Des Trappes dans les Clés

Éric Wegrzynowski

Université des Sciences et Technologies de Lille
Laboratoire d'Informatique Fondamentale de Lille
F-59655 Villeneuve d'Ascq Cedex
Eric.Wegrzynowski@lifl.fr

Résumé On mesure souvent la sécurité d'un système cryptographique à l'aide de la taille des clés utilisées. Cette mesure est-elle un véritable indice de sécurité ? Qu'elles soient publiques ou secrètes, des clés peuvent présenter des faiblesses volontaires ou non. Das le premier cas, les faiblesses peuvent provenir d'un mauvais générateur d'alea. Dans le second cas on parle de trappe.

1 Introduction

Jamais il n'a été autant fait usage de cryptographie qu'aujourd'hui. Le développement des technologies de l'information et de communication a accru la nécessité de protéger les données. Diplomates et militaires, autrefois principaux utilisateurs, partagent maintenant ce besoin avec les entreprises et le grand public (cartes bancaires, sites Internet sécurisés,).

L'une des fonctions de la cryptographie (et historiquement la première) est d'assurer la confidentialité des données et des transactions. À cette fin, de nombreux systèmes de chiffrement ont été conçus, qui rendent inaccessibles l'information à toute personne non autorisée.

Conformément au principe énoncé en 1883 par Auguste Kerckhoffs, la sécurité d'un système de chiffrement ne doit pas reposer sur le secret de sa procédure, mais uniquement sur le secret d'un paramètre utilisé à chacune de ses mises en œuvre, paramètre appelé *clé*. Il est à noter que la plupart des systèmes de chiffrement utilisés de nos jours (du moins ceux à usage civil) respectent ce principe, les algorithmes les définissant étant publics (le cas du chiffrement utilisé dans le protocole GSM et celui du système de protection des œuvres cinématographiques sur DVD sont des exceptions récentes, qui ont connu le succès que l'on sait). L'élaboration de standards cryptographiques fait même appel à des évaluations publiques (c'est le cas de l'AES ([1]) aux Etats-Unis, et du projet NESSIE ([5]) en Europe).

La sécurité des systèmes de chiffrement repose donc d'une part sur la robustesse des algorithmes sous-jacents, leur résistance aux différentes attaques connues (cryptanalyse différentielle, linéaire, ...), et d'autre part sur la taille de l'espace des clés et la qualité de leur génération. C'est essentiellement à ce

second aspect que nous nous intéresserons ici, en laissant de côté le problème de l'hypothétique faiblesse des algorithmes[1].

Après avoir évoqué l'importance d'utiliser un bon générateur d'alea, nous évoquerons la possibilité de concevoir des générateurs de clés délibérément biaisés. Nous montrerons sur des exemples comment on peut construire de tels générateurs aussi bien pour des systèmes de chiffrement à clé secrètes que des systèmes à clé publique, permettant à leurs auteurs de retrouver (assez) facilement les clés produites.

2 Entropie des clés

On distingue deux familles de systèmes de chiffrement. Les systèmes symétriques, ou à clés secrètes, avec lesquels la même clé est utilisée aussi bien pour le chiffrement que pour le déchiffrement, et les systèmes asymétriques, encore appelés à clés publiques, qui se distinguent des systèmes précédents par l'utilisation de deux clés différentes : l'une, publique, sert au chiffrement, l'autre, privée, au déchiffrement.

Les clés des systèmes symétriques sont des chaînes binaires plus ou moins longues (56 bits pour le DES, 128 à 256 bits pour l'AES, 40 à 2048 bits pour RC4). Le plus souvent aucune contrainte n'est imposée pour ces clés : toute chaîne binaire peut convenir, et en choisir une peut se ramener à tirer à pile ou face autant de fois que le nombre de bits de la clé.

En revanche, dans le cas des systèmes asymétriques, les clés, même si on exprime aussi leur taille en bits, ne sont jamais des chaînes binaires quelconques. Au contraire elles possèdent une structure mathématique telle que la clé privée puisse déchiffrer toute information chiffrée avec la clé publique correspondante. Leur génération ne peut donc pas se limiter à une succession de tirages à pile ou face.

2.1 Cas des systèmes symétriques

Un système de chiffrement symétrique se doit d'offrir un espace de clés suffisamment vaste pour éviter une attaque par recherche exhaustive de la clé utilisée. De telles attaques ont été menées avec succès ces dernières années contre des systèmes utilisant de "petites" clés. Citons le cas des deux étudiants de l'école Polytechnique, parvenus en 1995 en utilisant les ordinateurs de l'école à mener à bien une recherche exhaustive dans un espace de clés de 40 bits (système RC4 utilisé dans Netscape) en une trentaine d'heures. Mentionnons aussi, l'attaque menée contre le DES avec un espace de clés de 56 bits par l'Electronic Frontier Foundation qui a construit en 1997, pour un coût d'environ 200.000 dollars, une machine (EFF DES Cracker ([3])) capable d'examiner plus de 92 milliards de clés à la seconde et parcourant donc l'espace des clés en un peu plus de 4 jours en moyenne.

[1] D'autres facteurs interviennent aussi dans la sécurité que nous n'envisagerons pas ici (implémentations, matériels, utilisateurs...)

Il est communément admis aujourd'hui qu'afin d'échapper à une recherche exhaustive, il faut utiliser des clés de 80 bits au minimum (l'AES propose des clés de 128, 192 ou 256 bits).

2.2 Cas des systèmes asymétriques

L'espace des clés d'un système asymétrique est plus complexe que celui d'un système symétrique, car les clés doivent posséder une structure mathématique bien précise. Les systèmes asymétriques reposent sur la notion de fonctions à sens uniques (dont l'existence n'a pas encore été prouvée), c'est-à-dire de fonctions faciles à calculer, mais calculatoirement difficiles à inverser. Ainsi, s'il est facile de multiplier deux nombres entiers, il s'avère souvent très difficile de factoriser un nombre entier. Par exemple, le calul du produit des deux nombres de 78 chiffres

$$\begin{array}{c} 102639592829741105772054196573991675900 7 \\ 165678080380668033419335217907113077 79 \\ \times \\ 106603488380168454820927220360012878679 2 \\ 079585759892915222706082371930628086 43 \\ = \\ 109417386415705274218097073220403576120 0 \\ 373294544920599091384213147634998428893 4 \\ 784717997257891267332497625752899781833 7 \\ 970765372440271467435315933543338 97 \end{array} \tag{1}$$

est quasiment instantané sur un ordinateur aujourd'hui, en revanche la factorisation du nombre de 155 chiffres ainsi obtenu a demandé en 1999 un calcul distribué sur 300 ordinateurs pendant plusieurs mois ([6]).

Pour le système RSA, la fonction à sens unique utilisée est l'exponentiation modulaire. Rappelons rapidement le fonctionnement de RSA.

La partie publique d'une paire de clés RSA est la donnée d'un nombre entier n (le *modulus*) produit de deux nombres premiers p et q distincts (non publics), et d'un nombre e premier avec $\varphi(n) = (p-1)(q-1)$ (i.e. $pgcd(e, \varphi(n)) = 1$). La partie privée d est l'inverse de e modulo n, (i.e. $d = e^{-1} \pmod{n}$).

Pour chiffrer un message m (supposé codé sous forme d'un nombre inférieur à n) avec la clé publique (n, e) du destinataire, l'expéditeur calcule le nombre $c = m^e \pmod{n}$. Des algorithmes efficaces de calculs d'exponentiation modulaire (*square and multiply*) font de l'opération de chiffrement une fonction facile à calculer. En revanche, le calcul de m en connaissant c, n et e uniquement est très difficile[2].

Le destinataire peut déchiffrer le message qu'il reçoit grâce à sa clé privée d en effectuant le calcul $c^d \pmod{n}$. Sa clé privée lui ouvre donc une porte dérobée permettant une inversion facile de l'exponentiation effectuée au chiffrement.

[2] du moins dans les connaissances actuelles

Pour qu'un système asymétrique résiste aux attaquants, il faut que le calcul de la clé privée à partir de la seule connaissance de la clé publique soit lui aussi difficile. Dans le cas de RSA, cela implique une difficulté de factoriser l'entier n, car si un attaquant parvient à factoriser n, il est alors en mesure de calculer la clé privée. Compte tenu des derniers records de factorisation (cf ci-dessus), on considère que l'entier n doit être d'une taille d'au moins 1024 bits (chacun de ses facteurs premiers ayant au moins 512 bits).

2.3 Des clés faibles pour RSA

En 1990, Michael Wiener ([8]) a découvert que certaines clés RSA sont faibles.

Les opérations effectuées par le chiffreur ne sont pas exactement les mêmes que celles effectuées par le déchiffreur. En effet, si elles sont de même nature (exponentiation modulaire), elles ne se font pas avec le même exposant (e pour le chiffrement, d pour le déchiffrement). La quantité de calcul à effectuer croît avec l'exposant. Un débat a eu lieu qui discutait de savoir lequel du chiffreur ou du déchiffreur devait effectuer le plus de calcul. Autrement dit, lequel des deux exposants e et d doit être le plus petit.

La découverte de M. Wiener consiste en une procédure qui permet de calculer la clé privée à partir de la clé publique, à condition que la partie privée soit inférieure à la racine quatrième du modulus de la clé publique (i.e. comprend quatre fois moins de chiffres que n). Cette procédure, s'appuyant sur le développement en fractions continues de e/n, est très efficace (cf une implémentation en MAPLE dans l'annexe A). Cette découverte a mis un terme au débat : la clé privée doit être grande.

2.4 Entropie

La taille des clés des systèmes de chiffrement est une condition nécessaire de sécurité, mais certainement non suffisante. Ceux-ci sont souvent accompagnés de générateurs de clés, et la sécurité de l'ensemble du système dépend aussi de la qualité de ce dernier, en particulier son entropie.

L'entropie d'un générateur est égal au nombre moyen de questions binaires (questions à réponse oui/non) qu'il faut poser pour retrouver une clé. On peut montrer que l'entropie est maximale lorsque la distribution de probabilités sur l'espace des clés induite par le générateur est uniforme. Et dans le cas de clés de systèmes symétriques, cette entropie maximale coïncide avec la taille des clés.

Un système se trouve affaiblit s'il utilise un générateur d'entropie non maximale. Cela peut se produire dans le cas où la génération de clés s'appuie sur un mot de passe, car l'entropie du générateur est alors égale à l'entropie de l'espace des mots de passe, le plus souvent bien inférieure à ce que laisse espérer la taille des clés. Un générateur peut voir son entropie diminuée par l'utilisation d'un mauvais générateur de nombres aléatoires qui induit une distribution non uniforme sur l'espace des clés.

Ces deux cas d'affaiblissement de l'entropie du générateur ne sont pas dûs à une intention des concepteurs et/ou programmeurs. Ils sont conséquence du

mode de fonctionnement (cas de l'utilisation de mots de passe) ou du défaut du générateur d'alea utilisé, défaut qui peut très bien ne pas être connu lors de la conception. Il s'agit alors d'une faiblesse non intentionnelle.

En revanche, on peut imaginer qu'un producteur envisage délibérément d'introduire des faiblesses dans ses systèmes cryptographiques afin de pouvoir prendre connaissance de toute information chiffrée avec eux. Un exemple d'introduction de telle faiblesse a été révélée il y a quelques années, lorsque la presse a dévoilé que la NSA avait introduit dans les systèmes de chiffrement commercialisés par la société suisse CryptoAG un mécanisme qui révélait la clé utilisé dans le chiffrement.

Il est possible de concevoir des générateurs de clés dont l'entropie est considérablement affaiblie de telle sorte qu'elle rend envisageable une recherche exhaustive ou bien un moyen de calcul de la clé. Nous appelons *générateurs à trappe* ces générateurs intentionnellement affaiblis. Nous en présentons deux exemples.

3 Trappes dans les clés secrètes

Nous allons illustrer sur un exemple un générateur de clés secrètes dont l'entropie est bien inférieure à la taille des clés qu'il engendre. Ce générateur ne couvre qu'une infime partie de l'espace de toutes les clés, et offre la possibilité de mener une recherche exhaustive de la clé utilisé pour chiffrer un message à qui en connaît le principe (i.e. la trappe qu'il contient).

Considérons l'espace de toutes les clés de n bits. En définition l'opération d'addition des clés comme le ou-exclusif bit à bit, icet espace possède une structure algébrique d'espace vectoriel sur le corps à deux éléments $\mathbb{F}_2 = \{0, 1\}$ de dimension est n.

Le générateur annoncé n'engendre que des clés d'un sous–espace vectoriel de dimension $k < n$ (autrement dit un $[n, k]-$code). Pour cela, il suffit de fixer une fois pour toute k clés $\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_k$, linéairement indépendantes. Pour engendrer une clé $\mathbf{c}$ de n bits, il suffit de tirer au sort k bits $b_1, b_2, \ldots, b_k$ et de calculer la combinaison linéaire $\mathbf{c} = \sum_{i=1}^{k} b_i.\mathbf{c}_i$.

Par exemple, prenons $n = 10$ et $k = 4$. Fixons les quatre clés linéairement indépendantes

$$\mathbf{c}_1 = \texttt{100011001}, \mathbf{c}_2 = \texttt{010001011}, \mathbf{c}_3 = \texttt{001010101}, \mathbf{c}_4 = \texttt{000101100}$$

En supposant que les quatre bits tirés au hasard sont $b_1 = 0, b_2 = 1, b_3 = 1, b_4 = 1$, la clé produite par le générateur est

$$\begin{aligned} b_1.\mathbf{c}_1 + b_2.\mathbf{c}_2 + b_3.\mathbf{c}_3 + b_4.\mathbf{c}_4 &= \texttt{010001011} + \texttt{001010101} + \texttt{000101100} \\ &= \texttt{011110010} \end{aligned}$$

Ce générateur n'engendre seulement que 2^k clés de taille n sur les 2^n possibles normalement. Si les coefficients b_i sont tirés avec une distribution uniforme, l'entropie du générateur est de k bits au lieu de n. Si $k \approx 50$, cette différence peut

être exploitée pour quiconque connaît la base fixée, avec un effort comparable à celui menée par l'EFF contre le DES.

La trappe introduite dans ce générateur n'amène–t–elle pas des biais statistiques détectables ? Avec Vincent Benony à l'université de Lille, nous avons construit un tel générateur pour les paramètres $n = 128$ et $k = 50$ en choisissant correctement les k clés de base fixées. Nous avons généré un échantillon de 2^{25} clés, et effectué quelques tests statistiques classiques (distribution des poids (nombre de 1 dans les clés), répartition des clés octet par octet (une clé de 128 bits est constituée de 16 octets), collisions (répétition d'une clé dans l'échantillon)). Aucun d'eux n'a révélé le biais (cf quelques résultats à l'annexe B).

Cependant, il existe un moyen efficace de détecter le biais en exploitant la structure algébrique de l'espace des clés. En effet, si on génère $N > n$ clés de taille n avec un générateur d'entropie maximale, la probabilité que la matrice $N \times n$ obtenue avec une clé par ligne a un rang maximal (égal à n) avec une probabilité p égale à ([7])

$$p = \prod_{i=0}^{n-1}(1 - 2^{i-N}) > 1 - 2^{n-N} + 2^{-N}$$

On voit donc que la probabilité de ne pas avoir une matrice de rang maximal décroît exponentiellement avec N (avec $N = n + 10$, cette probabilité est inférieure à $\frac{1}{1000}$).

Si nous utilisons un générateur biaisé tel que celui décrit ci–dessus, il suffira de calculer le rang de matrices obtenues avec un peu plus de n clés pour constater qu'elles ne sont pas de rang n et ainsi détecter le biais.

Afin d'échapper à ce test du rang, on modifie simplement le générateur. Juste avant qu'il ne retourne la combinaison linéaire des clés de base, on la modifie en lui ajoutant un vecteur bruit choisi au hasard de poids limité t. Avec l'exemple précédent ($n = 10$, $k = 4$), on calcule un vecteur bruit de poids maximal $t = 1$. Si on obtient par exemple $\mathbf{e} = \texttt{000010000}$, le générateur retoune la clé

$$\mathbf{c}' = \mathbf{c} + \mathbf{e} = \texttt{011110010} + \texttt{000010000} = \texttt{011100010}$$

L'introduction du bruit fait que l'espace des clés produites par ce générateur ainsi modifié, n'est plus un code linéaire (car plus nécessairement stable par addition des clés), mais le code engendré par cet espace est le code de dimension n, c'est à dire l'espace entier des clés de n bits. Ce qui explique que le test du rang ne puisse plus s'appliquer.

Le nombre de clés pouvant être produites par ce générateur dépend d'un paramètre lié au code sous-jacent : la distance minimale d séparant deux clés du code ([4]). Si celle-ci est est supérieure à $2t + 1$, alors chaque clé ne peut être produite que d'une seule façon, et le nombre de clés est égal à $2^k . \sum_{i=0}^{t} \binom{n}{i}$. On en déduit que si la distribution de probabilité sur les vecteurs bruits est uniforme, l'entropie du générateur est alors égale à $k + \log_2 \sum_{i=0}^{t} \binom{n}{i}$ bits.

Nous avons repris notre générateur étudié ci-dessus ($n = 128$ et $k = 50$), et nous avons appliqué les mêmes tests avec $t = 1$ et $t = 2$ (l'entropie du générateur est alors de $57,01$ et 63.01 bits respectivement). Les résultats de ces tests n'ont rien révélé non plus.

4 Trappes dans les clés publiques

Il semble bien plus difficile, voire impossible, de produire un générateur de clés publiques contenant une trappe permettant de calculer la clé privée correspondante. Il n'en est rien comme l'ont mis en évidence Claude Crépeau et Alain Slakmon ([2]) à propos des clés RSA.

Le générateur qu'ils ont conçu s'appuie sur les clés RSA faibles révélées par M. Wiener ([8]), c'est–à–dire celles dont la partie privée est trop petite. Ces clés faibles sont de probabilité trop faible pour être obtenues par un générateur non biaisé. Produire de telles clés serait ainsi facilement repérable. L'idée de Crépeau et Slakmon consiste à masquer des clés faibles pour les transformer en clés d'apparence ordinaire.

La trappe de ce générateur est tout simplement un entier pair fixé M dont la taille est celle du modulus n de la clé à engendrer. La génération d'une paire de clés RSA consiste à construire une paire de clés faibles $\{(n, e_1), d_1\}$ (vérifiant donc $d_1 < \sqrt[4]{n}$) telle que $e = e_1 + M$ soit inversible modulo $\varphi(n)$. Une fois une telle paire trouvée, le générateur produit la paire clé $\{(n, e), d\}$ (avec $d = e^{-1} \pmod{\varphi(n)}$) dans laquelle, avec une probabilité très forte, l'entier d est du même ordre que n.

Toute personne connaissant la trappe M est en mesure de calculer d à partir de n et e. En effet, il est facile de retrouver $e_1 = e - M$ et d_1 avec la technique de Wiener. Une fois connue une paire d'exposants public/privé, il est possible de factoriser le modulus n grâce à un algorithme probabiliste. La factorisation de n établie, il ne reste plus qu'à calculer l'exposant privé d correspondant à l'exposant public e (cf une réalisation de ce générateur et de l'utilisation de la trappe en MAPLE dans l'annexe C).

Ce générateur est donc bien la preuve qu'il est possible d'introduire une trappe même dans des clés aussi structurée que des clés RSA. Existe-t-il un moyen de détecter la trappe contenue dans ce générateur ?

La réponse affirmative a été trouvée par Serge Vaudenay en étudiant la distribution des réductions de e modulo un petit nombre premier qui divise $\varphi(n)$ (cette étude est tout à fait réalisable, étant donné que celui qui génère la paire de clés peut connaître la factorisation de n). Cette distribution se distingue de celle obtenue avec un générateur ordinaire.

5 Conclusion

Aussi solides que peuvent être les algorithmes d'un système de chiffrement, celui–ci peut toujours être fragilisé par une production de mauvaises clés qui

peut être involontaire (clés faibles, génération fondée sur un mot de passe, mauvais générateur d'alea), ou au contraire totalement délibéré (introduction d'une trappe dans le générateur).

Nous avons donné deux exemples de générateur à trappe, l'un pour le cas de clés de systèmes symétriques, l'autre pour le système RSA. Il y en a certainement beaucoup d'autres.

La possibilité de produire de tels générateurs est inquiétante pour les utilisateurs. Elle est un argument supplémentaire pour l'ouverture des codes des programmes informatiques, surtout ceux touchant à la sécurité des systèmes d'information.

Références

1. AES. Advanced Encryption Standard. `http ://csrc.nist.gov/CryptoToolkit/aes`.
2. Claude Crépeau and Alain Slakmon. The hidden small exponent method for generating pseudo-rsa keys. `http ://crypto.cs.mcgill.ca/~crepeau/RSA`.
3. Electronic Frontier Foundation. *Cracking DES*. O'Reilly, 1998. `http ://www.eff.org`.
4. F.J. MacWilliams and N.J.A. Sloane. *The Theory of Error-Correcting Codes*. North-Holland, 1977.
5. NESSIE. New European Schemes for Signatures, Integrity, and Encryption. `http ://www.cryptonessie.org`.
6. RSA155. Factorization of a 512-bits rsa key using the number field sieve. `http ://www.inria.fr/actualites/1999/rsa155.html`, 1999.
7. Antoine Valembois. *Décodage, Détection et Reconnaissance des Codes Linéaires Binaires*. PhD thesis, Université de Limoges, 2000.
8. Michael J. Wiener. Cryptanalysis of short RSA secret exponents. *IEEE Transaction on information theory*, 36(3), may 1990.

A La procédure de Wiener contre les clés faibles RSA

Afin de faciliter sa tâche dans les opérations de déchiffrement (ou bien de signature), un titulaire d'une paire de clés RSA peut souhaiter disposer d'un petit exposant privé d. En 1990, Michael Wiener a montré comment il est possible de calculer d en connaissant n et e seulement [8].

Cette attaque repose sur le développement en fraction continue du rationnel e/n. Si d ne dépasse pas la racine quatrième de n environ (c'est-à-dire si d a quatre fois moins de chiffres que n), alors l'une des réduites du développement en fraction continue admet d pour dénominateur.

Voici un exemple traité en quelques secondes avec le logiciel MAPLE

```
# La clé publique est
> n := 1156874224948872299434900543036278825944114000890338816593 89\
        4582308391354154424869443317502827070573513839050173198155\
```

```
        18219733441860767640867004225260526772403436953261919442222\
        34070409320176244470672676087861485750413182473515607668644\
        45038494073721266549156734426375138934805688512598474499433\
        98392503648667529:

  e := 25721244518021878749600346452936422514825173496607690947210 4\
        58499822530829079073321948517571476394495871823168636529622\
        92081764108064486119179289022290076518772648688080658257854\
        12977269578183779400097761734183260692804643357850464093272\
        06784208938537319946762765511360042518180491580881837672777\
        1725313111888425:

# Développement en fraction continue de e/n et calcul des réduites
> convert(e/n,confrac,'reduites'):

# Chiffrement d'un message quelconque avec la clé publique
> M := 12345: C := M&^e mod n:

# Recherche de la réduite dont le dénominateur est d,
> i := 1:
> while C&^denom(reduites[i]) mod n <> M do i := i+1: od:

# La clé privée est
> d := denom(reduites[i]);

              502667943040009588806682046558996550593

# Vérification
> m := rand(0..n)():
> c := m&^e mod n:
> m - (c&^d mod n);

                                    0
```

B Un générateur de clés de 128 bits à trappe

Générateur sans bruit : La figure 1 ci-dessous montre la distribution des poids (nombre de bits à 1) dans un échantillon de 2^{25} clés obtenues avec une distribution uniforme dans un code de longueur 128 et de dimension 50. Cette répartition est tout à fait conforme à la distribution théorique (loi binomiale $\mathcal{B}(128, \frac{1}{2})$).

Les figures de la table 1 qui suivent, montrent pour chacun des 16 octets constituant une clé, la distribution des clés de l'échantillon selon la valeur de cet octet. On remarque une moyenne qui vaut approximativement 131000 et la théorie prévoit $\frac{2^{25}}{2^8} = 131072$.

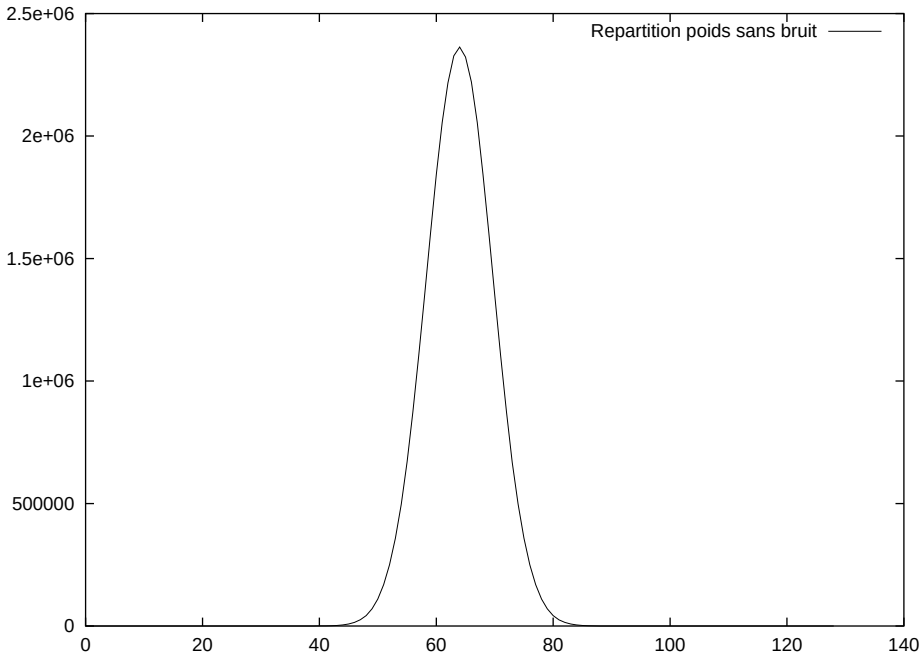
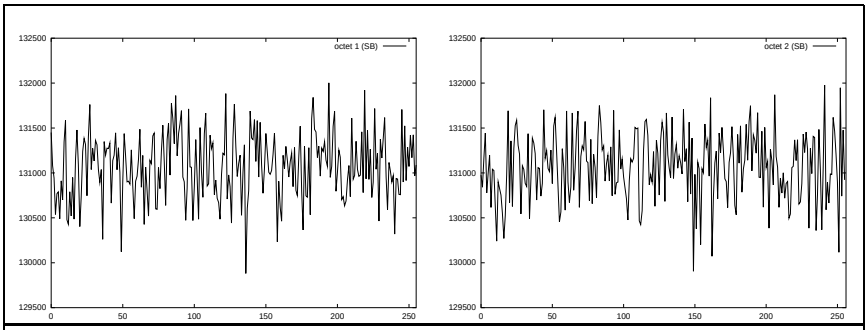


Fig. 1. Répartition des poids

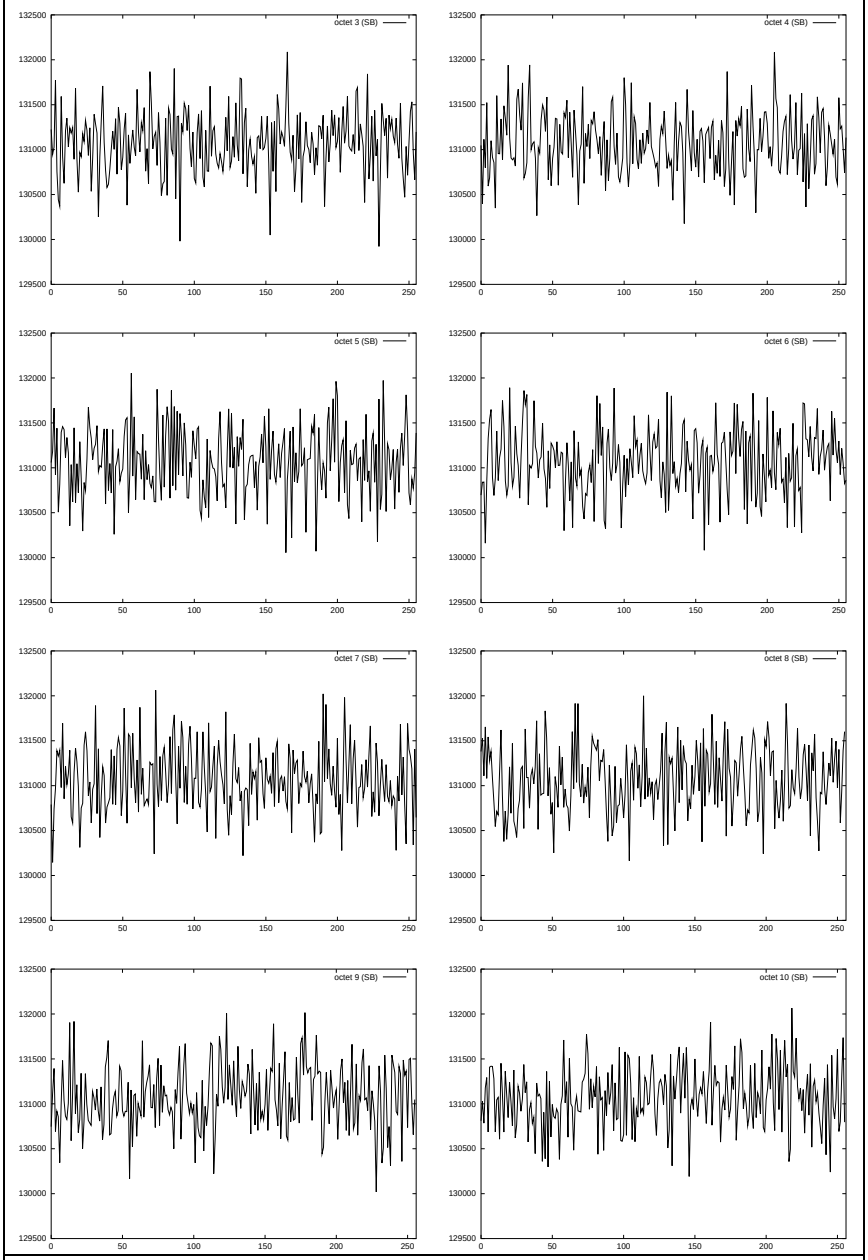
TAB. 1: Répartition des 16 octets



suite à la page suivante

TAB. 1: Répartition des 16 octets

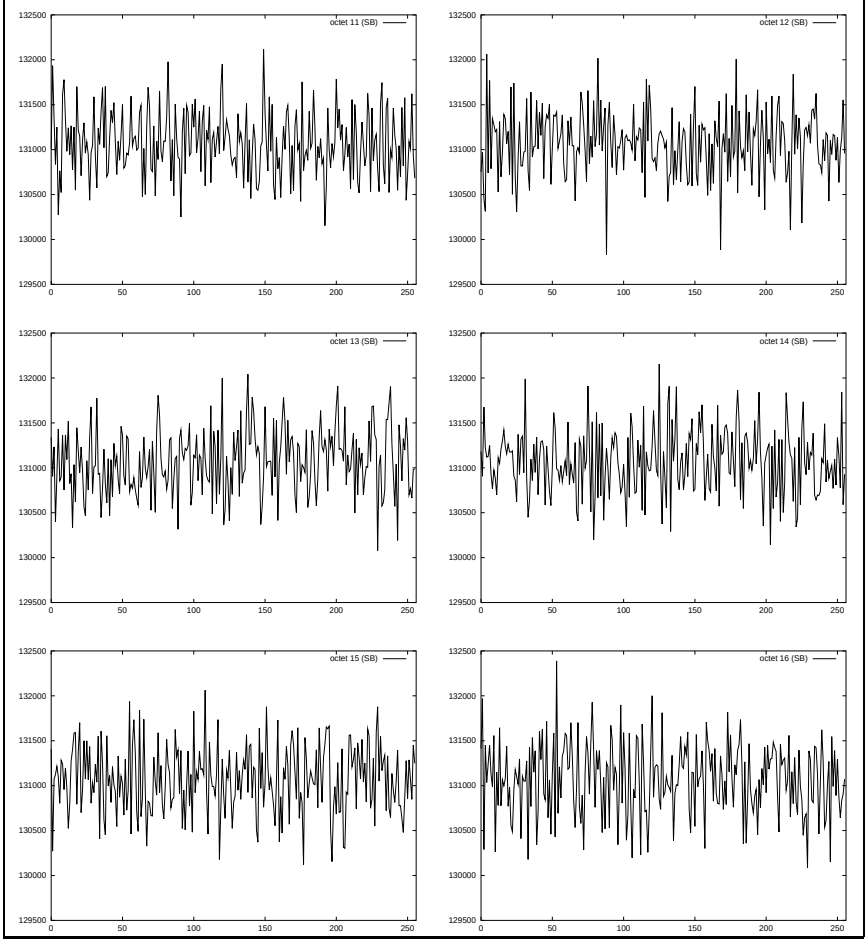
suite de la page précédente



suite à la page suivante

TAB. 1: Répartition des 16 octets

suite de la page précédente



Générateur avec bruit : Nous avons effectué des mesures analogues pour un générateur de clés choisies dans le même code bruité (avec des bruits de poids 1 et 2). Les résultats sont sensiblement identiques au cas sans bruit.

C Un générateur de clés RSA à trappe

On peut fabriquer un générateur à trappe qui produit des clés qui échappent en apparence à l'attaque de Wiener, sauf pour celui qui connaît la trappe. Ce générateur a été imaginé par C. Crépeau et A. Slakmon ([2]).

Voici le principe de la génération des clés avec le logiciel MAPLE :

```
# Initialisation du générateur de nombres aléatoires
```

```
> randomize():
# La taille des clés
> t := 1024:
# La trappe est un entier pair fixé inférieur à n
> M := 2*rand(2^(t-3)..2^(t-2))():
# On génère un modulus à partir de deux nombres premiers
> p := nextprime(rand(2^(t/2-1)..2^(t/2))()):
> q := nextprime(rand(2^(t/2-1)..2^(t/2))()):
> n := p*q:
> phi := (p-1)*(q-1):

# Puis on cherche un petit exposant privé
> d1 := 2*rand(2^(t/8-1)..2^(t/8))()+1:
> while igcdex(d1,phi,'e1')<>1 do d1 := d1+2;  od:
> while igcdex(e1+M,phi,'d')<>1 do
>   while igcdex(d1,phi,'e1')<>1 do d1 := d1+2; od:
> od:
# L'exposant privé est
> d := d mod phi:
# et le public est
> e := e1 + M:
```

Voici maintenant comment en connaissant la clé publique (n, e) et la trappe M on peut calculer la clé privée d.

```
# On commence par retrancher la trappe à e
> eprime := e-M:
# Ensuite on procède à l'attaque de Wiener sur la clé publique (n,e')
> convert(eprime/n,confrac,'reduites'):
> M := 12345: C := M&^eprime mod n:
> i := 1:
> while C&^denom(reduites[i]) mod n <> M do i := i+1: od:
# La clé privée est
> dprime := denom(reduites[i]);

          dprime := 646494819212925328043833613820642796535

# Ensuite, connaissant e' et d', on factorise n par une méthode probabiliste :
> u := eprime*dprime-1:
> while irem(u,2,'quot') = 0 do u := quot od:
> a := rand(2..n-1)(): while igcd(a,n)<>1 do a := rand(2..n-1)(); od:
> b := a&^u mod n:
> b2 := b*b mod n:
> while b2<>1 do
>   b := b2:
>   b2 := b*b mod n:
```

```
> od:

# on obtient les deux facteurs premiers
> p1 := igcd(b-1,n);q1 := igcd(b+1,n);

  p1 := 120328816104064584615853410051996559447373355205796383077\
        4542753034442728404920110514856119586932656913443763386080\
        4909768849735170358458751323561935471 79


  q1 := 104914027821708873342070815770733210136874164414257449402\
        6587007077346585764276789222815647080934040353167445328684\
        9045830490929788379445206857608471686497
# vérification
> n-p1*q1;

                                  0

# Il ne reste plus qu'à calculer l'exposant privé d correspondant à e
> phi1 := (p1-1)*(q1-1):
> igcdex(e,phi1,'d2'): d2 := d2 mod phi1:

#Vérification
> d-d2;

                                  0
```