Fiche technique

Attaque par factorisation contre RSA

Daniel Lerch Hostalot 

l'ordre des nombres décimaux à 100 chiffres.

Degré de difficulté

RSA représente sans conteste le système cryptographique à clé publique le plus utilisé de nos jours puisqu'il est parvenu à survivre aux critiques des spécialistes en cryptographie pendant plus d'un quart de siècle. La fiabilité de cet algorithme mondialement connu repose notamment sur la difficulté de décomposer en produit de facteurs premiers de gros chiffres, de

Nous allons étudier dans le présent article les travaux internes de RSA et la possibilité de lancer des attaques dites par factorisation. Nous verrons comment les clés de l'algorithme RSA ainsi que la procédure de l'attaque permettent d'obtenir la clé privée correspondant à la clé publique attaquée.

Afin de bien comprendre et réutiliser le présent article, le lecteur doit maîtriser les principes fondamentaux liés à la programmation en C et savoir manier certains concepts mathématiques. Vous trouverez dans le tableau des références des sources de documentation complémentaires pour de plus amples renseignements sur le sujet.

Tous les exemples ont été développés et testés sur un système GNU/Linux.

Cryptographie à clé publique

Contrairement à la cryptographie à clé privée, où une seule clé est utilisée pour crypter et décrypter les messages, la cryptographie à clé publique repose sur deux clés, concept également appelé biclé. Ces deux clés sont respectivement dénommées *clé publique* et *clé privée*. Afin de crypter la communication choisie, l'utilisateur aura besoin des deux clés. Alors que la clé privée doit demeurer confidentielle, la clé publique peut être mise à la disposition de quiconque désirant envoyer des messages cryptés à l'utilisateur. Un message crypté à l'aide d'une clé publique ne peut être décrypté qu'à l'aide de la clé privée correspondante. Or, pour ce faire, il faut aborder certains problèmes d'ordre mathématique. Notre choix s'est porté sur le système RSA en raison de la décomposition en produit de facteurs premiers, ou factorisation, de gros chiffres.

Les débuts de la cryptographie à clé publique remonte à la publication de Diffie et Hellman en 1976. Dans cette publication, les deux chercheurs ont introduit un protocole capable d'autoriser l'échange de certaines informations sur un canal

Cet article explique…

- Fonctionnement de RSA,
- Techniques permettant de lancer des attaques par factorisation.

Ce qu'il faut savoir…

- Principes fondamentaux de la programmation en C.

non sécurisé. Peu de temps après, en 1977, les chercheurs Rivest, Shamir et Adleman ont proposé leur propre système de cryptographie baptisé RSA, aujourd'hui le plus largement utilisé au monde.

Toujours en 1977, des documents ont démontré que les cryptographes du British Government Group pour la Sécurité des Communications Electroniques (GSEC) connaissaient déjà ce type de cryptographie en 1973.

Système cryptographique RSA

Comme nous venons de le dire précédemment, la sécurité du système RSA repose sur la difficulté mathématique de décomposer en produit de facteurs premiers, ou factoriser, de gros chiffres. Factoriser un nombre consiste à trouver les nombres premiers (facteurs) dont le produit donne pour résultat le nombre en question. Si, par exemple, vous souhaitez factoriser le nombre 12, vous allez obtenir un résultat de 2·2·3. La méthode la plus simple pour trouver les facteurs premiers d'un nombre n consiste à diviser ce dernier par l'ensemble des nombres premiers inférieurs à n. Quoique extrêmement simple, cette procédure peut se révéler très fastidieuse lorsque vous souhaitez factoriser de gros chiffres.

Procédons à quelques calculs pour vous donner une idée. Une clé de 256 bits (à l'instar de celle que nous allons attaquer ultérieurement) comprend près de 78 chiffres décimaux (1078). Dans la mesure où sur les clés du système RSA, ce nombre ne possède en règle générale que deux facteurs premiers, chacun de ces facteurs auront plus ou moins 39 chiffres. Autrement dit, pour factoriser le nombre en question, il faudra le diviser par l'ensemble des nombres premiers à 39 chiffres ou moins (1039). En supposant que seul 0,1% des nombres soient des nombres premiers, il faudra procéder à près de 1036 divisions. Imaginons maintenant que vous disposiez d'un système capable de réaliser 1020 divisions par seconde. Dans ce cas, vous passerez quelques 1016 secondes à attaquer la clé, autant dire plus de 300 millions d'années, soit un million l'âge de l'univers. Heureusement, nous serons bien plus rapide grâce à l'exemple décrit ci-après.

Analysons désormais le fonctionnement du système RSA. Il faut commencer par générer les clés publiques et privées (vous trouverez dans le tableau ci-joint quelques illustrations de concepts mathématiques particulièrement intéressants). Pour ce faire, il est nécessaire de bien respecter les étapes comme suit.

Etape 1

Il faut tout d'abord choisir deux nombres premiers p et q de manière aléatoire, puis les multiplier pour obtenir n : $n = p \cdot q$. Si vous choisissez par exemple, $p = 3$ et $q = 11$ vous obtiendrez $n = 33$.

Etape 2

Il faut ensuite calculer l'indicateur (ou fonction indicatrice) d'Euler à l'aide de la formule suivante : $\Phi(n) = \Phi(p \cdot q) = (p-1) \cdot (q-1)$. Dans cet exemple, vous devez obtenir $\Phi(n) = 20$.

Etape 3

Vous trouvez un exposant de cryptage (que vous allez utiliser par la suite pour le cryptage) que vous allez appeler e. Ce nombre doit être compatible avec $\mathrm{mcd}(e, \Phi(n)) = 1$ Un excellent exemple serait : $e = 3$ dans

Concepts mathématiques

Diviseur ou facteur

Un nombre entier a désigne un diviseur (ou facteur) de b lorsqu'il existe d'autres nombres entiers c compatibles avec $b = a \cdot c$. Exemple : $21 = 7 \cdot 3$

Nombres premiers et nombres composés

Un nombre entier est dit premier si ce dernier ne peut être divisé que par un et lui-même. Un nombre entier est dit composé lorsqu'il ne rentre pas dans la catégorie des nombres premiers.

Exemple : $21 = 7 \cdot 3$ est un nombre composé, alors que 7 et 3 sont des nombres premiers.

Factorisation

La factorisation d'un nombre entier n consiste à décomposer ce dernier en produit de ses facteurs premiers : $n = p_1^{e_1} \cdot p_2^{e_2} \cdots p_i^{e_i}$ où p_i représentent des nombres premiers et e_i représentent des nombres entiers positifs. Exemple : 84 peut être factorisé en $21 = 2^2 \cdot 3 \cdot 7$.

Module

Un module désigne et représente $a \bmod b$ le reste de la division entière de a entre b.

Exemple : $5 \bmod 3 = 2$, $10 \bmod 7 = 3$, $983 \bmod 3 = 2$, $1400 \bmod 2 = 0$.

a et b sont des modules de n : $b \equiv a \pmod n$ si leur différence (a-b) est un multiple den.

Diviseur commun maximum

Est désigné par diviseur commun maximum de deux nombres entiers a et b, représentée sous forme de $\mathrm{mcd}(a,b)$, le nombre entier le plus élevé pouvant diviser a et b.

Exemple : $\mathrm{mcd}(42,35) = \mathrm{mcd}(2 \cdot 3 \cdot 7, 5 \cdot 7) = 7$ Algorithme euclidien

L'algorithme euclidien permet de calculer le diviseur commun maximum de deux nombres selon $\mathrm{mcd}(a,b) = \mathrm{mcd}(b,r)$, où $a > b > 0$ représentent des nombres entiers, et r le reste de la division entre a et b Exemple : $\mathrm{mcd}(1470, 42)$

(1) $1470 \bmod 42 = 35 \rightarrow \mathrm{mcd}(1470,42) = \mathrm{mcd}(42,35)$

(2) $42 \bmod 35 = 7 \rightarrow \mathrm{mcd}(42,35) = \mathrm{mcd}(35,7)$

(3) $35 \bmod 7 = 0 \rightarrow \mathrm{mcd}(7,0) = 7$ Indicateur (ou fonction indicatrice) d'Euler

Soit $n \geq 0$ représentant $\Phi(n)$ le nombre d'entiers compris dans l'intervalle $[1,n]$ étant premiers* avec n. Pour $n = p \cdot q$, $\Phi(n) = (p-1)(q-1)$.

*Deux entiers a et b représentent les nombres premiers entre eux (ou co-premier) si $\mathrm{mcd}(a,b) = 1$.

la mesure où il ne possède aucun facteur commun avec 20 (facteurs 2 et 5).

Etape 4

Il faut ensuite calculer un exposant de décryptage que vous allez appeler d (il vous servira plus tard au décryptage). Ce nombre doit correspondre à $1 < d < \Phi(n)$ de sorte que $e \cdot d \equiv 1 (\mathrm{mod}\, \Phi(n))$. Autrement dit, d sera un nombre compris entre 1 et 20 qui, multiplié par 3 et divisé par 20 donnera 1. L'exposant de décryptage d peut alors être le nombre 7.

Clés

La clé publique de l'utilisateur appartient au couple (n,e), soit dans notre exemple (33, 3), alors que la clé privée correspond à d, soit 7. Logiquement, les nombres p, q et d $\Phi(n)$ doivent demeurer confidentiels.

(Dé)cryptage

Arrivé à ce stade, il suffit de procéder au cryptage à l'aide de $C = M^e \bmod n$ puis au décryptage grâce à $M = C^d \bmod n$. Si nous supposons que le message est $M = 5$, le chiffrage correspondant sera alors $C = 5^3 \bmod 33 = 26$. Afin de décrypter le message, il suffira d'appliquer $M = 26^7 \bmod 33 = 5$.

Comme nous l'avons mentionné en début d'article, et comme le démontre la procédure ci-dessus, la sécurité des systèmes cryptographiques repose sur le nombre n. En d'autres termes, si un pirate capable d'accéder à la clé publique parvient à factoriser n en obtenant p et q, il lui suffit alors d'appliquer les formules précédentes pour obtenir la clé privée tant convoitée.

Listing 1. *Transformation d'un chiffre hexadécimal en chiffre décimal*

```
#include <stdio.h>
#include <openssl/bn.h>
int main (int argc, char **argv) {
   BIGNUM *n = BN_new();
   if (argc!=2) {
      printf ("%s <hex>\n",
      argv[0]);
      return 0;
   }
   if(!BN_hex2bn(&n, argv[1])) {
      printf("error:
      BN_hex2bn()");
      return 0;
   }
   printf("%s\n", BN_bn2dec(n));
   BN_free(n);
}
```

Compétition de factorisation RSA

La Compétition de Factorisation RSA est un concours financé par les Laboratoires RSA au cours duquel d'importants prix sont décernés aux informaticiens capables de factoriser certains nombres particulièrement importants. Pour parvenir à connaître le statut des systèmes de factorisation, il faut savoir quelle longueur de clé va être nécessaire pour garantir la sécurité du système RSA.

À l'heure où nous rédigeons le présent article, le record de factorisation est RSA-640, nombre à 193 chiffres décomposé en produits de facteurs premiers le 2 novembre 2005 par F. Bahr et al. La prochaine compétition portera sur RSA-704, avec un prix de 30 000 dollars.

La Compétition de Factorisation RSA demeure sans aucun doute le moyen le plus pratique de se faire une idée sur l'état actuel des systèmes de factorisation.

Vous trouverez dans le tableau exposé à la fin du présent article les compétitions actuellement organisées en la matière.

Attaque par factorisation

Nous allons présenter ci-après un exemple d'attaque par factorisation dirigée contre une clé du système RSA. Afin de rendre les calculs plus rapides, nous utiliserons une clé bien plus courte que les clés normalement utilisées, afin d'en simplifier la décomposition en facteurs premiers. Même si cet exemple ne se conforme pas à la réalité, il permettra toutefois d'analyser les éléments constituant une attaque complète.

Il faut commencer par créer un environnement de travail au moyen de l'outil OpenSSL, chargé de générer les clés nécessaires et de crypter le message que vous allez ensuite utiliser sous forme de cible pour votre attaque. Il faudra ensuite factoriser le module n pour obtenir la clé privée, et enfin décrypter le message en question.

OpenSSL et RSA

OpenSSL est un outil cryptographique open source particulièrement pratique. Dans la partie consacrée aux références, vous trouverez toutes les informations nécessaires pour le télécharger, mais sachez que la plupart des distributions GNU/Linux le proposent par défaut. Nous allons donc l'utiliser ici pour configurer un environnement de test dans lequel l'attaque sera lancée.

La première étape consiste à générer un couple de clés pour le cryptage et le décryptage. Nous allons générer des clés de 256 bits, certes trop courtes pour maintenir un niveau de sécurité sur nos communications, mais largement suffisantes pour illustrer nos propos.

Nous générons donc une paire de clés, en maintenant notre clé privée confidentielle.

```
# Génération d'une paire de clés à 256
bits du système RSA
openssl genrsa -out rsa_privkey
.pem 256
cat rsa_privkey.pem
-----DEBUT DE LA CLE PRIVEE-----
MIGqAgEAAiEA26dbqzGRt31qincXxy
   4jjZMMOId/DVT8aTcq8aam
   DiMCAwEAAQIh
AmvT1oXa/rxF3mrVLrR/RS7vK1WT
   sQ5CWl/+37wztZOpAhEA+4jg
   EkfalFH+0S+1
IPKD5wIRAN+NmMH4AF0B8jz
   MAXHHXGUCEGRpRZnGmV
   kwSlrTgqj+Zu0CEA7v7CQR
yRxt09zCGNqcYo0CEDEW7mvoz
   MYYLC5o+zgfV4U=
-----FIN DE LA CLE PRIVEE RSA-----
```

Après cette étape, nous sauvegardons la clé publique dans un fichier. Il s'agit de la clé que nous allons publier afin de permettre à quiconque de nous envoyer des messages cryptés.

```
# Sauvegarde de la clé publique sur un
                        fichier
openssl rsa -in rsa_privkey.pem
   -pubout -out rsa_pubkey.pem
cat rsa_pubkey.pem
-----DEBUT DE LA CLE PUBLIQUE-----
MdwwDQYJKoZIhvcNAQEBB
   QADKwAwKAIhANunW6sxkbd
   9aop3F8cuI42TDDiHfw1U
/Gk3KvGmpg4jAgMBAAE=
-----FIN DE LA CLE PUBLIQUE-----
```

Après avoir générer cette paire de clés, il est désormais possible de crypter et de décrypter les messages. Nous allons travailler sur le message suivant :

```
echo "Forty-two" > plain.txt
```

Ce message peut être facilement crypté au moyen de la commande suivante et de la clé publique :

```
openssl rsautl -encrypt
   -pubin -inkey rsa_pubkey.pem \
   -in plain.txt -out cipher.txt
```

Afin de décrypter le message, nous utiliserons la clé privée :

```
openssl rsautl -decrypt -inkey
   rsa_privkey.pem -in cipher.txt
```

Une fois le fonctionnement de l'outil OpenSSL avec le système RSA maîtrisé, et en étant conscient de la nécessité de disposer de la clé privée pour pouvoir décrypter les messages, notre objectif consiste à obtenir cette clé privée sans avoir accès à l'originale. En d'autres termes, il faut tenter d'obtenir la clé privée en utilisant la clé publique. Il faut donc commencer par obtenir le module n ainsi que l'exposant de cryptage. Pour ce faire, il suffit d'utiliser la commande suivante avec la clé publique :

```
openssl rsa -in rsa_pubkey.pem
   -pubin -text -modulus
Modulus (256 bit):
   00:db:a7:5b:ab:31:91:b7:7d:
      6a:8a:77:17:c7:2e:
   23:8d:93:0c:38:87:7f:0d:54:
      fc:69:37:2a:f1:a6:
   a6:0e:23
Exponent: 65537 (0x10001)
Modulus=DBA75BAB3191B77D
   6A8A7717C72E238D930C38877
   F0D54FC69372AF1A6A60E23
writing RSA key
-----DEBUT DE LA CLE PUBLIQUE-----
MdwwDQYJKoZIhvcNAQEBBQ
   ADKwAwKAIhANunW6sxkbd9
```

Listing 2. *Clé privée*

```
#include <stdio.h>
#include <openssl/bn.h>
#include <openssl/rsa.h>
#include <openssl/engine.h>
#include <openssl/pem.h>
int main (int argc, char **argv) {
   RSA *keypair = RSA_new();
   BN_CTX *ctx = BN_CTX_new();
   BN_CTX_start(ctx);
   BIGNUM *n = BN_new();
   BIGNUM *d = BN_new();
   BIGNUM *e = BN_new();
   BIGNUM *p = BN_new();
   BIGNUM *q = BN_new();
   BIGNUM *dmp1 = BN_new();
   BIGNUM *dmq1 = BN_new();
   BIGNUM *iqmp = BN_new();
   BIGNUM *r0 = BN_CTX_get(ctx);
   BIGNUM *r1 = BN_CTX_get(ctx);
   BIGNUM *r2 = BN_CTX_get(ctx);
   BIGNUM *r3 = BN_CTX_get(ctx);
   if (argc!=4) {
      printf ("%s [p] [q] [exp]\n", argv[0]);
      return 0;
   }
   BN_dec2bn(&p, argv[1]);
   BN_dec2bn(&q, argv[2]);
   BN_dec2bn(&e, argv[3]);
   if(BN_cmp(p, q)<0) {
      BIGNUM *tmp = p;
      p = q;
      q = tmp;
   }
   // Nous calculons n
   BN_mul(n, p, q, ctx);
   // Nous calculons d
   BN_sub(r1, p, BN_value_one()); // p-1
   BN_sub(r2, q, BN_value_one()); // q-1/
   BN_mul(r0, r1, r2, ctx); // (p-1)(q-1)
   BN_mod_inverse(d, e, r0, ctx); // d
   BN_mod(dmp1, d, r1, ctx);
   BN_mod(dmq1, d, r2, ctx);
   // Nous calculons le module p inverse de q
   BN_mod_inverse(iqmp, q, p, ctx);
   // Clés RSA
   keypair->n = n;
   keypair->d = d;
   keypair->e = e;
   keypair->p = p;
   keypair->q = q;
   keypair->dmq1 = dmq1;
   keypair->dmp1 = dmp1;
   keypair->iqmp = iqmp;
   PEM_write_RSAPrivateKey(stdout, keypair, NULL, NULL, 0, NULL, NULL);
   BN_CTX_end(ctx);
   BN_CTX_free(ctx);
   RSA_free(keypair);
   return 0;
}
```

```
   aop3F8cuI42TDDiHfw1U
/Gk3KvGmpg4jAgMBAAE=
-----FIN DE LA CLE PUBLIQUE-----
```

Le module est exprimé en chiffres hexadécimaux. Afin de le convertir en chiffres décimaux, vous pouvez utiliser le programme exposé dans le Listing 1.

```
gcc hex2dec.c -lssl
```

Listing 3. *Modifier une clé privée en une clé publique*

```
gcc get_priv_key.c -lssl -o get_priv_key
      ./get_priv_key 297153055211137492311771648517932014693 \
        3343469240228704458360474938274848777799 65537
      -----BEGIN RSA PRIVATE KEY-----
                        MIGqAgEAAiEA26dbqzGRt31qincXxy4jjZMMOId/
DVT8aTcq8aamDiMCAwEAAQIh
       AMvT1oXa/rxF3mrVLrR/RS7vK1WTsQ5CWl/+37wztZOpAhEA+4j
gEkfalFH+0S+1
      IPKD5wIRAN+NmMH4AF0B8jzMAXHHXGUCEGRpRZnGmVkwS
lrTgqj+Zu0CEA7v7CQR
      yRxt09zCGNqcYo0CEDEW7mvozMYYLC5o+zgfV4U=
      -----END RSA PRIVATE KEY-----
```

Listing 4. *dfact_client*

```
...
for(;;) {
   get_random_seeds(&seed1, &seed2);
   switch(status) {
      case DF_CLIENT_STATUS_WAITING:
         N = recv_N_number(&rel_by_host, host);
         if(!N) sleep(DF_TIME_TO_RECV);
         else status = DF_CLIENT_STATUS_RUNNING;
      break;
      case DF_CLIENT_STATUS_RUNNING: {
         msieve_obj *obj = NULL;
         obj = msieve_obj_new(N, flags, relations, NULL,
         NULL, seed1, seed2, rel_by_host, 0, 0);
         if (obj == NULL) {
            syslog(LOG_ERR, "Factoring initialization failed");
            free(N);
            return 0;
         }
         msieve_run(obj);
         if(obj) msieve_obj_free(obj);
         while(!send_relations(N, host, relations)) sleep(DF_TIME_TO_SEND);
         if(unlink(relations)==-1) syslog(LOG_ERR, "unlink(): %s: %s",
                     relations, strerror(errno));
         status = DF_CLIENT_STATUS_WAITING;
         free(N);
      }
      break;
      default:
      break;
   }
...
```

```
/a.out DBA75BAB3191B77D6
   A8A7717C72E238D930C388
   77F0D54FC69372AF1A6A60E23
9935220997384201394973685017
   185769998267119089063339396
   575567287426977500707
```

Une fois le module obtenu en décimal, l'étape suivante consiste à le décomposer en produit de facteurs premiers.

Factorisation du module n

Dans la mesure où le nombre que nous voulons factoriser n'est pas trop important, il est plus rapide d'appliquer l'algorithme de factorisation baptisé QS (*Crible Quadratique*, en français). Cet algorithme est installé à l'aide de msieve, programme que vous pouvez télécharger en suivant le tableau des références ci-après. Msieve dispose d'une documentation suffisante qui vous permettra de l'installer et de l'utiliser rapidement sans trop de difficulté. Ce programme, combiné à la commande suivante suffit à factoriser le nombre proposé :

```
/msieve -v
99352209973842013949736850170
   17018576999826711908906333
   39396575567287426977500707
```

Un ordinateur moderne peut factoriser ce nombre en près de dix minutes, selon le matériel informatique disponible. Le résultat est le suivant :

```
factor: 297153055211137492311
   771648517932014693
factor: 334346924022870445836
   047493827484877799
```

À ce stade, une fois le module n factorisé et grâce à l'exposant de cryptage 65537 obtenu lors de l'étape précédente, nous disposons de toutes les données nécessaires pour l'obtention de la clé privée.

Obtention de la clé privée et décryptage du message

En raison des difficultés de ce processus avec les outils les plus communs, nous allons développer un programme capable de réaliser cette tâche à notre place. La source de ce programme est exposée dans le Listing 3.

Afin de réaliser les calculs nécessaires, nous avons utilisé la bibliothèque OpenSSL. Les variables BIGNUM sont employées par cette bibliothèque pour travailler sur les gros chiffres. Ces variables possèdent leur propre interface de programmation pour réaliser des opérations telles qu'additions, soustractions, opérations modulaires, etc.

Le programme donné en exemple commence par placer dans les variables *BIGNUM*, les paramètres *p*, *q* et *e*. Ensuite, si vous analysez le code en détails en vous aidant des commentaires, vous pourrez comprendre la procédure consistant à générer la clé privée. Il s'agit de la même procédure que nous avions expliquée précédemment de manière théorique. En réalité, la seule différence entre le test et la génération réelle de clés repose sur le fait que p et q auraient été choisis de manière aléatoire. Dans notre exemple, nous les avons obtenus à partir de la factorisation du module. Enfin, à l'aide de *PEM_write_RSAPrivateKey()*, nous pouvons rédiger la clé privée utilisée dans les exemples au format PEM. Si l'on compare la clé ainsi générée avec la clé privée originale, il est clair que notre objectif a bien été rempli, puisque nous avons obtenu la clé privée à partir de la clé publique.

Si nous appliquons notre nouvelle clé privée sur un fichier texte, comme par exemple *rsa_hacked_privkey.pem*, il est alors possible de décrypter le message :

```
openssl rsautl -decrypt
   -inkey rsa_hacked_privkey
   .pem -in cipher.txt
```

Algorithmes modernes de factorisation

Les algorithmes de factorisation ont énormément progressé au fil du temps, de sorte qu'aujourd'hui, nous disposons d'algorithmes particulièrement rapides tels que la Méthode de la Courbe Elliptique (*Elliptic Curve Method*, ou ECM), le Crible Quadratique (*Quadratic Sieve*, ou QS), ou le Crible de Corps de Nombre (*Number Field Sieve*, ou NFS). Ces algorithmes donnent également naissance aux certaines de variations selon le type de nombre à décomposer en produit de facteurs premiers où la manière de résoudre certaines parties du nombre en question. Ces algorithmes sont assez complexes. Ils sont en règle générale divisés en deux étapes au cours desquelles différents calculs conduisant à la factorisation du nombre en question sont effectués. Les algorithmes QS et NFS passent par une étape dite de crible. À ce stade, certains types de relations sont rassemblés, pour finalement construire un système d'équations permettant d'obtenir le résultat. L'étape dite de crible peut être réalisée par plusieurs machines fonctionnant de manière simultanée, dans la mesure où il s'agit normalement de l'étape la plus longue.

Dans les exemples exposés plus haut, nous avons utilisé le programme msieve, implémentation de l'algorithme de Crible Quadratique à Polynômes Multiples (*Multiple Polynomial Quadratic Sieve*, ou MPQS), lui-même étant dérivé de l'algorithme QS. L'algorithme QA se révèle plus rapide lorsque vous devez factoriser des nombres de moins de 110 chiffres. Si cette limite est dépassée, il vaut mieux avoir recours à l'algorithme NFS. Une variation de l'algorithme NFS est utilisée pour factoriser tout type de nombres ; il s'agit de l'algorithme GNFS (*General Number Field Sieve*). Peu de programmes libres et gratuits implémentent l'algorithme GNFS, et le seul programme de ce genre disponible ne dispose pas d'une bonne documentation, et n'est pas facile d'utilisation. Du moins, tel était le cas au moment où nous rédigions le présent article. Quoiqu'il en soit, nous allons nous

Listing 5. *dfact_server*

```
...
for(;;) {
   while(child_count >= DF_MAX_CLIENTS) sleep(1);
   sd_tmp = socket_server_accept(sd, client, sizeof(client));
   if((pid=fork())==0) {
      close(sd);
      process_client(sd_tmp, N, num_relations, rel_by_host, client);
   } else if (pid>0) {
      close(sd_tmp);
      child_count++;
   } else {
      perror("fork()");
   }
   close(sd_tmp);
}
close(sd);
...
```

Listing 6. *dfact_server (process_relations)*

```
void process_relations(char *N, int num_relations, int seconds) {
   for(;;) {
      int n_sieves = get_num_relations_in_file(DF_FILE_RELATIONS);
      printf ("relations: %d, need: %d \n", n_sieves, num_relations);
      // y a-t-il assez de relations ?
      if(n_sieves>=num_relations) {
         printf("Factoring %s\n", N);
         kill(0, SIGUSR1);
         uint32 seed1;
         uint32 seed2;
         uint32 flags;
         flags |= MSIEVE_FLAG_USE_LOGFILE;
         get_random_seeds(&seed1, &seed2);
         factor_integer(N,flags,DF_FILE_RELATIONS,NULL,&seed1,&seed2);
         printf("Factoring Done\n");
         kill(getppid(), SIGKILL);
         exit(0);
      }
      sleep(seconds);
   }
}
```

pencher sur le fonctionnement de l'algorithme GGNFS. Il s'agit d'une implémentation de l'algorithme GNFS qui, malgré un manque de stabilité, permet de décomposer des nombres en produit de facteurs premiers sans trop de problèmes.

L'algorithme GGNFS comprend en réalité un groupe d'outils, lesquels, utilisés un par un, peuvent procéder à l'ensemble des étapes composant cet algorithme. Toutefois, un novice en la matière risque d'avoir du mal à factoriser un nombre avec cette procédure assez complexe. C'est la raison pour laquelle l'algorithme GGNFS comprend un script rédigé en Perl qui se charge de toutes les tâches nécessaires. Ce script permet d'utiliser le programme sans aucun problème, mais ce n'est pas le meilleur moyen d'obtenir le meilleur des outils qui composent l'algorithme GGNFS.

Le plus simple pour tester ce programme reste encore d'éditer un fichier, que nous baptiserons *test.n*, dans lequel est indiqué le nombre que nous souhaitons décomposer en facteurs premiers.

```
cat test.n
n: 1522605027922533360535
   6183781326374297180681149
   6138068865790849458012296
   3258952897654000350692006139
```

Il faut ensuite lancer :

```
tests/factLat.pl test.n
```

Ce programme va effectivement factoriser le nombre, après quelques bonnes heures de traitement. La durée de traitement dépend du matériel utilisé. Afin d'optimiser l'algorithme GGNFS, il vaut mieux ne pas tenir compte du script factLat.pl pour utiliser les outils dont il dispose avec les bons paramètres. L'utilisation de l'algorithme mériterait d'y consacrer un article entier, mais ce n'est malheureusement pas le propos du présent article. Si vous souhaitez apprendre à vous en servir, le mieux reste de lire la documentation disponible avec le code source et de consulter le forum de discussions (voir les liens). Il est également recommandé de consulter quelques documents sur l'algorithme NFS. Quoiqu'il en soit, il est nécessaire de préciser qu'il faut maîtriser les principes de l'algèbre linéaire ainsi que la théorie des nombre.

Nécessité d'une attaque distribuée

La clé factorisée dans cet exemple est très modeste par rapport à la longueur des clés généralement utilisées aujourd'hui. Si nous décidions à l'instant de créer une clé RSA pour un usage personnel, nous utiliserions au minimum une clé de 1024 bits. Pour plus de sécurité, nous pourrions également utiliser une clé de 2048 ou 4096 bits. Si vous tentez de factoriser une de ces clés avec un PC familial, quelle que soit sa vitesse, vous verrez à quel point votre ordinateur va s'embarquer dans des calculs sans fin ne menant à rien. La vérité est qu'il est impossible d'attaquer ce genre de clés. Mais les avancées technologiques et mathématiques réduisent de jour en jour cet objectif. Sous certaines conditions, il est possible de réaliser des attaques distribuées en utilisant des centaines de machines de manière simultanée pour faciliter le processus de factorisation. De nombreuses études ont été menées dans ce domaine afin d'analyser les possibilités d'attaquer une clé de 1024 bits (voir la partie consacrée aux liens sur le Web). Aujourd'hui, ce genre d'attaque demeure bien loin de la portée de la plupart des accros d'informatique, mais pas de celle de certains gouvernements ou organisations.

En outre, l'existence de compétitions telles que la Compétition de Factorisation RSA mentionnée plus haut, aident considérablement les experts dans ce domaine en les motivant à créer des outils distribués destinés à la factorisation des gros chiffres.

Attaque distribuée

Nous avons évoqué dans les exemples précédents le programme msieve. Comme nous l'avons appris, ce programme est facile à utiliser et le programme est suffisamment développé pour ne pas créer trop de problèmes à l'utilisateur. Selon nous, il s'agit du meilleur programme capable jusqu'ici d'implémenter l'algorithme de Crible Quadratique (QS). Mais, ce programme n'est rien de plus qu'une démo illustrant un usage basique de la bibliothèque msieve, et ne peut donc être utilisé que sur une seule machine.

La documentation du programme comporte quelques méthodes permettant d'utiliser le programme démo sur différentes machines de sorte à réaliser une factorisation distribuée. Il s'agit toutefois d'une procédure manuelle, très peu pratique. C'est la raison pour laquelle nous avons décidé d'implémenter un modeste exemple de programme introduisant l'utilisation de la bibliothèque msieve pour réaliser une factorisation distribuée. Ce programme est baptisé dfact, et vous le trouverez sur le CD proposé avec ce magazine ainsi que dans la partie consacrée aux liens sur le Web.

Le programme peut être compilé avec un *Makefile* et n'exige que l'installation correcte de la bibliothèque msieve. Le chemin de cette bibliothèque doit être inclus dans le *Makefile*. Une fois compilé, vous trouverez deux binaires dans le dossier *bin/* correspondant aux côtés client et serveur. Le serveur (dfs) sera exécuté sur une machine avec une capacité de mémoire suffisante (plus le nombre est gros, plus il faudra de mémoire), et permettra de distribuer les charges et coordonner les clients. Le serveur prend quatre paramètres : le nombre à factoriser, le nombre de relations que nous voulons voir recompilée par le client pour chaque paquet envoyé et le nombre de secondes qu'il faut au serveur pour contrôler s'il dispose de données client en quantité suffisante pour terminer avec succès le processus de factorisation. Dans l'exemple suivant, nous allons demander aux clients d'envoyer des relations toutes les 5000 secondes et au serveur de vérifier le nombre de relations toutes les 60 secondes.

```
bin/dfs 9935220997384201394
   973685017018576999826
   711908906333939657556
   7287426977500707 5000 60
```

Nous allons lancer dfc sur quelques clients, en lui affectant comme paramètre l'IP du serveur et le chemin vers un fichier temporaire où les relations peuvent être sauvegardées. Par exemple :

```
bin/dfc /tmp/rel 192.168.1.7
```

Le programme *dfact* a été développé en utilisant comme support de base la bibliothèque msieve. Celle-ci dispose d'un exemple de programme baptisé *demo.c*, illustrant son fonctionnement simplifié. Si vous observez attentivement le code, vous constaterez qu'il n'y a rien de très difficile dans les instructions à suivre. Nous avons exposé dans le Listing 4 un extrait de code du client *dfact*. Ici, nous exposons le processus interne de la boucle principale au cours duquel le client obtient du serveur le nombre à factoriser, puis calcule les relations demandées via la bibliothèque msieve pour finir par les envoyer au serveur pour qu'il les traite.

Regardons comment le serveur gère la situation (Listing 5). Chaque client demandant l'envoi de la liste des relations au serveur est géré par *process_client()* via un processus séparé.

Une autre procédure distincte se charge de traiter les relations que les clients envoient à des intervalles de temps réguliers (Listing 6).

L'exemple de programme nous permet de factoriser un nombre à l'aide de plusieurs machines. Même si cette opération peut être réalisée via Internet, il est recommandé d'y renoncer compte tenu du manque d'authentification et/ou des mécanismes de décryptage. Il serait possible d'améliorer (ce serait par ailleurs un excellent exercice pour les lecteurs) l'usage de SSL, de renforcer la sécurité, d'optimiser la performance, etc...

Nous avions précédemment évoqué l'algorithme GNFS, plus efficace que l'algorithme MPQS lorsqu'il s'agit de factoriser des nombres à plus de 110 chiffres. À ce stade, il semble qu'aucun programme open source ne permette d'implémenter de manière simple un système distribué de factorisation avec l'algorithme GNFS, comme nous sommes parvenus à le faire avec la bibliothèque msieve (QS). L'auteur du programme msieve, toutefois, travaille sur un support pour l'algorithme GNFS. Même s'il n'est actuellement qu'à mi-chemin dans la programmation de ce support, il est fort possible que la solution définitive soit disponible dans un futur proche. Si tel était le cas, il ne serait alors guère difficile de modifier notre exemple (*dfact*) afin de réaliser une factorisation distribuée avec l'algorithme GNFS.

Quoiqu'il en soit, l'algorithme GGNFS permet d'utiliser plusieurs machines à des fins de factorisation. Ceci est possible grâce au script *factLat.pl*, comme nous l'avons démontré précédemment, mais sachez qu'il s'agit d'une version assez instable qui ne permet d'utiliser que très peu de machines seulement sur un réseau LAN.

Compétition de Factorisation RSA

- RSA-704 (30 000 dollars) : *http://www.rsasecurity.com/rsalabs/node.asp?id=2093#RSA704*
- RSA-768 (50 000 dollars) : *http://www.rsasecurity.com/rsalabs/node.asp?id=2093#RSA768*
- RSA-896 (75 000 dollars) : *http://www.rsasecurity.com/rsalabs/node.asp?id=2093#RSA896*
- RSA-1024 (100 000 dollars) : *http://www.rsasecurity.com/rsalabs/node.asp?id=2093#RSA1024*
- RSA-1536 (150 000 dollars) : *http://www.rsasecurity.com/rsalabs/node.asp?id=2093#RSA1536*
- RSA-2048 (200 000 dollars) : *http://www.rsasecurity.com/rsalabs/node.asp?id=2093#RSA2048*

Sur Internet

- Factorisation des gros nombres entiers – *http://factorizacion.blogspot.com*
- DFACT – *http://daniellerch.com/sources/projects/dfact/dfact-hakin9.tar.gz*
- GGNFS – Implémentation d'un Crible de Corps de Nombre : *http://www.math.ttu.edu/~cmonico/software/ggnfs/*
- Groupe Yahoo! Pour l'algorithme GGNFS – *http://www.groups.yahoo.com/group/ggnfs*
- MSIEVE – Factorisation d'entiers : *http://www.boo.net/~jasonp/qs.html*
- Compétition de Factorisation RSA : *http://www.rsasecurity.com/rsalabs/node.asp?id=2092*
- OpenSSL – *http://www.openssl.org*
- Algorithme Shor – *http://es.wikipedia.org/wiki/Algoritmo_de_Shor*
- Sur le coût de la factorisation RSA 1024 – *http://www.wisdom.weizmann.ac.il/%7Etromer/papers/cbtwirl.pdf*
- Estimations sur la factorisation du module RSA à 1024 bits – *http://www.wisdom.weizmann.ac.il/%7Etromer/papers/factorest.pdf*

À propos de l'auteur

Daniel Lerch Hostalot, ingénieur spécialisé en programmes C/C+ développés sur les plateformes GNU/Linux, est titulaire d'un MA en Techniques Sans Fil et Sécurité de Réseau du Programme de l'Académie Cisco sur Réseaux (*Cisco Networking Academy Program*, CCNA). Il travaille également en qualité d'Ingénieur Technique pour les systèmes informatiques, comme l'y autorise son diplôme délivré par l'Université Oberta, Catalogne (UOC). Daniel Lerch Hostalot travaille actuellement dans le secteur des télécommunications. Il maîtrise les langages de programmation suivants : C/C++, ShellScript, Java, Perl, PHP (programme de modules C).

Vous pouvez contacter l'auteur à l'adresse suivante : *dlerch@gmail.com*, url: *http://daniellerch.com*

Conclusion

Pour conclure, il est important de mentionner la répercussion que peuvent avoir les avancées mathématiques dans ce domaine. Un nombre impossible à factoriser aujourd'hui par les calculs informatiques, pourra l'être demain en quelques minutes à peine. Tout repose sur l'idée révolutionnaire d'un chercheur décidé à résoudre le problème. Toutefois, les quelques 20 ans de travaux réalisés sur les algorithmes du système RSA sont un véritable gage de sécurité, rendant cette éventualité encore lointaine.

La sortie d'ordinateurs quantiques va également considérablement menacer la sécurité de ce système cryptographique mondialement connu, notamment grâce à l'algorithme Shor (voir la section consacrée aux liens sur le Web) dont la procédure semble résoudre le problème de la complexité des polynômes. Cet algorithme va enfin permettre de factoriser une clé en un temps relativement raisonnable.