme GnuPG utilisent le système du **réseau de confiance**, ou *web of trust* en anglais. Dans les **infrastructures à clés publiques** comme le réseau de certification de SSL, une seule des deux parties certifie l’autre, ce qui génère la hiérarchisation dont on a parlé. Au contraire, dans un réseau de confiance, les deux parties se certifient mutuellement, créant ainsi un système décentralisé. Voyez plutôt.

À l’occasion d’une rencontre en personne, Brunehaut et Adalbéron s’échangent leurs clés publiques respectives. Quelques temps plus tard, Brunehaut rend visite à sa meilleure amie Chimnechilde, et elles échangent leurs clés. Lorsque Chimnechilde veut obtenir la clé publique d’Adalbéron, elle la

demande à Brunehaut, qui la lui donne en la certifiant avec sa propre clef privée. Comme Chimnechilde a une confiance totale en Brunehaut, elle accepte la clé sans problème. Dans son message à Adalbéron, elle lui donne sa clé publique, et comme c'est la meilleure amie de Brunehaut, Adalbéron la range dans ses contacts de confiance. De cette manière, si Brunehaut change de clé d'une manière ou d'une autre, elle n'a qu'à la donner à Chimnechilde en mains propres et celle-ci pourra la transmettre à Adalbéron, qui a toute confiance en elle.

On peut y ajouter encore une subtilité. Brunehaut est également amie avec Haremburgis et lui accorde toute sa confiance. Mais Adalbéron trouve que c'est une oie blanche, qui se laisse facilement berner, alors quand il obtient sa clé par l'intermédiaire de Brunehaut, il la range dans ses contacts de confiance limitée. Warnachaire subit le même sort : depuis que le comte Leudegisèle a menacé de massacrer toute sa famille s'il ne « coopérait pas avec la justice », tout le monde sait que ses informations sont sujettes à caution ; mais comme c'est un vieil ami, on ne va pas le rejeter totalement. Quand Adalbéron a besoin de la clé de quelqu'un, il demande à tous ses contacts. Si un de ses contacts de confiance lui donne une réponse, il l'accepte. Pour ses contacts de confiance limitée, en revanche, il faut qu'un nombre suffisant d'entre eux (généralement trois) lui donnent la même réponse pour qu'il l'accepte.

De cette manière, un réseau de confiance totalement décentralisé se constitue peu à peu et il est beaucoup plus difficile, voire impossible, à une agence gouvernementale de falsifier les informations.



Un réseau de confiance totalement décentralisé se constitue peu à peu

[[a]] | Si vous prêtez garde à ce que j'ai expliqué plus haut, il y a toujours un moment où il faut pouvoir s'échanger les clés de manière sécurisée : pour obtenir les clés publiques des AC de niveau maximal ou pour faire sa première entrée dans un réseau de confiance. Et malheureusement, le seul moyen sûr à 100 % pour se protéger d'une attaque de l'homme du milieu, c'est la rencontre physique. Dans la pratique, cela tient du bricolage. | | Côté SSL, les clés publiques des AC de niveau maximal sont écrites en dur dans les logiciels qui en ont besoin, par exemple les navigateurs, et elles sont fournies dans des documents officiels de l'administration publique, disponible également au format papier : l'Union Européenne, par exemple, édite régulièrement un PDF contenant les clés publiques

des AC de plus haut niveau ayant leur siège sur son territoire. Ensuite, quand vous voulez fournir votre clé publique à une AC pour qu'elle vous certifie, elle vous demandera généralement un justificatif d'identité pour prouver qui vous êtes. | | Côté réseau de confiance, on peut se réunir entre amis pour générer des clés tous ensemble et se les échanger en direct. Puis vous recommencez avec un autre groupe de potes et vos amis font pareil : petit à petit le réseau se constitue. Si vous voulez changer de clé et que la précédente n'a pas été compromise, vous pouvez alors diffuser votre nouvelle clé publique en la signant avec l'ancienne. Une solution intermédiaire, si vos amis ne sont pas intéressés par la mise en place d'un réseau de confiance, peut être de faire un échange de clés en main propre avec une association en qui vous avez confiance qui pourra alors vous mettre en contact avec d'autres gens.

Avant de terminer cette section, je voudrais revenir sur la remarque que j'ai faite en fin de section précédente. Vous vous souvenez que les poursuites contre Zimmerman ont cessé en 1996 sans raison apparente ? En 1995, le navigateur Netscape implémente la première version publique de SSL. Pour rappel, cette année-là, il était utilisé par environ 70 % des internautes. Ce n'est peut-être pas une coïncidence.

Bourrage papier

Abordons à présent certaines attaques plus fourbes à l'encontre de RSA.

La première d'entre elles a été démontrée par Johan Håstad en 1986 et utilise le théorème des restes chinois. Celui-ci sert à résoudre le difficile problème d'un système de n équations de la forme $x \equiv a_i\ [N_i]$. Or, si le même message est envoyé à plusieurs personnes ayant des N différents mais des E identiques, chaque fonction de chiffrement vaut $x^E \equiv y_i\ [N_i]$, où y_i est le texte chiffré envoyé à chaque destinataire : il suffit de E destinataires différents pour que le théorème fonctionne et qu'on puisse récupérer x^E, donc en un temps très réduit le message en clair.

Cela signifie qu'il est dangereux d'utiliser un E de petite taille. Premièrement, parce qu'il sera plus facile de réunir E messages identiques. Deuxièmement, parce que la probabilité que vous ayez le même E que d'autres gens est beaucoup plus forte avec $E = 3$ qu'avec $E = 2^{2^{42}} - 1$. Troisièmement, parce qu'avec un E suffisamment grand, même si l'attaque reste théoriquement possible, le temps nécessaire à la résolution du système d'équations devient trop important pour être rentable.

[[i]] | Dans le cas le plus extrême, avec un petit E, un grand N et un message court, $x^E < N$ donc une simple racine E-ième suffit à retrouver le message en clair.

En outre, Håstad a montré que rajouter du « bruit » aux différents messages pour qu'ils soient légèrement différents ne fonctionne pas si la fonction utilisée est déterministe. En 1997, Don Coppersmith a étendu l'attaque en démontrant que même un « bruit » aléatoire ne suffit pas à protéger le message si ce « bruit » est trop court, mais si je vous explique comment fonctionne l'attaque, on y est encore après-demain.

Dans un tout autre genre, RSA est sensible aux attaques par texte chiffré choisi. En effet, si un attaquant cherche à déchiffrer un message y donné, il peut demander au destinataire du message de lui déchiffrer le message $y \cdot r^E\ [N]$ où r est un entier pris au hasard : le résultat du déchiffrement sera x

$\cdot r$ [N], à partir duquel il est très simple de retrouver x. Cette attaque a toujours été théorique : pourquoi quelqu'un accepterait-il de déchiffrer pour vous l'intégralité d'un message chiffré d'une manière qui indique qu'il est destiné à lui seul ? Il faudrait vraiment que la victime soit idiote.

Mais un beau jour de 1998, Daniel Bleichenbacher a montré que cette attaque pouvait être beaucoup plus pernicieuse qu'on ne le pensait alors. Supposez un système informatique qui attend des messages formatés d'une certaine manière (donc à peu près tous les serveurs, HTTP, IRC, FTP ou n'importe quoi) mais chiffrés avec sa clé publique. Si un seul bit, ou mieux quelques octets, du texte clair sont strictement identiques dans tous les messages en clair qu'attend ce système — par exemple, le *magic number* qui identifie le format — et si le système renvoie un message d'erreur lorsque le message déchiffré ne correspond pas au format qu'il attendait, alors il existe un algorithme qui permet dans un temps très raisonnable de déchiffrer n'importe quel message chiffré adressé à ce système que l'on aurait intercepté.

[[i]] | Si l'on résume, pour être protégé contre ces deux types d'attaque, il faut respecter les recommandations suivantes. | | - Un E de taille suffisante. On recommande de ne pas descendre en-dessous de 65537, valeur qui permet malgré tout d'avoir de bonnes performances au chiffrement et à la vérification de signature par clé privée. | - Compléter les messages par un bourrage de « bruit » d'une taille suffisante. | - Pas le moindre bit du message en clair ne doit être prévisible.

Ces recommandations sont d'autant plus importantes quand le texte en clair est court, par exemple une clé de cryptographie symétrique ou un condensat à signer.

C'est là l'objet de l'***Optimal Asymmetric Encryption Padding*** (OAEP). Il s'agit d'un schéma de bourrage qui nécessite pour fonctionner un générateur de données aléatoires efficace et deux fonctions de hachage $G(x)$ et $H(x)$ donnant un résultat de taille fixe, respectivement k_1 et k_2. Les étapes de l'algorithme de bourrage sont alors les suivantes.

1. Le message est complété avec des 0 jusqu'à atteindre une taille de k_1 bits. On l'appelle m_0.
2. Le générateur de données aléatoires génère k_2 bits de données, appelées r.
3. On calcule $X = m_0 \oplus G(r)$ où $\oplus$ est la fonction « ou exclusif » ou *XOR*.
4. On calcule $Y = H(X) \oplus r$.
5. On concatène X et Y dans cet ordre, et c'est ce message seulement que l'on chiffre à l'aide de RSA.

La fonction « ou exclusif » étant son propre inverse, il est très simple de répéter l'opération en sens inverse. Ce schéma de bourrage a aussi comme intérêt qu'il n'est pas possible de reconstituer le message en clair bourré complet à partir d'un fragment de celui-ci.

Je n'ai pas les bonnes clés

Les attaques présentées dans la section précédente cherchaient à déchiffrer le message sans pour autant obtenir la clé privée du destinataire. Celles que nous allons voir ici sont beaucoup plus dangereuses, car elle permettent de casser une clé privée et donc d'usurper totalement l'identité de quelqu'un.

Tout d'abord, sans surprise, il faut que N soit suffisamment grand pour ne pas être factorisable dans

un délai raisonnable, même en faisant travailler des milliers d'ordinateurs en parallèle dessus. À l'heure actuelle, on est capable de factoriser des *N* de 768 bits de long en deux ans. Le *N* de 640 bits de long que je vous ai donné comme exemple de clé réelle a été factorisé en 2005 en seulement cinq mois à l'aide de tout juste 80 ordinateurs. Au vu des prédictions sur les capacités futures des ordinateurs, un *N* de 1024 bits peut être utilisé pour une information qui sera périmée sous un délai d'un an ou deux, un de 2048 bits est pleinement acceptable pour une utilisation longue, et un de 4096 bits garantit une inviolabilité à peu près définitive.

On a vu dans la section précédente qu'un *E* de petite taille était dangereux, mais il existe également des attaques efficaces contre les *D* de petite taille. Michael Wiener a exposé dans un article de 1990 que le développement en fractions continues de $\frac E {N}$ permettait de découvrir très vite *D* si celui-ci est inférieur à $N^{0,25}$ environ. Les gens ont tendance à choisir de petits *D* parce qu'ils rendent le déchiffrement ou la signature de messages beaucoup plus rapides, mais c'est une erreur : il existe une technique pour maintenir un bon niveau de performances même avec un *D* de grande taille, que nous exposerons dans le chapitre suivant.

En outre, la factorisation de Fermat est d'autant plus efficace que l'un des facteurs est proche de $\sqrt N$, c'est-à-dire, dans notre cas où il n'y a que deux facteurs, que *P* et *Q* ne doivent pas être trop proches, sans quoi la factorisation se voit grandement facilitée. Dans un autre ordre d'idée, si $P - 1$ ou $Q - 1$ se décompose en facteurs premiers exclusivement de petite taille, l'algorithme p-1 de Pollard est capable de factoriser *N* en peu de temps. Il faut donc prêter attention à ces deux faiblesses lorsque l'on génère une clé privée.

Il est enfin une « faiblesse » contre laquelle on ne peut pas grand chose. Si deux personnes utilisent deux *N* différents mais que par hasard ces deux *N* aient un facteur en commun, un simple calcul de PGCD permet de retrouver ce facteur commun et donc le deuxième facteur des deux *N* par division. Là où ça devient plus gênant, c'est qu'en collectant un grand nombre de clés publiques, la probabilité que l'une d'entre elles au moins ait un facteur commun avec la clé que l'on veut attaquer augmente. Et pour accélérer encore le processus, on peut multiplier entre elles toutes les clés publiques connues, donnant un nombre colossal mais encore manipulable, et procéder au test de PGCD avec ce nombre-là : seul le premier calcul sera vraiment long. Le seul moyen de se prémunir contre cette recherche semi-exhaustive est de disposer d'un bon générateur de nombres aléatoires lors de la création des clés privées.

Autres attaques

Il ne reste que trois attaques connues à mentionner.

Premièrement, à cause de la combine visant à améliorer les performances de déchiffrement dont on a parlé il y a peu, il est possible, en étudiant le temps que mettent plusieurs messages donnés à être déchiffrés, de déterminer rapidement la valeur de *D*. Le seul moyen de contrer cette méthode est de s'assurer que le temps de déchiffrement ne soit plus dépendant uniquement de *D*. Pour ce faire, on génère un nombre aléatoire *r*, et on déchiffre non pas *y* mais $r^E \cdot y \ [N]$: le résultat sera $r \cdot x \ [N]$, qui permet très simplement de retrouver *x*. Un même message mettra à chaque fois un temps différent à être déchiffré, rendant impuissante l'attaque par chronométrage.

Deuxièmement, la plupart des processeurs modernes utilisent un système de prédiction de

branchement pour accélérer le traitement des programmes. Dans deux articles parus en 2006, il a été présenté une utilisation possible des erreurs de prédiction de ces systèmes, qui permettrait à un programme de l'espace utilisateur sans permissions particulières de découvrir extrêmement rapidement la clé privée *D* utilisée par un déchiffrement de RSA ayant lieu en parallèle. Toutes les méthodes de protection connues actuellement seraient inefficaces face à ce genre d'attaques. Cela étant, à ma connaissance, cette attaque a été peu étudiée, donc on ne connaît pas encore bien son impact réel, mais on ne sait pas non plus comment s'en protéger.

Troisièmement, il est connu depuis bien longtemps que, si l'on parvient un jour à construire un ordinateur quantique, factoriser les clés publiques utilisées par RSA deviendra un jeu d'enfant. Cependant, ne vous inquiétez pas outre mesure : de tels ordinateurs sont à l'heure actuelle pure spéculation, et s'ils venaient à exister un jour, ils ouvriraient la porte à la cryptographie quantique (déjà théorisée depuis longtemps) à côté de laquelle RSA est aussi sécurisé que du morse.

À présent, vous savez tout ce qu'il y a à savoir sur RSA et son utilisation optimale. Il ne reste plus qu'à vous fournir quelques techniques d'optimisation des lourds calculs nécessaires à la cryptographie asymétrique et vous serez fin prêts à coder votre propre implémentation de RSA ou, mieux, à contribuer aux projets préexistants, comme OpenSSL ou GnuPG.

Mise en pratique

Dans ce chapitre, on va passer à la pratique en essayant de coder un système RSA. Pour commencer, je vous donnerai quelques outils mathématiques supplémentaires bien utiles pour ne pas créer un monstre qui va consommer toute l'énergie de votre PC jusqu'au dernier électron. Ensuite, on passera à la réflexion proprement informatique pour bien organiser le projet. Enfin, je vous proposerai des exemples de solution partielle.

Conseils mathématiques

L'opération la plus consommatrice de mémoire et donc de temps de calcul est l'**exponentiation modulaire**, celle qui consiste à chercher la valeur de *a* puissance *b* modulo *c*. La solution « naïve » pour calculer un tel nombre consiste à faire une série de multiplications pour obtenir la puissance, puis à passer le résultat à la division euclidienne pour trouver le modulo. Et c'est là que cela pêche. Prenons un cas simple : $4^{26} \equiv 233\ [437]$. Il ne met en jeu que des nombres de trois chiffres au maximum et pourtant, le résultat de l'opération puissance, avant qu'on lui applique le modulo, est un nombre à seize chiffres. Imaginez ce que ce serait avec des exposants et des bases à cent ou deux cents chiffres !

Il existe cependant un moyen d'aboutir au résultat en n'utilisant que des nombres de l'ordre de grandeur du modulo ou plus petits. Tout commence avec le raisonnement suivant. Si $y \equiv x\ [N]$ et si $x = x_0 \cdot x_1 \cdot x_2 \cdot x_3$, alors :

$$\begin{aligned} y \ &\equiv x_0 \cdot x_1 \cdot x_2 \cdot x_3 \ [N] \\ &\equiv (x_0 \ [N]) \cdot (x_1 \ [N]) \cdot (x_2 \ [N]) \cdot (x_3 \ [N]) \ [N] \end{aligned}$$

Prenons un cas un peu plus particulier. Comme n'importe quel nombre, l'exposant *E* peut être décomposé en une somme de puissances de 2, ça revient à l'écrire en notation binaire. Par exemple, $26 = 2^4 + 2^3 + 2^1$. De manière plus générale, on peut écrire que $E = \sum_{i = 0}^{n - 1} a_i \cdot 2^i$, avec a_i qui ne peut valoir que 0 ou 1 et *n* la première puissance de 2 supérieure à *E*. De sorte que la fonction de chiffrement peut se décomposer comme suit (on se rappellera que $a^{b_1 + b_2} = a^{b_1} \times a^{b_2}$).

$$\begin{align} y & \equiv x^E \ [N] \equiv x^{\sum_{i = 0}^{n - 1} a_i \cdot 2^i} \ [N] \\ \ & \equiv \begin{bmatrix}\prod_{i = 0}^{n - 1} (x^{2^i})^{a_i}\end{bmatrix} \ [N] \equiv \begin{bmatrix}\prod_{i = 0}^{n - 1} (x^{2^i} \ [N])^{a_i}\end{bmatrix} \ [N] \end{align}$$

En outre, $x^2 \equiv x \cdot x\ [N] \equiv (x\ [N]) \cdot (x\ [N])[N]$, ce qui signifie de manière plus générale que $x^n \equiv (x\ [N]) \cdot (x^{n - 1}\ [N])\ [N]$.

Ainsi, en introduisant le modulo à chaque étape de la multiplication, on réalise plus d'opérations mais avec des nombres beaucoup plus petits. On peut alors définir l'algorithme suivant pour calculer l'exponentiation modulaire $x^E\ [N]$.

[[i]] | 1. On décompose E en puissances de 2 pour obtenir les coefficients a_i. | 2. On pose $y_0 = x_0 = 1$. | 3. $x_i = (x_{i-1})^2$ [N]. | 4. $y_i = y_{i-1} \cdot (x_i)^{a_i}$ [N]. Cette étape peut être sautée si $a_i = 0$, car elle revient à multiplier par 1. | 5. Si $i + 1 < n$, retourner à l'étape 3 avec le i suivant. | 6. Le résultat voulu est le dernier y_i.

Voilà tout pour l'exponentiation modulaire. Je vous avais également parlé d'une combine qui permet de simplifier et donc accélérer le déchiffrement ou la signature par clé privé. Le principe général est d'utiliser le théorème des restes chinois pour utiliser des exposants beaucoup plus petits que D : il faudra faire deux exponentiations modulaires, mais tout mis bout à bout le calcul prendra en fait quatre fois moins de temps environ. Pour l'explication détaillée (en anglais, j'en suis le premier désolé), voyez ce lien.

1. On commence par calculer les valeurs $d_P = D\ [P - 1]$, $d_Q = D\ [Q - 1]$ et Q^{-1} l'inverse modulo P de Q. Pour calculer cette dernière valeur, on utilisera l'algorithme d'Euclide étendu que nous avons vu au chapitre 2 section 3. Celui-ci nous avait servi à calculer D à partir de E et M : ici, nous allons calculer Q^{-1} en utilisant Q à la place de E et P à la place de M.
2. Ensuite on calcule $x_1 = y^{d_P}\ [P]$ et $x_2 = y^{d_Q}\ [Q]$.
3. Enfin, on peut obtenir le message déchiffré avec la formule suivante : $x = x_2 + Q \times (Q^{-1} \cdot (x_1 - x_2)\ [P])$. Si $x_1 < x_2$, pensez à utiliser $(x_1 - x_2 + P)$ à la place de $(x_1 - x_2)$.

Conseils informatiques

2,21 gigowatts

Dans un langage comme le C, les types natifs d'entiers permettent de représenter des nombres sur 64 bits. Certaines instructions en assembleur x86 permettent de monter jusqu'à 128 bits. En tout état de cause, on est très loin des clés de 1024 voire 2048 ou 4096 bits qu'un programme implémentant RSA a besoin de pouvoir manipuler. Il faut donc recourir à une bibliothèque d'**arithmétique multi-précision**.

La plus utilisée en C (et en C++ par l'intermédiaire d'une surcouche) est la **GNU Multiple Precision Arithmetic Library** ou GMP pour les intimes. Ce n'est pas la seule : en C++, la **Class Library for Numbers** ou CLN est nettement moins utilisée mais présente l'avantage d'être codée nativement en C++, plutôt que « d'enrober » du C. Mais objectivement, la GMP domine le paysage informatique. En effet, de nombreux langages implémentent l'arithmétique multi-précision dans leur bibliothèque standard (C#, Java, OCaml, Perl, PHP, etc.) voire nativement dans le langage (Erlang, Haskell, Python, Ruby, etc.) ; cependant, la plupart de ces langages font en réalité appel à GMP de manière transparente pour l'utilisateur.

Cela étant, nous n'expliquerons pas comment fonctionne cette bibliothèque, pour deux raisons. Tout d'abord, c'est une bibliothèque extrêmement puissante, qui nécessiterait un tutoriel à elle seule, et ce ne serait pas lui faire justice que d'en expliquer quelques bribes sur un coin de table. Ensuite, il n'y a pas grand intérêt à créer une implémentation de RSA en C/C++ : la librairie OpenSSL, très largement utilisée par les logiciels ayant besoin de SSL, et la librairie libgcrypt, utilisée surtout par GnuPG, toutes deux écrites en C, font cela bien mieux que vous ou nous n'en serions capable. Si vraiment

vous voulez travailler en C/C++, essayez plutôt de trouver un moyen d'améliorer ces deux bibliothèques, ou un projet voisin (CyaSSL, GnuTLS, PolarSSL…). En revanche, il peut être intéressant de créer des implémentations natives dans d'autres langages, plutôt que de recourir indirectement à ces deux bibliothèques, en particulier pour les langages strictement fonctionnels.

Pensez donc simplement à utiliser des entiers n'ayant pas de limite de taille si le langage ne le fait pas automatiquement à votre place

Y'a écrit quoi, là ?

Le gros inconvénient du chiffrement RSA, c'est qu'une fois chiffré, un simple texte ne ressemble plus à rien, et cela devient excessivement difficile de l'enregistrer dans un fichier ou de le transmettre par mail. De même, les clés publiques et clés privées étant de grands nombres, il n'est pas dit que les interpréter comme du texte donne quoi que ce soit de potable. Ce n'est pas trop gênant pour les conserver, déjà plus pour les transmettre à quelqu'un.

La solution la plus couramment adoptée, et à mon avis la plus simple et la plus économique, consiste à convertir tout ce qui n'est pas du texte brut ou un type de fichier usuel (JPEG, etc.) en base64. Cet encodage consiste à découper les données en tronçons de trois octets soit 24 bits, et à représenter chaque groupe de 6 bits de ce tronçon par un caractère : comme il n'y a plus que 64 possibilités, on peut n'utiliser que des caractères courants. En l'occurrence, les lettres de l'alphabet en minuscule et majuscule, les dix chiffres et les caractères « + », « / » et « = », ou alors, dans une variante destinée à être utilisée dans les URL, « - », « _ » et « = ».

Je vous conseille vivement de l'utiliser aussi.

Mettons de l'ordre

Dans une implémentation complète d'un logiciel utilisant RSA, il y aura nécessairement plusieurs couches. La première correspond aux outils de bas niveau nécessaires pour faire fonctionner les algorithmes : je pense en particulier au générateur de données aléatoires. La deuxième regroupe les fonctions mathématiques qui sont utiles en cryptographie mais qui peuvent avoir d'autres usages : PGCD, exponentiation modulaire, inverse modulaire, etc. La troisième est la couche vraiment cryptographique : on y trouvera les fonctions de hachage, les algorithmes de cryptographie symétrique et l'algorithme RSA proprement dit. La quatrième implémente les outils nécessaires à une utilisation en situation réelle de RSA : génération de clés robustes, lutte contre les attaques par chronométrage, OAEP, stockage de clés, réseau de confiance, etc. La cinquième constitue l'interface utilisateur donnant accès aux quatre couches inférieures. Elle n'est pas indispensable si vous ne voulez pas faire un outil spécifique, par exemple un clavardeur en P2P chiffré, mais simplement une bibliothèque utilisable par d'autres gens.

À partir de là, tout dépend jusqu'à quel point vous voulez réinventer la roue. La plupart des langages ont dans leur bibliothèque standard une implémentation des deux premières couches. Nombre d'entre eux implémentent aussi tout ou partie de la troisième. Et si vous creusez, il y a des chances que quelqu'un ait déjà implémenté la quatrième couche aussi. Dans ce dernier cas, n'abandonnez pas tout de suite : voyez s'il s'agit d'une implémentation native ou seulement d'une interface avec une bibliothèque en C ou équivalent. Vous pourriez avoir tout de même quelque chose à apporter.

En tout état de cause, il peut être judicieux de regrouper les deux premières couches dans une bibliothèque, qui pourra toujours vous servir sur d'autres projets, et les deux suivantes dans une autre bibliothèque, que vous pourrez interfacer à plusieurs logiciels différents.

C'est pourquoi, dans la dernière section de ce tutoriel, je ne donnerai pas une implémentation complète, beaucoup trop longue à programmer et dont le code prendrait beaucoup trop de place, mais des exemples commentés, dans plusieurs langages, de certaines des fonctions à implémenter.

Exemples

Base64 (en PHP)

Un exemple d'implémentation du Base64 en PHP. En soi, ce n'est pas nécessaire puisque le langage possède nativement les fonctions `base64_encode()` et `base64_decode()`. Cependant, l'exemple donné ici a deux intérêts. Tout d'abord, il offre également une paire de fonctions pour le base64 « spécial URL ». Ensuite, il met en évidence la principale difficulté que vous serez amenés à rencontrer si vous voulez implémenter du RSA en PHP : quand un script PHP lit un fichier ou reçoit des arguments par l'URL, ces données sont de type texte, alors que toutes les fonctions mathématiques ou bit-à-bit travaillent sur des nombres, et il est excessivement fastidieux de faire la conversion de l'un à l'autre.

En d'autres termes, PHP est un mauvais langage pour la cryptographie.

```
[[s]] | php | <?php | | $b64tc = str_split("ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/", 1); | // Plus rapide à coder que de taper le tableau à la main. | // b64tc est pour base64_table_caracteres | | function base64($texte) { | | global $b64tc; | | if ($texte == "") return ""; // Pour une chaine vide, pas la peine de suivre toute la fonction. | | $morceaux = str_split($texte, 3); // Le texte d'entrée est découpé en tronçons de trois caractères de long. | $resultat = ""; | | for ($i = 0; $i < count($morceaux); $i++) { | | $caras = str_split($morceaux[$i], 1); // Les caractères individuels, pour pouvoir obtenir la valeur numérique de la chaîne. | // Sans cela, PHP ne nous permet pas d'utiliser les opérateurs bit-à-bit. | | if (strlen($morceaux[$i]) == 3) { // On a un tronçon complet. C'est le cas le plus courant, | // c'est pourquoi on le met en premier, pour économiser des comparaisons inutiles. | $morceaux[$i] = ord($caras[0])*0x10000 + ord($caras[1])*0x100 + ord($caras[2]); // On le convertit en un nombre. | | $bloc1 = ($morceaux[$i] & 0xFC0000) >> 18; | $bloc2 = ($morceaux[$i] & 0x03F000) >> 12; | $bloc3 = ($morceaux[$i] & 0x000FC0) >> 6; // On met à profit les opérateurs bit-à-bit pour | $bloc4 = ($morceaux[$i] & 0x00003F); // obtenir les 4 morceaux du tronçon. | | $resultat .= $b64tc[$bloc1] . $b64tc[$bloc2] . $b64tc[$bloc3] . $b64tc[$bloc4]; | } // Fin du cas principal | | else if (strlen($morceaux[$i]) == 2) { | | $morceaux[$i] = ord($caras[0])*0x100 + ord($caras[1]); // On le convertit en un nombre. | | $bloc1 = ($morceaux[$i] & 0xFC00) >> 10; | $bloc2 = ($morceaux[$i] & 0x03F0) >> 4; | $bloc3 = ($morceaux[$i] & 0x000F) << 2; | | $resultat .= $b64tc[$bloc1] . $b64tc[$bloc2] . $b64tc[$bloc3] . '='; | | } // Fin du premier cas particulier | | else { // Si strlen($morceaux[$i]) == 1, donc. | | $morceaux[$i] = ord($caras[0]); | | $bloc1 = ($morceaux[$i] & 0xFC) >> 2; | $bloc2 = ($morceaux[$i] & 0x03) << 4; | | $resultat .= $b64tc[$bloc1] . $b64tc[$bloc2] . '=='; | | } // Fin du second cas particulier | | } // Fin de la boucle | | return $resultat; | | } // Fin de function base64($texte) | | | | function base64_dechiffrer($texte) { | | global $b64tc; | | $morceaux = str_split($texte, 4); // Le texte d'entrée est découpé en tronçons de quatre caractères de long. | $resultat = ""; | | for ($i =
```

```
0; $i < count($morceaux); $i++) { | $blocs = str_split($morceaux[$i], 1); | | if ($blocs[3] != '=') { // Cas général, on a un tronçon complet. | | $en_numerique = (array_keys($b64tc, $blocs[0])[0] << 18) | (array_keys($b64tc, $blocs[1])[0] << 12) | (array_keys($b64tc, $blocs[2])[0] << 6) | array_keys($b64tc, $blocs[3])[0]; // D'abord on travaille sur des nombres, pour reconstituer la valeur numérique du tronçon de 3 caractères | $resultat .= chr(($en_numerique & 0xFF0000)>>16) . chr(($en_numerique & 0x00FF00)>>8) . chr($en_numerique & 0x0000FF); | // Puis on récupère chaque caractère du tronçon à partir de sa valeur numérique. | } // Fin du cas général | | else if ($blocs[2] != '=') { // Le tronçon n'a que deux caractères. | | $en_numerique = (array_keys($b64tc, $blocs[0])[0] << 10) | (array_keys($b64tc, $blocs[1])[0] << 4) | (array_keys($b64tc, $blocs[2])[0] >> 2); | $resultat .= chr(($en_numerique & 0xFF00)>>8) . chr($en_numerique & 0x00FF); | | } // Fin du premier cas particulier | | else { // Le tronçon n'a qu'un caractère. | | $resultat .= chr((array_keys($b64tc, $blocs[0])[0] << 2) | (array_keys($b64tc, $blocs[1])[0] >> 4)); | | } // Fin du second cas particulier | | } // Fin de la boucle | | return $resultat; | | } // Fin de function base64_dechiffrer($texte) | | | | /****** Mode URL ******/ | // Les deux fonctions suivantes se contentent de remplacer les deux caractères qui diffèrent d'avec le base64 normal. | | function base64_url($texte) { | | return str_replace(array('+', '/'), array('-', '_'), base64($texte)); | | } | | function base64_url_dechiffrer($texte) { | | return base64_dechiffrer(str_replace(array('-', '_'), array('+', '/'), $texte)); | | } | | ?> |
```

Fonction RSA et annexes (en Haskell)

On implémente ici la fonction de chiffrement/déchiffrement de RSA ainsi que deux variantes de la fonction de déchiffrement optimisé (cf. supra « Conseils mathématiques »), accompagnées des trois fonctions mathématiques indispensables pour les mener à bien : l'exponentiation modulaire, l'algorithme d'Euclide étendu et l'inverse modulaire.

Le Haskell est particulièrement adapté ici. Utiliser un langage strictement fonctionnel est une très bonne idée quand on ne s'intéresse pas directement à l'interface utilisateur mais seulement aux calculs qui doivent être faits. En outre, les principales fonctions mathématiques ici présentées utilisent massivement la récursivité et Haskell est très fort à ce jeu-là.

Voyez en outre la concision du code : il y a presque plus de commentaires que de code.

[[s]] | haskell

```
| -- Exponentiation modulaire | -- x = base, e = exposant, n = modulo | expoMod :: Integer -> Integer -> Integer -> Integer | expoMod x e 0 = error "expoMod : Le modulo 0 n'existe pas." | expoMod x 0 n = 1 | expoMod x e n = (base (e `mod` 2) * expoMod ((x*x) `mod` n) (e `div` 2) n) `mod` n | where base 0 = 1 | base 1 = x | | -- Algorithme d'Euclide étendu | -- Pour bien comprendre comment la récursivité est mise en place, il faut bien voir que | -- si b*u' + r*v' = pgcd, a*u + b*v = pgcd et a=b*q + r | -- alors r = a - b*q | -- donc pgcd = b*u' + (a - b*q)*v' = a*v' + b*(u' - q*v') | -- donc u = v' et v = (u' - q*v') | -- ce qui permet d'introduire la récursivité. | -- Pour plus de sécurité, on effectue beaucoup de vérifications sur les paramètres. | euclideExt :: Integer -> Integer -> (Integer, Integer, Integer) | euclideExt a 0 = (1, 0, a) | euclideExt a b | | a < b = euclideExt b a | | a == 0 = error "euclideExt : Au moins un des deux arguments doit être différent de 0." | | a < 0 = euclideExt (-a) b | | b < 0 = euclideExt a (-b) | | otherwise = (v', u' - q*v', pgcd) | where (q, r) = a `quotRem` b | (u', v', pgcd) = euclideExt b r | | -- Inverse modulaire | -- a = base, n = modulo | -- La fonction retourne 0 en cas d'erreur | invMod :: Integer -> Integer -> Integer | invMod a 0 = error "invMod : Le modulo 0 n'existe pas." | invMod a n | | pgcd == 1 = if inv < 0 then inv + n else inv | | otherwise = error "invMod : La base et le modulo doivent être premiers entre eux." | where (_, inv, pgcd) = euclideExt n a |
```

```
| -- L'algorithme de chiffrement, ou le déchiffrement non optimisé, n'est qu'un appel à | -- la fonction d'exponentiation modulaire avec les bons arguments. Il n'y a donc pas | -- lieu de l'implémenter indépendamment | | -- Déchiffrement optimisé avec toutes les infos précalculées | -- y = message chiffré, (p, q, dp, dq, qinv) = clé privée complète | rsaDechiffre :: Integer -> Integer -> Integer -> Integer -> Integer -> Integer -> Integer | rsaDechiffre y p q dp dq qinv = xbis + q * ((qinv * diffxs) `mod` p) | where diffxs | xsemel < xbis = xsemel - xbis + p | otherwise = xsemel - xbis | xsemel = expoMod y dp p | xbis = expoMod y dq q | | -- Déchiffrement semi-optimisé où la clé privée complète doit être calculée | -- y = message chiffré, (p, q, d) = clé privée de base | rsaDechiffre' :: Integer -> Integer -> Integer -> Integer -> Integer | rsaDechiffre' y p q d = rsaDechiffre y p q (d `mod` (p-1)) (d `mod` (q-1)) (invMod q p) |
```

OAEP (en Python)

La fonction présentée ici met en place l'OAEP : si vous ne vous souvenez plus de ce que c'est, courrez lire la section « Bourrage papier » du chapitre précédent. Elle n'agit que dans un sens, à savoir emballer le message pour qu'il respecte le standard, mais elle sait gérer à la fois des textes de la bonne taille et des textes trop long. Pour l'écrire, on a utilisé Python, pour la simplicité du code, mais aussi pour bénéficier des nombreuses fonctions de hachage que le langage implémente dans sa bibliothèque standard.

Si vous voulez vous entraîner, vous pouvez modifier la fonction de manière à ce qu'elle puisse utiliser d'autres fonctions de hachage produisant des condensats de taille différente à ce qui est présenté là. Puis, vous pourriez implémenter une nouvelle fonction qui, en fonction de la longueur de la clé, appelle oaep_base() avec les fonctions de hachage les plus adaptées.

```
[[s]] | py | # -*- coding: utf8 -*- | import os | import hashlib | | def xor_str(s1, s2): | # Effectue un "ou exclusif" entre les deux chaînes | return "".join(chr(ord(a) ^ ord(b)) for a, b in zip(s1, s2)) | | def oaep_base(msg): | if len(msg) <= 64: | # On complète la chaîne avec des "0" jusqu'à ce | # qu'elle ait une longueur de 64 | msg = msg + chr(0) * (64 - len(msg)) | | # 32 octets de données aléatoires | R = os.urandom(256) | | # On chiffre R | G = hashlib.sha512(R) | | # XOR | X = xor_str(msg, str(G)) | | # On chiffre X | H = hashlib.sha256(X) | | # XOR | Y = xor_str(str(H), str(R)) | | return X + Y | else: | # len(msg) = 64 * q + len(msg)%64 | # Donc len(msg) = 64 * (q+1) - 64 + len(msg)%64 | # Donc len(msg) + 64 - len(msg)%64 est un multiple de 64. | msg = msg + chr(0) * (64 - len(msg)%64) | | # Découpage en tronçons de 512 bits i.e. 64 caractères | data = [msg[64*i:64*(i+1)] for i in range(len(msg)/64)] | | # On applique oaep_base à chaque tronçon | data = map(oaep_base, data) | | return data |
```

Voilà, on a fait le tour de la question. Si vous avez des questions ou des remarques, n'hésitez pas à commenter. Par ailleurs, si vous avez des implémentations à proposer dans d'autres langages informatiques, nous serions ravis d'en entendre parler.

Conclusion

Petite note concernant le droit d'auteur.

- Le texte est l'œuvre de Dominus Carnufex et placé sous licence BiPu L. Le logo de même.
- Les images sont l'œuvre de Vayel, utilisent uniquement des images du domaine public comme base, et sont placées sous licence BiPu. Seule la photo du premier chapitre n'est pas de son fait, mais nous ne sommes pas parvenus à retrouver l'auteur original.
- Le code en Python est l'œuvre de Vayel et placé sous licence CeCIll-B.
- Le reste du code est l'œuvre de Dominus Carnufex et placé sous licence CeCIll.

Pour ceux qui ne seraient pas familiers des licences françaises, cela équivaut respectivement et dans les grandes lignes à CC BY-SA-NC, CC 0, licence MIT/licence X11, GPL.

Enfin, sachez que ce tutoriel est la complète réécriture d'un tutoriel écrit par chabotsi, dont vous pouvez lire la première version sur feu le *Site du Zéro*.