Factorisation d'entiers, cryptographie

Jean-Guillaume Dumas[1]

[1]Laboratoire Informatique et Distribution, ENSIMAG - antenne de Montbonnot.
ZIRST - 51, av. Jean Kuntzmann, 38330 Montbonnot Saint Martin.
http://www-id.imag.fr/~jgdumas ; e-mail : Jean-Guillaume.Dumas@imag.fr

Le problème de la cryptographie est de proposer des méthodes pour coder facilement de l'information de telle sorte que le décodage soit difficile si l'on ne possède pas la ***clé*** adéquate. Cet article présente le système de codage à clé publique mis au point par Rivest, Shamir et Adleman (R.S.A. en 1978 [18]) et les moyens de ***casser*** ce système, c'est-à-dire de décoder sans la clé à l'aide de la théorie des nombres.
La première partie présente brièvement le principe des systèmes à clé publique, la deuxième propose les fondements du système RSA et la génération des clés par ***composition de nombres premiers***. Les deux parties suivantes abordent les problèmes de tests de primalité (reconnaître si un nombre donné est premier ou composé) puis de factorisation (***décomposer en facteurs premiers***).

1. Principe des clés publiques

Il faut coder un message *x*, c'est-à-dire qu'il faut produire un nouveau message y à l'aide d'une fonction de codage *f* : *y = f(x)*. Cette fonction *f* doit être construite de sorte que y soit très différent de x et aussi de sorte que le décodage soit possible à l'aide d'une autre fonction g, elle aussi très différente de *f*. *f* et *g* sont appelées les **clés** du système de codage.
Dans les systèmes à clé publique, *f* est publiée dans le monde entier et *g* est secrète, ainsi tout le monde peut envoyer des messages secrets au possesseur de *g*. Seul celui-ci peut les décoder. Le problème du codage revient donc à construire deux fonctions *f* et *g* réciproques l'une de l'autre et telle que *g* soit difficile à trouver même si l'on connaît *f*.

2. Système de codage RSA

L'idée est d'utiliser un entier *m* (le modulo) et de représenter les messages comme des suites d'entiers compris entre *0* et *m-1* : il est possible de représenter tous les caractères alphanumériques d'un clavier d'ordinateur par un nombre compris entre 0 et 255 = 2^8-1. Cette représentation est appelée ASCII. On crée ensuite des groupes de k caractères (les mots) de sorte que 256^k soit juste plus petit que m. C'est une première manière de coder un texte par des nombres. Il faut ensuite crypter ces nombres de telle sorte qu'il soit difficile de retrouver le texte original.

La méthode de Rivest, Shamir et Adleman consiste à utiliser les congruences modulo *m* («*a et b sont congrus modulo m*» signifie que a et b donnent le même reste lorsqu'on les divise par *m* : $a \equiv b\ [m]$ veut dire $\exists\, \alpha \in Z,\ a-b = \alpha\, m$). En effet, pour certains *m*, il est possible de trouver deux entiers *k* et *u*, tels que pour n'importe quel reste *x*, l'élévation à la puissance *k*u* est la fonction identité, c'est-à-dire que par deux calculs successifs de puissance, on retombe sur le reste initial : $(x^k)^u \equiv x^{ku} \equiv x\ [m]$. À partir de cette propriété, on peut alors construire deux fonctions réciproques l'une de l'autre : $f(x) \equiv x^k\ [m]$ et $g(y) \equiv y^u\ [m]$. En outre nous allons voir que même en connaissant *m* et *k*, il est très difficile de calculer *u* si celui-ci est secret : cela donne donc un moyen de crypter des messages.

La section suivante donne les moyens de construire des entiers *m*, *k* et *u* vérifiant cette propriété.

2.1 Principe mathématique

La preuve de l'existence de tels nombres repose sur un résultat de Léonard Euler, établi vers 1760 et généralisant le petit théorème de Fermat :

> *Soit m un entier strictement positif, produit de nombres premiers distincts (m est «sans carré»). Si $\varphi(m)$ est le nombre des entiers a premiers avec m et tels que $0 \leq a < m$, alors pour tout multiple r de $\varphi(m)$ et pour tout entier x, on a $x^{r+1} \equiv x\ [m]$.*

Ainsi, si l'on peut décomposer $r+1$ en deux facteurs différents ($r+1 = k\,u$), on peut alors construire deux fonctions réciproques l'une de l'autre : $f(x) \equiv x^k [m]$ et $g(y) \equiv y^u [m]$. En outre, si $\varphi(m)$ n'est pas connu, nous allons voir qu'il est difficile de trouver *u* seulement en connaissant *k* et *m*.

Commençons par donner une idée plus précise de la valeur de $\varphi(m)$ avec deux exemples :

- Si *m* est premier, alors $\varphi(m) = m - 1$, puisque tous les nombres plus petits que *m* sont alors premiers avec *m* !
- Si *m* est un produit de deux nombres premiers $m = p\,q$, alors seuls les multiples de *p* et les multiples de *q* ne sont pas premiers avec *pq*. Il y en a $p+q-1$, d'où $\varphi(pq) = pq-(p+q-1) = (p-1)(q-1)$.

Peut-on utiliser un de ces deux exemples pour construire des entiers k et u ?

- En fait, le cas *m* est premier n'est pas intéressant pour le codage car si $k\,u = \varphi(m) + 1 = m$, alors, comme m est premier, on ne peut pas décomposer en deux facteurs intéressants : les seuls choix possibles sont *u* vaut *1* et *k* vaut *m* et *u* vaut *m* et *k* vaut *1*.
- Au contraire, si *m* est un produit de deux nombres on choisit alors *k* premier avec $\varphi(m)$: en effet, le théorème de Bézout et l'algorithme d'Euclide étendu nous donnent dans ce cas l'existence de *u* et *v* tels que $u\,k - v\,\varphi(m) = 1$ (voir par exemple [14, sections 4.2.3 et 4.3.2]). On pose alors $r = v\,\varphi(m)$ et l'on a bien réussi à décomposer $r+1$ en deux nombres *k et u* : $u\,k = v\,\varphi(m) + 1$. C'est ce moyen qu'ont choisi Rivest, Shamir et Adleman.

Donnons un petit exemple pour éclaircir la méthode : supposons que l'on veuille construire des clés avec $m = 55 = 5*11$.

- Calculons d'abord $\varphi(55) : (5-1)*(11-1) = 4*10 = 40$.
- Ensuite, choisissons un k premier avec $\varphi(55) = 40$: par exemple $k = 3$.
- L'algorithme d'Euclide étendu nous permet alors de trouver la formule suivante : $(27) * 3 - (2) * 40 = 81 - 80 = 1$. Nous pouvons donc utiliser $r = 80$ et $u = 27$. En effet, quel que soit *x*, compris entre *0* et *54*, on a alors le résultat suivant : $(x^3)^{27} \equiv x\,[55]$!

Maintenant que nos clés sont construites, codons un petit message : par exemple, le mot «bonjour».

- Faisons tout d'abord la correspondance entre les lettres et leur numéro dans l'alphabet : `B-o-n-j-o-u-r` donne `2-15-14-10-15-21-18.`
- Calculons les cubes de ces nombres modulo 55 : $2^3 \equiv 8$ [55]; $15^3 \equiv 225*15 \equiv 5*15 \equiv 75 \equiv 20$ [55]; $14^3 \equiv 49$ [55]; $10^3 \equiv 10$ [55]; $15^3 \equiv 20$ [55]; $21^3 \equiv 21$ [55]; $18^3 \equiv 2$ [55]. Le codage du mot 02151410152118 est donc 08204910202102.
- Pour décoder, il faut calculer tous les puissances $27^{\text{ièmes}}$! Le lecteur vérifiera que l'on retrouve bien $8^{27} \equiv 2$ [55] ; $20^{27} \equiv 15$ [55] ; $49^{27} \equiv 14$ [55] ; $10^{27} \equiv 10$ [55] ; $20^{27} \equiv 15$ [55] ; $21^{27} \equiv 21$ [55] et enfin $2^{27} \equiv 18$ [55].

2.2 Réalisation pratique

Une fois la théorie mathématique mise au point, il reste deux problèmes majeurs pour rendre cette technique utilisable en pratique :

- Combien de temps faut-il pour générer des clés de plusieurs centaines de chiffres, ou encore, combien de temps prend l'algorithme d'Euclide étendu sur des nombres de plusieurs centaines de chiffres ?
- Combien de temps faut-il pour coder (ou décoder), ou encore combien de temps prend le calcul des puissances modulo des nombres de plusieurs centaines de chiffres ?

1. Pour répondre à la première question, il faut tout d'abord être capable de produire des nombres premiers très grands, c'est-à-dire de tester rapidement si un nombre donné est premier ou non. Nous abordons cette question à la section 3. Il faut ensuite faire des calculs de «pgcd» (plus grand commun diviseur, par l'algorithme d'Euclide). Nous pouvons donner les résultats de la figure 1, obtenus à l'aide de GMP, une bibliothèque d'algorithmes écrits en C++ permettant de calculer avec des entiers de taille quelconque [9].

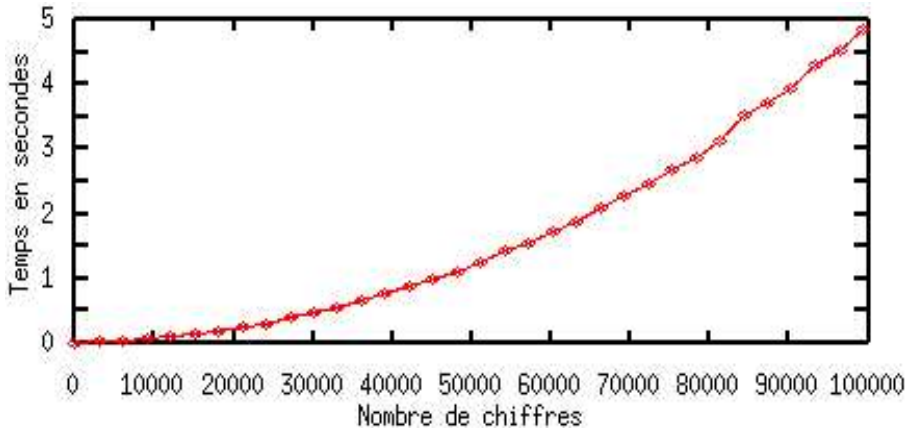

Figure 1: Algorithme d'Euclide sur un Pentium III à 735 MHz avec GMP

Nous constatons que nous pouvons calculer des pgcd de nombres de 100000 chiffres en seulement quelques secondes. Ce calcul est donc extrêmement rapide et ne limitera en rien la génération des clés. En outre, il est possible de prendre des nombres premiers déjà découverts (voir par exemple l'article [6], ou encore la page des nombres premiers : `http://www.utm.edu/research/primes`) et les combiner deux par deux pour obtenir directement le modulo m.

2. Pour le calcul des puissances, il existe de nombreux algorithmes rapides ; Donald Knuth [10, section 4.6.3], par exemple, a étudié ce problème en détails (voir aussi l'encadré 4 de [7]). Nous voyons sur la figure 2 que l'ordre de grandeur obtenu avec l'algorithme des carrés récursifs [14, algorithme 1.6], par exemple, et GMP, est d'environ 1500 chiffres en deux secondes, ou 500 chiffres en un dixième de seconde.

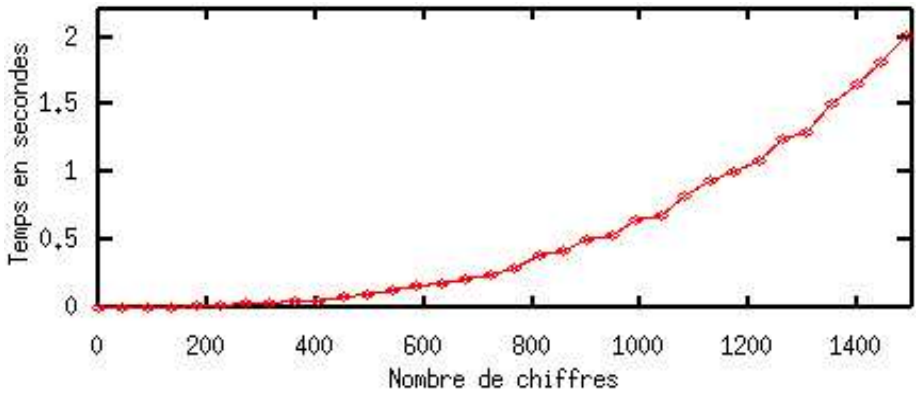

Figure 2: Calcul des puissances sur un Pentium III à 735 MHz avec GMP

C'est un temps relativement important. Cela veut dire qu'il faudrait environ 3 minutes et demie pour coder un fichier d'un mégaoctet (2^{20} chiffres) avec des clés de 500 chiffres et plus de 23 minutes pour le coder avec des clés de 1500 chiffres. Pour que le codage reste praticable, il ne faut donc guère dépasser ce nombre de chiffres pour les clés. Il nous reste à savoir si cet ordre de grandeur est suffisant pour garantir actuellement une bonne fiabilité du codage.

2.3 Comment évaluer la fiabilité du codage ?

Une fois des clés construites (dans notre cas $m = 55$ et $k = 3$ sont publics et $u = 27$ est gardé secret), il faut établir si notre système de cryptage est fiable, c'est-à-dire si u peut rester secret. Pour cela, il faut tout d'abord calculer $\varphi(m)$, puis ensuite u par l'algorithme d'Euclide. Or le seul moyen actuellement connu pour calculer $\varphi(m)$ est de factoriser m, c'est-à-dire de trouver p et q.

Le décodage des codes RSA à clés publiques semble donc se réduire à la factorisation d'entiers[2] !

[2]En fait, casser des codes RSA ne suffit pas pour factoriser les entiers [1] donc les deux problèmes ne sont pas équivalents. D'autre part, il existe peut-être d'autres façons de décoder, mais elles sont inconnues actuellement.

Si pour notre petit exemple (*m = 55*) le problème ne semble donc pas encore trop ardu, il en est tout autrement dès que *m* devient grand (plusieurs centaines de chiffres par exemple), comme nous allons le voir dans la section suivante.

3. Tests de primalité

Nous avons vu que le décodage des codes RSA est équivalent à la factorisation d'entiers. Avant toute factorisation, la première difficulté est de reconnaître les nombres premiers, c'est-à-dire d'avoir un algorithme qui nous permette de décider si un nombre donné est premier ou non.

3.1 Le crible d'Eratosthène

La première idée pour savoir si un nombre est premier est d'essayer de le diviser par tous les nombres plus petits que lui. Malheureusement, cette méthode risque de prendre un temps très important si le nombre à tester est très grand. En outre, la division est une opération qui est très lente pour un ordinateur (parfois 50 fois plus lente qu'une addition par exemple !) et l'on préfèrera donc l'éviter le plus possible lorsque l'on veut aller vite. Néanmoins, cette idée de tester tous les nombres plus petits peut être améliorée pour donner un algorithme relativement efficace en pratique : le crible d'Eratosthène [5,16]. L'algorithme identifie tous les nombres premiers jusqu'à un nombre donné n comme suit :

1. Notez les numéros *1, 2, 3, …, n*. Nous éliminerons les nombres composés (non-premiers) en les marquant. Au commencement, aucun nombre n'est marqué.

2. Marquez le numéro *1* comme spécial (il n'est ni premier ni composé).

3. Initialisez *k* à *1*.
   `Jusqu'à ce que` *k* excède ou égale la racine carrée de n `Faire` :
   - Trouvez le premier nombre *m*, dans la liste, plus grand que *k* et qui n'a pas été marqué (le tout premier nombre ainsi trouvé est 2). Marquez tous ses multiples (en ajoutant *m* jusqu'à ce que l'on dépasse *n*) en tant que nombres composés (dans le premier passage nous marquons ainsi tous les nombres pairs plus grands que *2*, dans le deuxième passage nous marquons tous les multiples de *3* plus grands que *3*, etc.).
   - *m* est premier.
   - Affectez *k* à *m* et recommencez.

4. Tous les nombres restant non-marqués sont aussi premiers.

Les seuls calculs nécessaires pour ce test sont donc un calcul de racine carrée (c'est cher mais on ne le fait qu'une seule fois !) et plusieurs additions (opération la plus rapide en machine). Malheureusement, cet algorithme n'en reste pas moins très lent : il faut déjà environ 1 seconde pour vérifier qu'un nombre de seulement *7* chiffres est premier, 10 secondes pour un nombre de *8* chiffres, etc. ! En outre, il lui faut beaucoup de mémoire pour stocker tous les nombres marqués : environ 10 Mégaoctets pour un nombre à 7 chiffres, 100 pour un nombre à 8 chiffres, etc. Pour

remédier à ce problème de mémoire, on peut alors utiliser l'algorithme avec division, mais nous avons vu que celui-ci sera encore plus lent. Le crible n'est donc pas du tout utilisable en pratique pour factoriser nos nombres de *1000* chiffres puisque que, même sans problème de mémoire, il lui faudrait alors environ 10^{992} secondes (ce qui est un très gros facteur de l'âge de l'univers : 15 milliards d'années $\approx 10^{18}$ secondes !).

3.2 Test probabiliste de Rabin-Miller

Il faut donc chercher d'autres tests de primalité. Le plus célèbre est sans conteste celui de Rabin [17] d'après une idée de Miller [13].
Celui-ci utilise directement le petit théorème de Fermat qui précise que pour tout nombre premier *p*, et pour n'importe quel nombre *a* plus petit que *p* alors $a^{p-1} \equiv 1 \ [p]$.
Au contraire, cela n'est pas toujours vrai si *p* n'est pas premier : pour un nombre composé *p*, environ un quart seulement des nombres *a* vérifient $a^{p-1} \equiv 1 \ [p]$, pour les autres a^{p-1} ne vaut pas *1*.
Une première idée d'algorithme est donc de choisir au hasard un nombre *a* entre *2* et *p*, puis de calculer a^{p-1} modulo *p*. Si le résultat ne vaut pas *1* alors *p n'est pas* premier. Au contraire, si l'on trouve *1* alors soit *p* est premier, soit *p* est composé et on est tombé sur un des nombres *a* qui ne le révèle pas. Donc, quand l'algorithme trouve *1*, *p* est premier avec une probabilité de *1 - 1/4 = 75 %*. Il suffit alors de relancer l'algorithme avec un autre *a* pour augmenter cette probabilité : si l'algorithme exécuté *k* fois trouve un *a* pour lequel la puissance ne vaut pas *1* alors *p* est composé, si l'algorithme a trouvé *1* à chaque fois alors *p* est premier avec une probabilité de $1 - 1/4^k$ (c'est-à-dire que l'on a environ une chance sur un million de milliards que l'algorithme se soit trompé si on l'a lancé *25* fois !). En fait, le test de Miller-Rabin est une amélioration de cette idée qui permet de calculer des puissances un peu moins grandes que *p-1* [10, algorithme P]. La figure 3 montre qu'une itération de ce test nécessite environ 1 seconde pour vérifier qu'un nombre de 1000 chiffres est sans doute premier.

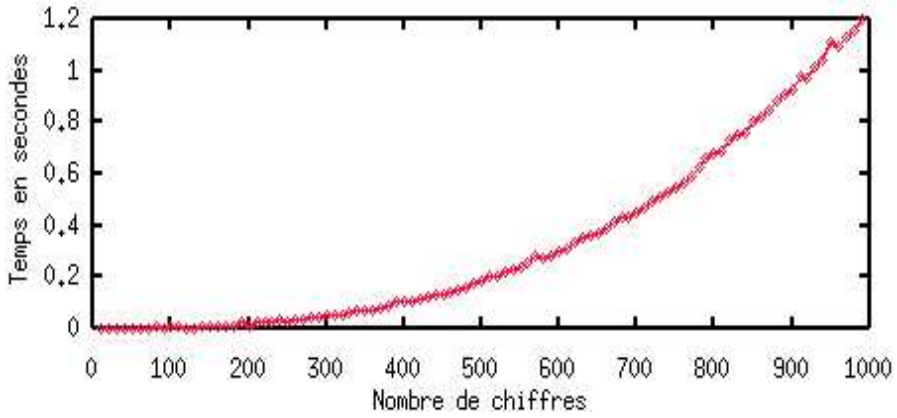

Figure 3: Test de Miller-Rabin sur un Pentium III à 735 MHz avec GMP

Le temps mis par cet algorithme est donc très proche du temps mis pour calculer des puissances modulaires (cf. figure 2 : le test reste raisonnable pour des nombres de 1000 chiffres). Une autre variante, légèrement plus rapide mais avec de moins bonnes probabilités de réussite, est le test de Lehmann [11] sur une idée de Solovay-Strassen [19]. Dans les deux cas, le test de primalité n'est donc pas un problème lorsque l'on veut casser un code RSA. Nous allons voir que la difficulté vient de la factorisation en elle-même.

4. Factorisation

La factorisation d'entiers est un problème qui peut s'exprimer de manière relativement simple mais qui n'a pas, jusqu'à présent, de solution vraiment efficace (nous avons vu que le crible d'Eratosthène, par exemple est inutilisable pour des nombres de plus de *10* chiffres).
De nombreux algorithmes très différents existent, le but de cette section n'est pas de les énumérer tous mais plutôt de proposer ceux qui sont les plus efficaces pour différentes tailles de nombres à factoriser.

4.1 Nombres de quelques chiffres

La première catégorie de nombres à factoriser est celle des « nombres composés de tous les jours », c'est-à-dire les nombres qui ont moins de *20* chiffres. Un algorithme très efficace et est celui de Pollard [15]. L'algorithme ne nécessite que quelques lignes de code (une quarantaine seulement) et est très simple à programmer : l'idée est de calculer une suite du type $u_{k+1} \equiv f(u_k)\ [m]$. En effet, on peut montrer que pour certains écarts fixés entre k_1 et $k_2 = g(k_1)$, alors $u_{k2} - u_{k1}$ est un multiple d'un facteur de m. Il s'agit alors de calculer le pgcd de $u_{g(k)} - u_k$ et de m pour les différentes valeurs de k, celui-ci finira par tomber sur un facteur non trivial de m. Si l'on prend $f(u) = u^2+1$, il faudra de l'ordre de $\sqrt{p}$ itérations si p est le plus petit facteur de m. En pratique, cet algorithme factorise en quelques secondes les nombres de *1* à environ *25* chiffres (avec des facteurs de *12* ou *13* chiffres, cela donne environ 10 millions d'itérations !) et devient très rapidement inutilisable pour des facteurs au-delà de *15* chiffres.

4.2 Nombres de quelques dizaines de chiffres

Pour aller plus loin, la méthode de Lenstra [12] est une solution. Celle-ci utilise des courbes elliptiques (courbes du type $y^2 = x^3+a\,x\,+b$; l'article de J. Buchmann, par exemple, est une bonne introduction [2]) dont l'étude dépasse le cadre de cet article. Néanmoins, sa programmation reste simple (environ 150 lignes de code) et nous pouvons donner quelques idées des propriétés de cet algorithme : il est conjecturé que cet algorithme nécessite un nombre moyen d'opérations de l'ordre de

$$O\left(\ln(m)^2\, e^{\sqrt{2\ln(p)\ln(\ln(p))}}\right).$$

En pratique, cet algorithme factorise en quelques secondes les nombres de *25* à *40* chiffres (avec deux facteurs de tailles semblables). En outre, si l'on a de la chance et que l'on choisit une courbe elliptique particulière, celle-ci peut permettre de factoriser très rapidement : le projet ECMNET (Elliptic Curve Method on the Net) consiste à fournir une l'implémentation de cet algorithme, disponible sur internet [`http://www.loria.fr/~zimmerma/records/ecmnet.html`]. Cet projet a permis de factoriser des nombres avec des facteurs jusqu'à *54* chiffres (record au 3 Juin 2001 : un facteur de 54 chiffres trouvé le 26 décembre 1999 en seulement 454 secondes de calcul sur un Dec Alpha EV6 (21264) cadencé a 500Mhz [`http://www.desargues.univ-lyon1.fr/home/mizony/premiers.html`]).
Le problème est que les bonnes courbes elliptiques varient pour chaque nombre à factoriser et qu'il n'existe pas encore de moyen de trouver la bonne courbe pour un nombre donné. Toutefois, cette rapidité de calcul lorsque l'on a une «bonne» courbe elliptique est à l'origine du programme de factorisation sur internet [`http://www.npac.syr.edu/factoring/overview.html`] : en effet, de nombreux internautes peuvent récupérer le programme ECM et le lancer pour qu'il essaye différentes courbes elliptiques [`http://www.loria.fr/~zimmerma/ecmnet`]. Ainsi, un nombre très important de courbes elliptiques peut être exploré dans le même temps et accélérer potentiellement la recherche de facteurs premiers.

4.3 Le champion du monde

Enfin, le champion actuel pour la factorisation des codes RSA est l'algorithme « Number Field Sieve » [3] ou crible quadratique (là encore, l'article de Buchmann est une bonne introduction [2]), qui, pour factoriser un nombre *m* composé de deux facteurs de tailles respectives similaires, semble nécessiter un nombre moyen d'opérations de l'ordre de

$$O\left(e^{\sqrt[3]{(7.11112)\ln(m)\ln(\ln(m))^2}}\right).$$

Il s'agit de trouver des couples de nombres tels que leurs carrés soient congrus modulo *m* : $x^2 \equiv y^2$ [m]. Alors, $x^2-y^2=(x-y)(x+y)$ est un multiple de *m* et, avec de la chance, l'un de ces deux nombres permet donc de décomposer *m*. Toute la difficulté de l'algorithme consiste à trouver de tels entiers *x* et *y* !
Si l'idée de base est relativement simple, la programmation est un peu plus délicate que pour les précédents algorithmes mais cet algorithme détient les records actuels avec, en particulier, la factorisation, le 22 août 1999, d'une clé RSA de *155* chiffres (*512* bits) [4, `http://www.cwi.nl/cwi/ publications/reports/abs/MAS-R0007.html`]. Le temps de calcul nécessaire pour cette dernière factorisation a été gigantesque : *7* mois de calcul sur *300* PC et stations de travail repartis dans *12* sites et *6* pays, une machine SGI origin 2000 et un Cray C196 !!! Pour des nombres spéciaux, cet algorithme est même parvenu en novembre 2000 à factoriser un nombre de *233* chiffres [`ftp://ftp.cwi.nl/pub/herman/SNFSrecords`] en *5* mois sur *150* stations SGI et Sun cadencées entre 180 et 450 MHz, et sur environ *100* PC cadencés entre 266 et 600 MHz.

5. Conclusion : des clés robustes ?

Nous avons donc vu que les meilleurs algorithmes de factorisation actuels peuvent factoriser des nombres jusqu'à 230 chiffres en plusieurs mois de calculs avec quelques centaines de machines. Cela semble être la limite actuelle de calcul. De plus, cette limite est régulièrement reculée : ainsi, entre 1999 et 2000, le record de factorisation pour les clés RSA est passé de 140 à 155 chiffres. Cette progression est due pour partie à la progression des vitesses des machines utilisées (ce facteur n'est pas vraiment gênant puisqu'il permet aussi la génération des clés plus grandes) mais aussi et surtout grâce à des progrès théoriques et pratiques pour améliorer à la fois les algorithmes de factorisation et leurs implémentations. Quant à savoir quelle taille de clé est nécessaire pour les années à venir, il est bien sûr difficile de répondre de manière certaine. Toutefois, pour les nombres à 1000 chiffres, nous avons vu qu'il était raisonnablement possible de les utiliser pour les clés et il semble que la factorisation de tels «monstres» restera encore impossible par les méthodes connues à ce jour avant plusieurs années.
Quant aux limites légales, le décret n°99-199 du 17 mars 1999 précise que les clés de chiffrement autorisées en France ne doivent pas dépasser *128* bits, c'est-à-dire environ *39* chiffres. Ces clés n'offrent donc qu'une sécurité extrêmement relative lorsque que l'on sait qu'il ne faut que quelques secondes pour les factoriser avec l'algorithme des courbes elliptiques !

6. Bibliographie

6.1 Livres et articles généraux

[2] Johannes Buchmann. La factorisation des grands nombres. *Pour la Science*, (251), septembre 1998. `http://www.pourlascience.com/numeros/pls-251/art-12.htm`.

[5] Henri Cohen. Les nombres premiers. *La Recherche*, (278), juillet 1995.

[6] Jean-Paul Delahaye. Les chasseurs de nombres premiers. *Pour la Science*, (258), avril 1999. `http://www.pourlascience.com/numeros/pls-258/logique.htm`.

[7] Jean-Paul Delahaye. La cryptographie RSA vingt ans après. *Pour la Science*, (267), janvier 2000. `http://www.pourlascience.com/numeros/pls-267/logique.htm`.

[8] Michel Demazure. *Cours d'alg*èbre. Primalité, Divisibilité, Codes, volume XIII de *Nouvelle bibliothè*que Mathématique. Cassini, Paris, 1997.

[10] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 de *The Art of Computer Programming*. Addison-Wesley, Reading, MA, USA, troisième édition, 1997.

[14] Jean-Marie Monier. *Alg*èbre I : Cours et 600 exercices corrigés. J'intègre. Dunod, Paris, seconde édition, 2000.

[16] Carl Pomerance. La recherche des nombres premiers. *Pour la Science*, (64), février 1983.

[20] Philip Zimmermann. Cryptographie et réseau. *Pour la Science*, (260), juin 1999. `http://www.pourlascience.com/numeros/pls-260/art-2.htm`.

6.2 Pour aller plus loin

[1] Dan Boneh et Ramarathnam Venkatesan. Breaking RSA may not be equivalent to factoring. Dans *Advances in Cryptology – EUROCRYPT* '98, volume 1403, pages 59–71, 1998. `http://theory.stanford.edu/~dabo/papers/no_rsa_red.ps.gz`.

[3] J. P. Buhler, H. W. Lenstra, et Carl Pomerance. *Factoring integers with the number field sieve*, volume 1554 de *Lecture Notes in Mathematics*, pages 50–94. Springer-Verlag, 1993.

[4] Stefania Cavallar, Bruce Dodson, Arjen K. Lenstra, Walter Lioen, Peter L. Montgomery, Brian Murphy, Herman te Riele, Karen Aardal, Jeff Gilchrist, Gérard Guillerm, Paul Leyland, Joël Marchand, François Morain, Alec Muffett, Chris Putnam, Craig Putnam, et Paul Zimmermann. Factorization of a 512-bit rsa modulus. Dans Bart Preneel, éditeur, *Advances in cryptology: EUROCRYPT* 2000: International Conference on the Theory and Application of Cryptographic Techniques, Bruges, Belgium, May 14–18, 2000: proceedings, volume 1807 de *Lecture Notes in Computer Science*, pages xiii + 608, New York, NY, USA, 2000. Springer-Verlag Inc.
`http://www.cwi.nl/cwi/publications/reports/abs/MAS-R0007.html`.

[9] Torbjörn Granlund. *The GNU* multiple precision arithmetic library, 2000. Version 3.0, `http://www.gnu.org/gnulist/production/gnump.html`.

[11] Daniel J. Lehmann. On primality tests. *SIAM Journal on Computing*, 11(2), mai 1982.

[12] Hendrik Lenstra. Factoring integers with elliptic curves. *The Annals of Mathematics*, 126(3):649–673, novembre 1987.

[13] Gary L. Miller. Riemann's hypothesis and tests for primality. Dans *Conference Record of Seventh Annual ACM* Symposium on Theory of Computation, pages 234–239, Albuquerque, New Mexico, mai 1975.

[15] John M. Pollard. A Monte Carlo method for factorization. *BIT Numerical Mathematics*, 15(3):331–334, 1975.

[17] Michaël O. Rabin. A probabilistic algorithm for testing primality. *Journal of Number Theory*, 12, 1980.

[18] Ronald L. Rivest, Adi Shamir, et Leonard Adleman. A method for obtaining digital signature and public-key cryptosystems. *Communication of the ACM*, 21(2), 1978.

[19] Robert M. Solovay et Volker Strassen. A fast Monte-Carlo test for primality. *SIAM Journal on Computing*, 6(1):84–85, mars 1977.