100 % SÉCURITÉ INFORMATIQUE

MISC

Multi-System & Internet Security Cookbook

France METRO : 9 €
DOM : 9 €
CAN : 13,50 $CAD
CH : 15 CHF
BEL : 9,90 €
POL/S : 1100 CFP
POL/A : 1400 CFP

L 16844 - 5 H - F: 9,00 € - RD

AVRIL / MAI 2012

HORS - SERIE N°5

UTILISATION POUR NÉOPHYTES

- OpenSSL ou PolarSSL à l'heure du choix

CRYPTOGRAPHIE MODERNE

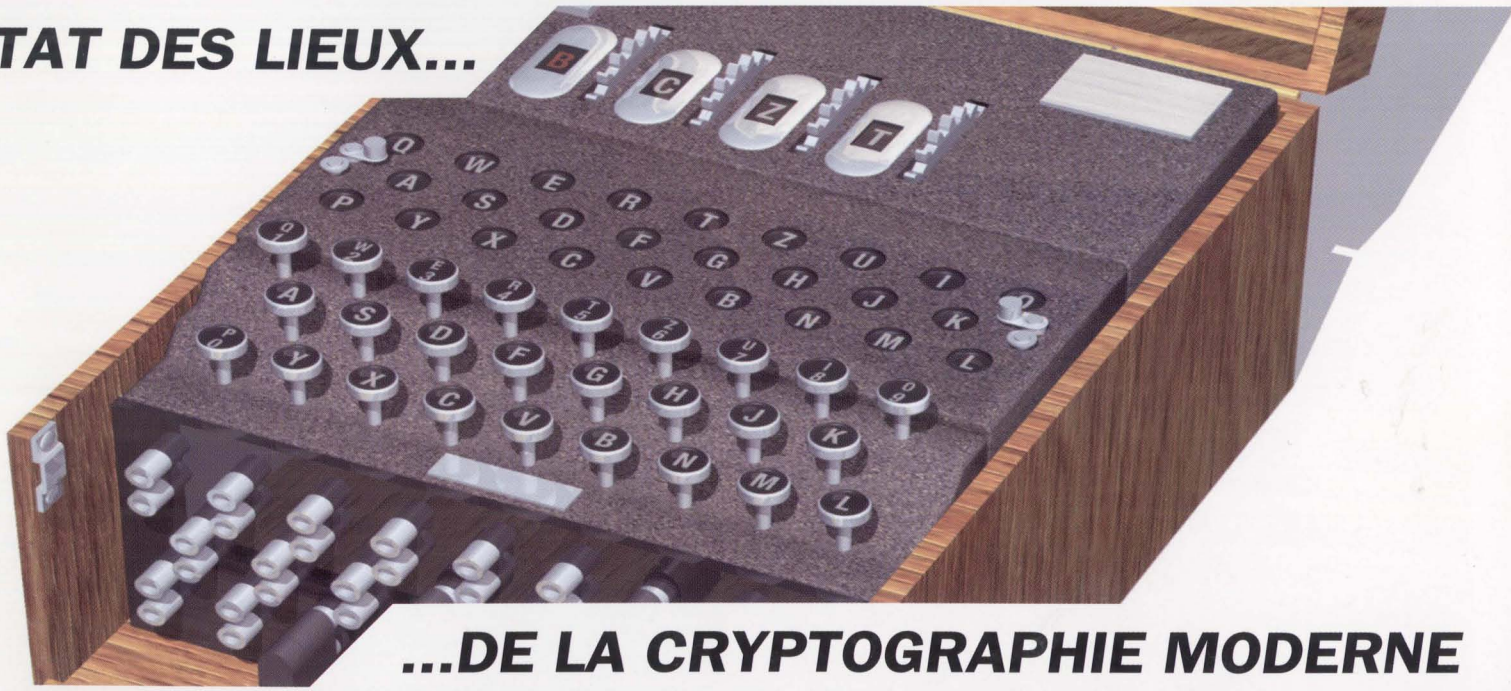
- DES, AES : de l'espérance de vie d'un algorithme symétrique
- Factorisation d'entiers : la voie royale du cassage de RSA

CRYPTOGRAPHIE DU FUTUR

- Chiffrement homomorphe ou comment ne manipuler que des données chiffrées
- Cryptographie quantique : science ou fiction ?

CRYPTOGRAPHIE : VOS SECRETS SONT-ILS BIEN GARDÉS ?

ÉTAT DES LIEUX...



...DE LA CRYPTOGRAPHIE MODERNE

ATTAQUES PAR CANAUX AUXILIAIRES

- Quand le temps ou la consommation laissent fuir les clés
- Techniques de programmation pour empêcher les fuites

APPLICATION : LES DRM

- Les DRM : au-delà d'un empêcheur de copier en rond
- Cryptographie boîte blanche, ou comment cacher ses clés en pleine lumière

REVERSE ET CRYPTO

- Reconstruire un algorithme cryptographique
- Virus, crypto et mobiles : nous avons les moyens de vous faire parler !

ÉDITO

Le lundi, c'est raviolis. Et quand Véro me demande de rendre mon édito dans 2 jours alors que je pensais avoir une semaine, je me dis que raviolis ou pas, je vais rater *Top Chef* à la télé.

Et ça tombe bien, parce que nous vous avons concocté un double numéro hors-série sur le thème de la cryptographie. Il est loin le temps du code de César. À l'époque, ça laissait baba à Rome, mais aujourd'hui, les choses ont bien changé. C'est pour ça que nous vous emmenons dans les cuisines.

Mais avant tout, je cède à la gourmandise et ne peux résister à vous annoncer une excellente nouvelle : il y aura un second hors-série, sur ce même thème. Le sujet est riche en calories, on a mis les petits plats dans les grands, et vous allez vous en mettre plein la panse (donc j'essuie).

Ce numéro est plutôt axé sur les fondements de la cryptographie contemporaine et sa mise en œuvre, là où le prochain insistera plus sur l'histoire et les attaques.

Passons donc au menu ! Je ne vais pas le détailler, vous avez le sommaire à côté. Mais je vais quand même revenir sur quelques articles. C'est ça l'avantage d'être rédacteur en chef, on peut goûter avant les autres :)

On le sait maintenant, la crypto, ce n'est pas pour les enfants ou Madame Michu. Ça demande de sacrées connaissances en mathématiques. Toutefois, nous n'avons pas voulu faire une énième plongée dans les rappels consultables (de restaurant) partout. Je tiens donc à remercier les auteurs de tous les articles qui ont fait des efforts pédagogiques impressionnants.

On retrouve là un problème identique à celui de la sécurité informatique : les bases ne sont plus maîtrisées par les nouveaux arrivants, et en même temps, ces bases deviennent de plus en plus complexes. Je suis encore sidéré quand je vois en cours des étudiants en dernière année d'école d'ingénieur qui ne savent pas ce qu'est un *debugger* ou comment ça fonctionne.

Alors, en tant que vieux aigri qui vient de fêter ses 25 ans (Véro, ne dis toujours rien, STP) et qui radote : oui, la sécurité, tout comme la crypto, sont des disciplines difficiles.

Mais revenons aux fourneaux. J'ai été impressionné par le recul de 3 articles en particulier.

Les 2 premiers portent sur la question de la factorisation d'une part, et sur les attaques contre les algorithmes symétriques d'autre part. Ces articles sont ardus, mais ils sont riches. On se rend alors compte de la similitude entre un algo crypto et un vin. Quand il sort, il est tout frais, mais il demande à être examiné. S'il passe cette première inspection, il acquiert ses lettres de noblesse. De plus en plus de monde s'intéresse à lui, il est à son apogée. Puis il dépérit au gré des attaques, jusqu'à tourner au vinaigre.

Les auteurs de ces articles (qu'ils soient encore remerciés de leurs efforts) reviennent sur les évolutions de la recherche au cours de ces 25 dernières années et expliquent en conséquence les évolutions passées et ce que l'avenir nous réserve.

Le dernier article que je voulais mettre en avant est celui sur les DRM, article qui n'est pas du tout technique. Déjà, vous pouvez chercher, la littérature sur ce sujet est quasi inexistante. Pour tout le monde, les DRM sont juste un truc pénible qui empêche de copier ses fichiers comme on veut. Grâce à cet article, on se rend compte que cette vision des DRM n'est que le petit bout de la lorgnette, et révèle surtout un système DRM mal foutu. L'auteur travaille dans ce domaine depuis des années, il pourrait être taxé de prosélytisme. Mais lisez-le, faites preuve de *fair-play*, et vous ne regarderez plus jamais votre iPod de la même manière.

Enfin, je voudrais également associer à ce numéro et au prochain Marc Rybowicz pour l'inspiration tirée de la journée anniversaire des 25 ans du Master CRYPTIS qu'il a organisée, et pour son aide dans la réalisation des numéros. Nous espérons qu'ils vous plairont comme nous nous sommes régalés à rassembler tout ce savoir.

Bon appétit,

Fred Raynal

SOMMAIRE

Rendez-vous au 27 avril 2012 pour le n°61 ! **www.miscmag.com**

MISC est édité par **Les Éditions Diamond**
B.P. 20142 / 67603 Sélestat Cedex
Tél. : 03 67 10 00 20 - **Fax** : 03 67 10 00 21
E-mail : cial@ed-diamond.com
Service commercial : abo@ed-diamond.com
Sites : www.miscmag.com
www.ed-diamond.com
IMPRIMÉ en Allemagne - PRINTED in Germany
Dépôt légal : A parution
N° ISSN : 1631-9036
Commission Paritaire : K 81190
Périodicité : Bimestrielle
Prix de vente : 8 Euros

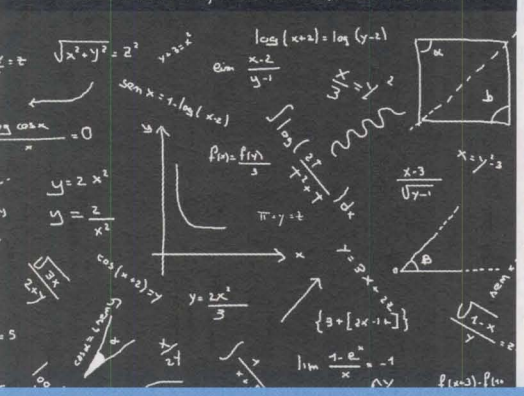
Directeur de publication : Arnaud Metzler
Chef des rédactions : Denis Bodor
Rédacteur en chef : Frédéric Raynal
Secrétaire de rédaction : Véronique Sittler
Conception graphique : Kathrin Troeger
Responsable publicité : Tél. : 03 67 10 00 27
Service abonnement : Tél. : 03 67 10 00 20
Impression : VPM Druck Rastatt / Allemagne
Distribution France : (uniquement pour les dépositaires de presse)
MLP Réassort :
Plate-forme de Saint-Barthélemy-d'Anjou. Tél. : 02 41 27 53 12
Plate-forme de Saint-Quentin-Fallavier. Tél. : 04 74 82 63 04
Service des ventes : Distri-médias : Tél. : 05 34 52 34 01

Membre April www.april.org

Charte de MISC

MISC est un magazine consacré à la sécurité informatique sous tous ses aspects (comme le système, le réseau ou encore la programmation) et où les perspectives techniques et scientifiques occupent une place prépondérante. Toutefois, les questions connexes (modalités juridiques, menaces informationnelles) sont également considérées, ce qui fait de MISC une revue capable d'appréhender la complexité croissante des systèmes d'information, et les problèmes de sécurité qui l'accompagnent. MISC vise un large public de personnes souhaitant élargir ses connaissances en se tenant informées des dernières techniques et des outils utilisés afin de mettre en place une défense adéquate.
MISC propose des articles complets et pédagogiques afin d'anticiper au mieux les risques liés au piratage et les solutions pour y remédier, présentant pour cela des techniques offensives autant que défensives, leurs avantages et leurs limites, des facettes indissociables pour considérer tous les enjeux de la sécurité informatique.

LA CRYPTO, COMMENT L'UTILISER, POUR QUOI FAIRE ?



DU BON USAGE DE LA CRYPTOGRAPHIE : OPENSSL ET POLARSSL POUR VOUS SERVIR

Olivier Tuchon – DGA/MI

mots-clés : *RAPPELS DE CRYPTOLOGIE / BIBLIOTHÈQUES CRYPTOGRAPHIQUES / OPENSSL / POLARSSL / CRYPTOGRAPHIE APPLIQUÉE*

L'allergie aux mathématiques est un phénomène assez courant chez la plupart d'entre nous. Concevoir ou casser un algorithme cryptographique sans de très solides bases théoriques paraît impensable à certains. Et pourtant, l'orchestration des mécanismes nécessaires à l'obtention d'une solution robuste est potentiellement à la portée de tous ceux qui n'ont pas un doctorat en théorie des nombres ! Cet article aidera le lecteur à se poser les bonnes questions et à faire le bon choix parmi le trop grand nombre d'algorithmes et de bibliothèques, le tout illustré à travers différents cas concrets avec OpenSSL et PolarSSL. Faites chauffer xterm, vim et gcc !

1 Objectifs de sécurité

La cryptographie ne résout pas tous les problèmes de sécurité mais elle aide dans bien des cas à les limiter quand on utilise de la bonne cryptographie. La question légitime qui se pose alors est « Qu'est ce qu'un bon algorithme de cryptographie ? ». On aurait tendance à répondre « un algorithme pas encore cassé... ».

La cryptographie a pour but d'assurer la confidentialité et l'intégrité des données :

- La confidentialité garantit que les personnes ne disposant pas du secret ne puissent pas découvrir le contenu réel du message.
- L'intégrité garantit l'identité numérique d'une personne, qu'un message n'a pas été modifié ou alors qu'il provient bien d'une personne numérique donnée.

Dans le cas de la preuve d'intégrité d'un message, on parle de signature ou d'authenticité, et dans le cas de la preuve d'identité d'une personne, on parle d'authentification.

1.1 Confidentialité

C'est l'opération de chiffrement qui garantit la confidentialité. Les algorithmes associés sont divisés en 2 grandes catégories :

- Le chiffrement symétrique : une seule clé, qui sert pour le chiffrement et le déchiffrement, est partagée entre toutes les personnes participant à un échange.
- Le chiffrement asymétrique : la clé publique du destinataire sert au chiffrement et sa clé privée sert pour le déchiffrement, on parle quelquefois de bi-clé.

En général, ces deux catégories ne sont pas concurrentes mais ont chacune des domaines spécifiques d'utilisation. Le coût élevé en ressources du chiffrement asymétrique le réserve au cas où le message à protéger est court, tandis que le chiffrement symétrique s'adapte très bien, par sa vitesse, aux messages très longs. Il est possible de mixer les deux, comme nous le verrons plus tard.

1.2 Intégrité

Comme pour la confidentialité, on observe une division entre le monde de la cryptographie symétrique où l'intégrité est assurée par les algorithmes dits « MAC » (pour *Message Authentication Code*) et le monde de la cryptographie asymétrique où l'on parle d'algorithmes de signature (la clé privée est utilisée pour signer et la clé publique associée vérifie la signature).

2 Briques de base

Cette section identifie les algorithmes satisfaisant les objectifs décrits ci-dessus. Bien sûr, une liste exhaustive

ne peut pas être donnée car elle serait bien trop longue. Cependant, nous nous appuierons sur les publications des organismes de standardisation tels que le NIST **[NIST]**, mais aussi d'un document édité par l'ANSSI qui regroupe un ensemble de règles et de recommandations concernant le choix et le dimensionnement des mécanismes cryptographiques **[RGS_B1]**.

2.1 Chiffrement

2.1.1 Chiffrement symétrique

La taille des clés pour un algorithme symétrique est un élément de sécurité très important. Une longueur suffisante rend la recherche exhaustive impossible. La taille minimale recommandée par exemple dans **[RGS_B1]** est de 128 bits.

Un chiffrement symétrique par bloc est également défini par la taille des blocs traités, **[RGS_B1]** recommande la taille des blocs à 128 bits également. Différents modes opératoires sont associés à ces primitives : ECB, CBC, CFB, CTS, XTS, XEX, ... Ils définissent la façon dont les blocs sont liés entre eux. Le NIST les spécifie à travers 5 documents (SP800-38{A,B,C,D,E}) **[SP800-38]**. Un aperçu visuel très clair est disponible sur la page Wikipédia **[MODES]**.

L'autre grande famille des mécanismes de chiffrement symétrique regroupe les algorithmes de chiffrement par flot. Ils ont l'avantage d'être extrêmement rapides mais ils ont souvent mauvaise presse : RC4, A5/1 et A5/2 sont tous très utilisés mais aussi cassés. Il est d'ailleurs recommandé dans **[RGS_B1]** « *d'employer des algorithmes de chiffrement par bloc et non des algorithmes de chiffrement par flot dédiés [...]* ». Nous pouvons toutefois lire dans ce même document qu' « *il est recommandé d'employer des algorithmes de chiffrement par flot largement éprouvés dans le milieu académique* ». Il faut comprendre ici que « largement éprouvés » = « pas encore cassés » !

Algorithme	Tailles de clé (en bits)	Taille de bloc (en bits)
3-DES	112, 168	64
RC5	32, 64, 128	jusqu'à 2048
AES	128, 192, 256	128
RC6	jusqu'à 2048	128
Camelia	128, 192, 256	128
Serpent	128, 192, 256	128

Liste non exhaustive d'algorithmes de chiffrement symétrique pouvant satisfaire au moins une des recommandations (RGS_B1-RecomCléSym-1, RGS_B1-RecomBlocSym-1)

SOSEMANUK
Salsa20/12
Trivium
Snow3G

Liste non exhaustive d'algorithmes de chiffrement par flot pouvant satisfaire la recommandation RGS_B1-RecomChiffFlot-1

2.1.2 Chiffrement asymétrique

À la différence de la cryptographie symétrique, la cryptographie asymétrique se base sur des problèmes mathématiques difficiles. Deux problèmes ont ainsi donné naissance à de nombreux algorithmes : la factorisation et le logarithme discret.

Étant donné $n = p * q$ avec p et q deux nombres entiers premiers, factoriser n (i.e. trouver p et q) est difficile si leur taille est suffisamment grande. Le dernier record public est pour un n de 768 bits **[RSA-768]**.

Le logarithme discret consiste à retrouver l'exposant e si on connaît y, x et p (un nombre premier) dans la formule suivante : $y = x^e \bmod p$. Le dernier record public est pour un p de 676 bits **[DLP-676]**.

Le cryptosystème RSA **[RSA]** est basé sur la factorisation et voici quelques règles/recommandations qu'on peut trouver dans **[RGS_B1]** :

- RègleFact-1 : « *La taille minimale du module est de 2048 bits, pour une utilisation ne devant pas dépasser l'année 2020* ».
- RègleFact-2 : « *Pour une utilisation au-delà de 2020, la taille minimale du module est de 4096 bits* ».
- RecomFact-1 : « *Il est recommandé, pour toute application, d'employer des exposants publics strictement supérieurs à 216 = 65536* ».

Pour le problème du logarithme discret, nous pouvons citer parmi les plus utilisés l'algorithme ElGamal **[ElGamal]**. La taille minimale du module recommandée pour cet algorithme est de « *2048 bits pour une utilisation ne devant pas dépasser 2020* » (RègleLogp-1).

2.2 Intégrité

2.2.1 MAC

Les MAC sont basés sur la cryptographie symétrique et sont de 2 types :

- ceux qui utilisent les fonctions de hachage (HMAC) ;
- ceux qui sont basés sur le mode opératoire d'un algorithme de chiffrement (CBC-MAC, GCM, ...).

Le choix des fonctions de hachage doit être fait aussi avec beaucoup de précaution. Le MD5 et le SHA1 ne sont plus considérés comme des algorithmes de hachages cryptographiques sûrs. La taille de l'empreinte est aussi un critère déterminant :

- RégleHash-1 : « *Pour une utilisation ne devant pas dépasser 2020, la taille minimale des empreintes générées par une fonction de hachage est de 200 bits* ».
- RecomHash-1 : « *L'emploi de fonctions de hachage pour lesquelles des attaques partielles sont connues est déconseillé* ».

Fonction de hachage	Taille de l'empreinte en bits
SHA256	256
SHA512	512
BLAKE	512
Skein	512

Quelques fonctions de hachage satisfaisant aux critères de [RGS_B1]

2.2.2 Signature

Basée sur la cryptographie asymétrique, la signature garantit également la non-répudiation. En effet, l'action de signer fait intervenir la clé privée de la bi-clé qui par définition ne doit rester qu'en la possession de la personne qui détient cette bi-clé. Elle ne pourra donc nier avoir signé tout document qui pourra être vérifié par la clé publique.

Les cryptosystèmes RSA, ElGamal et DSA peuvent être utilisés pour le calcul de signatures. Le document Digital Signature Standard **[FIPS186-3]** du NIST donne toutes les recommandations nécessaires pour utiliser au mieux RSA et DSA. Le cryptosystème RSA peut être appliqué indifféremment avec les normes PKCS#1v1.5 et PKCS#1v2.1 mais avec une clé de 1024 bits au minimum, tout comme le DSA. Enfin, pour l'utilisation de ElGamal, la règle RègleLogp-1 de **[RGS_B1]** doit être appliquée.

3 Assemblage des briques de base

Couvrir tous les problèmes de la cryptographie serait impossible en si peu de pages, mais nous allons nous concentrer sur une des problématiques : le chiffrement de fichier. En règle générale, protéger un fichier revient à assurer sa confidentialité et garantir son intégrité. Nous allons nous intéresser à 2 scénarios parmi les plus courants :

- protection par mot de passe ;
- protection par clé publique.

3.1 Dérivation d'un mot de passe

Le mot de passe que l'utilisateur doit saisir dans son logiciel cryptographique préféré ne constitue pas en soi une clé de chiffrement ou d'intégrité, il faut le dériver. Un bon mécanisme de dérivation doit respecter les 4 principes suivants :

- Avoir un bon mot de passe.
- Utiliser un sel aléatoire afin de se prémunir contre les attaques de type *RainbowTables*.
- Utiliser une bonne fonction pseudo-aléatoire non inversible afin de ne pas pouvoir remonter au mot de passe si on connaît la clé dérivée.
- Itérer un nombre conséquent de fois le processus afin d'obtenir un temps de dérivation raisonnable pour l'utilisateur mais suffisamment long pour ralentir l'attaquant.

L'algorithme PBKDF2 décrit dans la norme PKCS #5 v2.0 de *RSA Laboratories* **[PKCS5]** peut être un excellent choix **[PBKDF2]**.

Un bon mot de passe dépend de :

- sa longueur L ;
- la longueur de l'alphabet A auquel il appartient ;
- sa génération aléatoire.

La formule mathématique suivante donne l'équivalent en bits d'un mot de passe :

$$bits = L * \log_2(|A|)$$

On peut grossièrement apporter une appréciation sur un mot de passe correctement généré :

Taille en bits d'un mot de passe	Force d'un mot de passe
< 64	Très faible
64 < 80	Faible
80 < 100	Moyen
> 100	Très fort

« Sans entropie, la longueur n'est rien »

Une fois ce bon mot de passe généré se pose la question du stockage... Malheureusement, seule votre mémoire sera l'endroit idéal pour l'enregistrer, oubliez donc le bon vieux post-it sur l'écran et aussi celui sous votre clavier !

```
from math import log
import string

# calcule la force en bits d'un mot de passe de longueur L et
appartenant à l'alphabet A
def password_strength(alphabet, length):
  return length * (log(len(alphabet)) / log(2))

A = string.ascii_letters + string.digits  # A-Za-z0-9
L = 8
print "équivalent à %.2f bits" %(password_strength(A, L))

>>> équivalent à 47.63 bits
```

Code python2 pour calculer la force en bits d'un mot de passe étant donné son alphabet et sa longueur

3.2 Protection d'un fichier

3.2.1 Protection par mot de passe

Le mot de passe peut être une *passphrase*, qui, à robustesse identique, est certes plus longue qu'un mot de passe mais elle est surtout plus facile à retenir **[ARS]**.

Le schéma suivant est un schéma classique de protection de fichier basée sur l'utilisation d'un mot de passe :

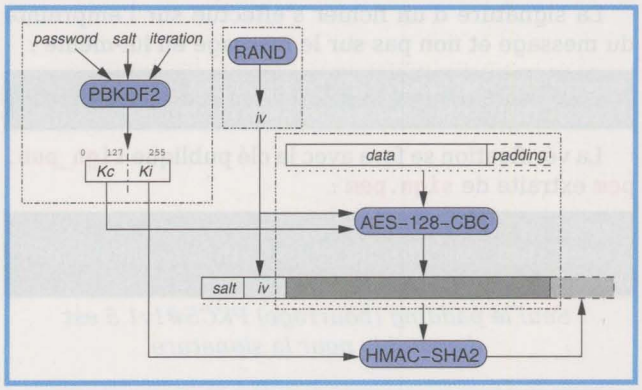


Figure 1 : Confidentialité et authenticité d'un fichier à partir d'un mot de passe

Ce mécanisme se base entièrement sur la cryptographie symétrique **[Encrypt-Then-MAC]** pour ses différentes étapes. Ainsi le message est chiffré dans un premier temps, ensuite c'est l'empreinte MAC du chiffré qui est calculée. Les algorithmes ont été choisis de façon arbitraire à partir des recommandations vues dans la partie 2 :

- dérivation du mot de passe à l'aide de l'algorithme PBKDF2 ;
- chiffrement des données claires avec AES-128-CBC ;
- intégrité des données chiffrées, du salt et de l'IV (pour *Initilisation Vector*) avec HMAC-SHA256.

Il est important de noter que l'iv IV utilisé par l'algorithme de chiffrement en mode CBC doit être différent à chaque utilisation pour que 2 clairs ne produisent pas le même chiffré pour une même clé Kc (sauf dans des cas très particuliers). En effet, dans un monde paranoïaque, cela peut être une information à l'avantage de l'attaquant que de posséder ou transmettre deux fois le même message chiffré...

De plus, le calcul d'intégrité s'effectue bien sur les données chiffrées et non pas sur les données claires de façon à ne pas apprendre le moindre bit sur les données en entrée de la fonction qui calcule le MAC si jamais cette dernière s'avérait vulnérable.

Enfin, la clé de chiffrement Kc et la clé d'intégrité Ki doivent être différentes et aléatoires.

3.2.2 Protection par clé publique

Le schéma dans la Figure 2 peut convenir pour l'envoi d'un même message chiffré à une liste de destinataires.

Ce modèle fonctionne sur la cryptographie hybride (mélange de cryptographie symétrique et de cryptographie asymétrique). Le principe est de chiffrer une seule fois les données claires avec un algorithme symétrique et d'en encapsuler la clé pour chaque destinataire (U1, ..., Un) avec un chiffrement asymétrique. Le message entier est signé par un mécanisme asymétrique afin de garantir à la fois l'identité de l'expéditeur et l'intégrité du message.

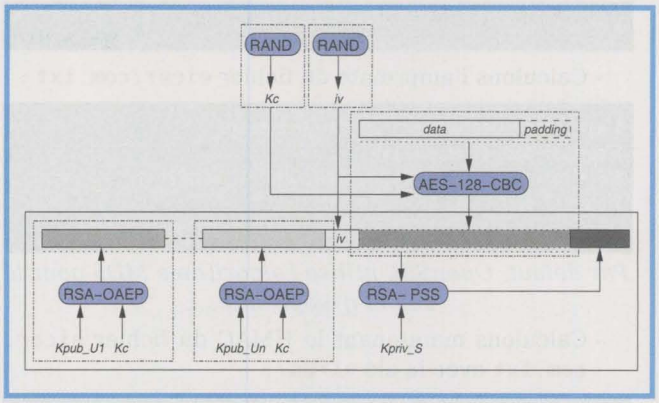


Figure 2 : Confidentialité et intégrité d'un fichier à destination de plusieurs personnes

Il est important de noter que chacun des protagonistes dispose de 2 bi-clés distinctes : une pour le chiffrement asymétrique et une pour la signature.

4 Bibliothèques cryptographiques : OpenSSL et PolarSSL

Dans cette dernière partie, nous allons nous concentrer sur la mise en application de tout ce que nous avons vu jusque-là à travers 2 bibliothèques : OpenSSL **[OpenSSL]** et PolarSSL **[PolarSSL]**.

OpenSSL n'est plus à présenter **[MISC32]**, elle est de loin la plus utilisée des bibliothèques cryptographiques mais elle n'en est pas moins complexe à prendre en main de par son manque de documentation ! Aussi, nous considérerons OpenSSL uniquement en lignes de commandes et nous aborderons la programmation de solutions cryptographiques en C avec la bibliothèque PolarSSL, qui a le bon goût d'être légère, bien écrite et surtout très documentée.

4.1 Voyage en lignes de commandes avec OpenSSL

Pour avoir les commandes disponibles avec OpenSSL, il suffit de taper dans votre terminal :

```
$ openssl -h
```

même si l'option **-h** n'est pas valide, elle permet d'afficher les informations qu'on souhaite !

4.1.1 Digest / HMAC

L'ensemble des algorithmes permettant de gérer l'intégrité avec la cryptographie symétrique est accessible avec la commande :


```
$ openssl dgst -h
```

- Calculons l'empreinte du fichier **eicar.com.txt** :

```
$ wget https://secure.eicar.org/eicar.com.txt
$ openssl dgst eicar.com.txt
MD5(eicar.com.txt)= 44d88612fea8a8f36de82e1278abb02f
$ openssl dgst -sha1 eicar.com.txt
SHA1(eicar.com.txt)= 3395856ce81f2b7382dee72602f798b642f14140
```

Par défaut, OpenSSL utilise l'algorithme MD5 pour le calcul d'empreinte...

- Calculons maintenant le HMAC du fichier **eicar.com.txt** avec la clé **v1rUs?** :

```
$ openssl dgst -hmac v1rUs? eicar.com.txt
HMAC-SHA1(plain.txt)= 919dc5fb01b1a34a2ed427b2d095cfcef6e1a578
$ openssl dgst -md5 -hmac v1rUs? eicar.com.txt
HMAC-MD5(eicar.com.txt)= 77a080bc5e329a32ccc96d69c1805051
```

Par défaut, OpenSSL utilise un HMAC-SHA1

4.1.2 Chiffrement symétrique

L'ensemble des algorithmes permettant d'assurer la confidentialité avec la cryptographie symétrique est accessible avec la commande :

```
$ openssl enc -h
```

- Chiffrement du fichier **plain.txt** avec le mot de passe MISC en utilisant l'AES-128-CBC :

```
$ echo Misc_Mag > plain.txt
$ openssl enc -k MISC -e -aes-128-cbc -in plain.txt -out cipher.raw -p
salt=A91D9E8920974263
key=C3E953F81962D118FADF783F6468D298
iv =2821FB4AD692031649AE2608B2A37498
```

Si on exécute de nouveau cette commande, on obtient un résultat différent :

```
$ openssl enc -k MISC -e -aes-128-cbc -in plain.txt -out cipher_2.raw -p
salt=4600803ECEBA1FE9
key=84A2096A3383B9659D1067E5F71F6E1F
iv =72015A110B08CC6416B44102474A4F4F
```

Les valeurs **salt**, **iv** et **key** ne sont pas identiques, ce qui indique qu'OpenSSL utilise une dérivation de mot de passe. Il faut cependant lire les sources de la bibliothèque (excellent exercice par ailleurs que l'auteur recommande vivement au lecteur) pour savoir quel mécanisme précis est implémenté pour cette tâche !

4.1.3 Signature

Nous allons utiliser l'algorithme asymétrique RSA pour signer notre fichier **plain.txt**. Il faut dans un premier temps générer notre bi-clé de taille 1024 bits qui sera stockée dans le fichier **sign.pem**.

```
$ openssl genrsa  -out sign.pem 1024
Generating RSA private key, 1024 bit long modulus
...++++++
.....................................++++++
e is 65537 (0x10001)
```

La signature d'un fichier s'effectue sur l'empreinte du message et non pas sur le message en lui-même :

```
$ openssl dgst -sha1 -binary plain.txt > plain.sha1
$ openssl rsautl -sign -inkey sign.pem -in plain.sha1 -out plain.sign
```

La vérification se fera avec la clé publique **sign_pub.pem** extraite de **sign.pem** :

```
$ openssl rsa -in sign.pem -pubout > sign_pub.pem
$ openssl rsautl -pubin -inkey sign_pub.pem -in plain.sign -out plain.verif
$ diff plain.sha1 plain.verif
```

Seul le padding (bourrage) PKCS#1v1.5 est disponible pour la signature

4.1.4 Chiffrement asymétrique

Le RSA est utilisé ici pour chiffrer une clé symétrique de 128 bits.

```
$ openssl genrsa -out ciph.pem 1024
$ openssl rsa -in ciph.pem -pubout > ciph_pub.pem
$ openssl rand 16 > key.plain
$ openssl rsautl -encrypt -oaep -inkey ciph_pub.pem -pubin -in key.
plain -out key.ciph
```

4.1.5 Certificats

Nous allons décrire les étapes de la création d'un certificat SSL auto-signé pour son serveur web.

1- Création de l'autorité de certification (CA)

```
$ openssl genrsa 1024 > ca.key
$ openssl req -new -x509 -days 365 -key ca.key > ca.crt
```

Avec son certificat **ca.crt**, la CA signera les requêtes de certifications qu'elle recevra.

2- Génération d'une bi-clé du serveur (**server.key**) et sa demande de certification (**server.csr**) :

```
$ openssl genrsa -out server.key 1024
$ openssl req -new -key server.pem > server.csr
```

Il faut remplir les champs comme vous le souhaitez, mais le champ *Common Name* doit correspondre à l'adresse URL de votre site.

3- Signature du certificat du serveur par la CA

```
$ openssl x509 -req -in server.csr -out server.crt -CA ca.crt -CAkey
ca.key -CAcreateserial -CAserial ca.srl
```

Le fichier **server.crt** est le certificat du serveur.

4.2 Du côté de PolarSSL

4.2.1 Présentation

La bibliothèque PolarSSL **[POLAR]** implémente le protocole SSL/TLS en C avec comme cible principale les systèmes embarqués. La documentation est basée sur les

commentaires des fichiers en-têtes et dans les sources des programmes d'exemples fournis dans l'archive. Pour le côté *people*, cette bibliothèque a été initialement conçue par Christophe Devine et s'appelait à l'époque XySSL, elle est actuellement maintenue par Paul Baker.

Chiffrement symétrique	AES, DES, 3DES, ARC4, Camellia, XTEA
Mode opératoire	ECB, CBC, CTR, CFB
Chiffrement asymétrique	RSA PKCS#1v1.5, RSA PKCS#1v2.1
Signature	RSA PKCS#1v1.5, RSA PKCS#1v2.1
Fonctions de hachage + HMAC	MD2, MD4, MD5, SHA-1, SHA-224, SHA-256, SHA-384, SHA-512
Protocoles	DH, SSLv3, TLSv1.0, TLSv1.1 (partie client)
PRNG	CTR_DRBG, HAVEGE
PKI	Certificats X.509v3, lecture de CRL
Carte à puce	Interface PKCS#11 via OpenSC

Liste exhaustive des fonctionnalités offertes par la bibliothèque

En plus d'être légère, elle est très modulable car chaque algorithme est codé dans un seul fichier. Il est donc très facile d'isoler uniquement le mécanisme ciblé et de ne prendre que son fichier source et son en-tête pour les inclure dans un autre projet sans aucune autre modification.

Il faut noter que le protocole TLSv1.2 n'est pas présent car il fait appel aux courbes elliptiques dont le support n'a pas encore officiellement été ajouté. Il est cependant possible de coder l'arithmétique sur courbes elliptiques **[ECC]** facilement puisque la bibliothèque « Grands Nombres » fournie est relativement facile à prendre en main. Cela constitue un très bon exercice pour le lecteur ;-)

Par souci de place, seules les lignes de codes les plus pertinentes ont été retenues. Les codes complets sont disponibles sur la Toile.

4.2.2 Exemple 1 : PBKDF2

Nous allons ici coder l'algorithme de dérivation de mot de passe PBKDF2.

Notre fonction prend en arguments un mot de passe, un sel, le nombre de tours de boucle interne à effectuer et la longueur souhaitée de la clé.

```
#include "polarssl/sha2.h" /* *** PRF = HMAC_SHA256 *** */
#define LEN_DIGEST 32
int32_t pbkdf2(
    u_int8_t *dkey, size_t len_dkey,
    int8_t *password, size_t len_password,
    u_int8_t *salt, size_t len_salt,
    u_int32_t iterations)
{
  int32_t ret;
  u_int count, i, j;
  size_t r;
  u_int8_t *salt_ext;  // S||i, pour le calcul de Ui
  u_int8_t d0[LEN_DIGEST], d1[LEN_DIGEST], d2[LEN_DIGEST]; // variables temporaires
  [...]
  salt_ext = (u_int8_t *)malloc((len_salt + 4) * sizeof(u_int8_t));

  [...]
  /* *** PBKDF2 ! *** */
  memcpy(salt_ext, salt, len_salt);
  for(count = 1; len_dkey > 0; count++)
  {
    salt_ext[len_salt+0] = (count >> 24) & 0xFF;
    salt_ext[len_salt+1] = (count >> 16) & 0xFF;
    salt_ext[len_salt+2] = (count >>  8) & 0xFF;
    salt_ext[len_salt+3] = (count      ) & 0xFF;
    sha2_hmac((unsigned char *)password,len_password,salt_ext,len_salt+4,d1, 0);
    memcpy(d0, d1, LEN_DIGEST);
    /* *** Calcul des Ti = F(P, S, c, i) *** */
    for(i = 1; i < iterations; i++)
    {
      sha2_hmac((unsigned char *)password, len_password, d1, LEN_DIGEST, d2, 0);
      memcpy(d1, d2, LEN_DIGEST);
      for(j = 0; j < LEN_DIGEST; j++)
        d0[j] ^= d2[j];
    }
    r = MIN(len_dkey, LEN_DIGEST);
    /* *** d0 contient un morceau de la clé dérivée *** */
    memcpy(dkey, d0, r);
    dkey += r;
    len_dkey -= r;
  }

  [...]

  return ret;
}
```

Le code complet est disponible ici **[PASTEBIN1]**.

4.2.3 Exemple 2 : protection d'un buffer

Il s'agit ici d'implémenter le schéma vu dans la partie 3.2.1.

Tout ce dont nous avons besoin est déjà codé à présent : HMAC-SHA256, PBKDF2, AES-128-CBC à la gestion près du *padding* pour le mode CBC. En effet, PolarSSL ne gère pas le padding dans les modes qui en nécessitent un, il revient à l'application qui appelle la fonction de chiffrement de le faire.

En regardant le prototype de la fonction `aes_crypt_cbc`, on s'aperçoit effectivement que si la longueur `length` de l'entrée `input` n'est pas correcte (i.e. multiple de la taille d'un bloc), elle renvoie le code d'erreur `POLARSSL_ERR_AES_INVALID_INPUT_LENGTH`.

Nous allons donc utiliser et coder le padding appelé 0x80 qui consiste à ajouter le motif 1000...0002 afin de compléter le dernier bloc pour qu'il ait la bonne taille.

Les étapes de notre protection seront les suivantes :

1- dérivation du mot de passe ;

2- dérivation des sous-clés (chiffrement et intégrité) ;

3- ajout du padding 0x80 ;

4- génération de l'IV ;

5- chiffrement ;

6- calcul de l'intégrité ;

7- effacement de la pile.

Ce qui donne le code :

```
#include "polarssl/sha2.h"
#include "polarssl/aes.h"
#include "polarssl/havege.h"
#include "pbkdf2.h"

const u_int8_t padding[16] = {
  0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};

int32_t protect_buffer(
    u_int8_t **output, size_t *len_output,
    u_int8_t *input, size_t len_input,
    int8_t *password, size_t len_password,
    u_int8_t *salt, size_t len_salt,
    u_int32_t iterations)
{
  u_int8_t master_key[32]; //Cle maitre Km = Kc || Ki
  u_int8_t cipher_key[16]; //Cle de chiffrement, Kc
  u_int8_t auth_key[16]; //Cle d'authenticite, Ki

  u_int8_t *input_padd;     // input + padding
  size_t len_padd;
  u_int8_t *cipher;     // AES-CBC-128(Kc, input_padd)
  u_int8_t iv[16];

  aes_context aes_ctx;    // contexte de chiffrement
  sha2_context sha2_ctx; // contexte d'intégrité
  havege_state prng_ctx; // contexte du PRNG
  [...]

  /* *** 1/ Derivation du mot de passe *** */
  local_ret = pbkdf2(master_key, 32, password, len_password, salt, len_salt,
iterations);

  /* *** 2/ Derivation des sous cles → Km = Kc || Ki *** */
  memcpy(cipher_key, master_key, 16);
  memcpy(auth_key, master_key+16, 16);

  /* *** 3/ Ajout du padding *** */
  if (len_input % 16 != 0) //Le dernier bloc n'est pas de la bonne taille
    len_padd = 16 - (len_input % 16);
  else //Sinon on ajoute un bloc entier de padding
    len_padd = 16;
  //input_padd = input + padding
  input_padd = (u_int8_t *)malloc((len_input+len_padd)*sizeof(u_int8_t));
  //cipher = salt + iv + AES-CBC-128(Kc, input_padd) + HMAC-SHA256(...)
  cipher = (u_int8_t *)malloc((len_salt + sizeof(iv) +
    len_input + len_padd + 32)*sizeof(u_int8_t));
  memcpy(input_padd, input, len_input);
  memcpy(input_padd+len_input, padding, len_padd);

  /* *** 4/ Génération aléatoire de l'IV *** */
  havege_init(&prng_ctx);
  havege_random(&prng_ctx, (unsigned char *)iv, sizeof(iv));
  memcpy(cipher+len_salt, iv, sizeof(iv));
  memcpy(cipher, salt, len_salt);

  /* *** 5/ Chiffrement *** */
  aes_setkey_enc(&aes_ctx, (unsigned char *)cipher_key, 128);
  aes_crypt_cbc(&aes_ctx, AES_ENCRYPT, len_input+len_padd,
    (unsigned char *)iv, input_padd, cipher+len_salt+sizeof(iv));

  /* *** 6/ Ajout de l'authenticité *** */
  sha2_hmac(auth_key, 16, cipher, len_salt + sizeof(iv) + len_input + len_padd,
    cipher+len_padd+sizeof(iv)+len_input+len_padd, 0);

  *output = cipher;
  *len_output = len_salt + sizeof(iv) + len_input + len_padd + 32;
  [...]
}
```

Le code complet est disponible ici **[PASTEBIN2]**.

La bibliothèque contient le même type de programme à titre d'exemple (**programs/aes/aescrypt2.c**) mais nous pouvons apporter quelques critiques quant aux mécanismes choisis :

- L'IV est l'empreinte de la taille du fichier et du nom du fichier. Ce qui signifie que si on chiffre 2 fois le même fichier avec la même clé, le résultat sera le même. On peut noter que le calcul de l'IV donnera le même résultat pour 2 fichiers différents mais de même longueur et qui ont le même nom !
- La dérivation de mot de passe n'est pas basée sur un mécanisme standard et elle ne comporte pas de sel.
- La clé de chiffrement est la même que la clé d'intégrité.

4.2.4 Exemple 3 : échange d'un secret

Nous allons ici nous intéresser à l'échange d'un secret avec l'algorithme Diffie-Hellman signé. Le protocole Diffie-Hellman **[DH]** dans sa forme originale est vulnérable à l'attaque *Man-In-The-Middle*, nous allons introduire une contre-mesure possible : la signature des échanges.

Alice et Bob (Enfin ! Ils en ont mis du temps à apparaître dans cet article) disposent chacun d'une bi-clé de signature et des paramètres publics Diffie-Hellman : un générateur g et un module p.

1- Alice envoie $g^a \bmod p$ et la signature $S_{Alice} = Sign(g^a \bmod p)$.

2- Bob vérifie la signature S_{Alice}.

3- Si la signature est bien vérifiée, il envoie $g^b \bmod p$ et la signature $S_{Bob} = Sign(g^b \bmod p)$.

4- Alice vérifie la signature S_{bob}.

5- Alice et Bob peuvent dériver le secret commun $g^{ab} \bmod p$.

Si les vérifications échouent, cela signifie que les messages ont été altérés pendant la transmission. Il appartient donc à l'application qui implémente le protocole de décider de la marche à suivre en cas d'erreur.

Dans notre exemple avec PolarSSL, nous allons signer les messages avec RSA-PKCS#1v1.5.

Nous allons nous concentrer sur :

1- l'utilisation du contexte DH ;

2- la génération d'une bi-clé RSA ;

3- la signature d'un message.

```
/* *** Utilsation de Diffie-Hellman *** */
#include "polarssl/havege.h"
#include "polarssl/dhm.h"
#include "polarssl/bignum.h"

#define DH_SZ_BITS      2048

const int8_t *dh_P = "FD.. .. ..."; // Valeur du module P en hexadécimal
const int8_t *dh_G = "04"; // Valeur du générateur G  en hexadécimal

int32_t test_dh(void)
{

  /* *** Parametres d'Alice *** */
  dhm_context dh_alice;
  u_int8_t dh_public_alice[DH_SZ_BITS>>3]; //g^a mod p
  u_int8_t dh_secret_alice[DH_SZ_BITS>>3]; //g^ab mod p
  /* *** Parametres de Bob *** */
  dhm_context dh_bob;
  u_int8_t dh_public_bob[DH_SZ_BITS>>3]; //g^b mod p
```

```
  u_int8_t dh_secret_alice[DH_SZ_BITS>>3]; //g^ab mod p

  size_t dh_public_size;
  size_t dh_private_size;
  havege_state prng_ctx;

  /* *** Initialisation *** */
  [...]
  havege_init(&prng_ctx);

  /* *** Chargement des paramètres publiques DH *** */
  [...]

  /* *** Génération de la valeur publique d'Alice *** */
  dhm_make_params(&dh_alice, 128, dh_public_alice, &dh_public_size,
       havege_random, &prng_ctx);
  /* *** Génération de la valeur publique d'Alice *** */
  dhm_make_params(&dh_alice, 128, dh_public_bob, &dh_public_size,
       havege_random, &prng_ctx);

  /* *** Alice récupère la valeur publique de Bob et calcule le secret commun *** */
  mpi_cpy(&dh_alice.GY, &dh_bob.GX);
  dhm_calc_secret(&dh_alice, dh_secret_alice, &dh_private_size);

  /* *** Bob récupère la valeur publique d'Alice et calcule le secret commun *** */
  mpi_cpy(&dh_bob.GY, &dh_alice.GX);
  dhm_calc_secret(&dh_bob, dh_secret_bob, &dh_private_size);
  [...]
}
```

Le code complet est disponible ici **[PASTEBIN3]**.

```
/* *** Génération d'une bi-clé RSA puis signature d'un message *** */

#include "polarssl/havege.h"
#include "polarssl/rsa.h"
#include "polarssl/sha2.h"


#define RSA_SZ_BITS    2048

int32_t test_sign_rsa(u_int8_t message, size_t len_msg)
{
  rsa_context rsa_alice;
  havege_state prng_ctx;

  u_int8_t digest[32];
  u_int8_t signature[RSA_SZ_BITS>>3];

  [...]
  havege_init(&prng_ctx);

  /* *** Génération de la bi-clé RSA-2048 de signature d'Alice
              *** e = 0x10001
           */
  rsa_init(&rsa_alice, RSA_PKCS_V15, 0);
  rsa_gen_key(&rsa_alice, havege_random, &prng_ctx, RSA_SZ_BITS, 0x10001);

  /* *** Calcul de l'empreinte SHA256 de 'message' qui sera signée *** */
  sha2(message, len_msg, digest, 0);

  /* *** Signature RSA PKCS#1v1.5 *** */
  rsa_pkcs1_sign(&rsa_alice, NULL, NULL, RSA_PRIVATE, SIG_RSA_SHA256, 32,
digest, signature);
  /* *** Effacement de la bi-clé RSA *** */
  rsa_free(&rsa_alice);
  [...]
}
```

Le code complet est disponible ici **[PASTEBIN4]**.

Nous avons maintenant toutes les fonctions à utiliser pour écrire un échange Diffie-Hellman signé... Enfin, vous avez maintenant toutes les fonctions à utiliser pour écrire cet échange !

Conclusion

Nous n'avons qu'entre-aperçu au travers de la simple problématique du chiffrement de fichier que beaucoup de questions pouvaient se poser. Le choix des algorithmes et les tailles de clé ont été abordés en début d'article, mais une lecture attentive et très longue (qui a dit fastidieuse ?) des standards pourra vous donner une idée bien plus claire même s'ils ne sont valides qu'au moment où ils ont été rédigés. Le choix de la bibliothèque cryptographique n'est pas à prendre à la légère, elle est un véritable problème. En effet, même si un AES est normalisé, il n'est pas rare de voir des erreurs de programmation qui peuvent se révéler fondamentales pour la sécurité globale d'une application (mauvaise génération de clés, secrets non effacés en mémoire, ...).

J'espère que vous êtes mieux armé (la cryptographie était considérée comme une arme il n'y a pas si longtemps...) pour survivre dans le monde de la programmation de solutions cryptographiques. Avec tout le choix qui s'offre à vous, chiffrez bien ! ■

REMERCIEMENTS

Je tiens à remercier Robert Erra pour m'avoir incité à écrire cet article et aux étudiants des Mastères Spécialisés de l'ESIEA pour avoir été les cobayes du pseudo-culte que je voue à PolarSSL ;-).

BIBLIOGRAPHIE

[NIST] http://www.nist.gov

[RGS_B1] http://www.ssi.gouv.fr/IMG/pdf/RGS_B_1.pdf

[SP800-38] http://csrc.nist.gov/groups/ST/toolkit/BCM/current_modes.html

[MODES] http://fr.wikipedia.org/wiki/Mode_d%27op%C3%A9ration_%28cryptographie%29

[ElGamal] Taher ElGamal (1985). "A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms"

[PKCS5] ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-5v2/pkcs5v2-0.pdf

[PBKDF2] http://sid.rstack.org/blog/index.php/400-pbkdf2-a-l-epreuve-du-fbi

[POLAR] http://polarssl.org/

[ECC] Guide to Elliptic Curve Cryptography, de Darrel Hankerson, Alfred Menezes et Scott Vanstone aux éditions Springer

[DH] New Directions in Cryptography, Whitfield Diffie , Martin E. Hellman, 1976

[Encrypt-Then-MAC] Mihir Bellare, Chanathip Namprempre. Authenticated Encryption: Relations among notions and analysis of the generic composition paradigm, 2007

[RSA] R. Rivest, A. Shamir, L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. Communications of the ACM, Vol. 21 (2), pp.120–126. 1978

[RSA768] http://www.rsa.com/rsalabs/node.asp?id=2093

[MISC32] A. Apvrille, Conception et architecture de la bibliothèque cryptographique d'OpenSSL, MISC 32

[DLP-676] http://eprint.iacr.org/2010/090

[ARS] http://arstechnica.com/business/news/2011/10/when-passwords-attack-the-problem-with-aggressive-password-policies.ars

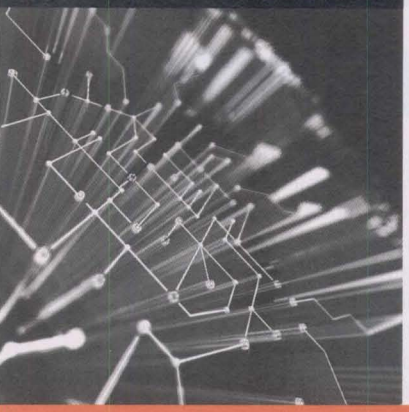
[PASTEBIN1] http://pastebin.com/xUDpESzj

[PASTEBIN2] http://pastebin.com/2DgfsLvP

[PASTEBIN3] http://pastebin.com/pnJaugEj

[PASTEBIN4] http://pastebin.com/0mS7eEn6

LES AVANCÉES DE LA CRYPTO MODERNE



DE L'ESPÉRANCE DE VIE D'UN ALGORITHME SYMÉTRIQUE

(OU L'AES DIX ANS APRÈS)

Anne Canteaut et Marine Minier

mots-clés : CRYPTOGRAPHIE / CHIFFREMENT / AES / CRYPTANALYSE

La cryptographie symétrique reste l'un des moyens les plus rapides de chiffrer des messages échangés par deux parties qui possèdent une clé commune, cette clé étant le plus souvent transmise de façon sûre via l'utilisation de la cryptographie asymétrique. La définition de primitives communes en cryptographie symétrique reconnues au niveau international remonte à la standardisation du DES en 1977 comme algorithme de chiffrement par bloc. Depuis cette date, les fonctions de hachage MD5 et SHA-1 ont été standardisées dans le courant des années 90 et l'AES a été choisi comme nouveau standard de chiffrements par bloc en octobre 2000 par le NIST au terme d'une compétition internationale publique de trois ans réunissant quinze candidats.

1 Introduction

Les progrès importants faits en cryptanalyse au cours des dernières années montrent que peu d'algorithmes symétriques résistent plus de quinze ans aux attaques. Les fonctions de hachage MD5 et SHA-1 ont été cassées respectivement treize et dix ans après leur publication. Parmi les huit chiffrements à flot recommandés en 2008 à l'issue du concours international eSTREAM, un a déjà été cryptanalysé et un deuxième présente des faiblesses importantes. Et l'exemple le plus frappant est naturellement celui de l'AES. L'AES vient en effet d'être victime pour la première fois d'une attaque qui semble légèrement moins coûteuse que la recherche exhaustive de la clé. Mais il a surtout motivé d'importants développements en cryptanalyse qui, s'ils ne mettent pas en danger la sécurité du chiffrement par bloc lui-même, se sont révélés plus graves lorsque l'AES était employé dans d'autres contextes (fonctions de hachage, scénarios à plusieurs clés, ...). Doit-on en conclure que tous les algorithmes symétriques récents, même conçus par des spécialistes reconnus, ont toutes les chances d'être cassés au cours de la décennie ?

Nous tenterons donc dans cet article de faire le point sur l'état des recherches concernant la cryptanalyse de l'AES. Nous décrirons dans une première partie cet algorithme. Et nous nous intéresserons ensuite à trois périodes de sa vie : les années d'incertitude (2001-2004), l'âge d'or (2005-2008), et finalement ce qu'on pourrait considérer comme les premières alertes (2009-2011).

2 Description de l'AES

2.1 Comment construire un chiffrement par bloc ?

Un chiffrement par bloc découpe un message en blocs de même taille, n bits (n doit être suffisamment grand pour éviter les attaques par dictionnaire, typiquement n=128 bits pour l'AES), et ensuite, chiffre chacun de ces blocs (nous n'aborderons pas ici le mode opératoire d'un tel système, c'est-à-dire la façon dont on l'utilise pour traiter les blocs de message successifs, mais celui-ci est extrêmement important). De façon générique, on décrit un algorithme de chiffrement par bloc E comme une application qui associe à un bloc de message de n bits, m, et à une clé de l bits, K, un bloc chiffré de n bits, c : $E_K(m)=c$. Pour pouvoir déchiffrer, il faut naturellement

que E_K soit une permutation pour toutes les clés possibles K. L'algorithme de déchiffrement correspondant, D_K, permet de déchiffrer c avec la même clé K : $D_K(c)=m$, autrement dit D_K est l'inverse de la fonction E_K pour toute clé K. La permutation de chiffrement E_K paramétrée par la clé K doit donc être facile à inverser. La clé K doit être également prise dans un espace suffisamment grand pour se prémunir contre la recherche exhaustive (essayer toutes les clés). Par exemple dans le cas de l'AES, K est de taille 128, 192 ou 256 bits, ce qui fait que la recherche exhaustive nécessite 2^{128}, 2^{192} ou 2^{256} essais, cette dernière borne étant très proche du nombre d'atomes présents dans l'univers estimé à 2^{265}.

Les méthodes de construction de tels algorithmes restent très empiriques mais la plus usitée est l'utilisation d'algorithmes itératifs. Ces algorithmes sont composés de r étages et à chaque étage, ils font agir la même permutation interne f_{K_i}, appelée fonction de tour, paramétrée par une sous-clé K_i. Les sous-clés K_1,..., K_r sont différentes à chaque étage et générées à partir de la clé secrète K et d'un algorithme de génération de sous-clés (voir Figure 1).

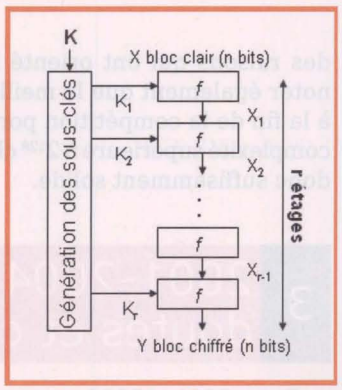


Figure 1 : Structure générale d'un chiffrement par bloc itératif

Une méthode classique de construction de la permutation interne f est fondée sur ce qu'on appelle les réseaux de substitution-permutation (voir Figure 2). À chaque étage, on applique au bloc d'entrée X_i une substitution non linéaire puis une fonction généralement linéaire. Ces réseaux prennent au mot les deux concepts définis par Shannon : la confusion et la diffusion. La substitution, en général représentée par une série de boîtes S (des permutations non linéaires) appliquées en parallèle, garantit, si elle est bien choisie, une bonne confusion : elle fait disparaître les structures tant linéaires qu'algébriques du chiffrement. La permutation, elle, garantit une bonne diffusion de l'information : elle fait en sorte que chaque bit de sortie soit influencé par le plus grand nombre possible de bits d'entrée. Précisons également que l'usage des mots substitution et permutation est abusif. On peut effectivement parler de substitution en ce qui concerne les boîtes S même s'il peut y avoir confusion avec le terme de permutation, mais le terme de permutation est beaucoup trop approximatif pour désigner l'application linéaire qui suit. Cependant, l'emploi de ces termes reste classique en cryptographie, nous continuerons donc à les utiliser dans la suite de ce paragraphe.

Ce réseau doit également vérifier ce qu'on appelle l'effet avalanche : les modifications du texte clair doivent être de plus en plus importantes au fur et à mesure que les données se propagent dans la structure de l'algorithme. De ce fait, en perturbant un seul bit en entrée, on obtient idéalement une sortie totalement différente, d'où le nom de ce phénomène. En effet, si la diffusion des modifications des entrées sur les sorties n'est pas suffisamment grande, il sera possible d'établir des prédictions sur les entrées à partir de l'observation des sorties et donc d'obtenir un biais statistique.

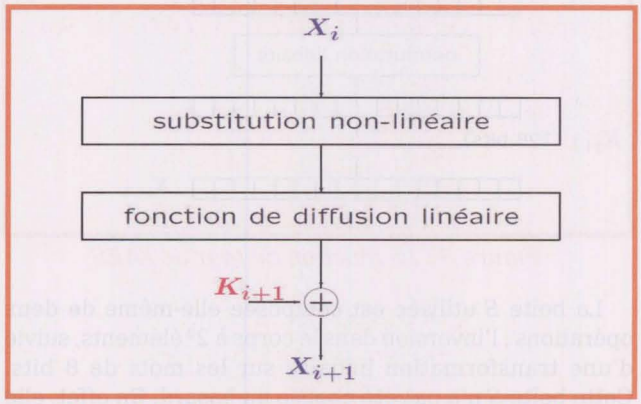


Figure 2 : Structure générale d'un étage d'un réseau de substitution-permutation

2.2 Description de l'AES

L'AES est le standard actuel de chiffrement par bloc et utilise un réseau de substitution-permutation. Mais avant de décrire l'algorithme en lui-même, rappelons tout d'abord dans quel contexte Rijndael a été choisi pour devenir l'AES. En 1997, sachant déjà l'ancien standard de chiffrement par bloc, le DES, vulnérable, le NIST (*National Institute for Standard and Technology*) lançait un appel pour choisir un nouvel algorithme de chiffrement par bloc dont la longueur des clés de 128, 192 et 256 bits devait permettre de l'utiliser au cours du 21ème siècle. Ce nouvel algorithme porterait le nom d'AES (*Advanced Encryption Standard*). 15 propositions émanant des quatre coins du monde ont été soumises. En août 1999, après une première phase de sélection, 5 finalistes ont été retenus, il s'agissait de Mars, Serpent, Twofish, RC6 et Rijndael. C'est finalement Rijndael qui a été choisi comme AES le 2 octobre 2000. Il est aujourd'hui spécifié dans la norme FIPS-197 **[13]**.

L'AES chiffre des blocs de 128 bits sous des clés de 128, 192 ou 256 bits. Le nombre d'étages est variable en fonction de la longueur de la clé : 10 tours si la clé maître fait 128 bits, 12 tours pour une clé de 192 bits et 14 tours pour une clé de 256 bits. Ces tours sont précédés d'une addition avec la sous-clé K_0 (un ou-exclusif bit à bit) tandis que le dernier tour ne contient pas l'opération *MixColumns* décrite plus loin. L'AES est un chiffrement orienté octet, c'est-à-dire que le bloc de 128 bits à chiffrer est représenté par 16 octets.

La fonction de tour de l'AES décrite à la figure 3 fait agir une même boîte *S* 16 fois en parallèle sur chacun des octets (cette étape s'appelle *SubBytes* dans la FIPS-197)

puis une permutation linéaire (composée des opérations *ShiftRows* et MixColumns) suivie par un ou-exclusif bit à bit avec la sous-clé K_i du tour (appelé *AddRoundKey*).

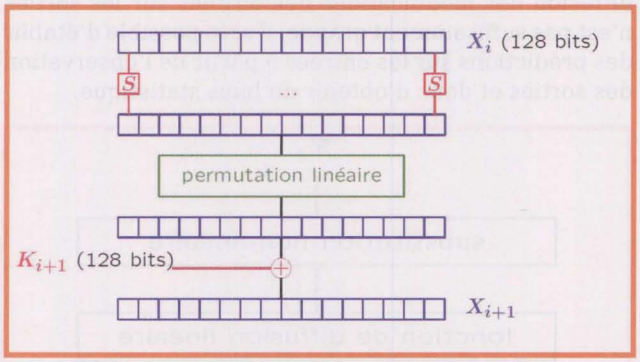
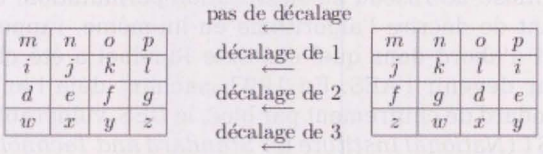


Figure 3 : La fonction de tour de l'AES

La boîte *S* utilisée est composée elle-même de deux opérations : l'inversion dans le corps à 2^8 éléments, suivie d'une transformation linéaire sur les mots de 8 bits. Cette boîte *S* n'a pas été choisie au hasard. En effet, elle offre une bonne résistance aux cryptanalyses linéaires et différentielles et suit le principe de confusion.

Quant à la partie linéaire, elle est composée des opérations ShiftRows et Mixcolumns qui agissent sur une matrice d'octets de taille 4 × 4. L'opération ShiftRows consiste à faire subir une permutation circulaire vers la gauche aux lignes de la matrice à chiffrer :



MixColumns est une fonction de diffusion optimale sur les mots de quatre octets qui porte sur chaque colonne de la matrice représentant le bloc d'entrée.

Le cadencement de clé qui permet de générer les sous-clés de chaque étage à partir de la clé maître *K* utilise les opérations déjà définies dans le chiffrement. Plus précisément, la clé maître *K* est représentée par *Nk* mots de 32 bits, *Nk* valant 4, 6 ou 8 en fonction de la longueur de la clé. Il s'agit donc d'étendre cette clé en *Nb*+1 clés K_i de 128 bits où *Nb* désigne le nombre de tours. Le schéma général du cadencement de clé est présenté à la figure 4 où *F* est la fonction qui décale de 1 octet vers la gauche les octets du mot de 32 bits et qui ensuite applique à ce mot de 32 bits 4 fois la boîte *S* de l'AES en parallèle et finalement ajoute une constante. Une fois ce tableau construit, les sous-clés sont prises dans l'ordre de leur utilisation : les 4 premiers mots de 32 bits vont donner la sous-clé K_0, les 4 suivants, la sous-clé K_1, et ainsi de suite.

Nous venons de voir les détails de l'algorithme Rijndael devenu AES le 2 octobre 2000. Cet algorithme est très rapide, facile à implémenter, les tables nécessaires aux calculs ne prennent pas beaucoup de place en mémoire et il est très facile à inverser. C'est sans doute une partie

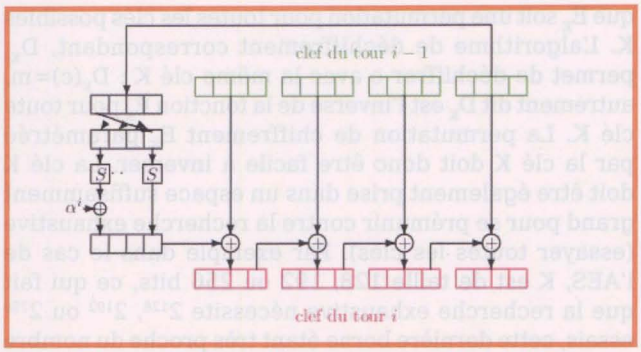


Figure 4 : Le cadencement de clé de l'AES

des raisons qui ont orienté le choix du NIST. Il est à noter également que la meilleure attaque contre l'AES à la fin de la compétition portait sur 7 étages avec une complexité supérieure à 2^{128} chiffrements. L'AES semblait donc suffisamment solide.

3 2001-2004 : doutes et certitudes

Entre 2001 et 2004 est apparue une nouvelle forme de cryptanalyse : la cryptanalyse algébrique. Il s'agissait de mettre en pratique l'attaque déjà proposée par Shannon dans son article fondateur de 1949 **[22]**. Cette attaque consiste à mettre en équation les relations liant un ensemble de messages clairs, l'ensemble correspondant de messages chiffrés en fonction des bits des sous-clés intervenant à chaque étage. Plus précisément et comme écrit dans **[22]** : étant donné un message clair de *n* bits, $M = m_1, m_2, \cdots, m_n$, une clé de l bits, $K=K_1, \ldots, K_l$ et le message chiffré correspondant $C = c_1, c_2, \cdots, c_n$, le cryptanalyste peut écrire les équations de chiffrement :

$$\begin{aligned} c_1 &= f_1(m_1,m_2,\cdots,m_n,K_1,\cdots,K_\ell) \\ c_2 &= f_2(m_1,m_2,\cdots,m_n,K_1,\cdots,K_\ell) \\ &\vdots \\ c_n &= f_s(m_1,m_2,\cdots,m_n,K_1,\cdots,K_\ell) \end{aligned}$$

Dans ce système, tout est connu sauf les bits de clé K_i. Chacune de ces équations doit donc être complexe en les K_i et impliquer beaucoup d'entre eux. En particulier, elle doit avoir un degré élevé. Sinon, l'ennemi peut résoudre les équations les plus simples et ensuite les plus complexes par substitution.

3.1 L'attaque de Shannon sur l'AES

Si on applique cette idée au cas de l'AES, on génère les équations binaires suivantes :

- Pour le premier tour dont la sortie est le mot de 128 bits, x_1, l'équation impliquant les bits de $x_1=(x_1^1, \cdots, x_1^{128})$, les bits du message $m=(m^1, \cdots, m^{128})$ et les bits de la sous-clé $K_0=(K_0^1, \cdots, K_0^{128})$ s'écrit :

$$(x_1^1, \cdots, x_1^{128}) = F(m^1 + K_0^1, \cdots, m^{128} + K_0^{128}) \text{ avec } \deg F = 7.$$

- De même, si on écrit les relations liant les bits de la sous-clé $K_1=(K_1^1, \cdots, K_1^{128})$ en fonction des bits de la clé maître, on obtient (la sous-clé K_0 étant en fait les 128 premiers bits de la clé maître K) pour une clé maître de 128 bits :

$$\begin{aligned}(K_1^1, \cdots, K_1^{32}) &= G(K_0^1, \cdots, K_0^{128}) \text{ avec } \deg G = 7 \text{ et} \\ (K_1^{33}, \cdots, K_1^{128}) &= \ell(K_0^1, \cdots, K_0^{128}, K_1^1, \cdots, K_1^{32}) \text{ avec } \deg \ell = 1.\end{aligned}$$

- Pour le deuxième tour où la sortie est x_2, on obtient :

$$(x_2^1, \cdots, x_2^{128}) = F(x_1^1 + K_1^1, \cdots, x_1^{128} + K_1^{128}) \text{ avec } \deg F = 7.$$

En poursuivant ainsi le raisonnement, on obtient un système d'équations dépendant des 128 variables de la clé maître (pour une clé de 128 bits), de 10 × 32 variables intermédiaires pour les sous-clés et de 9 × 128 variables intermédiaires pour l'état, soit un système de 1600 équations de degré 7 en 1600 variables.

3.2 Amélioration de Courtois et Pieprzyk [8]

On peut faire diminuer le degré algébrique du système précédent en utilisant l'idée développée par N. Courtois et J. Pieprzyk. Dans **[8]**, les auteurs remarquent qu'il existe des relations de degré 2 entre les entrées et les sorties de la boîte S de l'AES. En effet, on a :

$$S(x) = x^{-1} \text{ pour tout } x \neq 0, \text{ soit } xS(x) = 1, \text{ pour tout } x \neq 0$$

Ce que l'on peut réécrire en $x^2 S(x) = x$ pour tout x.

Plus précisément, on peut obtenir ainsi 39 relations booléennes de degré 2 entre chaque octet en entrée et sortie d'un tour de chiffrement (ou de cadencement de clé). Dans le système précédent, on remplace 8 relations de degré 7 (les expressions dépendant de S) par 39 relations de degré 2. Ce qui revient à construire un système de 8000 équations de degré 2 en 1600 variables.

3.3 Comment résoudre ces systèmes ?

La question qui se pose alors est de trouver une méthode efficace pour résoudre un tel système polynomial. Il existe différentes techniques pour ce faire :

- La linéarisation qui consiste à remplacer toute variable de degré supérieur à 1 par une nouvelle variable puis à appliquer un pivot de Gauss sur le système linéaire qui en résulte.
- La résolution utilisant les bases de Gröbner dont le principe est d'utiliser une division euclidienne généralisée dans les idéaux de polynômes. On dispose d'un certain nombre d'algorithmes pour calculer les bases de Gröbner. Parmi les implémentations les plus efficaces de ces techniques développées à ce jour, on peut citer F4 et F5 développés par J.-C. Faugère **[11]** **[12]**.
- Les *SAT-solvers* sont une classe d'algorithmes qui permettent de résoudre des instances particulières SAT. Rappelons que le problème de satisfaisabilité (SAT) est un problème de décision qui consiste à savoir s'il existe une solution à une série d'équations booléennes données.
- Évidemment, il existe des techniques ad-hoc qui permettent d'utiliser des méthodes particulières fonction du système lui-même (s'il est creux, sur-défini, sous-défini, etc.).

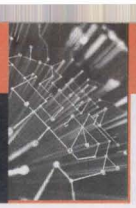
Mais il reste impossible de résoudre le système généré par un AES complet (comme précisé dans l'AES *security report* produit par le réseau d'excellence européen ECRYPT **[19]**). Pour mesurer à quel point ces attaques sont loin d'être effectives, on peut tenter de proposer un système réduit approprié que l'on peut casser, puis en déduire la complexité de résolution pour un AES complet. Parmi les principales tentatives, on peut citer le travail de C. Cid, S. Murphy et M. Robshaw qui, dans **[7]**, ont créé une version réduite de l'AES sur F_{2^4} et ont pu attaquer jusqu'à 4 tours de cette version réduite montrant ainsi que l'AES complet était hors de portée d'une telle attaque. Mais il reste de nombreux problèmes ouverts sur le sujet, en particulier, il reste à déterminer l'influence du caractère creux du système sur sa résolution.

3.4 Prolongement des attaques algébriques

L'émergence des attaques algébriques a bien sûr eu des conséquences sur le monde de la cryptographie symétrique. D'une part, elles ont conduit à la définition des critères de résistance aux attaques algébriques afin de pouvoir choisir les meilleures composantes non linéaires pour les systèmes de chiffrement (dans le cas des algorithmes de chiffrement par bloc, il s'agit surtout des boîtes S). D'autre part, si les attaques algébriques semblent ne pas mettre en danger les chiffrements par bloc, elles se sont montrées efficaces contre de nombreux chiffrements à flot.

3.4.1 Critères de résistance aux attaques algébriques

Afin de résister aux attaques algébriques, l'immunité algébrique d'une fonction S opérant sur les mots de n bits est définie comme étant le degré minimal d d'une relation booléenne non nulle f telle que $f(x,S(x)) = 0$ pour tout x. On peut montrer qu'il existe une relation de degré inférieur ou égal à d dès que :



$$\sum_{i=0}^{d} \binom{2n}{i} > 2^n .$$

Cela implique que des relations de degré 2 existent si n ≤ 6 et que toute fonction avec 8 variables en entrée et en sortie a des relations de degré 3. Cependant, à l'heure actuelle, il n'existe pas de critère précis lié au nombre de relations de petit degré.

3.4.2 Attaques algébriques contre les chiffrements à flot

Même si les attaques algébriques semblent avoir une complexité trop élevée pour mettre en danger les chiffrements par bloc, elles se sont avérées un outil puissant quand il s'est agi d'attaquer les chiffrements à flot. Rappelons rapidement qu'un chiffrement à flot est un générateur pseudo-aléatoire qui produit une suite pseudo-aléatoire générée à partir d'une clé secrète partagée par les entités souhaitant communiquer. Jusqu'en 2003, la plupart des chiffrements à flot (comme E0, le chiffrement utilisé par Bluetooth) étaient composés d'un état interne dont le contenu était mis à jour via des relations linéaires et ensuite, le contenu de cet état était filtré par une fonction booléenne f choisie pour ses bonnes propriétés cryptographiques. C'est la sortie de cette fonction f qui constituait la suite pseudo-aléatoire binaire produite par le chiffrement.

Le principe général des attaques algébriques est d'exploiter la linéarité de la fonction de mise à jour du registre et ensuite d'utiliser des relations de bas degré induites par la fonction de filtrage f. Ainsi, il était possible de construire des relations de la forme : si $f(x_1, \cdots, x_n)=1$, alors $g(x_1, \cdots, x_n) = 1$ avec deg g < deg f. Cette technique a permis d'attaquer avec succès des algorithmes comme Toyocrypt, E0, Sober, LILI-128 ou Sfinks. Ainsi, en 2003, à la fin du projet NESSIE, processus de choix de primitives cryptographiques lancé par l'Europe, il ne restait aucun candidat (indemne d'attaques) dans la catégorie chiffrements à flot et générateurs de nombres pseudo-aléatoires : le portfolio NESSIE pour cette catégorie était vide !

C'est pour cette raison qu'en 2004, le réseau d'excellence européen ECRYPT a lancé le projet eSTREAM **[10]** d'appel à chiffrements à flot. En 2008, à la fin du processus de soumission et d'évaluation, ce sont finalement 8 (puis 7) algorithmes de chiffrement à flot qui ont été recommandés sur les 34 soumissions originelles. Tous ces finalistes utilisent une fonction non linéaire de mise à jour de l'état interne.

4 2005-2008 : l'âge d'or

La certitude que l'AES résistait aux attaques algébriques l'a alors transformé en couteau suisse de la cryptographie. Cela a été rendu possible grâce à la démocratisation de l'AES, notamment à travers le nouvel ensemble de 7 instructions Intel®, appelé Intel® AES-NI, implémenté en dur dans les processeurs Intel® Xeon et dans la deuxième génération des processeurs Intel® Core™. Cet ensemble d'instructions a rendu l'AES encore plus rapide qu'il n'était et l'a introduit dans énormément de machines grand public.

L'actualité de l'AES a été cependant moins prégnante durant la période 2005-2008. En effet, c'étaient les fonctions de hachage qui étaient sous les feux de l'actualité. Tout d'abord parce que tous les standards choisis durant les années 90 étaient en train d'être attaqués avec beaucoup de succès. Déjà, MD4, standardisé en 1992 **[20]**, avait été attaqué avec succès par H. Dobbertin en 1996 **[9]**. En ce qui concerne MD5 **[21]**, il a fallu attendre 2004 et l'attaque de X. Wang **[26]** qui, après diverses améliorations, permet de trouver des collisions sur MD5 en quelques millisecondes. Dans le même temps, la famille SHA **[16] [17] [18]** ne sortait pas indemne de ces attaques **[27] [25]** : il est possible de trouver des collisions sur SHA-0 en 1 heure et sur SHA-1 en un temps calcul certes encore hors de portée, mais bien inférieur à ce que l'on attend pour une fonction bien conçue. De même, les autres fonctions de hachage existantes comme HAVAL **[23]** ou RIPEMD **[24]** ont été attaquées avec succès.

Ne survivait donc en 2005 comme standard pour les fonctions de hachage que SHA-2. C'est ce qui a poussé le NIST à lancer en 2007 une nouvelle compétition internationale pour choisir et standardiser une nouvelle fonction de hachage, SHA-3 (http://csrc.nist.gov/groups/ST/hash/sha-3/), qui devra être utilisée dans les années à venir. La date limite de soumission était octobre 2008. Le NIST a reçu 64 algorithmes des 4 coins du monde. Après une première phase de sélection en décembre 2008, 51 candidats ont été retenus pour le premier tour. Après le deuxième tour, 14 candidats ont été retenus, puis en décembre 2010, ce sont finalement 5 finalistes qui ont été choisis. La sélection du vainqueur devrait avoir lieu fin 2012.

Parmi les 14 finalistes du deuxième tour, 4 utilisaient des fonctions de l'AES ou s'inspiraient de sa structure et parmi les 5 finalistes restants, Grøstl **[14]** utilise une structure similaire à celle de l'AES.

Ainsi, l'âge d'or de l'AES a consisté en sa popularisation et en la diversification de l'utilisation des fonctions le composant.

5 2009-2011 : premières alertes ?

Depuis 2008, au travers de la compétition SHA-3, le monde de la cryptographie symétrique s'est concentré sur la conception et les attaques de fonctions de hachage. Cela a permis des avancées notables en cryptanalyse, notamment sur la définition même de ce qu'est une attaque. Une partie de ces résultats a pu être appliquée à l'AES.

5.1 Attaque par rebond

Cette attaque a été introduite dans **[15]** et peut être vue comme une amélioration de la cryptanalyse différentielle.

5.1.1 Cryptanalyse différentielle

La cryptanalyse différentielle introduite par E. Biham et A. Shamir en 1991 **[2]** consiste à étudier l'évolution au cours du chiffrement d'une différence introduite dans un couple de messages m et m' : $m' = m + a$. Le chemin différentiel à travers le chiffrement est probabiliste et dépend essentiellement du passage de la différence à travers les boîtes S. Pour calculer cette probabilité, on s'attache donc tout d'abord à calculer la probabilité de passage de la différence à travers chacune des boîtes S. On calcule donc la probabilité, évaluée sur toutes entrées X que la différence entre $S(X+a)$ et $S(X)$ soit égale à b, pour un couple donné (a,b) de différences en entrée et sortie.

Si on cherche à appliquer cette technique à 4 tours de l'AES, l'un des meilleurs chemins différentiels que l'on peut construire est celui représenté à la figure 5. Il s'agit de trouver un couple d'entrées qui diffèrent sur un octet et dont les sorties après 4 tours diffèrent également sur un octet.

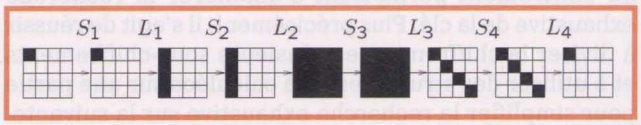


Figure 5 : Chemin différentiel sur 4 étages de l'AES où S représente l'opération SubBytes et L les opérations ShiftRows et MixColumns. Les octets en noir représentent les octets avec une différence. Les autres ne contiennent pas de différence.

Pour la plupart des couples d'octets d'entrées/sorties (a,b) de différence de la boîte S de l'AES, on a : $Pr_X[S(X + a) + S(X) = b] \leq 2^{-7}$. Si l'on cherche à construire le chemin différentiel complet de la figure 5, il faut donc passer toutes les boîtes S de l'étage S_3, soit 16 boîtes S à traverser en parallèle. On obtient donc une probabilité de passage pour un couple de différences donné (a,b) à travers S_3 qui est : $Pr_X[S_3(X + a) + S_3(X) = b] \leq (2^{-7})^{16}$. Ce qui nous donne une probabilité du chemin différentiel de la figure 5 pour quatre tours de l'AES qui vérifie : $Pr_{X,K}[AES^4(X + a) + AES^4(X) = b] \leq (2^{-7})^{16}$, soit une probabilité de réalisation du chemin différentiel extrêmement faible.

5.1.2 Attaque par rebond

L'attaque par rebond est une amélioration de l'attaque différentielle classique : il ne s'agit plus de regarder un couple de différences d'entrée/sortie précis mais de s'intéresser à l'ensemble des couples de différences et de valeurs qui vont vérifier le passage à travers S_3 (voir Figure 6).

On choisit tout d'abord un couple de différences (δ_1, δ_2) sur 4 octets comportant des différences, comme indiqué à la figure 6. On traverse la partie linéaire pour, à partir de δ_1, obtenir une différence a sur 16 octets en entrée de S_3. De même, à partir de δ_2, on remonte jusqu'à la sortie de S_3 pour obtenir une différence b sur 16 octets en sortie de S_3. La probabilité qu'on puisse connecter les différences a et b en entrée/sortie de S_3 pour les 16 octets est 2^{-16} car pour une différence donnée en entrée/sortie (α, β) d'une boîte S, la probabilité que cette différence soit valide est 1/2. En d'autres termes : $Pr[\exists X / S(X+\alpha) + S(X) = \beta] = 1/2$. Mais, à chaque fois qu'on trouve une paire (α, β) valide, cette différentielle est vérifiée par deux valeurs. Donc le nombre de valeurs qui vérifie la différence (a, b) est 2^{16} pour les 16 octets.

Il faut donc en moyenne essayer 2^{16} différences (δ_1, δ_2) pour obtenir un couple de différences valide (a, b) qui lui-même va donner 2^{16} valeurs qui vont vérifier cette différence. Le coût amorti pour obtenir un couple de valeurs vérifiant la différence (δ_1, δ_2) est donc de 1.

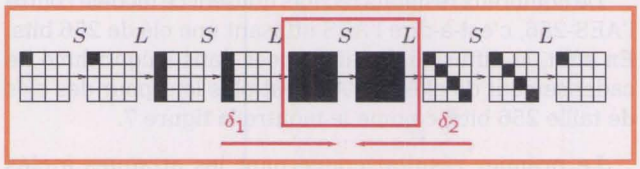


Figure 6 : Principe de l'attaque par rebond

Afin de finir l'attaque, on remonte les tours restants au début et à la fin du chemin pour les valeurs trouvées et on teste si les valeurs obtenues sont correctes sur le reste du chemin.

Cette nouvelle technique a permis de montrer que 8 tours de l'AES ne se comportent pas comme une permutation aléatoire parfaite, autrement dit, pour 8 tours de l'AES, il existe des structures sous-jacentes qui permettent de le différencier d'une vraie permutation aléatoire. Cette attaque a également permis d'une part de cryptanalyser avec un succès raisonnable un certain nombre de soumissions SHA-3 comme LANE ou la version 2 de Grøstl, et d'autre part, de montrer le comportement imparfait de composants des fonctions de hachage Grøstl, ECHO ou Whirlpool. Cette technique d'attaque peut également être généralisée à d'autres types de structures comme celles des candidats SHA-3 JH ou Skein.

5.2 Attaques à clés liées

Le principe des attaques à clés liées a été introduit en 1993 par E. Biham **[1]**, il est le suivant : l'attaquant dispose de couples clairs-chiffrés obtenus avec différentes clés liées par une relation de son choix. Cela peut paraître étonnant comme forme d'attaque et semble ne pas remettre en cause la sécurité d'un chiffrement par bloc lorsqu'on l'utilise de façon classique. Cependant, ces attaques deviennent pertinentes quand le chiffrement par bloc considéré est employé dans une fonction de hachage, par exemple avec la construction de Davies-Meyer pour laquelle le message joue le rôle de la clé. L'attaquant qui, dans le cas d'un chiffrement par bloc, n'avait aucun contrôle sur la clé, peut ici agir à sa guise sur le message.

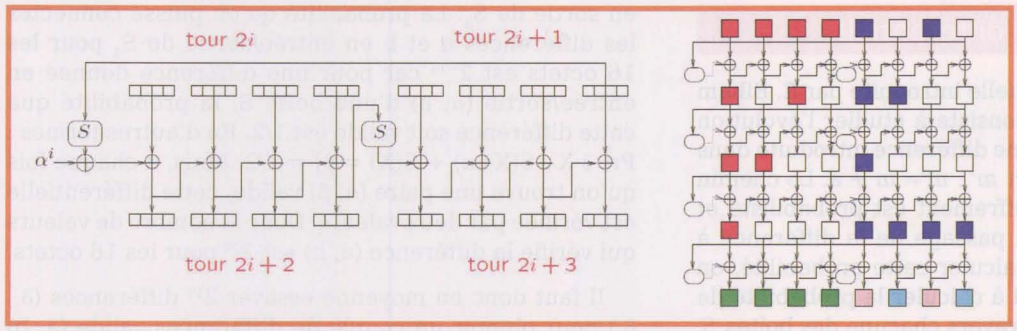


Figure 7 : À gauche, le détail du cadencement de clé pour l'AES-256. À droite, un chemin différentiel sur 4 tours du cadencement de clé de l'AES-256 qui a une probabilité 1 sur les trois premiers tours.

De nombreux résultats récents utilisent ce modèle contre l'AES-256, c'est-à-dire l'AES utilisant une clé de 256 bits. En effet, la diffusion de différences dans l'algorithme de cadencement de clés de l'AES est plus lent pour des clés de taille 256 bits, comme le montre la figure 7.

Le premier résultat concernant les attaques à clés liées sur les 14 tours de l'AES-256 est dû à A. Birykov, D. Khovratovich et I. Nikolic dans **[5]**. Ils utilisent une différence sur les clés comme celle présentée à la figure 7 pour annuler les différences sur les états internes du chiffrement. Un exemple d'une telle annulation est présenté à la figure 8 pour les deux premiers tours de l'AES-256.

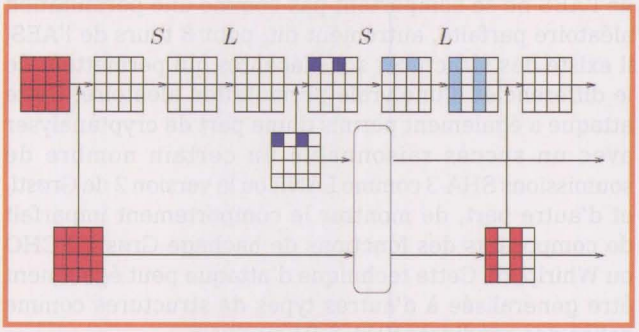


Figure 8 : Exemple d'annulation des différences sur deux tours de l'AES-256

En continuant ainsi de façon probabiliste, le processus d'annulation des différences entre les sous-clés et les états internes du chiffrement, A. Birykov, D. Khovratovich et I. Nikolic arrivent à construire une attaque à clés liées sur l'AES-256 en entier nécessitant 2^{131} calculs, une place en mémoire de 2^{65} en utilisant 2^{35} clés différentes et permettant de retrouver la clé. La complexité de cette attaque est bien en-dessous de la recherche exhaustive de la clé qui nécessite 2^{256} chiffrements.

Dans **[4]**, A. Birykov et D. Khovratovich ont amélioré ce résultat et leur nouvelle attaque à clés liées nécessite $2^{99.5}$ calculs, une place en mémoire de 2^{77} en utilisant 4 clés différentes et permet de retrouver la clé. Ils proposent également la première attaque contre une version complète de l'AES-192 avec une complexité de 2^{176} calculs, une place en mémoire de 2^{152} en utilisant 4 clés différentes et 2^{123} couples clairs-chiffrés. Finalement, dans **[3]**, une attaque sur 9 des 14 tours de l'AES-256 permet de retrouver la clé avec une complexité de 2^{39} calculs, une place en mémoire de 2^{32} en utilisant seulement 2 clés différentes. L'intérêt d'une telle attaque est qu'elle peut être réalisée en pratique.

5.3 Attaque par bicliques

Depuis 2009, est également apparue une nouvelle forme d'attaque appelée attaque par bicliques. Le but de cette attaque est de trouver des structures internes au chiffrement permettant d'améliorer la recherche exhaustive de la clé. Plus précisément, il s'agit de réussir à diviser le chiffrement en plusieurs sous-chiffrements et à utiliser des structures déjà calculées sur une partie pour simplifier la recherche exhaustive sur la suivante.

Dans le cas de l'AES et des résultats proposés dans **[6]**, les structures utilisées pour améliorer la recherche exhaustive sont fondées sur des chemins différentiels à clés liées. Cela permet de construire les attaques résumées dans la table 1.

	temps	mémoire	données
AES-128	$2^{126.2}$	2^{8}	2^{88}
AES-192	$2^{189.8}$	2^{8}	2^{80}
AES-256	$2^{254.4}$	2^{8}	2^{40}

Table 1 : Attaques par bicliques

Il s'agit donc bien des premières attaques moins chères que la recherche exhaustive sur les trois versions complètes de l'AES. Elles ne mettent cependant pas en danger l'algorithme car leurs complexités sont extrêmement élevées. Elles révèlent cependant des faiblesses potentielles qui seront éventuellement mieux exploitées dans les années à venir.

Conclusion

Même si l'attention du monde cryptographique se concentre aujourd'hui sur la compétition SHA-3 et ses candidats, attaquer l'AES reste un défi permanent pour tout cryptanalyste et beaucoup d'efforts ont été faits pour tenter de l'attaquer. On peut cependant dire qu'actuellement, l'AES ne comporte pas de faiblesse notable et peut toujours être utilisé comme chiffrement par bloc même s'il semble que l'AES-256 n'offre pas une sécurité de 256 bits en raison des attaques à clés liées existant contre lui.

Comme nous venons de le voir, il est très risqué d'utiliser une primitive cryptographique pour une fonctionnalité pour laquelle elle n'a pas été conçue : il semble aujourd'hui dangereux d'utiliser l'AES comme fonction de hachage en mode Davis-Meyer par exemple. Il est également important de noter qu'à cause des attaques à clés liées, la clé maître doit être un secret tiré uniformément au hasard et que, par exemple, utiliser une clé, puis la modifier suivant une procédure déterministe au lieu d'en tirer une nouvelle, fait partie des méthodes à proscrire.

Les différentes expériences de standardisation de primitives de cryptographie symétrique ont montré que la durée de vie moyenne d'un algorithme est d'une quinzaine d'années, ce qui peut sembler peu mais ne semble pas si étonnant quand on sait qu'un standard concentre les tentatives d'attaques d'une grande partie de la communauté cryptographique. Est-ce alors qu'il faut déjà s'interroger sur la future durée de vie du finaliste de la compétition SHA-3 ? ■

RÉFÉRENCES

[1] Eli Biham, New types of cryptoanalytic attacks using related keys (extended abstract), in Advances in Cryptology - EUROCRYPT '93, volume 765 of Lecture Notes in Computer Science, pages 398–409. Springer, 1993

[2] Eli Biham and Adi Shamir, Differential Cryptanalysis of DES-like Cryptosystems, J. Cryptology, 4(1):3–72, 1991

[3] Alex Biryukov, Orr Dunkelman, Nathan Keller, Dmitry Khovratovich, and Adi Shamir, Key recovery attacks of practical complexity on AES-256 variants with up to 10 rounds, in Advances in Cryptology - EUROCRYPT 2010, volume 6110 of Lecture Notes in Computer Science, pages 299–319. Springer, 2010

[4] Alex Biryukov and Dmitry Khovratovich, Related-key cryptanalysis of the full AES-192 and AES-256, in Advances in Cryptology - ASIACRYPT 2009, volume 5912 of Lecture Notes in Computer Science, pages 1–18. Springer, 2009

[5] Alex Biryukov, Dmitry Khovratovich, and Ivica Nikolic, Distinguisher and related-key attack on the full AES-256, in Advances in Cryptology - CRYPTO 2009, volume 5677 of Lecture Notes in Computer Science, pages 231–249. Springer, 2009

[6] Andrey Bogdanov, Dmitry Khovratovich, and Christian Rechberger, Biclique cryptanalysis of the full AES. In Advances in Cryptology - ASIACRYPT 2011, volume 7073 of Lecture Notes in Computer Science, pages 344–371, Springer, 2011

[7] Carlos Cid, Sean Murphy, and Matthew J. B. Robshaw, Small scale variants of the AES, in Fast Software Encryption: 12th International Workshop, FSE 2005, volume 3557 of Lecture Notes in Computer Science, pages 145–162. Springer, 2005

[8] Nicolas Courtois and Josef Pieprzyk, Cryptanalysis of block ciphers with overdefined systems of equations, in Advances in Cryptology - ASIACRYPT 2002, volume 2501 of Lecture Notes in Computer Science, pages 267–287. Springer, 2002

[9] Hans Dobbertin, Cryptanalysis of MD4, in Fast Software Encryption, Third International Workshop - FSE 1996, volume 1039 of Lecture Notes in Computer Science, pages 53–69, Springer, 1996

[10] ECRYPT Stream Cipher Project eSTREAM, The current estream portfolio, eSTREAM, ECRYPT Stream Cipher Project, 2008, http://www.ecrypt.eu.org/stream

[11] J. Faugère, A new efficient algorithm for computing Gröbner basis (F4), Journal of Pure and Applied Algebra, 139(1-3):61–88, 6 1999

[12] Jean Charles Faugère, A new efficient algorithm for computing Gröbner bases without reduction to zero (F5), in Proceedings of the 2002 international symposium on Symbolic and algebraic computation, ISSAC '02, pages 75–83, ACM, 2002.

[13] FIPS 197, Advanced Encryption Standard, Federal Information Processing Standards Publication 197, 2001, U.S. Department of Commerce/N.I.S.T.

[14] Praveen Gauravaram, Lars R. Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen, Grøstl – a SHA-3 candidate, Submission to NIST (Round 3), 2011

[15] Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen, The Rebound Attack: Cryptanalysis of Reduced Whirlpool and Grøstl, in Fast Software Encryption - FSE 2009, volume 5665 of Lecture Notes in Computer Science, pages 260–276, Springer, 2009

[16] U.S. Department of Commerce, FIPS 180: Secure Hash Standard, Federal Information Processing Standards Publication, N.I.S.T., 1993

[17] U.S. Department of Commerce. FIPS 180-1: Secure Hash Standard (SHS), Federal Information Processing Standards Publication, N.I.S.T., 1995

[18] U.S. Department of Commerce, FIPS 180-2: Secure Hash Standard. Federal Information Processing Standards Publication, N.I.S.T., 2002

[19] European Network of Excellence ECRYPT, AES security report (ecrypt 2006), 2006, http://www.ecrypt.eu.org/ecrypt1/documents/D.STVL.2-1.0.pdf

[20] Ronald L. Rivest, The MD4 message-digest algorithm, Internet RFC 1320, 1992

[21] Ronald L. Rivest, The MD5 message-digest algorithm, Internet RFC 1321, 1992

[22] C. E. Shannon, Communication theory of secrecy systems, Bell System Technical Journal, 28:656–715, 1949

[23] Xiaoyun Wang, Dengguo Feng, and Xiuyuan Yu, An attack on hash function HAVAL-128, Science in China Series F: Information Sciences, 48(5):545–556, 2005

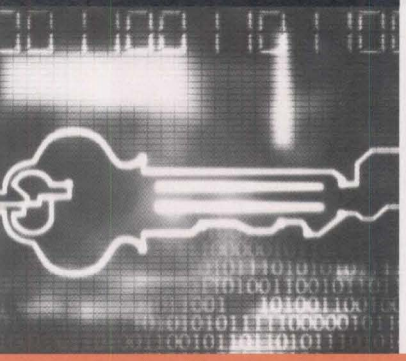
[24] Xiaoyun Wang, Xuejia Lai, Dengguo Feng, Hui Chen, and Xiuyuan Yu, Cryptanalysis of the hash functions MD4 and RIPEMD, in Advances in Cryptology - EUROCRYPT 2005, volume 3494 of Lecture Notes in Computer Science, pages 1–18, Springer, 2005

[25] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu, Finding collisions in the full SHA-1, in Advances in Cryptology - CRYPTO 2005, volume 3621 of Lecture Notes in Computer Science, pages 17–36, Springer, 2005

[26] Xiaoyun Wang and Hongbo Yu, How to break MD5 and other hash functions, in Advances in Cryptology - EUROCRYPT 2005, volume 3494 of Lecture Notes in Computer Science, pages 19–35, Springer, 2005

[27] Xiaoyun Wang, Hongbo Yu, and Yiqun Lisa Yin, Efficient collision search attacks on SHA-0, in Advances in Cryptology - CRYPTO 2005, volume 3621 of Lecture Notes in Computer Science, pages 1–16, Springer, 2005

LES AVANCÉES DE LA CRYPTO MODERNE



FACTORISATION D'ENTIERS : LA VOIE ROYALE DU CASSAGE DE RSA

F. Morain - morain@lix.polytechnique.fr - INRIA & LIX École polytechnique

mots-clés : *RSA / FACTORISATION D'ENTIERS / CRIBLE QUADRATIQUE / CRIBLE ALGÉBRIQUE / CALCULS MASSIFS*

La façon la plus directe de casser le cryptosystème RSA est de factoriser la clé publique. Nous donnons ici quelques éléments scientifiques et historiques pour comprendre le fonctionnement des algorithmes de factorisation.

1 Introduction : émergence d'une communauté

De tous temps, il s'est trouvé des mathématiciens pour aimer faire des calculs que d'aucuns trouvaient anecdotiques, voire inutiles. Ces passionnés passaient de longs jours, à la main, à prouver la primalité de grands entiers (souvent de la forme 2^p-1), calculer des zéros de la fonction de Riemann, factoriser des entiers, faire des tables de polynômes. Cela les a souvent conduits à inventer des méthodes astucieuses qui pour certaines n'ont pu être mises en œuvre avec succès qu'à partir du moment où des ordinateurs ont été disponibles.

La factorisation des entiers a donc sa préhistoire (cf. **[WS94]**), et son histoire récente est liée à l'informatique. La théorie de la complexité a été formalisée dans le but de classifier les problèmes à résoudre en fonction de leur facilité ou leur difficulté, leur vitesse. La cryptographie asymétrique (à clé publique) est née dans ce mouvement vers le milieu des années 1970. Le plus vieil exemple de système de chiffrement moderne est celui de RSA (rappelé en appendice), dont la sécurité a des liens avec la factorisation des entiers, qui de fait est devenue à la mode.

Il faut tout de suite expliquer pourquoi la factorisation est un problème aux frontières de l'informatique et des mathématiques. Pour un mathématicien, il n'y a pas de problème : un entier a des facteurs premiers, on peut les trouver par divisions successives, voire en n'essayant que les facteurs plus petits que la racine carrée du nombre qui nous intéresse. L'informaticien va quant à lui ajouter la notion de temps : dans quel temps peut-on factoriser N ? Peut-on le faire en temps polynomial (le rêve) ? Quel est le meilleur algorithme pour ce faire ? Et les allers-retours entre les deux domaines peuvent démarrer.

La communauté des théoriciens des nombres voulant utiliser des ordinateurs comme aide à leurs calculs est née au début des années 1970, un peu avant l'apparition de RSA. Les ordinateurs d'alors étaient rudimentaires, mais il existe encore des nostalgiques des premières calculettes programmables qui ont permis de faire des calculs non triviaux. De concert avec la cryptologie moderne, les machines se sont démocratisées et tout le monde a pu participer à l'aventure : n'importe quel système de calcul mathématique possède quelques algorithmes de factorisation, et qui peut prétendre n'avoir jamais joué avec ?

Signalons en passant que la factorisation d'entiers est un point bloquant dans beaucoup de recherches, par exemple en théorie algébrique des nombres.

Avec de fortes motivations, les algorithmes de multiprécision, déjà exposés par Knuth en 1973, sont implantés massivement et toujours le sujet de travaux très actifs. Par exemple, on sait qu'on peut multiplier deux entiers de n bits en temps quasi-linéaire en n, et c'est devenu une réalité pratique, ce qui permet d'optimiser de nombreux autres algorithmes (division, racine carrée, etc.). La référence incontournable est la bibliothèque GMP de T. Granlund. Elle est intégrée dans de nombreux systèmes de calcul, et elle est au cœur d'une communauté très active.

Des bibles existent sur le domaine : Knuth **[Knu97]** a été rejoint par le livre de Cohen **[Coh00]**, Crandall & Pomerance **[CP05]**. La grande revue scientifique est *Mathematics of Computation* depuis les origines, un congrès s'est mis en place (*Algorithmic Number Theory Symposium* – ANTS), dont la précédente édition a eu lieu à Nancy et la prochaine à San Diego.

Quel rôle pour cette communauté académique publique ? Nous sommes là pour étudier les objets de base de la cryptologie et en tester la sécurité intrinsèque. Dans le cas

de RSA, nous justifions les équations, puis nous donnons les outils pour fabriquer des clés, et nous établissons l'état de l'art dans le cassage de ces mêmes clés. Notre rôle dans la chaîne s'arrête là, laissant aux fabricants de cartes à puce la tâche de veiller à d'autres aspects de la sécurité, ou bien aux concepteurs de protocoles qui en prouvent les propriétés de sûreté, de sécurité, etc. La cryptologie asymétrique ne se réduit pas à cet apport des mathématiciens/informaticiens.

2 Les grands principes des algorithmes de factorisation

Il n'existe aujourd'hui que deux grandes classes d'algorithmes pour factoriser les entiers : les algorithmes par réduction et les algorithmes de combinaisons de congruences. Dans la première catégorie, on exploite des propriétés des facteurs premiers p de N : des objets modulo N se réduisent à 0 modulo p, ce qui permet de trouver p. On trouve dans cette catégorie les méthodes de type p−1 à la Pollard (ce qui inclut les courbes elliptiques).

Expliquons le principe des algorithmes à combinaisons de congruences, les seuls actuellement à pouvoir casser des clés RSA. Pour factoriser un entier N, il suffit de résoudre l'équation $x^2 \equiv 1 \bmod N$ et en particulier de trouver les racines différentes de ± 1. Par exemple pour N = 13290059, on trouve (par énumération) deux racines carrées non triviales : 2164587 et 11125472. Pour factoriser N, il suffit alors de calculer pgcd(2164587−1, N) = 3119, ou encore pgcd(11125472−1, N) = 4261.

Toute la difficulté est de trouver ces racines carrées. Suivant une idée remontant à Kraitchik (en 1921 !), on cherche des factorisations auxiliaires

$$x_i^2 \bmod N = y_i = \prod_{j=1}^{k} p_j^{\alpha_j}$$

sur une base de nombres premiers $\{p_1, p_2, ..., p_k\}$ choisie à l'avance. Avec notre exemple, choisissons k = 7 avec $p_1 = 2$, $p_2 = 3$, ..., $p_7 = 17$. Dans la méthode la plus simple (due à Dixon), on cherche ces factorisations auxiliaires pour des x_i dans un intervalle. On trouvera dans la table 1 quelques-unes de ces équations pour $x \geqslant \sqrt{N}$.

x	$y = x^2 \bmod 13290059$	factorisation de y	ligne
6321	84864	$2^7 \cdot 3 \cdot 13 \cdot 17$	L_1
6711	5167344	$2^4 \cdot 3 \cdot 7^2 \cdot 13^3$	L_2
12091	1632	$2^5 \cdot 3 \cdot 17$	L_3
12093	50000	$2^4 \cdot 5^5$	L_4
12107	388800	$2^6 \cdot 3^5 \cdot 5^2$	L_5
12193	2478600	$2^3 \cdot 3^6 \cdot 5^2 \cdot 17$	L_6
12593	12393000	$2^3 \cdot 3^6 \cdot 5^3 \cdot 17$	L_7
13649	234375	$3 \cdot 5^7$	L_8
18947	157216	$2^5 \cdot 17^3$	L_9

Table 1 : Quelques factorisations auxiliaires pour la factorisation de N = 13290059

Il ne reste plus qu'à combiner certaines lignes pour faire apparaître des carrés, qui nous fourniront des racines carrées de 1. À la main, on multiplie les lignes L_6 et L_9 pour trouver

$$(12193 \cdot 18947)^2 \equiv (2^4 \cdot 3^3 \cdot 5 \cdot 17^2)^2 \bmod N.$$

On calcule alors d'une part

$$N_1 = (12193 \cdot 18947) \equiv 5089768 \bmod N,$$

$$N_2 = (2^4 \cdot 3^3 \cdot 5 \cdot 17^2) \equiv 624240 \bmod N,$$

d'où l'on déduit $N_1^2 \equiv N_2^2 \bmod N$, ou $(N_1/N_2)^2 \equiv 1 \bmod N$, soit $5089768/624240 \equiv 11125472 \bmod N$.

Combiner des lignes pour faire apparaître des carrés revient à chercher des produits dont la factorisation ne contient que des exposants pairs. Dit autrement, on cherche une combinaison linéaire dans la matrice des exposants des lignes modulo 2. Dans notre cas, la matrice serait (avec v_p désignant la valeur de l'exposant modulo 2 de la puissance de p divisant le nombre $x^2 \bmod N$) :

ligne	v_2	v_3	v_5	v_7	v_{11}	v_{13}	v_{17}
L_1	1	1	0	0	0	1	1
L_2	0	1	0	0	0	1	0
L_3	1	1	0	0	0	0	1
L_4	0	0	1	0	0	0	0
L_5	0	1	0	0	0	0	0
L_6	1	0	0	0	0	0	1
L_7	1	0	1	0	0	0	1
L_8	0	1	1	0	0	0	0
L_9	1	0	0	0	0	0	1

La somme des lignes L_6 et L_9 correspond bien à une relation de dépendance et donc à une solution pour notre problème. On laisse au lecteur le soin de trouver d'autres combinaisons.

On renvoie à la littérature pour les algorithmes utilisés dans cette partie d'algèbre linéaire. Dans tous les cas, on travaille sur des matrices booléennes creuses, qui sont souvent gigantesques, comme celle du récent record dont nous parlerons plus loin. Cette partie des algorithmes est très technique à distribuer entre processeurs, et se révèle souvent le point bloquant dans les calculs.

L'analyse globale de l'algorithme fait apparaître qu'il faut un nombre gigantesque de factorisations auxiliaires, mais celles-ci peuvent être cherchées de façon indépendante et donc massivement distribuables (par exemple en donnant des intervalles de variation distincts à chaque nœud de calcul).

Il peut paraître paradoxal de factoriser un nombre N en s'aidant de nombreuses factorisations auxiliaires de la même taille. Cette méthode marche pourtant, et on peut même l'améliorer, en cherchant à factoriser des nombres auxiliaires plus petits. La méthode du crible quadratique (*quadratic sieve* – QS, due à C. Pomerance améliorant le crible linéaire de R. Schroeppel) consiste à consider le polynôme.

$$Q(Z) = (Z + \lfloor\sqrt{N}\rfloor)^2 - N$$

où $\lfloor r \rfloor$ désigne la partie entière du réel r. Dans notre exemple : Q(Z) = (Z+3645)²−13290059. Tant que z est petit, Q(z) sera proche de 2z√N, ce qui est plus petit que N. Par exemple :

$$Q(1) = 3257, Q(2) = 10550 = 2 \cdot 5^2 \cdot 211, \ldots$$

Il ne reste plus qu'à faire varier z dans un intervalle pour collecter des factorisations auxiliaires $Q(z_i)$. Posant $x_i = z_i + 3645$, on peut écrire :

$$x_i^2 \equiv (z_i + 3645)^2 \equiv Q(z_i) \bmod N$$

et nous retombons sur le formalisme précédent.

Le choix d'un polynôme quadratique permet également d'accélérer la phase de factorisation des entiers auxiliaires. On peut prévoir quand 5 divise les valeurs de Q(Z) : il suffit de résoudre l'équation Q(Z) ≡ 0 mod5, dont les racines, dans notre exemple, sont 2 et 3. Si k est un entier, on trouve que :

$$Q(2 + 5k) = 5(5\,k^2 + 7294\,k + 2110),$$

donc on sait que toutes ces valeurs de Q seront divisibles par 5. Dans le même esprit que le crible d'Eratosthene qui permet de trouver rapidement les nombres premiers, on peut trouver tous les diviseurs premiers des Q(z) dans un intervalle.

On peut voir le crible algébrique (*number field sieve* – NFS, inventé par J. Pollard) comme une généralisation de la méthode précédente avec des polynômes de degré plus élevé. La théorie est plus compliquée, mais les factorisations auxiliaires seront faites sur des nombres plus petits.

On peut démontrer que la complexité (le nombre d'opérations) des algorithmes de factorisation s'écrit à l'aide de la fonction :

$$L_x[\alpha, c] = \exp\left(c(\log x)^{\alpha}(\log\log x)^{1-\alpha}\right),$$

avec $0 \leq \alpha \leq 1$. Cette fonction interpole les algorithmes de temps de calcul polynomial $L_x[0, c] = (\log x)^c$ et de temps de calcul exponentiel $L_x[1, c] = x^c$. On donne dans la table ci-dessous les complexités des trois algorithmes : division jusqu'à la racine carrée, QS et NFS, ainsi que quelques valeurs numériques.

	$p \leqslant \sqrt{N}$	QS	NFS
N	$L_N[1, 1/2]$	$L_N[1/2, 1]$	$L_N[1/3, 1]$
$2^{128} \approx 10^{38}$	1.84×10^{19}	4.61×10^{8}	1.85×10^{5}
$2^{256} \approx 10^{77}$	3.40×10^{38}	1.46×10^{13}	2.02×10^{7}
$2^{512} \approx 10^{154}$	1.16×10^{77}	6.69×10^{19}	1.02×10^{10}
$2^{768} \approx 10^{231}$	3.94×10^{115}	1.27×10^{25}	9.49×10^{11}
$2^{1024} \approx 10^{308}$	1.34×10^{154}	4.42×10^{29}	3.82×10^{13}

On peut constater sur ce tableau que casser un nombre de 1024 bits à l'aide de NFS ne semble pas hors de portée. Comme un nombre de 768 bits a été factorisé, il ne faut en théorie guère plus que 40 fois plus de calculs pour factoriser un nombre de 1024 bits.

Rentrer plus avant dans la description de ces cribles nous entraînerait trop loin, mais les références citées assouviront la soif du lecteur.

3 La factorisation à travers les âges

Historiquement, les factorisateurs appelaient les nombres par leur prénom : $F_n = 2^{2^n}+1$ désigne le n-ième nombre de Fermat, 3, 101+ désigne $3^{101}+1$, etc. Le Lieutenant Colonel Allan J. C. Cunningham a laissé son nom à un projet de factorisation commencé en 1925 (avec H. J. Woodall), et dont le secrétaire perpétuel est Sam Wagstaff, qui en assure les éditions papier et électronique **[BLS+88]**. On trouve sur le Web ces tables, qui ont servi de pierre de touche pour tous les algorithmes de factorisation et de primalité. Ces tables sont sans cesse remplies et étendues, même si une des méthodes modernes (celle du crible algébrique avec polynôme creux) permet de mieux les factoriser que les autres nombres.

Les factorisateurs ont constaté avec le temps que les nombres les plus difficiles à factoriser semblent être ceux qui ont deux grands facteurs premiers. C'est ainsi que les clés RSA sont formées d'un produit de deux grands nombres premiers. Dans les années 1990, RSA, INC. avait lancé un concours de factorisation en fabriquant des clés spéciales. Ce ne sont pas des clés opérationnelles, ce qui évite des ennuis évidents. Gardons à l'esprit que si la communauté académique factorise un nombre de n bits réputé difficile, alors tous les nombres de cette taille sont vulnérables. En fonction de votre degré de paranoïa, vous pouvez croire que les grandes agences de sécurité ont au choix : déjà trouvé un algorithme de factorisation en temps polynomial (même sans ordinateur quantique) ; savent factoriser 100 bits de plus que la communauté académique.

Les deux graphiques qui suivent démontrent leur double origine mathématique et informatique : les mathématiciens parlent plutôt de la taille des nombres en chiffres décimaux, les informaticiens en bits. Passer de l'une à l'autre des échelles se fait facilement à l'aide de l'équation fondamentale 155dd = 512b. On trouvera en abscisse les années, en ordonnée la taille en chiffres décimaux. Ainsi, le nombre $F_7 = 2^{2^7}+1$ a été factorisé en 1970, comme l'indique la figure 1 : il a été par la première méthode tous-usages (CFRAC – basée sur les fractions continues). C'est un record qui précède RSA dans le temps. Jusqu'au milieu des années 1980, tous les records de factorisation ont été réalisés sur des ordinateurs relativement monstrueux : IBM, CDC, CRAY, etc., surtout à partir du moment où la factorisation est devenue stratégique. Puis les réseaux de station de travail ont démocratisé les calculs et permis de gagner en taille (le premier à le faire a été R. D. Silverman sur des stations SUN). Puis Internet a permis de distribuer encore plus les calculs, dans tous les pays, tous les continents, cela bien

avant sa découverte par le grand public dans les années 1990. La participation massive des internautes a été de mode pendant un temps (culminant avec la factorisation de RSA-129), mais abandonnée pour les derniers records en date, en partie à cause des méthodes plus difficiles à mettre en œuvre (grande mémoire centrale, caches, etc.), en partie à cause du coût social de tels calculs : gérer des millions de mails automatiquement est prenant en temps humain, sans parler des problèmes de sauvegarde, etc. On peut cependant dans certains cas utiliser des réseaux locaux monstrueux (même de Playstations !).

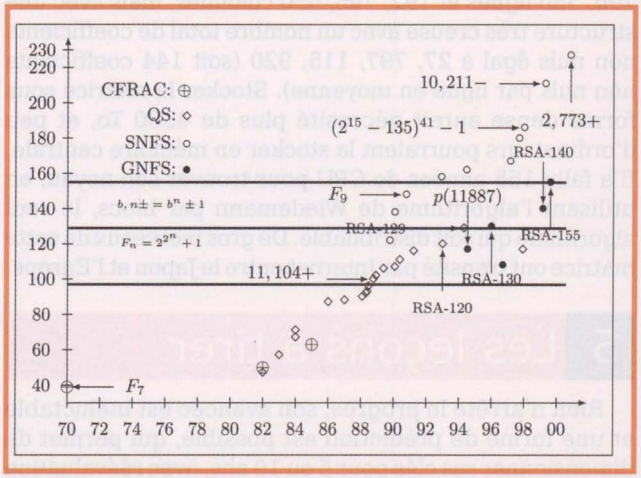


Figure 1 : Records de factorisation entre 1970 et 2000 ; SNFS est une variante de NFS adaptée aux nombres du type bn± 1 ; GNFS s'attaque à tous les nombres. Les noms sont expliqués dans un glossaire à la fin de l'article.

La factorisation a au minimum bénéficié de la loi de Moore, ce qui est visible sur les portions de droites présentes sur les graphiques. Un angle brutal désigne un progrès d'origine mathématique (apparition d'une nouvelle variante, d'un nouvel algorithme), ou d'origine pratique (mise en réseaux, algorithmes mieux distribués). On voit également sur les figures la vie et la mort des algorithmes qui battent les records. De façon duale, on peut y lire aussi les performances des différents algorithmes en fonction de la taille des entiers que l'on cherche à factoriser.

La figure 1 contient également trois « barres » mythiques. La plus basse se situe au niveau 97dd, la deuxième 129dd, la troisième 512 bits. Cette dernière est symbolique et a marqué quelques esprits. Celle de 129dd correspond à la taille du message défi de l'article original de RSA. Quant à la première, elle rappelle que la Carte Bleue utilisait une clé RSA déjà un peu trop petite à son lancement, même s'il a fallu attendre 10 ans après le franchissement de cette barre pour que quelqu'un prenne le risque de casser cette clé opérationnelle... Il semblerait que la leçon ait été à peu près comprise, les nouvelles clés ayant au moins 900 bits en général (mais il faudra songer à les changer régulièrement).

De même, le cassage de clés de 512 bits n'a pas empêché, encore aujourd'hui, l'utilisation de certificats avec des clés de cette taille. Il était vrai que peu de gens avaient les compétences pour factoriser ces nombres ; la situation a changé, on trouve des programmes très efficaces sur le réseau et on peut casser tranquillement des clés dans sa cave avec quelques ordinateurs.

La figure 2, page suivante montre que l'écart entre les deux variantes de NFS semble augmenter. On peut voir qu'il y a environ 10 ans d'écart pour des nombres de même taille, mais des deux types différents. Ainsi, $F_9 = 2^{2^9}+1$ a été factorisé 10 ans plus tôt qu'une clé RSA de même taille.

AUTOUR DE L'ARTICLE...

▪ LE CRYPTOSYSTÈME RSA

Rappelons comment marche le système inventé par Rivest, Shamir et Adleman en 1976. Nous présentons une version simplifiée.

Fabrication des clés : on choisit p et q premiers, $p \neq q$; on calcule $N = p\,q$, $\lambda(N) = \text{ppcm}(p-1, q-1)$; on choisit e tel que $\text{pgcd}(e, \lambda(N))=1$, $d \equiv 1/e \bmod \lambda(N)$. La clé publique est (N, e), la clé privée est d.

Chiffrement/déchiffrement : pour chiffrer, Bob calcule $y = x^e \bmod N$ et l'envoie à Alice. Alice déchiffre en calculant $y^d \bmod N \equiv x$.

Utiliser ce chiffrement tel quel est dangereux, à cause des propriétés arithmétiques trop fortes du chiffrement. Des chercheurs ont imaginé ajouter une indication de temps aux messages envoyés. Par exemple, si on envoie le même message aux temps t et t+1, on se retrouve à chiffrer :

$$C_1 \equiv M^3 \bmod N, \quad C_2 \equiv (M+1)^3 \bmod N.$$

Malheureusement, c'est encore dangereux, comme montré par Franklin et Reiter. À partir des deux messages chiffrés, on calcule facilement :

$$\begin{cases} C_2 + 2C_1 - 1 &= 3M^3 + 3M^2 + 3M \\ C_2 - C_1 + 2 &= 3M^2 + 3M + 3 \end{cases}$$

d'où $M = (C_2+2C_1-1)/(C_2-C_1+2) \bmod N$, sans factoriser N, ni rien apprendre sur N !. C'est ce qui explique pourquoi de nombreux chercheurs pensent que casser RSA serait plus facile que factoriser N.

Quelle solution pour avoir un RSA mieux défendu ? Il faut rendre le chiffrement aléatoire, à l'aide de fonctions de hachage cryptographique. Bellare et Rogaway ont proposé la variante OAEP (pour *Optimal Asymmetric Encryption Padding*) qui demande de chiffrer des messages M en utilisant un nombre aléatoire protégé par des fonctions de hachage H et G comme suit :

$$((M \oplus G(r)) || r \oplus \mathcal{H}(M \oplus G(r)))^e \bmod N.$$

AUTOUR DE L'ARTICLE...

▪ GLOSSAIRE DES NOMS DE NOMBRES

Le projet Cunningham donne des codes aux nombres : $b^n \pm 1$ est noté b, n± 1. Des factorisations très particulières sont les factorisations d'Aurifeuille. On trouve par exemple :

$$2^{4k-2}+1 = L \cdot M = (2^{2k-1}-2k+1)(2^{2k-1}+2k+1)$$

ce qui explique le 2, 1642M de la figure 2.

Dans l'ancien challenge RSA, des tables de nombres de partitions étaient également données à factoriser. Pour n entier, p(n) désigne le nombre de façons d'écrire n comme somme d'entiers strictement positifs. Par exemple : p(4) = 5 car

$$4 = 3+1 = 2+2 = 2+1+1 = 1+1+1+1.$$

Les nombres RSA sont écrits aussi bien en chiffres décimaux qu'en chiffres binaires : RSA-120 est une clé RSA de 120 chiffres décimaux ; RSA-768b est une clé de 768 chiffres binaires.

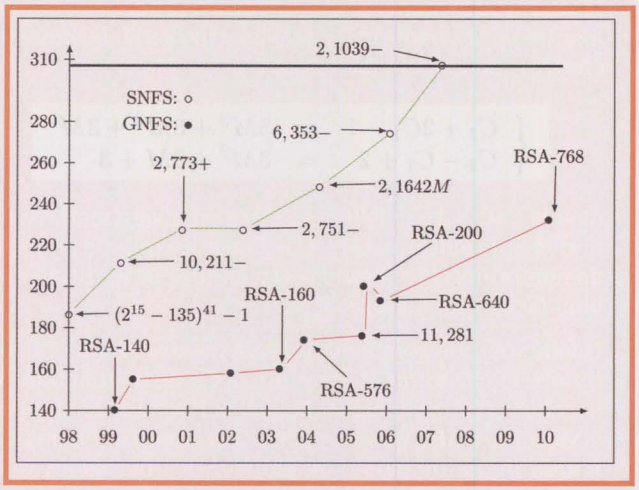


Figure 2 : Records de factorisation depuis 1998

Une interpolation linéaire réalisée sur un graphique comparable à nos deux figures par R. Brent lui a permis de calculer la fonction suivante qui donne l'année de factorisation en fonction de la taille du nombre.

$$\text{année} = 2.15(\log N)^{1/3}(\log\log N)^{2/3} + 1951.45$$

À cette aune, 896 bits seraient atteints en 2014, et 1024 bits en 2018.

4 Le dernier record en date : RSA 768 (232dd)

Les chasseurs de prime rassemblés pour traquer les facteurs de ce nombre sont peu nombreux : Thorsten Kleinjung ; Kazumaro Aoki ; Jens Franke ; Arjen Lenstra ; Emmanuel Thomé ; Joppe Bos ; Pierrick Gaudry ; Alexander Kruppa ; Peter Montgomery ; Dag Arne Osvik ; Herman Te Riele ; Andrey Timofeev ; Paul Zimmermann. Mais ils ont beaucoup de machines : les calculs de la première phase (crible, cf. appendice) se sont déroulés d'août 2007 à avril 2009, et ont nécessité l'équivalent de 1500 années AMD64 sur plusieurs sites de par le monde. Les 64, 334, 489, 730 factorisations auxiliaires obtenues durant la première phase occupaient 5 To (fichiers compressés).

La seconde phase d'algèbre linéaire est titanesque : après nettoyage, la matrice booléenne finale avait 192, 796, 550 lignes et 192, 795, 550 colonnes, mais avec une structure très creuse avec un nombre total de coefficients non nuls égal à 27, 797, 115, 920 (soit 144 coefficients non nuls par ligne en moyenne). Stocker la matrice sous forme dense aurait nécessité plus de 4500 To, et peu d'ordinateurs pourraient la stocker en mémoire centrale. Il a fallu 155 années de CPU pour trouver son noyau, en utilisant l'algorithme de Wiedemann par blocs, le seul algorithme qui soit distribuable. De gros morceaux de cette matrice ont transité par Internet entre le Japon et l'Europe.

5 Les leçons à tirer

Rien n'arrête le progrès, son avancée est inéluctable et une forme de prédiction est possible, qui permet de dimensionner ses clés pour 5 ou 10 ans, avec réévaluation périodique. Il semble qu'il y ait 10 ans entre le cassage académique, la prise de conscience et les attaques éventuelles sur des clés opérationnelles. Il faut donc effectuer une veille de tous les instants.

Les attaques contre RSA sont multiples : canaux cachés, défaillances, mauvaises utilisations, mauvaise fabrication ou gestion de clés et/ou de certificats (voir le très récent article de Lenstra *et alii* sur un recensement des mauvaises clés sur le réseau). La factorisation d'entiers est emblématique, mais il ne faut pas que ce soit l'arbre qui cache la forêt. ■

▪ RÉFÉRENCES

- [BLS+88] J. Brillhart, D. H. Lehmer, J. L. Selfridge, B. Tuckerman, and S. S. Wagstaff, Jr. Factorizations of bn± 1, b=2,3,5,6,7,10,11,12 up to high powers, Number 22 in Contemporary Mathematics, AMS, 2 edition, 1988

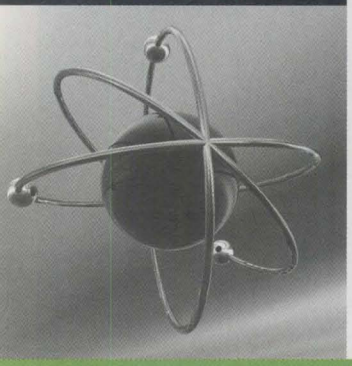
-[Coh00] H. Cohen, A course in algorithmic algebraic number theory, volume 138 of Graduate Texts in Mathematics, Springer–Verlag, 2000, Fourth printing

-[CP05] R. Crandall and C. Pomerance, Prime numbers – A Computational Perspective, Springer Verlag, 2nd edition, 2005

-[Knu97] D. E. Knuth, The Art of Computer Programming: Seminumerical Algorithms, Addison-Wesley, 3e edition, 1997

-[WS94] H. C. Williams and J. O. Shallit, Factoring integers before computers, in Mathematics of Computation 1943–1993: a half-century of computational mathematics (Vancouver, BC, 1993), volume 48 of Proc. Sympos. Appl. Math., pages 481–531, Amer. Math. Soc., Providence, RI, 1994


POUR RENFORCER
LA SÉCURITÉ
DE VOTRE ENTREPRISE,
GLISSEZ-VOUS DANS
LA PEAU D'UN HACKER !
FORMATIONS INTRUSIONS
Cours SANS Institute
Certifications GIAC
SEC 542
Tests d'intrusion applicatifs
et hacking éthique
SEC 560
Network Penetration Testing and
Ethical Hacking
SEC 660
Tests d'intrusion avancés, exploits,
hacking éthique
Dates et plan disponibles
Renseignements et inscriptions
par téléphone +33 (0) 141 409 700
ou par courriel à : formations@hsc.fr
www.hsc-formation.fr
SANS
GIAC
HSC
création agence Comellink - crédit photo : Hedge



CRYPTOGRAPHIE QUANTIQUE ET CRYPTOGRAPHIE POST-QUANTIQUE : MYTHES, RÉALITÉS, FUTUR

Robert Erra - erra@esiea.fr

« Quantum theory is a theory of parallel interfering universe » - **David Deutsch**

mots-clés : *INFORMATIQUE QUANTIQUE / CALCUL QUANTIQUE / QUBIT / SHOR / CRYPTOGRAPHIE / POST-QUANTIQUE*

Imaginons un instant que vous puissiez acheter un calculateur quantique, appelons-le (Q)HAL, capable de casser en quelques instants une clé RSA de 1024 voire de 2048 bits, vous imaginez aisément l'impact sur la sécurité de l'information ? Catastrophique ! En un mot, tout ce qui repose sur la cryptographie asymétrique (SSL/TLS) est mort ! Tout ? Hum, pas tout à fait, un calculateur quantique pourrait bien casser tout système de chiffrement asymétrique dont la sécurité repose sur la difficulté de factoriser un module RSA (ou sur la difficulté de résoudre le problème du logarithme discret). Mais pour les autres systèmes qui ne reposent pas sur l'un de ces deux problèmes ? Rien n'est moins sûr. Mais alors, qu'en est-il de la cryptographie quantique aujourd'hui ? Elle existe pratiquement mais permet essentiellement l'échange, le transport et la mise en commun de clés. C'est de la distribution quantique de clés. On a ainsi un paradoxe : l'existence de (Q)HAL ne remet(tra) pas forcément en cause la cryptographie quantique car si celle-ci utilise bien des systèmes quantiques, elle n'utilise pas de calculateur quantique ! Cet article s'adresse à des non spécialistes de la mécanique ou de l'informatique quantique, groupe dont fait partie l'auteur, qui s'est posé la question : l'informatique quantique va-t-elle tuer la cryptographie (vision pessimiste) que nous connaissons, et si oui, pourra-t-elle en même temps permettre d'inventer une nouvelle cryptographie (vision optimiste) qui lui résistera ? Cet article se propose de présenter rapidement l'informatique quantique, en tuant au passage certaines idées reçues, et en cherchant à préciser les limites de l'informatique quantique.

1 Introduction

L'informatique quantique **[WIKIIQ]** se propose d'exploiter les propriétés quantiques de la matière **[LEB]** afin, entre autres, de calculer vite et mieux qu'un ordinateur classique ou de faire des actions qu'un ordinateur classique ne peut faire (par exemple, téléporter un photon).

Là où un bit classique ne peut représenter que 0 ou 1, un bit quantique (qubit, de *quantum bit*) peut représenter simultanément 0 et 1. Deux qubits peuvent représenter simultanément 0, 1, 2 et 3 (ou 00, 01, 10 et 11 en binaire), et ainsi de suite. Un ensemble de 11 qubits peut donc théoriquement représenter simultanément tous les entiers de 0 à 2^{11}-1 et effectuer certains calculs sur ces nombres.

Si Eve, attaquante anonyme, disposait d'un calculateur quantique à $2^{11} = 2048$ qubits, il pourrait, en simplifiant un peu, factoriser une clé RSA de 2048 bits en quelques instants. Enfin, Eve pourra le faire si jamais un tel calculateur existe un jour.

Mais aujourd'hui, un calculateur aussi puissant n'existe pas encore, a priori. Si en revanche Alice, qui comme d'habitude continue à vouloir communiquer de manière discrète avec Bernard, décide d'échanger des clés de manière quantique, elle peut, sous certaines contraintes que nous évoquerons plus loin, acheter ou construire un dispositif qui lui permet d'effectuer un échange de clé de manière sécurisée.

Ce qui relie la cryptographie quantique et le calcul quantique, c'est la mécanique quantique, science des phénomènes atomiques et subatomiques. La cryptographie quantique et le calcul quantique reposent sur deux propriétés essentielles (mais mystérieuses) : la superposition et l'intrication, voir **[LEB]**.

Une première remarque, il arrive fréquemment que cryptographie post-quantique et cryptographie quantique soient confondues. Néanmoins, ce sont des notions très différentes :

- La cryptographie quantique est quelque chose de concret de nos jours. Elle permet, par exemple, de distribuer des clés de manière sécurisée. C'est encore sur elle que reposent l'algorithme BB84 et les autres algorithmes de distribution de clés (B92, EPR). Mais elle ne permet pas encore de chiffrer de manière symétrique un fichier.
- La cryptographie post-quantique pose la question suivante : si (Q)HAL existe un jour, quelle cryptographie devra-t-on utiliser afin de sécuriser les communications du futur ? En clair, quelle cryptographie résistant à un calculateur quantique pourrait continuer à être utilisée de manière sécurisée ?

Pourrait-on d'ailleurs utiliser un calculateur quantique pour chiffrer de manière quantique un disque dur ou même un simple fichier ? C'est loin d'être évident car il existe très peu d'algorithmes de chiffrement quantique permettant de faire cela, les seuls qui ont été publiés sont très récents (et doivent donc encore être évalués) et comme ils sont asymétriques...

2 Qubits

Un bit classique peut valoir 0 ou 1, que ce soit sur une carte perforée, une disquette, un disque SSD ou une barrette mémoire. Ainsi, un mot de N bits classiques ne peut prendre qu'une des 2^N valeurs possibles.

Un photon est l'une des rares particules quantiques qu'on peut « voir » et manipuler aisément : on peut le générer, mesurer sa polarité, etc. Bien que les photons ne puissent a priori être utilisés aujourd'hui pour construire un calculateur quantique, ils sont au cœur des protocoles de distribution quantique de clés car on peut aussi le faire voyager sur de longues distances, grâce par exemple aux fibres optiques.

Nielsen **[NIEL]** explique que pour un qubit, le support n'importe pas : des calculateurs quantiques pourraient voir le jour à partir d'un photon polarisé ou des ions piégés, ce qui change, c'est qu'un qubit peut prendre autant de valeurs qu'il y a de mélanges possibles des valeurs 0 et 1.

Soit un système physique qui peut se trouver soit dans un état Ψ, soit dans un état Φ, le principe de superposition quantique stipule que le système peut alors se trouver dans un état composé d'une combinaison simultanée des deux états, combinaison linéaire qui s'écrit ainsi dans le formalisme proposé par Dirac :

$$\alpha|\Psi>+\beta|\Phi>$$

(avec α et β deux nombres complexes vérifiant $|\alpha|^2+|\beta|^2=1$).

Comment se représenter un qubit ? Les physiciens aiment bien dire que c'est un vecteur de l'espace vectoriel complexe $\mathbb{C}^2$ pour lequel une base particulière a été choisie :

$$B=\{|0>,|1>\}$$

Les vecteurs $|0>$ et $|1>$ forment une base orthogonale (en pratique cette base correspond par exemple à deux des polarisations « opposées » possibles d'un photon). Lorsqu'on manipule un ou des qubits, on doit choisir une base, et B est la base standard du calcul quantique. Un qubit peut alors être représenté sur ce qu'on appelle la sphère de Bloch (ou Poincaré-Bloch) (voir figure 1).

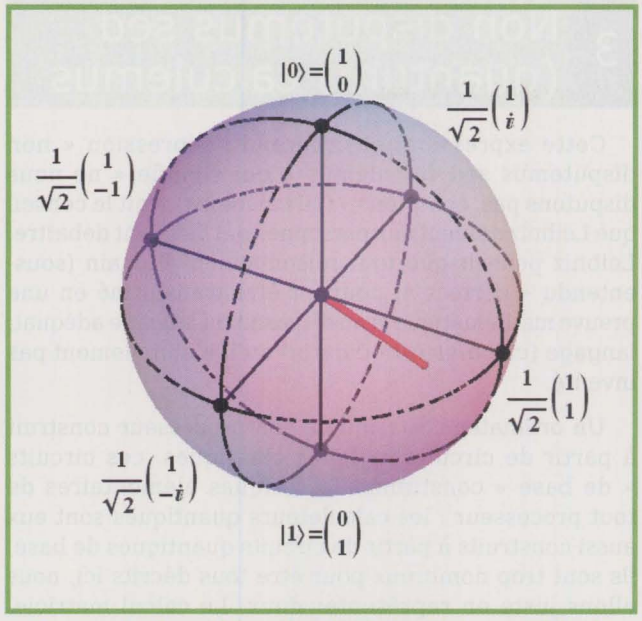


Figure 1 : La sphère de Bloch (ou Poincaré-Bloch)

On peut aussi dire que $|0>$ représente le vecteur $\begin{pmatrix}0\\1\end{pmatrix}$ tandis que $|1>$ représente $\begin{pmatrix}1\\0\end{pmatrix}$ mais ce qui est important, c'est qu'un qubit étant un système quantique, il peut réellement « valoir » $\alpha|0>+\beta|1>$ avec encore une fois α et β deux nombres complexes vérifiant $|\alpha|^2+|\beta|^2=1$. Ainsi, un qubit peut être simultanément dans plusieurs états à la fois et cette propriété s'étend à un ensemble de qubits, qu'on appelle

un registre quantique. Le principe de superposition quantique est à la base de presque tous les algorithmes quantiques. C'est l'un des extraordinaires avantages d'un registre quantique, c'est comme si on avait accès à une énorme quantité de registres classiques, et ce, en un instant.

Un phénomène limite cependant la puissance du calcul quantique : mesurer l'état d'un registre quantique détruit de l'information ; (Q)HAL0 était quant à lui construit à partir de molécules de chloroforme dont la durée de vie ne dépassait pas quelques minutes **[WIKICQ]**. Et si on « mesure » l'état du qubit $\alpha|0>+\beta|1>$ on aura $|0>$ avec une probabilité $|\alpha|^2$ et $|1>$ avec une probabilité $|\beta|^2$. On a le même problème avec un système à plusieurs qubits : mesurer un registre quantique fait donc « perdre » des valeurs. Ce qui interdit toute copie d'une valeur manipulée, c'est un des problèmes du calculateur quantique.

L'intrication est un autre phénomène encore plus mystérieux, qui concerne les états quantiques d'un objet. Deux objets quantiques intriqués se doivent d'être considérés comme un seul et unique objet et ce de manière inséparable. Une conséquence majeure est qu'une modification sur l'un des objets modifie également le second de manière instantanée et quelle que soit la distance qui les sépare. C'est ce qu'on appelle la propriété de non-localisation quantique.

3 Non disputemus sed (quantum) calculemus

Cette expression, qui reprend l'expression « non disputemus sed calculemus » qui signifie « ne nous disputons pas, comptons » (ou calculons) était le conseil que Leibniz donnait aux personnes qui devaient débattre. Leibniz pensait que tout raisonnement humain (sous-entendu « correct ») pourrait être transformé en une preuve mathématique si on disposait du langage adéquat, langage (*calculus ratiocinator*) qu'il n'a finalement pas inventé.

Un ordinateur classique a son processeur construit à partir de circuits logiques classiques, ces circuits « de base » constituent les briques élémentaires de tout processeur ; les calculateurs quantiques sont eux aussi construits à partir de circuits quantiques de base, ils sont trop nombreux pour être tous décrits ici, nous allons juste en représenter deux. Le calcul matriciel est très souvent utilisé pour représenter les circuits quantiques de base. Une transformation linéaire sur un espace vectoriel est parfaitement définie en donnant l'image de chaque élément d'une base. Soit par exemple la transformation linéaire X qui associe le vecteur $|0>$ au vecteur $|1>$ et le vecteur $|1>$ au vecteur $|0>$; elle est représentée par la matrice $X=\begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}$, elle est nommée X car elle représente la négation. Ce circuit logique quantique est dit unitaire car la matrice X est unitaire **[WIKIUNI]** : i.e. elle vérifie $XX^H = X^H X = I$ où X^H représente la matrice adjointe (conjuguée et transposée) et I est la matrice identité ; cette matrice X est inversible.

On a aussi la matrice de Hadamard (qui effectue la transformation dite de Walsh-Hadamard) $H=\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}$, elle représente la transformation linéaire qui associe le vecteur $|0>$ au vecteur $\frac{1}{\sqrt{2}}(|0>+|1>)$ et le vecteur $|1>$ au même vecteur $\frac{1}{\sqrt{2}}(|0>+|1>)$. Cette transformation, voir la figure 2, est fondamentale en calcul quantique car elle permet, appliquée à N qubits, (c'est un peu plus compliqué mais similaire) de générer les 2^N superposés possibles, ce qui correspond aux 2^N nombres possibles allant de 0 à 2^N - 1. La figure 3 représente un circuit quantique à deux qubits composé de deux portes de Hadamard et d'une porte X.

La matrice de Hadamard est unitaire, c'est pour cela qu'on ajoute le facteur $\frac{1}{\sqrt{2}}$ et inversible. Pourquoi insister sur unitaire et inversible ? Parce qu'un calculateur quantique ne peut être réalisé qu'à partir de portes logiques quantiques dont la transformation linéaire associée est unitaire, et donc inversible. C'est vraisemblablement la limite la plus dramatique des calculateurs quantiques. En fait, on peut effectuer des calculs à partir de transformations non unitaires mais cela revient à dire au calculateur quantique de sortir une valeur au hasard. En résumé : un calculateur quantique permet un parallélisme massif, mais limité dans ses capacités.

Que peut-on calculer aujourd'hui avec un ordinateur ? Il faut d'abord se demander ce qui est calculable tout court. Il y a diverses réponses, Church, Turing, Post, ainsi que Kleene et d'autres ont essayé de répondre à cette difficile question. La thèse dite de Church-Turing (qui est en fait une hypothèse et non une chose prouvée) stipule que toute fonction qui peut être regardée comme naturellement calculable (*naturally computable*) peut être calculée par une machine de Turing universelle, et donc, par votre ordinateur. Nous nous contenterons de dire, en simplifiant pour cet article : toute fonction calculable par une machine de Turing universelle est une fonction primitive récursive, voir **[REY]**.

Mais le modèle de calcul posé par le calcul quantique est difficilement comparable avec celui d'une machine de Turing. Soulevons la question que le lecteur attentif n'aura pas manqué de se poser, depuis le début, nous utilisons l'expression calculateur quantique et non pas ordinateur quantique. Pourquoi ? Pour mieux souligner qu'un calculateur quantique ne peut ou ne pourra pas « faire » tout ce que permet de faire aujourd'hui un ordinateur classique. Un ordinateur classique est une machine de Turing universelle tandis qu'un calculateur quantique ne l'est pas, même si Deutsch parle de machine de Turing quantique. À la lecture des nombreux ouvrages et articles qui présentent les algorithmes quantiques, force est de constater que la liste de ces algorithmes est assez courte, même si elle s'allonge constamment.

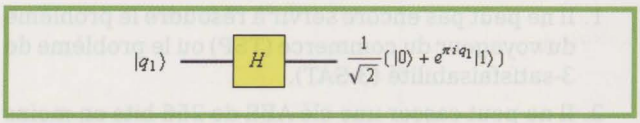


Figure 2 : Un exemple de circuit quantique à un qubit

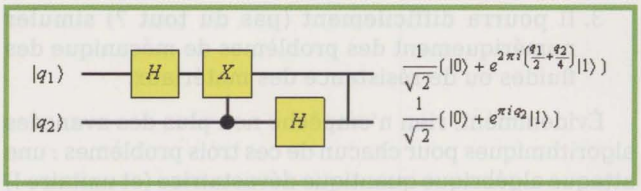


Figure 3 : Un autre exemple de circuit quantique à deux qubits (ce qui revient à appliquer une matrice (4,4) à l'entrée)

Mais tout d'abord : peut-on réaliser un calculateur quantique? Oui, il existe des moyens très différents : utiliser des « ions piégés », la RMN, des molécules de chloroforme ; c'était le cas de (Q)HAL0). Mais nul doute que tous ces moyens sont complexes et pour l'instant assez chers. L'ironie, c'est que Feynman avait imaginé un calculateur quantique car les ordinateurs classiques ne permettent pas de simuler des systèmes quantiques, il avait alors proposé d'utiliser les principes de la mécanique quantique pour simuler un système quantique. Le calculateur quantique était né, au moins en théorie. La progression des ordinateurs quantiques est tout de même très loin de suivre la loi de Moore. En effet, avec 3 qubits opérationnels en 1998, 4 qubits opérationnels en 2000, on n'en est aujourd'hui qu'à 16 qubits, à ce rythme-là, on pourra acheter un calculateur quantique utile au XXIème siècle !

Mais les ordinateurs quantiques existent bel et bien, IBM a semble-t-il été la première entreprise à construire un tel ordinateur quantique, appelons-le (Q) HAL0. Véritable prouesse technologique, cet ordinateur quantique à 7 qubits a permis en 2001 de factoriser le nombre 15 par l'algorithme de factorisation quantique de Shor. C'est aussi probablement la factorisation d'un entier à quatre bits la plus chère de l'histoire ! Le site **[IBM]** explique : « ... IBM scientists controlled a vial of a billion-billion (10**18) of these molecules so they executed Shor's algorithm and correctly identified 3 and 5 as the factors of 15. »

On voit que cela n'est pas si simple et bien que (Q) HAL0 existe, il semble qu'un courant se dessine pour prouver qu'un (Q)HAL à 256 qubits ou plus ne pourra peut-être pas être utilisable en pratique à cause justement de phénomènes quantiques comme la décohérence quantique. Wikipédia définit ce phénomène de la manière suivante : « *... phénomène physique susceptible d'expliquer la transition entre les règles physiques quantiques et les règles physiques classiques [...], à un niveau macroscopique. Plus spécifiquement, cette théorie apporte une réponse, considérée comme étant la plus complète à ce jour, au paradoxe du chat de Schrödinger et au problème de la mesure quantique* ».

Harley **[HAR]**, qui fait clairement partie des anti (Q) HAL, a déclaré : « *We can factor the number 15 with quantum computers. We can also factor the number 15 with a dog trained to bark three times.* » (On conseille vivement la lecture du réjouissant article **[BEBE]**).

4 Algorithmes quantiques

À partir des circuits quantiques de base comme la porte de Hadamard ou la porte X, on peut construire un calculateur quantique à plusieurs qubits. Mais que peut calculer un tel calculateur, ou plus exactement, quels types de problèmes algorithmiques un tel calculateur quantique peut-il résoudre ?

Une lecture de quelques ouvrages **[PQC] [QAI] [IQC] [QCS]** montre qu'il existe essentiellement 4 types d'algorithmes qu'un calculateur quantique peut exécuter :

- l'algorithme de Deutsch et Deutsch-Josza ;
- l'algorithme de Simon ;
- l'algorithme de factorisation de Shor ;
- l'algorithme de Grover.

L'algorithme de Deutsch **[WIKID]** et Deustsh-Josza (qui en est une généralisation) est un des tout premiers algorithmes quantiques. On pose la question suivante, soit une fonction $f:\{0,1\}^n \rightarrow \{0,1\}$, nous faisons l'hypothèse que cette fonction est soit constante (0 ou 1), soit équilibrée : pour la moitié des entrées, elle vaut 0 et pour l'autre moitié des entrées, elle vaut 1. Avec un ordinateur classique, il faut, pour savoir si la fonction est constante ou équilibrée, tester au moins $2^{n-1}+1$ entrées, soit un temps exponentiel. Deutsch a proposé d'utiliser, dans le cas où n=1, une transformation de Hadamard sur deux qubits. On part de l'état $(|0>|1>)$ une transformation de Hadamard sur chacun des deux qubits met le système dans l'état $\frac{1}{\sqrt{2}}(|0>+|1>)(|0>-|1>)$. Supposons que nous ayons une implémentation quantique de la fonction f sous forme de boîte noire (on dit aussi un oracle quantique), celle-ci permet de faire passer l'état $(|x>|y>)$ à l'état $(|x>|f(x)+y>)$ (où + désigne le XOR sur un bit), après divers calculs que le lecteur pourra suivre sur la page **[WIKID] [PQC] [QAI] [IQC] [QCS]**, avec une seule lecture quantique, nous savons si $f(x)+f(y)=0$ auquel cas la fonction est constante, sinon la fonction est équilibrée.

L'algorithme de Simon résout le problème suivant : soit une fonction $f:\{0,1\}^n \rightarrow \{0,1\}^{n-1}$ telle qu'il existe un z non nul tel que $\forall x \neq y: f(x)=f(y) \Leftrightarrow x=y+z$ où + désigne encore le XOR sur un bit, le problème est de trouver ce z. Un algorithme classique devra faire dans l'ordre de $O(2^{(\frac{1}{2}-\epsilon)n})$ avec ϵ petit. Un calculateur quantique peut résoudre le problème en $O(n)$ étapes, c'est un gain considérable, on parle d'accélération exponentielle (*exponential speedup*). L'algorithme de Simon est important car il est à la base de presque tous les algorithmes intéressants comme celui de Shor sur la factorisation. Il résout en fait un problème de calcul de période, z est la période de la fonction $f(x)$

et l'algorithme de Shor, comme nous le verrons, calcule une telle période pour factoriser un entier. Ce sont, en simplifiant à l'extrême, des cas particuliers de ce qu'on appelle le problème du sous-groupe caché :

Problème du sous-groupe caché (hidden subgoup problem)

Soit G un groupe, X un ensemble fini et $f : G \to X$ une fonction qui « cache » un sous-groupe H de G, c'est-à-dire que $\forall g_1, g_2 \in G, f(g_1) = f(g_2) \Leftrightarrow g_1 H = H g_2$ pour les cosets de H. Si la fonction f est donnée sous forme d'oracle, déterminer le sous-groupe caché H via un ensemble générateur de H avec le minimum d'évaluations de la fonction f.

Ce problème, grâce à une généralisation de l'algorithme de Simon, peut se résoudre en un temps polynomial si le groupe G est abélien (ou commutatif), la version la plus simple du problème **[PQC] [QAI] [IQC] [QCS]** est bel et bien de trouver la période d'une fonction sur $\mathbb{Z}$ ou dans le cas de l'algorithme de Shor, trouver la période d'une fonction sur $\mathbb{Z}_N$. Tous ces algorithmes nécessitent et utilisent la transformation de Fourier quantique (QFT) qui, comme son nom l'indique, est la version quantique de la transformation de Fourier classique, qui est une transformation unitaire. Nous ne décrirons pas cette QFT mais elle est fondamentale dans l'algorithme de Shor, elle peut se réaliser grâce à la transformation de Walsh-Hadamard qui généralise la transformation de Hadamard *H* vue précédemment.

Sur le problème de Grover, la recherche d'un élément donné dans une base de N éléments, la performance d'un calculateur quantique est un peu pessimiste : on ne peut faire mieux que $\sqrt{N}$ étapes. En fait, disons le tout net : on ne sait même pas si un calculateur quantique pourrait un jour résoudre tout problème que le modeste ordinateur utilisé pour écrire cet article résout sans difficulté.

Imaginons qu'il existe un algorithme qui permette de résoudre un système d'équations $Ax = b$ où les composantes de *A* et de b sont entiers, avec un calculateur quantique à plusieurs centaines de qubits, cet algorithme pourrait permettre de résoudre des systèmes avec plus de 10^{30} inconnues, ce qui serait à la fois extraordinaire et néanmoins problématique : on ne peut pas stocker 10^{30} données !

Mais, si un tel algorithme n'existe pas (encore ?), il existe cependant un algorithme qui permet de calculer $x^t_0 M x_0$ où x_0 est une solution de $Ax = b$, x^t_0 le vecteur transposé de x_0 et *M* une matrice donnée. On récupère alors une seule donnée.

Alors, que peut-on vraiment faire avec un calculateur quantique ? Avec un ordinateur classique, la réponse est simple : tout algorithme qu'on peut coder peut être exécuté sur un ordinateur classique. Avec un calculateur quantique, la situation est totalement différente. Tout calcul sur un calculateur quantique (sauf la mesure) doit être réversible, et donc tout algorithme quantique doit être une succession d'opérations unitaires (via les portes logiques quantiques unitaires).

En fait, hors la recherche de périodes, il semble que le calculateur quantique possède plusieurs limitations, ainsi :

1. Il ne peut pas encore servir à résoudre le problème du voyageur du commerce (TSP) ou le problème de 3-satisfaisabilité (3-SAT).
2. Il ne peut casser une clé AES de 256 bits en moins de 2^{128} opérations par force brute via l'algorithme de Grover.
3. Il pourra difficilement (pas du tout ?) simuler numériquement des problèmes de mécanique des fluides ou de résistance des matériaux.

Évidemment, rien n'empêche non plus des avancées algorithmiques pour chacun de ces trois problèmes : une attaque algébrique quantique dévastatrice (et unitaire !) pour (2), par exemple.

5 L'algorithme de Shor

Lorsque Shor a en 1994 publié son algorithme de factorisation quantique, qui est clairement la star des algorithmes quantiques, cela a entraîné un extraordinaire engouement pour l'informatique quantique. Pour la requête *quantum computing*, Google donne plus de 9 millions de réponses. Voyons ce que donne cet algorithme sur un cas « simple » : n=17*19=323.

Soit a=2, si on considère les puissances de a modulo n : $\{a^i \bmod n, i \in [0, \ldots, n-1]\}$ cette suite est périodique, voir la figure 4.

Soit *r* la plus petite valeur de cette période, c'est l'ordre de a modulo n qui vaut ici 72, ce qui signifie que $a^i = a^{i+72} \bmod n$, ceci entraîne qu'on a $a^{72} = 1 \bmod n$, et donc que $PGCD(a^{r/2}+1, n) \neq 1$ et aussi que $PGCD(a^{r/2}-1, n) \neq 1$, ainsi comme ces deux PGCD sont différents de n, cela signifie qu'on a un facteur de n.

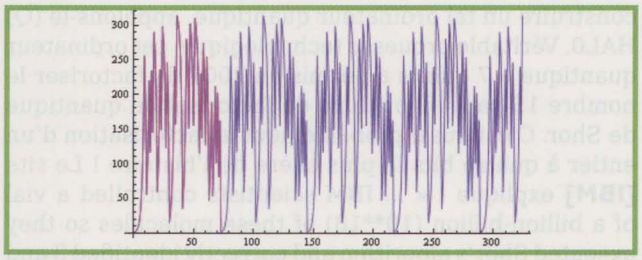


Figure 4 : les puissances successives de a^i mod n, la partie rouge montre la « période »

Algorithme de factorisation quantique de Shor

Entrée : un entier n supposé non premier

Sortie : un facteur, (avec une probabilité de 50%)

[1] prendre un nombre a au hasard tel que PGCD(a,n)=1 (auquel cas on a gagné)

[2] Utiliser l'algorithme quantique de la recherche de la période r de la fonction $f(x) = a^x \bmod n$

[3] Si r est pair : calculez $PGCD(a^{r/2}+1, n) \neq 1$ et/ou $PGCD(a^{r/2}-1, n) \neq 1$ avec l'algorithme d'Euclide étendu, sur un ordinateur classique.

[4] Si r est impair, recommencer l'étape [1].

C'est la version naïve de l'algorithme de Shor, voir **[WIKISH] [PQC] [QAI] [IQC] [QCS]** pour plus de détails, l'étape [4] signifie en fait un échec, l'algorithme de Shor sur un calculateur quantique est en fait un algorithme probabiliste. Mais sa probabilité de réussite est telle que quelques essais permettent de factoriser un nombre, si on a un calculateur avec suffisamment de qubits pour le nombre n. La partie [3] se réalise grâce à la transformée rapide quantique de Fourier (QFT), voir **[PQC] [QAI] [IQC] [QCS]**.

On peut aussi faire la remarque que (Q)HAL0 était un calculateur quantique à 7 qubits qui a servi à factoriser un nombre (n=15) à 4 qubits. À quoi servent ces qubits surnuméraires ? Il est nécessaire d'utiliser des codes quantiques correcteurs d'erreurs pour lutter contre le bruit quantique. On ne sait pas aujourd'hui avec précision quelle taille de nombre un calculateur quantique à 1024 qubits pourrait permettre de réellement factoriser, mais on sait d'ores et déjà que ce ne sera pas un nombre de 1024 bits.

6 Distribution quantique de clés

Le grand succès de la cryptographie quantique est la distribution quantique de clés (*quantum key distribution* ou QKD). Si Alice désire calculer une clé commune avec Bernard, clé qui pourra par exemple servir comme clé de chiffrement symétrique secrète, elle peut se fournir auprès de deux entreprises commerciales (ID Quantique en Suisse et MagiQ technologies aux US). Ces systèmes, qui ne sont pas très bon marché, garantissent le transport de clés sur des fibres optiques, et ce, de manière a priori sécurisée, jusqu'à plusieurs dizaines de kilomètres.

Néanmoins, il y a quelques petits soucis à prendre en compte. D'abord, Alice doit évidemment savoir si elle parle bien à Bernard. Et oui, si vous avez suivi, vous avez deviné, fidèle lecteur de *MISC*, la bonne vieille attaque de *Man in the Middle*, l'attaque de l'homme du milieu, peut très bien fonctionner ici, s'il n'y a pas authentification. Vous voyez l'autre piège ? Dans le cas d'une utilisation, disons avec 1000 personnes différentes, Alice doit envoyer ou recevoir ou calculer avec chacune de ces 1000 personnes une clé d'authentification. Sur un « autre canal » sécurisé ! Ces systèmes sont donc pour l'instant cantonnés à des utilisations très spécifiques (centralisation de résultats d'élections, centre de commandement nucléaire tactique, etc.).

La sécurité de RSA repose sur une hypothèse mathématique non prouvée : il est calculatoirement difficile de factoriser un module RSA « bien choisi », on peut faire la remarque que la cryptographie quantique dans sa version distribution quantique de clés repose sur une hypothèse, physique cette fois-ci, non prouvée elle aussi : il est impossible d'effectuer des mesures sur un système quantique à l'insu du « propriétaire » dudit système sans que celui-ci ne s'en aperçoive.

Le premier protocole d'échange/distribution quantique de clés a été le protocole BB84 **[QAI] [IQC] [QCS]**, proposé en 1984 par Bradley et Brassard, deux chercheurs de Montréal. Leur idée repose sur l'utilisation de photons, dont nous avons vu que c'est un système quantique simple mais intéressant. La sécurité repose sur l'idée que lorsqu'un photon est envoyé par Alice à Bernard, si Eve essaye de le lire, l'état quantique du photon va changer et Bernard peut s'en rendre compte. Or, en 2010, des chercheurs norvégiens **[QC]** ont montré qu'ils étaient capables, sur deux systèmes commerciaux quantiques, de retrouver 100% de la clé secrète, car les « émetteurs » photoniques envoient en fait plusieurs photons qui hélas, ne sont pas intriqués. Ce qui est très dommage, car si ces photons étaient intriqués, l'attaque serait inopérante ! D'autres protocoles ont depuis été inventés **[QAI] [IQC] [QCS]**, qui utilisent aussi des photons ; E91 utilise des photons intriqués et on ne lui connaît aucune faiblesse. En conclusion : même si la théorie quantique est totalement « vraie », il reste le problème de réaliser un système quantique physiquement parfait. Dans le cas du protocole BB84, cela revient à dire : il faut envoyer un seul et unique photon à la fois. Et cela semble coûter cher alors que l'attaque décrite dans **[QC]** a été faite avec du matériel peu cher.

7 Cryptographie post-quantique

Alors, si un calculateur quantique fonctionnel existe en 2020, par exemple, il en sera fini de RSA et de El Gamal et de tous ces algorithmes qui ont fait la joie des étudiants dans les cours de cryptographie du monde entier ? Soit. Mais un calculateur quantique à 256 qubits ne pourra pas casser une clé AES de 256 bits classiques en moins de 2^{128} étapes, tant qu'il n'y aura pas eu d'avancée algorithmique, ce qui laisse encore une bonne marge de sécurité. De même, certains algorithmes anciens comme celui de McEliece semblent résister à toute attaque par un algorithme quantique.

Les actes **[PQC]** du premier colloque sur la cryptographie post-quantique sont à ce titre très intéressants. On y trouve en résumé :

- La cryptographie à base de fonctions de hachage (*hash-based cryptography*) ; les arbres de hachage de Merkle, qui datent de 1978, en font partie.
- La cryptographie multivariable **[CM]** : « *Il s'agit d'étudier comment l'on peut utiliser les problèmes qui apparaissent autour de la résolution des systèmes d'équations en plusieurs variables sur les corps finis pour faire de la cryptographie.* ».
- La cryptographie à base de code (de type code correcteur d'erreurs) : le système de chiffrement de McEliece (à base de code de Goppa) est un excellent exemple.

- La cryptographie à partir de réseaux euclidiens (*lattice based cryptography*) : l'exemple peut-être le plus connu est le système NTRU.

  En fait, on se rend compte que RSA et les autres algorithmes comme El Gamal ont un peu « tué » certains algorithmes dès lors que ceux-ci avaient soit une faiblesse avérée, soit un petit défaut :

- Les arbres de hachage de Merkle datent de 1978, et sont très utilisés dans les réseaux pair-à-pair (voir **[MHT]** : « *Les arbres de Merkle sont aussi utilisés dans le système de fichiers ZFS, dans le protocole Google Wave ou dans le système de monnaie virtuelle bitcoin* »).

- Le système de chiffrement de McEliece : inventé en 1978, est un algorithme asymétrique mais sa clé est très longue (les clés publiques et privées étant de très grandes matrices, il faut plusieurs dizaines de Ko voire plus pour les stocker) et il double la taille du message chiffré ; mais il a un avantage : le déchiffrement peut être très rapide et on lui connaît très peu de faiblesses dès lors que la clé est assez grande et bien choisie.

- Le système NTRU : élégant, à base de réseaux euclidiens symplectiques (pour simplifier), autant dans sa version chiffrement NTRUEncrypt que dans sa version signature NTRUSign, pour ceux qui veulent en savoir plus : lire la thèse de N. Gama **[TG]** ; certaines variantes ont été cassées, même si aucune attaque n'a été dévastatrice sur les dernières variantes.

- Le système de cryptographie multivariable : leur sécurité repose sur le statut NP-complet du problème de résolution d'équations polynomiales multivariables de nombreux algorithmes, *Hidden Field Equations* (HFE), Sflash et autres ; le premier algorithme connu, Matsumoto-Imai publié en 1995, a été cassé par J. Patarin qui a lui-même proposé de nombreux algorithmes ; thème prometteur.

Ainsi, la relève semble assurée, et si certains algorithmes sont aujourd'hui délaissés, quelquefois à juste titre ou pas, comme celui de McEliece, nul doute que le jour où vous pourrez acheter un calculateur quantique comme on achète *MISC*, ces algorithmes seront bel et bien utilisés, même s'ils doublent la taille du message, même si la clé publique fait 1Mo ou 100 Mo, même si etc. Car il n'y aura pas d'autre choix en fait.

Concluons avec quelques problèmes ouverts

Exécuter un code arbitraire sur un calculateur quantique semble clairement hors de portée. Un ordinateur quantique est d'abord un calculateur quantique qui peut résoudre certains problèmes, en nombre limité, mais de manière très rapide. Ainsi, une conclusion un peu pessimiste s'impose : le calcul quantique permet un parallélisme massif mais limité à certains types de problèmes.

Tout le monde peut constater en regardant son chat (ou celui du voisin, ou son chien, ou un éléphant) qu'au niveau macroscopique, la matière ne se comporte pas du tout de manière quantique, le phénomène qui pourrait expliquer cela est la décohérence quantique **[HEI]**. Il est tout à fait possible que la décohérence quantique puisse empêcher le bon fonctionnement d'un (Q)HAL à 2048 qubits.

Qui dit cryptographie quantique dit aussi cryptanalyse quantique. Même si celle-ci est débutante, elle pourra se faire avec un calculateur quantique. Et savoir si l'informatique (et l'algorithmique) quantique va tuer la cryptographie actuelle (vision pessimiste) sans permettre à une nouvelle cryptographie d'émerger, cryptographie résistante face aux attaques des calculateurs quantiques, reste LA question qu'il va falloir étudier. Nul doute qu'on devrait lire des choses passionnantes dans les années qui viennent. (Même si la lecture de cet article semble montrer un certain scepticisme de l'auteur quant au futur des calculateurs quantiques, force est de constater que l'informatique quantique a ouvert des pistes fantastiques, et c'est déjà énorme). Il se pourrait même un jour qu'on puisse résoudre ou tout au moins mieux comprendre la fameuse conjecture $P = NP$? Rappelons que même si l'algorithme de Shor est bel et bien une prouesse algorithmique, on ne sait toujours pas si factoriser une clé RSA est un problème NP-complet ou pas, et finalement, malgré Shor, Simon et Grover, on ne sait toujours pas ce que donneront les calculateurs quantiques sur les difficiles problèmes NP-complets.

Posons un problème ouvert qui illustre bien la situation actuelle : imaginons disposer d'un (au moins) ordinateur quantique pouvant casser une clé RSA de 1024 bits, pourrait-on s'en servir pour factoriser « assez » rapidement une clé RSA de 2048 bits ? Éventuellement en terminant le calcul sur un ordinateur classique ? Rien n'est moins sûr (l'auteur pense même que non) mais :

- Si c'est possible, cela pose une autre question intéressante : si avec un ordinateur quantique pouvant casser une clé RSA de N bits, on peut casser une clé RSA de 2N bits, alors... il suffirait de pratiquer la méthode « Diviser Pour Régner » jusqu'à l'utilisation d'un (Q)UAL à 16 qubits pour... on laisse au lecteur le soin de continuer.
- Si ce n'est pas possible, alors pourquoi, et surtout, peut-on le prouver ? Si oui, alors, lorsque sortira un calculateur quantique capable de casser une clé RSA de 1024 bits, on passera toutes les clés à 2048, 4086 bits ou plus et on gagnera encore quelques années.

Aux lecteurs non spécialistes de la mécanique ou de l'informatique quantique, qui ont l'impression de n'avoir rien compris à cet article, l'auteur (qui se demande lui-même...) rappelle qu'en mécanique quantique, on peut très bien refaire plusieurs fois la même expérience et avoir des résultats différents, alors relisez, vous verrez bien ! Aux lecteurs spécialistes de la mécanique ou de

l'informatique quantique, l'auteur demande un peu d'indulgence (quantique ou non) sur les simplifications (quelquefois rudes) faites ici, et attend d'eux toutes les remarques qu'ils voudront bien lui adresser sur ses erreurs.

L'informatique quantique a clairement rappelé aux informaticiens qu'un calcul est un processus physique, comme le dit A. Eckert **[HEI]** : « *a computation is a physical process. It may be performed by a piece of electronics [...] and as such it is subject to the laws of physics* »... et le plus surprenant, c'est que cela a et aura un impact sur toute l'algorithmique, quantique ou pas. ■

■ REMERCIEMENTS

L'auteur remercie le très courageux relecteur anonyme, qui a donné des conseils et pointé de nombreuses erreurs, et deux de ses étudiants, J.-P. Roliers et S. Leens pour leur relecture attentive. Tout ce qui reste est dû à l'auteur. Les figures 2, 3 et 4 sont obtenues à partir d'une modification d'un code écrit par R. Muradian [MUR].

■ RÉFÉRENCES

- [IBM] http://www-03.ibm.com/press/us/en/pressrelease/965.wss
- [BEBE] http://photonics.anu.edu.au/qoptics/Research/Resources/HeisenbergDog.pdf
- [HAR] Robert Harley, 5/12/01, Sci.crypt
- [REY] J.-F. Rey, Calculabilité, complexité et approximation, Vuibert.
- [DEUTSCH] http://www.ceid.upatras.gr/tech_news/papers/quantum_theory.pdf
- [NIEL] J. Nielsen, Les règles du monde quantique, Dossier Pour la Science, Juillet-Septembre 2010
- [WIKICQ] http://fr.wikipedia.org/wiki/Ordinateur_quantique
- [LEB] M. Le Bellac, Introduction à l'information quantique, Éditions Belin, 2005
- [KHT] http://fr.wikipedia.org/wiki/Arbre_de_Merkle
- [TG] N. Gama, thèse disponible à http://gama.nicolas.free.fr/these.pdf
- [QWIKI] http://www.quantiki.org/
- [HEI] D. Heiss, Fundamentals of quantum computing, Springer
- [PQC] D.J. Bernstein, J. Buchmann, E. Dahmen, Post-Quantum Cryptography, Springer
- [QAI] S. Stenholm, K.-A. Suominen, Quantum Approach to Informatics, Wiley
- [IQC] P. Kaye, R. Laflamme, M. Mosca, An introduction to quantum computing, Oxford Press
- [QCS] N. S. Yanofsky, M. A. Manucci, Quantum computing for computer scientists, Cambridge Press
- [WIKIIQ] http://fr.wikipedia.org/wiki/Informatique_quantique
- [WIKIUNI] http://fr.wikipedia.org/wiki/Matrice_unitaire
- [WIKID] http://fr.wikipedia.org/wiki/Algorithme_de_Deutsch-Jozsa
- [WIKISH] http://fr.wikipedia.org/wiki/Algorithme_de_Shor
- [QC] http://www.nature.com/nphoton/journal/v4/n10/abs/nphoton.2010.214.html
- [CM] http://www.prism.uvsq.fr/index.php?id=113
- [MUR] R. Muradian, « Quantum Fourier Transform Circuit », http://demonstrations.wolfram.com/QuantumFourierTransformCircuit/

LE CHIFFREMENT HOMOMORPHE

OU COMMENT EFFECTUER DES TRAITEMENTS SUR DES DONNÉES CHIFFRÉES

Carlos AGUILAR MELCHOR - carlos.aguilar@xlim.fr et
Marc RYBOWICZ - marc.rybowicz@xlim.fr
Laboratoire XLIM, UMR CNRS 7252 - Université de Limoges

mots-clés : CRYPTOGRAPHIE / PROTECTION DE LA VIE PRIVÉE / CLOUD COMPUTING

Et si vos données n'étaient jamais déchiffrées, même en mémoire, même quand vos applications ou le système d'exploitation les utilise ? Le chiffrement dit « complètement homomorphe » permettrait de réaliser de tels exploits. Seul hic, la construction d'un tel système de chiffrement est restée pendant trente ans un problème ouvert, jusqu'à récemment où des avancées spectaculaires ont eu lieu...

1 Le Graal de la cryptographie : une conjecture de 1978

Alice n'a qu'une confiance modérée en son fournisseur de service de courriel. Elle voudrait que les messages de ses interlocuteurs arrivent chiffrés chez son hébergeur, et y restent stockés chiffrés, sans que jamais celui-ci ne puisse avoir accès aux clairs. Mais dans ce cas, elle doit renoncer à de nombreux services : comment demander au serveur de sélectionner les messages contenant un certain mot-clé ou correspondant à un ensemble de critères plus complexes, sans que le serveur ne déchiffre le contenu de la boîte aux lettres ? Comment le serveur va-t-il appliquer des filtres automatiques à l'arrivée d'un courrier : traitement du spam, ajout automatique d'événements au calendrier, tri automatique dans des dossiers, etc. ?

Ce problème est bien sûr générique et peut être décliné en de nombreuses variantes. Dans un contexte plus professionnel, la tendance est à l'externalisation du stockage et du traitement des données de l'entreprise, que ce soit dans le « cloud » ou ailleurs. Mais même si les données sont transmises et stockées chiffrées, le prestataire, et parfois toute la chaîne de prestataires, a nécessairement accès aux données en clair au moment des traitements puisque les processeurs n'opèrent que sur des données en clair.

L'idéal serait donc de disposer d'un système de chiffrement tel que les algorithmes de traitement des données puissent « passer à travers la couche de chiffrement ». Ainsi, le client transmettrait ses données chiffrées au prestataire, ce dernier appliquerait le traitement sans jamais déchiffrer les données, puis renverrait les résultats chiffrés au client. Un tel système de chiffrement est qualifié d'homomorphe, voire de complètement homomorphe s'il y a peu de limitations sur la nature des traitements capables de traverser la couche de chiffrement.

Dès 1978, c'est-à-dire dès le tout début de l'histoire de la cryptographie à clé publique et du déploiement massif de la cryptographie dans le domaine civil, Rivest, Adleman et Dertouzos, trois chercheurs de premier plan **(1)**, conjecturent l'existence d'un système de chiffrement complètement homomorphe **[RAD78]**. Ce type de système ouvrirait la voie à de nombreuses applications et la communauté cryptographique mondiale déploie pendant 30 ans une intense activité pour en construire un. Mais ce n'est que dans les années 2008/2009, notamment grâce aux travaux de Craig Gentry **[Gen09]** alors chercheur à IBM et doctorant à l'Université de Stanford, qu'une rupture scientifique permet enfin de répondre positivement à la conjecture RAD et de construire un système permettant d'envisager une industrialisation. Depuis, les progrès ont été nombreux et rapides, et le chiffrement homomorphe reste l'un des sujets « chauds » des grandes conférences de cryptologie.

Dans cet article, version rédigée de l'exposé donné par le premier auteur aux journées des 25 ans du master Cryptis de l'Université de Limoges **(2)**, nous tenterons d'expliquer les grands principes du chiffrement homomorphe, de relater quelques moments décisifs de cette recherche du Graal de la cryptographie moderne, et de faire le point sur les perspectives.

2 Principes du chiffrement homomorphe

Le chiffrement homomorphe se place dans le modèle suivant : les données à chiffrer ainsi que les chiffrés de ces données sont des entiers, les traitements que l'on souhaite effectuer à travers le chiffrement sont des additions et des multiplications, et les calculs s'effectuent modulo **(3)** un entier p. En d'autres termes, chaque traitement possible s'écrit sous la forme d'un polynôme :

$$f(X_1, X_2, \ldots, X_v) = \sum a_{e_1,\ldots,e_v} X_1^{e_1} X_2^{e_2} \ldots X_v^{e_v} \bmod p.$$

Est-ce que cela veut dire que la classe des traitements considérés est extrêmement limitée et que les algorithmes les plus génériques **(4)** ne peuvent pas traverser la couche de chiffrement ? La réponse est « ça dépend ». En effet, on peut montrer que la classe des algorithmes qui peuvent s'exprimer par un polynôme comme ci-dessus est très vaste. L'objectif de cet article n'est pas de présenter au lecteur la théorie des modèles de calcul ni de prouver quoi que ce soit à ce sujet, mais nous pouvons énoncer quelques points qui aideront le lecteur à comprendre la portée des résultats obtenus sur le chiffrement homomorphe.

Il est tout d'abord important de comprendre que certains algorithmes ne peuvent pas être représentés par des polynômes. Ceci dit, un algorithme dont la taille des entrées est bornée peut être décrit par un polynôme, en passant par une représentation intermédiaire sous forme de circuit booléen **(5)** **[Gol01]**. Ceci fait donc le lien entre les algorithmes et les polynômes **(6)**, et a pour conséquence que **si l'on veut exécuter un algorithme sur des données chiffrées, il suffit de pouvoir transformer l'algorithme en un polynôme et d'utiliser un chiffrement homomorphe permettant d'évaluer ce polynôme.**

Ainsi, un système de chiffrement tel que tout polynôme peut traverser la couche de chiffrement permet en théorie d'exécuter une très large classe d'algorithmes sur des données chiffrées : c'est la notion de chiffrement complètement homomorphe.

Néanmoins, l'application directe des principes ci-dessus pour exprimer un algorithme sous une forme polynomiale est inutilisable dans de nombreux cas car trop coûteuse : le nombre de variables ainsi que le degré du polynôme sont généralement énormes et le coût de la transformation est rédhibitoire. Fort heureusement, beaucoup d'algorithmes peuvent se représenter comme polynômes de façon beaucoup plus naturelle qu'en passant par un circuit booléen. De plus, le chiffrement complètement homomorphe n'est pas le seul cas intéressant, loin de là : même des systèmes dont la couche de chiffrement ne peut être traversée que par des polynômes très spécifiques peuvent permettre d'exécuter des algorithmes très intéressants sur des données chiffrées (c.f. Section 3.2).

Quoi qu'il en soit, un important travail d'ingénierie devra encore être effectué pour exploiter tout le potentiel de ces systèmes de chiffrement.

AUTOUR DE L'ARTICLE...

▪ SÉCURITÉ DES SYSTÈMES DE CHIFFREMENT

Paramètre de sécurité : entier généralement noté k que l'utilisateur d'un système de chiffrement fixe lors de la génération des clés. Il faut que toute attaque pouvant briser les propriétés de sécurité du système ait un coût d'au moins 2^k opérations. De nos jours, on considère que k=100 est un choix sûr. En effet, il y a environ 2^{30} ordinateurs dans le monde pouvant réaliser environ 2^{30} opérations par seconde, et donc un attaquant utilisant tous ces ordinateurs en parallèle aurait besoin de 2^{40} secondes, c'est-à-dire 31000 années pour compromettre la sécurité.

IND-CPA : l'indistinguabilité contre des attaques à textes clairs choisis, traduction de l'anglais *indistinguishability against chosen plaintext attacks*. C'est une propriété de sécurité standard en cryptographie pour un système de chiffrement randomisé (voir autre encadré). L'idée est que, si un attaquant choisit un couple de messages m_1 et m_2 en clair, et si on lui fournit un chiffré α de m_1 ou de m_2, alors l'attaquant est dans l'incapacité de deviner, avec une probabilité significativement supérieure à ½, si α est le chiffré de m_1 ou de m_2. En d'autres termes, l'analyse du chiffré n'apporte quasiment aucune information : elle ne permet pas à l'attaquant, pour distinguer un chiffré de m_1 d'un chiffré de m_2, de faire significativement mieux qu'en choisissant au hasard. Cette propriété donne des garanties fortes sur la confidentialité des données chiffrées, comme prouvé par Goldwasser et Micali **[GM 84]**.

▪ ALGORITHMES ET CHIFFREMENTS RANDOMISÉS

Algorithmes randomisés/déterministes : On dit qu'un algorithme *Algo* est randomisé quand pour une entrée donnée il peut donner lieu à plusieurs sorties en fonction de choix aléatoires réalisés pendant son exécution. Si pour une entrée E, l'algorithme Algo donne lieu à une sortie S, on représentera cet événement par S ← Algo(E). Avec un tel algorithme, on peut très bien avoir deux exécutions sur une même entrée donnant lieu à deux sorties différentes, c'est d'ailleurs ce qui arrivera avec une probabilité extrêmement forte dans le cas des algorithmes associés aux systèmes de chiffrement. Par opposition, un algorithme déterministe est un algorithme ne faisant pas de choix aléatoire pendant son exécution et ayant donc une unique sortie pour chaque entrée. Pour un tel algorithme, Algo2 donnant lieu pour une entrée E à une sortie S, on note S=Algo2(E).

Chiffrement randomisé : chiffrement caractérisé par trois algorithmes et deux espaces (M espace des claris, C espace des chiffrés) définis implicitement par ces fonctions :

- KeyGen, algorithme randomisé prenant en entrée le paramètre de sécurité (voir autre encadré) et donnant en sortie une bi-clé (pk, sk), où pk et sk désignent respectivement une clé publique et la clé secrète associée. On note (pk, sk) ← KeyGen(k).
- Enc, algorithme de chiffrement randomisé, prenant en entrée une clé publique pk et un message m appartenant à M et renvoyant un chiffré α appartenant à C choisi aléatoirement parmi les nombreux chiffrés possibles du message. On note α ← Enc(pk, m).
- Dec, algorithme de déchiffrement déterministe, tel que pour tout m de M et tout α ← Enc(pk, m), on ait Dec(sk, α) = m.

2.1 Définition

Un système de chiffrement homomorphe est avant tout un système de chiffrement randomisé (voir encadré) et est donc défini par un espace de messages en clair M, un espace de messages chiffrés C et les trois algorithmes KeyGen, Enc, et Dec, associés à un tel système. Le lecteur comprendra facilement l'importance de la randomisation, notamment si on manipule des bits chiffrés avec un système à clé publique. En effet si les 0 et les 1 étaient toujours chiffrés de la même façon, l'attaquant n'aurait qu'à chiffrer ces deux messages pour être capable de déchiffrer par comparaison.

Mais un système de chiffrement homomorphe est aussi caractérisé par deux éléments supplémentaires :

- F, la classe des fonctions calculables, formée des polynômes qui peuvent « traverser » la couche de chiffrement ;
- Eval, l'algorithme déterministe (voir encadré) d'évaluation d'un polynôme sur des données chiffrées.

Dans la suite, $\alpha_1, \ldots, \alpha_v$ désignent toujours les chiffrés des clairs $m_1, \ldots, m_v$ par une clé publique pk. Si f est un polynôme de F à v variables, alors Eval(pk, $(\alpha_1, \ldots, \alpha_v)$, f) doit être un chiffré de $f(m_1, \ldots, m_v)$. Ainsi, on doit avoir la relation (voir figure 1) :

$$Dec(sk, Eval(pk, (\alpha_1, \ldots, \alpha_v), f)) = f(m_1, \ldots, m_v).$$

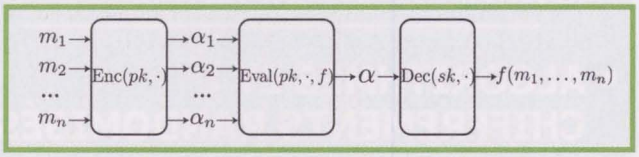


Figure 1 : Évaluation d'une fonction f sur des données chiffrées. Généralement, on considère que c'est un utilisateur qui réalise les étapes de chiffrement et déchiffrement et un prestataire qui réalise l'évaluation de la fonction sur les données chiffrées.

2.2 Critères de performances et sécurité

Les performances d'un système de chiffrement homomorphe s'évaluent selon les critères suivants :

- Le nombre maximal de multiplications et d'additions que l'on peut effectuer, qui se traduit par des restrictions sur le degré, le nombre de variables, et la taille des coefficients des polynômes de F. Dans le cas du chiffrement complètement homomorphe, il n'y a pas de restrictions sur ces quantités.
- Le coût des calculs, en temps et en espace, pour évaluer Enc, Eval, et Dec.
- Le facteur d'expansion. En effet, il peut être nécessaire d'utiliser des chiffrés de taille plus importante que la taille des clairs, le rapport entre ces tailles, dit facteur d'expansion, pouvant dépendre du nombre d'opérations arithmétiques que l'on souhaite effectuer « à travers » le chiffrement.

Un « bon » système de chiffrement homomorphe permet d'atteindre un niveau de sécurité IND-CPA (voir encadré), avec un facteur d'expansion raisonnable, des temps de calculs acceptables, peu de limitations sur les polynômes de F, et ce, pour des tailles de clés utilisables en pratique. C'est ce défi qui a résisté pendant plus de 30 ans à la communauté des cryptologues et qui semble maintenant à portée de main.

Note

Un chiffrement homomorphe est, par nature, malléable, c'est-à-dire qu'on peut construire de nouveaux chiffrés valides à partir d'autres chiffrés. Cette propriété est perçue comme une source potentielle de vulnérabilités pour les cryptosystèmes d'usage traditionnel. Par exemple, l'« Optimal Asymetric Encryption Padding » (OAEP), introduite par Bellare et Rogaway **[BR95]** et intégrée au standard PKCS1 protège RSA contre la malléabilité, comme montré par Fujisaki, Okamoto, Pointcheval et Stern **[FOPS01]**. Ainsi, quand on utilisera un cryptosystème homomorphe, il faudra faire attention à l'intégrité des messages (puisqu'un attaquant peut modifier les clairs) ainsi qu'à éviter les attaques dites par « oracle de déchiffrement » (i.e. il ne faut pas qu'un utilisateur diffuse des informations sur des clairs correspondant à des chiffrés manipulés par un attaquant).

3 1978-2008 : les premiers pas

Dans la foulée de l'invention des premiers cryptosystèmes asymétriques, fondés sur l'arithmétique modulo un grand entier (RSA **[RSA78]**, Diffie-Hellman **[DH76]**), il était naturel de rechercher des systèmes homomorphes fondés sur la théorie des nombres. D'ailleurs, le cryptosystème RSA, dans sa version initiale, possède nativement des propriétés homomorphes, mais ne permet de réaliser que des multiplications. Cet exemple, peu utile en pratique, a l'avantage de la simplicité ; il sera détaillé ci-dessous pour illustrer les concepts. D'autres systèmes permettent de ne réaliser que des sommes. Malgré cette limitation, ils se sont révélés utiles, l'application phare étant la construction de protocoles de Retrait d'Informations Privé **(7)**, ou protocoles RIP. On en illustrera les principes. Enfin, on dira quelques mots de certaines tentatives infructueuses pour créer un système de chiffrement complètement homomorphe.

3.1 Le chiffrement homomorphe multiplicatif

Probablement, le cas le plus simple de chiffrement homomorphe **(8)** est RSA, quand il est utilisé dans sa version déterministe. Rappelons qu'une clé publique RSA est un couple pk=(e,N), e étant un petit entier et N grand nombre difficile à factoriser. Un message m est

simplement un entier inférieur à N-1, et son chiffré est l'entier m^e mod N, c'est-à-dire le reste de la division de m^e par N. Avec quelques connaissances en arithmétique élémentaire, on voit facilement que :

$$(m_1^e \bmod N)(m_2^e \bmod N) \bmod N = (m_1 m_2)^e \bmod N,$$

d'où l'on déduit :

$$Dec(sk, \alpha_1 \alpha_2 \bmod N) = m_1 m_2 \bmod N.$$

En considérant que l'espace des clairs M et l'espace des chiffrés C sont formés des entiers modulo N et que l'opération de multiplication sur les clairs et les chiffrés est la multiplication modulo N, on voit que la classe des fonctions calculables contient le polynôme $f(X_1, X_2) = X_1 X_2$, et que Eval(pk, (α_1, α_2), $X_1 X_2$) est simplement la réduction modulo N du produit $\alpha_1 \alpha_2$.

Un autre point important à comprendre est que généralement, on peut utiliser la propriété homomorphe itérativement. Ainsi, la formule :

$$((m_1 m_2)^e \bmod N)(m_3^e \bmod N) \bmod N = (m_1 m_2 m_3)^e \bmod N$$

montre que l'on peut effectuer deux multiplications. En généralisant, on voit facilement que la classe F contient tous les monômes de la forme :

$$f(X_1, X_2, \ldots, X_v) = X_1^{e_1} X_2^{e_2} \ldots X_v^{e_v}$$

La fonction Eval consiste tout simplement à calculer $f(\alpha_1,\ldots,\alpha_v)$ et à réduire le résultat modulo N, ce qui donnera bien un chiffré de $f(m_1,\ldots,m_v)$.

On a ainsi construit un système dit de chiffrement homomorphe multiplicatif, c'est-à-dire permettant de calculer des produits sur les données. Les fonctions Enc, Dec, et Eval peuvent être implémentées relativement efficacement, mais il n'est pas randomisé, donc pas IND-CPA, et n'est pas utilisé en pratique.

Un autre exemple de système homomorphe multiplicatif est El Gamal **[EIG85]**, qui a lui l'avantage d'être randomisé.

3.2 Le chiffrement homomorphe additif

De façon analogue, il existe des cryptosystèmes permettant de calculer des sommes, dits systèmes de chiffrement homomorphe additif. Le cryptosystème de Pailler **[Pai99]** est sans doute le représentant le plus connu de cette famille. Sans entrer dans les détails mathématiques, il s'agit d'un cryptosystème randomisé pour lequel les clairs sont des entiers entre 0 et N-1, les chiffrés sont des entiers entre 0 et N^2-1 (facteur d'expansion 2), et qui vérifie la propriété :

$$Dec(sk, \alpha_1 \alpha_2 \bmod N^2) = m_1 + m_2 \bmod N.$$

En d'autres termes, le produit des chiffrés modulo N^2 correspond à l'addition des clairs modulo N. En remarquant que la multiplication d'un clair par un entier positif b_i correspond à ajouter le clair (b_i -1) fois à lui-même et en généralisant la somme ci-dessus en une somme à v termes, on déduit que les fonctions calculables sont de la forme :

$$f(X_1, X_2, \ldots, X_v) = b_1 X_1 + b_2 X_2 + \ldots + b_v X_v. \quad (*)$$

On obtient un système de chiffrement homomorphe additif, IND-CPA, avec un facteur d'expansion acceptable et des performances satisfaisantes pour Enc, Dec et Eval. Un autre exemple de système additif est celui de Goldwasser et Micali **[GM 84]**.

Une application intéressante de cette classe de cryptosystèmes est la construction de protocoles de retrait d'informations privé (protocoles RIP). En général, pour accéder à un élément d'une base de données, un utilisateur envoie une requête indiquant quel élément l'intéresse, puis la base renvoie l'élément en question. Quel élément de la base intéresse un utilisateur peut être une information que celui-ci souhaite garder confidentielle, même vis-à-vis de l'administrateur de la base. Par exemple, on peut imaginer qu'un important investisseur abonné à un service d'informations financières puisse ne pas vouloir révéler quelles sociétés l'intéressent. Ou qu'un particulier adhérent à une bibliothèque électronique souhaite conserver la confidentialité sur la nature de ses centres d'intérêt.

On considère un modèle particulièrement simple de base de données, l'adaptation de situations réelles à ce modèle relevant d'un travail d'ingénierie hors du champ de cet article. La base contient des éléments $b_1, b_2, \ldots, b_v$ qui sont des entiers, et l'utilisateur identifie l'élément qui l'intéresse par son indice. Voici un exemple de protocole RIP fondé sur le chiffrement homomorphe additif **[Ste98]**, dans un scénario où Alice souhaite obtenir l'information d'indice j de la base de données contrôlée par Bob (voir la figure 2, page suivante) :

- Alice calcule des chiffrés $\alpha_1,\ldots,\alpha_v$ de sorte que α_j soit un chiffré de 1 et que α_i soit un chiffré de 0 pour tout i différent de j. On remarque que les chiffrés de 0 sont deux à deux distincts **(9)** puisque le système est randomisé. De plus, ils sont indiscernables du chiffré de 1, puisque le système est IND-CPA.
- Alice transmet $\alpha_1,\ldots,\alpha_v$ à Bob, qui ne peut savoir quel chiffré est celui de 1.
- Bob calcule Eval(pk, $(\alpha_1,\ldots,\alpha_v)$, f), où f est la fonction linéaire donnée par la formule (*) ci-dessus et formée à partir des éléments b_i de la base de données. Il obtient donc un chiffré c_j de f(0,0,0,1,0...0), le 1 se trouvant à la position j, c'est-à-dire un chiffré de b_j.
- Bob envoie ce chiffré c_j à Alice.
- Alice calcule Dec(sk, c_j)=b_j et obtient bien l'élément cherché, sans avoir révélé à Bob duquel il s'agit.

La bibliothèque de cryptographie avancée de l'Université du Texas met à la disposition du public une implémentation GPL d'un protocole RIP (plus précisément d'un protocole contenant un protocole RIP) utilisant le cryptosystème de Paillier **[PSS09]**. Une autre implémentation de ce protocole a été réalisée dans **[ACGJR08]** en utilisant

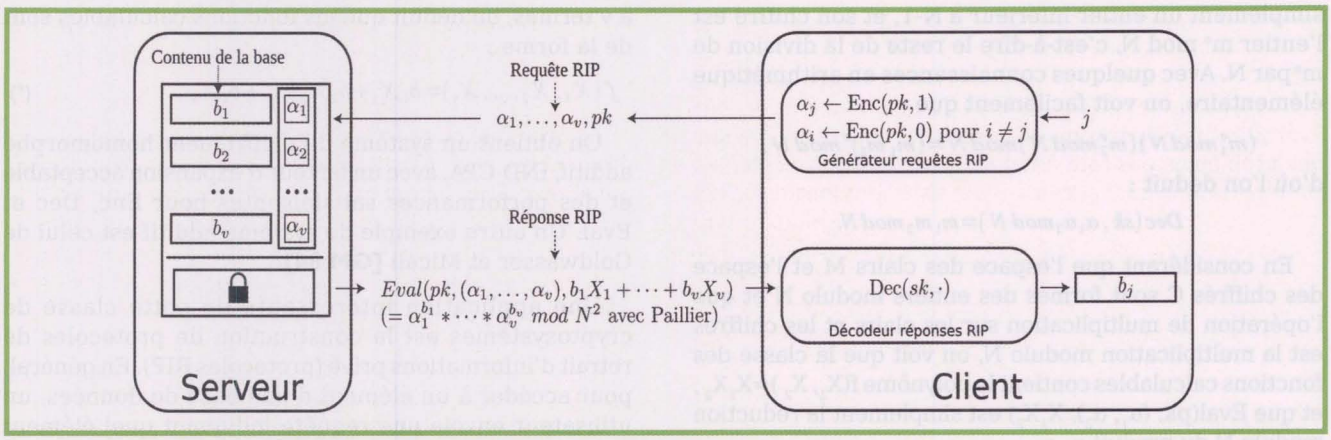


Figure 2 : Un protocole de Retrait d'Informations Privé basé sur le chiffrement homomorphe additif

un cryptosystème construit sur les réseaux euclidiens (voir section 4) à la place de celui de Pailler ainsi que des traitements sur GPU pour augmenter les performances au niveau calculatoire.

Les protocoles RIP ont bien sûr des inconvénients : à l'étape 1, le coût du calcul est proportionnel au nombre d'éléments de la base de données et il en est de même du coût de la transmission à l'étape 2. Quant à l'étape 3, le coût calculatoire dépend de la taille de la base, en incluant donc la taille des b_i. Mais des améliorations significatives sont possibles. Les protocoles RIP, qu'ils reposent sur le chiffrement homomorphe ou sur d'autres paradigmes, ont fait l'objet de nombreuses recherches ces quinze dernières années. Nous n'en dirons pas plus dans cet article et renvoyons le lecteur intéressé au site maintenu par Helger Lipmaa **[Lip--]**.

3.3 Vers un chiffrement complètement homomorphe

Les premières propositions de systèmes complètement homomorphes commencèrent il y a presque vingt ans : en 1994 Fellow et Koblitz présentèrent Polly Craker **[FK94]** ; en 1996 et 2002 Domingo-Ferrer proposa deux systèmes **[DF96] [DF02]** ; et en 2006 Grigoriev et Ponomarenko proposèrent un autre système **[GP06]**. Tous ces systèmes furent rapidement cassés (voir **[AGH10]** pour une liste de références).

Le premier pas dans la direction du chiffrement complètement homomorphe fut fait en 2005 par Boneh, Goh et Nissim **[BGN05]** qui proposèrent le premier système permettant d'évaluer de façon concise des polynômes de second degré tant que l'évaluation du polynôme est un petit nombre. Malheureusement, l'approche proposée par ces trois auteurs est difficile à généraliser à des polynômes de degré supérieur, et bien qu'ils introduisent des idées très intéressantes, celles-ci ne peuvent pas développer leur plein potentiel avec les systèmes de chiffrement basés sur des problèmes classiques de la théorie des nombres.

4 2008 : la rupture

4.1 Les réseaux euclidiens et les premiers résultats

Indépendamment des premiers travaux sur le chiffrement homomorphe, une branche de l'informatique théorique, la théorie de la complexité des réseaux euclidiens **(10)**, entre avec force dans le monde de la cryptographie dans les années 90 à cause des révolutionnaires preuves de sécurité qu'elle apporte : les réductions dites « pire-cas/cas-moyen » **[Ajt96]**. Ces derniers sont de plus résistants aux attaques par ordinateurs quantiques **(11)**, contrairement aux cryptosystèmes dont la sécurité repose sur la difficulté de factoriser de grands entiers ou de calculer des logarithmes discrets dans les corps finis **[Sho94]**. Le lien entre la complexité des réseaux euclidiens, la cryptographie fondée sur les réseaux euclidiens, et celui du chiffrement homomorphe est évident depuis très tôt, mais les propriétés homomorphes que l'on peut obtenir semblent à première vue beaucoup plus faibles que celles déjà atteintes par les systèmes de chiffrement classiques et cette connexion n'est que peu étudiée par la communauté.

Ce n'est qu'à partir de 2008 que les systèmes de chiffrement basés sur les réseaux donnent lieu à des avancées fondamentales vers le chiffrement complètement homomorphe. Cette année, un groupe de chercheurs de l'université de Limoges et de l'Université Polytechnique de Catalogne (Espagne) propose une approche **[AGH08] [AGH10]** permettant d'évaluer sur des données chiffrées des fonctions avec un nombre théoriquement arbitraire de sommes et de produits. La sécurité du système repose sur des problèmes complètement standards **(12)** en cryptographie, mais a un coût qui grandit rapidement avec le nombre de multiplications, ce qui le rend utilisable en pratique que pour des fonctions ayant peu de produits : on parlera dans ce cas de chiffrement presque complètement homomorphe. Cette même année, Craig Gentry, d'IBM Watson (New York), présente au séminaire de Dagstuhl une approche complètement différente permettant également d'évaluer des fonctions avec un nombre arbitraire de

sommes et produits **[Gen09]**. Cette approche a un coût qui est uniquement linéaire en le nombre de sommes et produits et n'est donc pas restreinte aux fonctions avec peu de produits. Malheureusement, le système a un coût exorbitant par opération et sa sécurité est construite sur des problèmes pas tout à fait standards. Malgré cela, les problèmes ont quand même l'air suffisamment difficiles aux yeux de la communauté et, tout au moins du point de vue théorique, on a là le premier système de chiffrement complètement homomorphe. Ce résultat a suscité un très grand émoi dans la communauté des cryptologues, et même au-delà, puisqu'il figure dans la liste des 10 avancées scientifiques principales de l'année 2009 établie par le magazine *La Recherche*, toutes disciplines confondues. L'information a aussi été largement reprise par la presse généraliste internationale (Forbes, New York Times, etc.). Cette percée théorique a valu à Craig Gentry deux prix de la prestigieuse *Association for Computing Machinery*.

4.2 Évolution des performances

Très vite, de nombreux résultats font progresser l'approche de Gentry (voir **[BGV12]** pour un historique) de façon à la rendre moins coûteuse et à éviter les liaisons avec des problèmes peu standards. Enfin, en 2012, Brakerski et al. **[BGV12]** présentent un protocole à partir de l'approche proposée dans **[AGH10]** qui, tout en gardant la solidité des hypothèses de sécurité présentes dans cet article, atteint la généralité des résultats de Gentry en utilisant plusieurs nouvelles techniques d'une grande subtilité. Au vu des présentations à la conférence CRYPTO 2011, des implémentations pratiques sont en cours, mais n'ont pas été publiées pour le moment.

La seule implémentation disponible d'un système complètement homomorphe correspond aux propositions de 2008-2009 mais n'est qu'une preuve de concept, la clé publique étant de plusieurs gigabits et le calcul d'une seule multiplication sur des données chiffrées nécessitant 31 minutes sur un serveur IBM xserver 3500 avec 24 Go de RAM ! En ce qui concerne le chiffrement presque complètement homomorphe, Naehrig et al. ont proposé une implémentation **[NLV11]** permettant de réaliser le produit d'une quinzaine d'éléments ainsi qu'un très grand nombre de sommes avec des chiffrés d'environ un méga-octet et un coût par produit de quelques secondes. Ces implémentations sont très vite largement dépassées par l'avalanche de résultats théoriques permettant d'améliorer fortement les performances. Ceci explique que la communauté attende une stabilisation du domaine avant de proposer des nouvelles implémentations.

En effet, la progression est fulgurante. Chaque année les coûts ont été réduits au logarithme ou à la racine carrée de ceux de l'année précédente, comme le montre le tableau ci-dessous. Ce tableau est extrêmement simplifié ; son objet est de faire ressortir les ordres de grandeur des coûts et leur évolution, en concentrant des informations (approximativement) correctes. Pour ce qui est du Chiffrement Presque Complètement Homomorphe (CPCH), chaque case de la ligne indique à la fois :

- l'ordre de grandeur du nombre d'opérations binaires à réaliser pour évaluer une somme ou un produit ;
- l'ordre de grandeur de la taille des clés et des chiffrés.

Le principal facteur de croissance pour ces valeurs est d, le nombre maximal d'éléments qu'on souhaite multiplier entre eux dans les algorithmes. En ce qui concerne le Chiffrement Complètement Homomorphe (CCH), les coûts sont par définition indépendants du nombre de produits à réaliser, et dépendent principalement du paramètre de sécurité k (voir encadré). Dans les protocoles CCH, le coût majeur est celui du calcul d'une multiplication sur des données chiffrées, valeur qui est donnée dans ce tableau. Cette valeur donne également un ordre d'idée de la taille de la clé publique du système et des chiffrés.

	2008	2009	2010	2011	2012
CPCH	k^{2d}	$k^6 * d^5$	$k^4 * d^3$	$k^2 * d^3$	$k^2 * \log^3(d)$
CCH	k^{14}	k^{14}	k^7	k^4	??

On insiste sur le fait que ces informations ne servent qu'à donner une idée approximative de l'évolution des coûts et tailles. Elles ne sauraient se substituer à une analyse détaillée des différents protocoles.

5 Et maintenant ?

Au niveau des protocoles complètement homomorphes, de nouveaux résultats semblent indiquer que les coûts vont encore fortement diminuer, mais il est encore un peu tôt pour faire une estimation de ce que donnera la prochaine implémentation. En ce qui concerne les systèmes de chiffrement presque complètement homomorphe, d'après l'implémentation de Naehrig et al. **[NLV11]** des résultats de 2011 (coût en $k^2 * d^3$), il est raisonnable de penser qu'une implémentation fondée sur les résultats de 2012 (en $k^2 * \log^3(d)$) devrait permettre d'obtenir, pour des coûts raisonnables, des systèmes de chiffrement presque complètement homomorphe pouvant évaluer des polynômes de degré jusqu'à 2^{15}=32768, au lieu de seulement 15 actuellement. Avec de tels polynômes, on peut commencer à envisager l'exécution d'algorithmes très complexes sur des données chiffrées ; reste à savoir si une « killer application » émergera parmi ceux-ci.

L'utilisation du chiffrement homomorphe demandera de toutes façons le développement d'une ingénierie complexe qui lui sera propre. On peut s'attendre à ce que les résultats dans ce domaine soient rapidement brevetés. Gentry et d'autres membres de l'équipe d'IBM Watson ont déjà déposé des demandes en ce sens pour des résultats théoriques **[Gen11] [HGV11] [GH12]**. Leur implémentation du chiffrement complètement homomorphe a été gardée jalousement secrète alors qu'il s'agissait seulement d'un prototype très loin d'être utilisable en pratique. Il en sera sans doute de même pour les implémentations futures. Du côté de l'implémentation de Naehrig et al., les auteurs sont membres de Microsoft Research.

L'intérêt de l'industrie pour ces avancées scientifiques est donc bien là. Fort heureusement, de nombreux résultats sont diffusés dans la communauté scientifique sans brevets associés. On ne peut qu'espérer que des alternatives libres voient le jour, malgré le poids des concurrents industriels et la difficulté de la tâche. ■

■ NOTES

(1) Les deux premiers étant co-inventeurs du célèbre cryptosystème RSA [RSA78].

(2) http://www.cryptis.fr

(3) C'est-à-dire qu'à chaque addition ou multiplication, on remplace le résultat par le reste de sa division par p.

(4) C'est-à-dire correspondant à une machine de Turing arbitraire.

(5) À titre d'exemple, un algorithme prenant en entrée un octet et en sortie un autre octet peut être représentée par un polynôme de degré 255.

(6) Les coefficients des polynômes correspondant aux circuits booléens ne sont pas a priori des entiers naturels, mais un encodage sous forme d'entiers naturels est toujours possible, ce qui permet de se ramener au modèle indiqué.

(7) Attention à l'accord, c'est bien le retrait qui est privé, pas les informations.

(8) En relâchant la contrainte, donnée dans la section précédente, que le chiffrement soit randomisé.

(9) Avec une très forte probabilité, car l'ensemble des chiffrés d'un message doit être de grande taille.

(10) Dans ce contexte cryptographique, un réseau euclidien est l'ensemble des combinaisons linéaires à coefficients entiers d'un nombre fini de vecteurs de base, eux-mêmes à coefficients entiers. La sécurité des algorithmes cryptographiques associés est souvent basée sur la difficulté de trouver dans le réseau un vecteur de petite longueur. Les réseaux euclidiens sont appelés « lattices » en anglais.

(11) Qui restent théoriques tant que l'on n'aura pas réussi à construire d'ordinateurs quantiques assez puissants.

(12) On entend par « problème standard » un problème dont la difficulté est reconnue par la communauté cryptologique et sur lequel on peut faire reposer la sécurité d'un cryptosystème asymétrique. Exemple : la factorisation des grands entiers.

■ BIBLIOGRAPHIE

- [ACGJR08] Carlos Aguilar Melchor, Benoit Crespin, Philippe Gaborit, Vincent Jolivet, Pierre Rousseau, High-Speed Private Information Retrieval Computation on GPU, in SECUREWARE 2008, pages 263-272
- [AGH08] Carlos Aguilar, Philippe Gaborit, and Javier Herranz, Additively homomorphic encryption with d-operand multiplications, Cryptology ePrint Archive, Report 2008/378, 2008. http://eprint.iacr.org/, dernière version dans [AGH10]
- [AGH10] Carlos Aguilar, Philippe Gaborit, and Javier Herranz. Additively homomorphic encryption with d-operand multiplications, in CRYPTO'10, volume 6223 of Lecture Notes in Computer Science, pages 138–154, Springer, 2010
- [Ajt96] Miklos Ajtai, Generating hard instances of lattice problems (extended abstract), in Proceedings of the twenty-eighth annual ACM symposium on Theory of computing (STOC '96), ACM, New York, NY, USA, pages 99–108, 1996
- [BGN05] Dan Boneh, Eu-Jin Goh, and Kobbi Nissim, Evaluating 2-DNF formulas on ciphertexts, in Theory of Cryptography Conference, TCC'2005, volume 3378 of Lecture Notes in Computer Science, pages 325–341, Springer, 2005.
- [BR95] M. Bellare, P. Rogaway, Optimal Asymmetric Encryption - How to encrypt with RSA, Extended abstract in Advances in Cryptology - Eurocrypt '94 Proceedings, Lecture Notes in Computer Science Vol. 950, A. De Santis ed, Springer-Verlag, 1995
- [BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan, (Leveled) fully homomorphic encryption without bootstrapping, in Proceedings of the 3rd Innovations in Theoretical Computer Science Conference (ITCS '12), ACM, New York, NY, USA, 309–325, 2012
- [DF96] Josep Domingo-Ferrer. A new privacy homomorphism and applications, Information Processing Letters, 60(5) :277–282, 1996
- [DF02] Josep Domingo-Ferrer, A provably secure additive and multiplicative privacy homomorphism, in Agnes Hui Chan and Virgil D. Gligor, editors, Information Security, 5th International Conference, ISC 2002 Sao Paulo, Brazil, September 30 - October 2, 2002, Proceedings, volume 2433 of Lecture Notes in Computer Science, pages 471–483, Springer, 2002
- [DH76] W. Diffie and M. E. Hellman, New Directions in Cryptography, IEEE Transactions on Information Theory, vol. IT-22, Nov. 1976, pp: 644–654
- [ElG85] Taher ElGamal, A public key cryptosystem and a signature scheme based on discrete logarithms, IEEE Transactions on Information Theory, 31(4) :469–472, 1985
- [FK94] Michael Fellows and Neal Koblitz. Combinatorial cryptosystems galore!, in Finite fields : theory, applications, and algorithms (Las Vegas, NV, 1993), volume 168 of Contemp. Math., pages 51–61, Amer. Math. Soc., 1994
- [FOPS01] E. Fujisaki, T. Okamoto, D. Pointcheval, and J. Stern, RSA-OAEP is secure under the RSA assumption, in J. Kilian, ed., Advances in Cryptology-CRYPTO 2001, vol. 2139 of Lecture Notes in Computer Science, Springr-Verlag, 2001
- [Gen09] Craig B. Gentry, Fully homomorphic encryption using ideal lattices, in Proceedings of STOC'09, pages 169–178, ACM Press, 2009
- [Gen11] Craig B. Gentry, Fully homomorphic encryption method based on a bootstrappable encryption scheme, computer program and apparatus, United States Patent Application 20110110525
- [GH12] Craig B. Genry, Efficient Implementation Of Fully Homomorphic Encryption. United States Patent Application 20120039473
- [GM84] Shafi Goldwasser and Silvio Micali, Probabilistic encryption, Journal of Computer and System Sciences, 28(2) :270–299, 1984
- [Gol01] Oded Goldreich, Foundations of Cryptography, Cambridge University Press. 2001
- [GP06] Dima Grigoriev and Ilia Ponomarenko, Homomorphic public-key cryptosystems and encrypting boolean circuits, Applicable Algebra in Engineering, Communication and Computing 17(3), 239-255, (2006), 17 :239–255, 2006
- [HGV11] Shai Halevi, Craig B. Gentry, Vinod Vaikuntanathan, Efficient Homomorphic Encryption Scheme For Bilinear Forms, United States Patent Application 20110243320
- [Lip--] Helger Lipmaa, http://www.cs.ut.ee/~lipmaa/crypto/link/protocols/oblivious.php
- [NLV11] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan, Can homomorphic encryption be practical?, in Proceedings of the 3rd ACM workshop on Cloud computing security workshop (CCSW '11), ACM, New York, NY, USA, 113–124, 2011
- [Pai99] Pascal Paillier, Public-key cryptosystems based on composite degree residuosity classes. In 18th Annual Eurocrypt Conference (EUROCRYPT'99), Prague, Czech Republic, volume 1592 of Lecture Notes in Computer Science, pages 223–238, Springer, 1999
- [RAD78] Ronald L. Rivest, Leonard Adleman, and Michael L. Dertouzos, On Data Banks and Privacy Homomorphisms, chapter On Data Banks and Privacy Homomorphisms, pages 169–180, Academic Press, 1978
- [RSA78] R. Rivest, A. Shamir, and L. Adleman, A method for obtaining digital signatures and public key cryptosystems, Communications of the ACM, 21(2) :120–126, 1978
- [Sho94] P. W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring, Proc. 35nd Annual Symposium on Foundations of Computer Science (Shafi Goldwasser, ed.), IEEE Computer Society Press (1994), 124-134
- [Ste98] Julien P. Stern, A New Efficient All-Or-Nothing Disclosure of Secrets Protocol, in 13th Annual International Conference on the Theory and Application of Cryptology & Information Security (ASIACRYPT'98), Beijing, China, volume 1514 of Lecture Notes in Computer Science, pages 357–371, Springer, 1998

d'attaques exploite les variations de temps de traitement d'un programme. Elle se rapproche un peu du principe de fonctionnement d'un canal caché de type timing channel du fait que l'information est extraite en mesurant un temps d'exécution, au détail près que ce canal est établi de manière fortuite, sans entité réceptrice.

Dans son article, Kocher illustre cette vulnérabilité avec l'opération d'exponentiation modulaire $R = Y^X \bmod N$, en utilisant le programme suivant :

```
Exponentiation_modulaire (Y, X, n)

$1
resultat = 1
Tant que "X" > 0
        $11
        Si le bit de poid faible de "X" == 1 alors
                $111
                resultat = (resultat * Y) mod (n)
                111$
        decalage à droite de "X"
        Y = Y^2 mod (n)
        11$
retourner resultat
1$
```

Figure 1 : Algorithme Square & Multiply

On voit bien sur cet exemple que le traitement est différent en fonction de la valeur du bit de X_i. Dans la cas X_i = "1", il faut effectuer une multiplication supplémentaire. On comprend bien que cette implémentation « naïve » de l'algorithme « Square & Multiply » va fuir car en fonction de la valeur du bit X_i, l'algorithme effectue des opérations supplémentaires qui vont consommer du temps et de l'énergie. La figure 2 illustre le déroulement temporel de l'algorithme pour deux valeurs différentes de l'exposant secret X.

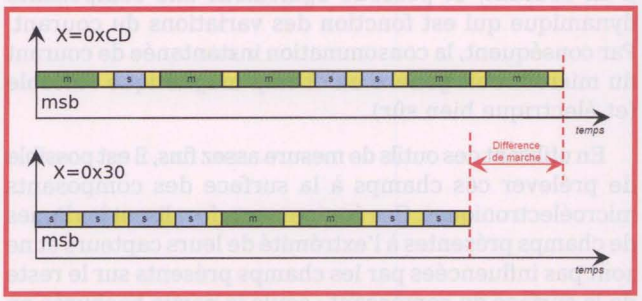


Figure 2 : Différence de marche entre deux calculs d'exponentiation modulaire

On dit que cette implémentation est déséquilibrée. La contre-mesure est de rendre identique le temps de passage dans chaque branche de l'algorithme.

La différence de marche illustrée dans la figure précédente n'est qu'une seule source de canal auxiliaire. En effet, en choisissant des valeurs particulières pour Y, on peut également jouer sur le temps d'exécution de la multiplication modulaire (branche $X_i = 1$) et faire fuir un peu plus cet algorithme.

2.2 La consommation électrique globale instantanée (Power Analysis)

De la même manière qu'il y a des pics de consommations électriques en hiver sur le réseau EDF, un microcircuit a une consommation variable en fonction du travail qu'il doit fournir pour traiter ses données. Ce phénomène est dû à la variation du nombre de transistors qui commutent.

Pour mieux comprendre ce problème de commutation, rappelons le fonctionnement de la brique élémentaire de tout processeur, à savoir l'inverseur CMOS (*Complementary Metal Oxide Semiconductor*) avec ses deux transistors à effet de champ (MOSFET) P et N (d'où le nom de *complementary*).

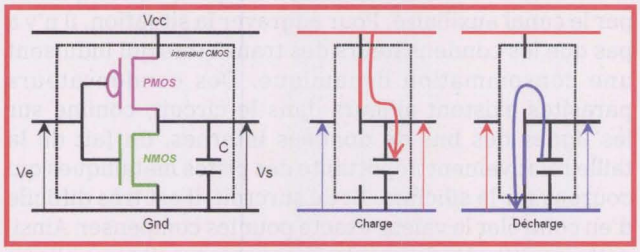


Figure 3 : Origine de la consommation des circuits CMOS

Pour rappel, les transistors se comportent comme des vannes ou des interrupteurs. Dans notre schéma (cf. figure 3), lorsque V_e passe à l'état logique « 0 », le transistor NMOS s'ouvre, le PMOS se ferme et il charge le condensateur C (cf. fig. 3 « charge »), ce qui a pour effet de faire basculer la sortie Vs à l'état logique « 1 ». Par la suite, lorsque la tension d'entrée V_e est à l'état logique 1, le transistor NMOS se ferme et le PMOS s'ouvre, ce qui ramène à la masse les deux bornes du condensateur C et induit le phénomène de décharge (cf. figure 3 « décharge »). La sortie du circuit V_s passe donc à l'état logique « 0 ». Nous avons donc bien un inverseur !

À l'état stationnaire « 0 » ou « 1 », lorsqu'il n'y a pas de variation de la tension d'entrée V_e, la consommation du circuit, appelée consommation statique, est très faible : les transistors CMOS ne consomment presque pas de courant et le condensateur C fuit très peu.

En contrepartie, pendant la phase de commutation (0 à 1 ou 1 à 0), le condensateur consomme du courant, soit en se vidant dans la masse (Gnd) via le transistor NMOS, soit en se chargeant, via le transistor PMOS. Ce phénomène est connu sous le nom de consommation dynamique. Il apparaît à chaque changement d'état du circuit, à savoir lors d'un coup d'horloge interne.

À l'aide d'un oscilloscope suffisamment rapide, on peut observer ces fluctuations de consommation de courant instantané. Ces variations se mesurent directement sur le circuit d'alimentation du composant, soit de manière détournée à l'aide d'une petite résistance de *shunt* par mesure de la variation de tension (ou différence de potentiels - ddp), soit directement par effet *Hall*, à l'aide d'une mini sonde en courant (cf. figure 4, page suivante).

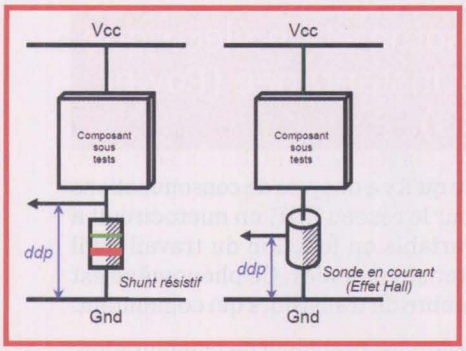


Figure 4 : Principe de la mesure de consommation en courant

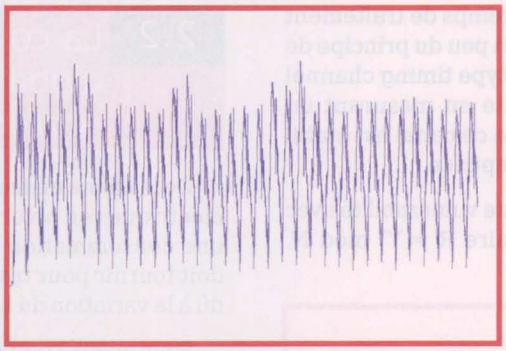


Figure 5 : Une implémentation matérielle d'un DES s'exécutant en quelques coups d'horloge

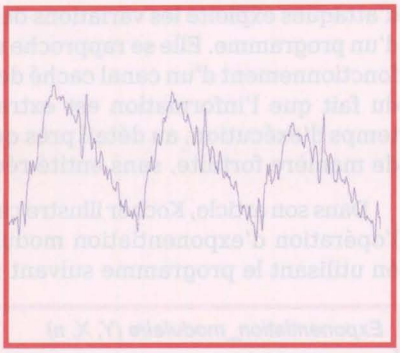


Figure 6 : Zoom sur les pics 5, 6 et 7

C'est cette consommation dynamique qui est exploitée par le canal auxiliaire. Pour aggraver la situation, il n'y a pas que les condensateurs des transistors qui induisent une consommation dynamique. Des condensateurs parasites existent ailleurs dans le circuit, comme sur les lignes des bus de données internes, du fait de la taille relativement importante des pistes métalliques qui courent sur le silicium. Et de surcroît, il est très difficile d'en contrôler la valeur exacte pour les compenser. Ainsi, cette multitude de condensateurs parasites explique que deux circuits routés différemment n'ont pas les mêmes signatures électriques puisque leurs bus ou leurs technologies de fonderie varient entre deux modèles de puces différents.

À titre d'illustration de la consommation dynamique d'un circuit, nous utilisons les courbes fournies dans le cadre du *challenge DpaContest* organisé par l'ENST ParisTech en 2008 (fig. 5 et 6).

Ces relevés sont des courbes de type SPA (*Simple Power Analysis*) représentatives du fonctionnement d'un algorithme DES câblé. On y aperçoit des pics de tailles variables, dont certains présentent des surconsommations de courant (notamment le pic 5, 13 et 21), ce qui pourrait porter à penser que le « gros » du calcul des 16 rondes s'effectue entre le 5ème et le 21ème pic.

Un zoom sur les premiers pics de forte consommation ne nous apporte apparemment pas d'information complémentaire en analyse directe.

Mesurées aux bornes d'un shunt mis en dérivation du circuit d'alimentation, ce type de relevé ne reflète malheureusement que l'activité globale du composant. Or, lors de la manipulation d'une donnée rouge, tous les transistors qui commutent ne sont pas corrélés avec le secret.

Afin de focaliser les mesures sur l'activité des transistors en corrélation avec la donnée secrète, il est intéressant de pouvoir faire abstraction des consommations parasites générées par la plus grande partie des transistors et se concentrer sur ceux utilisés par le cryptoprocesseur par exemple. Cette loupe existe, ce sont les fuites électromagnétiques.

2.3 Le rayonnement électromagnétique local (Electromagnetic Analysis)

Si on en croit James Clark Maxwell, la circulation d'un courant dans un conducteur génère des champs électriques et magnétiques au voisinage de ce dernier. L'équation de Maxwell-Gauss nous éclaire même sur les relations entre champ électrique (E), champ magnétique (B) et circulation de courant (j) (cf. figure 7).

$$rot\vec{B} = \mu_0\vec{j} + \mu_0\varepsilon_0\frac{\delta\vec{E}}{\delta t}, \quad \vec{j}_D = \varepsilon_0\frac{\delta\vec{E}}{\delta t}$$
$$rot\vec{B} = \mu_0(\vec{j} + \vec{j}_D)$$

Figure 7 : Équation de Maxwell-Gauss

Dans cette équation, on lit que le champ magnétique est issu d'une constante statique (la circulation permanente d'un courant) et possède également une composante dynamique qui est fonction des variations du courant. Par conséquent, la consommation instantanée de courant du microcircuit génère un champ magnétique variable (et électrique bien sûr).

En utilisant des outils de mesure assez fins, il est possible de prélever ces champs à la surface des composants microélectroniques. Ces équipements focalisent les lignes de champs présentes à l'extrémité de leurs capteurs et ne sont pas influencées par les champs présents sur le reste de la surface du composant : seule la partie hachurée en bleu dans la figure 8 sera à l'origine du champ capté par le solénoïde.

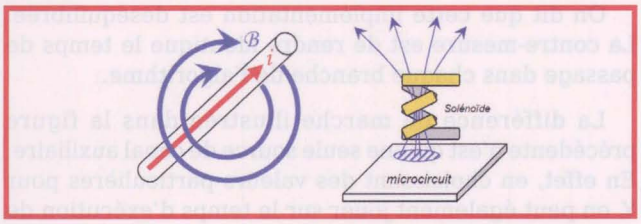
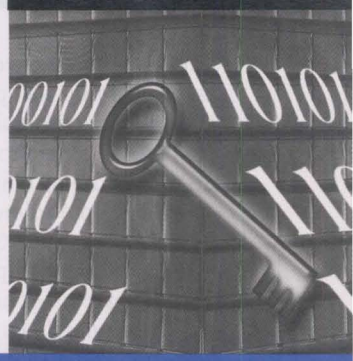


Figure 8 : Formation du champ magnétique et captation



CRYPTOGRAPHIE EN BOÎTE BLANCHE :
CACHER DES CLÉS DANS DU LOGICIEL

Brecht Wyseur - brecht.wyseur@nagra.com -
Security Architect & Cryptography Expert - NAGRA Kudelski group

mots-clés : CRYPTOGRAPHIE / PROTECTION LOGICIELLE / OBFUSCATION

Le défi que la cryptographie en boîte blanche vise à relever consiste à mettre en œuvre un algorithme cryptographique en logiciel de telle manière que les éléments cryptographiques restent sécurisés même en cas d'attaque en boîte blanche.

1 Introduction

L'objectif initial de la cryptographie a été de concevoir des algorithmes et des protocoles pour protéger un canal de communication contre l'espionnage. Cela a été la principale activité de la cryptographie moderne dans les 30 dernières années et a produit de nombreux algorithmes de chiffrement (comme AES, (T)DES, RSA, ECC) et de nombreux protocoles. Les éléments source et destination sont supposés sûrs (boîte noire) : l'attaquant a seulement accès aux entrées sorties de l'algorithme. Pour se conformer à un tel modèle, l'algorithme doit être exécuté dans un environnement sécurisé. L'activité de recherche dans ce domaine comprend des chiffrements améliorés et à usage spécifique (par exemple, le chiffrement par blocs légers, des schémas d'authentification, les algorithmes à clé publique homomorphiques), et leur cryptanalyse. Toutefois, du point de vue industriel, le problème principal est le déploiement pratique. Des protocoles déployés dans le mauvais contexte, des algorithmes mal mis en œuvre, ou des paramètres inappropriés peuvent offrir un point d'entrée pour les attaquants. Le déploiement de la cryptographie dans la pratique est devenu un enjeu en soi. Le problème empire lorsque le modèle d'attaque en boîte noire n'est plus satisfait. Au cours des dix dernières années, nous avons assisté à l'apparition d'un grand nombre de techniques de cryptanalyse nouvelles qui exploitent la présence d'informations additionnelles qui peuvent être observées lors de l'exécution d'un algorithme de chiffrement sur des canaux auxiliaires (*side channel attacks* en anglais) ; informations telles que le temps d'exécution, le rayonnement électromagnétique et la consommation d'énergie. Pallier ces attaques par canaux auxiliaires est un défi, car il est difficile de décorréler cette information des opérations sur des clés secrètes. De plus, la plateforme de déploiement impose souvent des contraintes de taille et de performance qui rendent difficile l'utilisation de techniques de protection.

Aujourd'hui, nous sommes confrontés à un nouveau modèle d'attaque, pire encore, car la cryptographie est déployée dans des applications qui sont exécutées sur des appareils ouverts tels que PC, tablettes ou *smartphones*, sans utiliser d'éléments sécurisés. Dans ce contexte, on parle d'attaque en boîte blanche. Ainsi, un attaquant en boîte blanche a un accès complet à l'implémentation logicielle d'un algorithme cryptographique : le binaire est complètement visible et modifiable par l'attaquant et celui-ci a le plein contrôle de la plateforme d'exécution (appels CPU, registres mémoire, etc.). Par conséquent, l'implémentation elle-même est la seule ligne de défense.

Les implémentations logicielles qui résistent à ces attaques en boîte blanche sont notées implémentations en boîte blanche (*white-box* en anglais). Elles sont la pierre angulaire d'applications où une clé cryptographique est utilisée pour protéger des biens, comme par exemple dans le domaine des DRM. Dans de tels cas, l'utilisateur du logiciel a une incitation à rétroconcevoir l'application et à en extraire la clé. Des attaques similaires peuvent se produire à l'encontre de l'intérêt de l'utilisateur, par exemple quand une application bancaire est exécutée sur un dispositif infecté par des logiciels malveillants ou sur un système multi-utilisateurs où certains utilisateurs ont des privilèges élevés. Dans de telles situations, lorsque les opérations cryptographiques sont implémentées au travers de bibliothèques cryptographiques standards comme OpenSSL ou que les clés cryptographiques sont stockées simplement dans la mémoire, la clé secrète sera découverte sans trop d'efforts, et ce quelle que soit la force de la primitive cryptographique utilisée.

Pour illustrer ces attaques en boîte blanche, nous présentons l'attaque *Key Whitening* de Kerins et Kursaw **[Kerins06]**. Cette attaque peut être déployée sur la plupart des implémentations de l'AES, y compris dans OpenSSL. On observe que l'AES utilise une opération de masquage par la clé du dernier tour comme étape finale de l'algorithme, afin de protéger l'itération finale de l'algorithme de chiffrement (voir Figure 1). L'avant-dernière opération

consiste en une consultation de table. Comme les spécifications de l'algorithme sont publiques, la table est connue et est accessible par un attaquant boîte blanche. En effet, avec un simple éditeur hexadécimal, cette table peut être située dans le binaire, et remplacée par des zéros. En conséquence, la sortie de l'avant-dernière opération sera zéro, et l'exécution de l'opération de masquage va simplement afficher la sous-clé finale, à partir de laquelle la clé d'origine AES peut être calculée.

```
000c83a0h: B3 00 00 00 7D 00 00 00 FA 00 00 00 EF 00 00 00
000c83b0h: C5 00 00 00 91 00 00 00 00 00 00 00 00 00 00 00
000c83c0h: 63 7C 77 7B F2 6B 6F C5 30 01 67 2B FE D7 AB 76
000c83d0h: CA 82 C9 7D FA 59 47 F0 AD D4 A2 AF 9C A4 72 C0
000c83e0h: B7 FD 93 26 36 3F F7 CC 34 A5 E5 F1 71 D8 31 15
000c83f0h: 04 C7 23 C3 18 96 05 9A 07 12 80 E2 EB 27 B2 75
000c8400h: 09 83 2C 1A 1B 6E 5A A0 52 3B D6 B3 29 E3 2F 84
000c8410h: 53 D1 00 ED 20 FC B1 5B 6A CB BE 39 4A 4C 58 CF
000c8420h: D0 EF AA FB 43 4D 33 85 45 F9 02 7F 50 3C 9F A8
000c8430h: 51 A3 40 8F 92 9D 38 F5 BC B6 DA 21 10 FF F3 D2
000c8440h: CD 0C 13 EC 5F 97 44 17 C4 A7 7E 3D 64 5D 19 73
000c8450h: 60 81 4F DC 22 2A 90 88 46 EE B8 14 DE 5E 0B DB
000c8460h: E0 32 3A 0A 49 06 24 5C C2 D3 AC 62 91 95 E4 79
000c8470h: E7 C8 37 6D 8D D5 4E A9 6C 56 F4 EA 65 7A AE 08
000c8480h: BA 78 25 2E 1C A6 B4 C6 E8 DD 74 1F 4B BD 8B 8A
000c8490h: 70 3E B5 66 48 03 F6 0E 61 35 57 B9 86 C1 1D 9E
```

Figure 1 : L'attaque key whitening

Cet article aborde le large spectre de la cryptographie en boîte blanche. Nous allons introduire les premières implémentations en boîte blanche qui ont été présentées et décrire les principaux résultats de cryptanalyse. Les questions existentielles qui ont été soulevées à la suite de ces cryptanalyses, comme les preuves d'existence et les limites, sont étudiées dans la branche théorique de la recherche sur la cryptographie en boîte blanche ; nous n'aborderons ce sujet que brièvement. Enfin, nous allons couvrir des aspects plus pratiques : comment les implémentations en boîte blanche sont attaquées dans la pratique, quels mécanismes de sécurité additionnels peuvent être déployés pour atténuer les problèmes pratiques, et quels sont les défis du futur.

2 La cryptographie en boîte blanche

Le principe de la cryptographie en boîte blanche a d'abord été publié par Chow *et al.* **[Chow02DES]** et s'intéressait au cas des implémentations à clés fixes du DES. Le défi consiste à coder en dur la clé symétrique dans l'implémentation du chiffrement DES : une implémentation implique une clé (on parle aussi d'algorithme boîte blanche statique, à la différence d'algorithme dynamique où la clé est paramétrable). L'idée principale est d'intégrer à la fois la clé fixe (sous la forme de données, mais aussi sous la forme de code) et des données aléatoires instanciées au moment de la compilation dans une composition dont il est alors difficile de déduire la clé d'origine. Un des principes fondamentaux de la cryptographie est le principe de Kerckhoffs, qui dit que la sécurité d'un système doit se fonder uniquement sur la confidentialité de la clé secrète, tous les autres aspects pouvant être publics. La cryptographie en boîte blanche aspire elle aussi à résister à une telle exposition publique : un attaquant a accès à l'implémentation, il connaît l'algorithme implémenté, ainsi que les techniques de protection en boîte blanche mises en œuvre ; la sécurité s'appuie sur la confidentialité de la clé secrète et des données aléatoires. Ceci est totalement différent de l'approche de la sécurité par obscurcissement, généralement obtenue par offuscation du code, qui vise à prévenir la rétro-ingénierie d'une application de façon que l'attaquant ne puisse pas trouver facilement un point d'attaque ou divulguer de la propriété intellectuelle.

2.1 Implémentations en boîte blanche

Les premières implémentations en boîte blanche ont été présentées par Chow *et al.* **[Chow02DE] [Chow02AES]** en 2002 sur le DES et l'AES. Leurs techniques en boîte blanche transforment un chiffrement en une série de tables dépendant de la clé (voir la figure 2 (a)). La clé secrète est codée en dur dans les tables et protégée par des techniques de randomisation. L'un des procédés employés est l'injection de codages aléatoires annihilants qui sont fusionnés avec les tables de façon que les tables et le flux de données soient aléatoires, tout en conservant la fonctionnalité sémantique globale de l'implémentation. Ceci est représenté dans la figure 2 (b), où F et G sont des codages aléatoires injectés entre A et B, puis B et C respectivement. La fonctionnalité globale (entrée A - sortie C) reste la même.

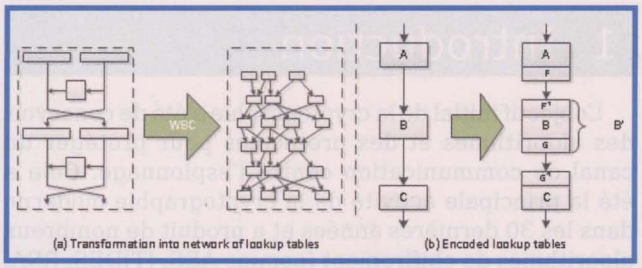


Figure 2 : Chow et al. implémentations en boîte blanche

Les techniques pour transformer un chiffrement par bloc en un réseau de tables peuvent être facilement étendues à d'autres chiffrements par réseaux de substitution-permutation (SPN). Très grossièrement, les étapes sont les suivantes :

- Réorganiser le chiffrement de sorte que les opérations boîte-S et l'opération exploitant la clé de tour soient contiguës, puis coder en dur la clé secrète dans la boîte-S, par exemple $T_k(x) = S(x) + K$.
- Injecter les opérations affines annihilantes dans la couche affine du chiffrement. De cette façon, les boîtes-S peuvent être ré-ordonnancées et l'opération affine (généralement conçue pour être aussi efficace que possible) peut être rendue moins creuse. (Ceci est référencé comme l'introduction de bijections de mélange dans les articles de Chow *et al* .
- Décomposer toutes les opérations affines en une série de tables, implémentant même l'opération XOR en une table.
- Injecter des codages aléatoires annihilants dans la séquence de tables.

Il s'agit d'une technique très *ad hoc* d'implémenter une primitive cryptographique et il est difficile d'en quantifier la sécurité. La seule affirmation de sécurité pouvant être faite concerne « la sécurité locale » : lorsque B est une bijection et F et G sont choisis aléatoirement, la fonction B' ne laisse pas fuir d'information. En effet, étant donné

B', toute bijection B avec un nombre équivalent de bits en entrée et sortie est un candidat car il y aura toujours une paire (F, G) telle que B' = GBF-1. Par conséquent, l'attaquant est obligé d'analyser également d'autres composants de l'implémentation, l'objectif étant que l'attaquant doive prendre trop de variables en compte, ce qui finalement rendrait son attaque impossible. Malheureusement, il n'existe pas de preuves de sécurité qui soutiennent cette déclaration ; dans la section suivante, nous montrerons même comment l'approche de Chow *et al.* peut être vaincue.

Comme des opérations très efficaces (telles que le XOR) sont transformées en tables, il y a un important surcoût en performance et en taille.

Implémentation	Taille	Référence
WB-AES **[Chow02AES]**	770 KBytes	14.5 KBytes (OpenSSL 1.0)
WB-DES **[Chow02DES]**	4.5 MBytes	8.25 KBytes (OpenSSL 1.0)
WB-DES **[Link05]**	2.3 MBytes	

2.2 Cryptanalyse en boîte blanche

L'implémentation en boîte blanche du DES **[Chow02DES]** a été la première prouvée non sûre. Sa vulnérabilité est due principalement à la structure de Feistel du DES qui peut être distinguée dans une représentation en table. Les premières techniques de cryptanalyse en boîte blanche trouvent leur origine dans des attaques par canaux auxiliaires telles que la corrélation de la propagation des fautes **[Jacob02]** ou les attaques deviner & déterminer (*guess and determine*) **[Lin05]**. Néanmoins, ces techniques font des hypothèses sur l'implémentation qui peuvent facilement être contournées **[Link05]**. Ce n'est qu'en 2007 que l'implémentation en boîte blanche du DES a été complètement cassée grâce à l'utilisation de la cryptanalyse différentielle tronquée par Wyseur *et al.* **[Wyseur07]** et Goubin *et al.* **[Goubin07]**.

L'implémentation en boîte blanche de l'AES a été cassée par Billet *et al.* **[Billet04]** en utilisant une technique de cryptanalyse algébrique. La technique algébrique présentée s'est montrée très puissante car elle cible directement la stratégie de randomisation des tables. Sans entrer trop dans les détails, l'idée de l'attaque est la suivante :

1. Isoler l'ensemble des tables qui représentent une itération (codée) de l'implémentation AES sous-jacente. Ceci est trivial, puisque les transformations employées sont connues (à clé secrète et aléa près). La figure 3 illustre la composition d'une telle itération, où P et Q sont des codages internes, Ti des boîtes-S avec les clés codées en dur, et M représente la partie linéaire de l'algorithme de chiffrement (comme l'opération MixColumn).
2. Définir les fonctions f_i comme des fonctions bijectives entre un octet à l'entrée et un octet à la sortie de l'itération ; les entrées vers les autres octets sont constantes mais différentes pour chaque f_i.
3. Étant donné que ces f_i sont bijectives et opèrent sur 8 bits, elles peuvent facilement être inversées et donc les fonctions composées $h_{ij} = f_j . f_{i-1}$ peuvent être calculées.
4. De l'ensemble des fonctions h_{ij}, la composante non linéaire de Q_0 peut être obtenue. Ceci est dû au fait que ces fonctions h_{ij} ne dépendent que de Q_0 et des constantes C_i et C_j ; elles ne dépendent pas de P_0 et $_0$.
5. L'information sur Q_0 conduit au P_0 de l'itération suivante (puisque ce sont des codages annihilants - inverses les uns des autres). Par conséquent, répéter les étapes 1 à 4 sur plusieurs itérations conduit à une implémentation de l'AES qui est seulement protégée par des codages linéaires. Il suffit alors de résoudre les équations restantes pour obtenir la clé secrète dissimulée.

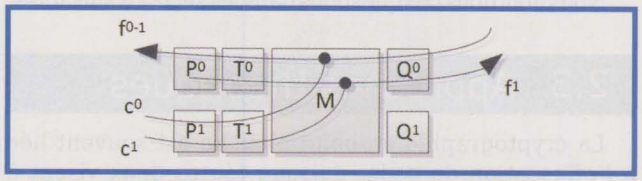


Figure 3 : Analyse d'une ronde de l'implémentation en boîte blanche d'AES

Michiels *et al.* **[Mich08]** ont démontré que cette technique de cryptanalyse algébrique peut être exploitée sur n'importe quel algorithme de chiffrement qui a des propriétés similaires à celles de l'AES (*i.e.* chiffrements SPN avec des matrices de code MDS). En fait, ils ont montré que ce sont précisément les propriétés qui ont été sélectionnées du point de vue de la sécurité boîte noire qui rendent le chiffrement vulnérable aux attaques en boîte blanche. Dans **[Wyseur09]**, nous avons montré comment les attaques algébriques peuvent être utilisées contre toute la stratégie d'implémentation en boîte blanche basée sur les tables : toute table avec perte (ce qui est inévitable dans les implémentations en boîte blanche à base de tables) laisse fuir de l'information qui peut être exploitée.

Les techniques de cryptanalyse algébrique ont conduit à la conception de nouvelles constructions dépassant la stratégie des tables, vers des implémentations basées sur des équations algébriques randomisées **[Bil03]** et l'introduction de perturbations pour casser la structure algébrique qui permet de monter des attaques algébriques **[Bri06wbc]**. L'implémentation de Billet et Gilbert **[Bil03]** a été cassée en raison d'un résultat de cryptanalyse sur la primitive sous-jacente qui a été utilisée. Des tentatives de résolution du problème ont été introduites par Ding **[Ding04]** en incluant des fonctions de perturbation pour détruire la structure algébrique. Cela a conduit à un schéma amélioré de chiffrement traçable **[Bri06trace]** et a finalement été appliqué à des implémentations en boîte blanche d'AES **[Bri06wbc]**. Néanmoins, même ces nouvelles constructions améliorées ont été démontrées non sûres par De Mulder *et al.* **[Dem10]**

En dépit de toutes les nouvelles constructions qui ont été présentées, la sécurité de la cryptographie en boîte blanche est très floue. La quasi-totalité des constructions présentées ont été prouvées non sûres dans des articles de cryptanalyse publiés par des universitaires. Seules quelques constructions restent « non cassées », mais

ce ne sont que des variantes mineures de constructions existantes - donc les attaques existantes pourront être appliquées avec peu d'efforts.

Considérant l'état de l'art actuel de la cryptographie en boîte blanche dans le domaine académique, les questions suivantes demeurent :

- Existe-t-il des implémentations en boîte blanche « fortes », et quelle sécurité est atteignable en boîte blanche ?
- Quelle est la difficulté d'exploiter les attaques dans un scénario du monde réel ?

Nous abordons ces questions dans les sections suivantes.

2.3 Approches théoriques

La cryptographie en boîte blanche est souvent liée à l'offuscation de code, puisque toutes deux visent à protéger les implémentations logicielles. Toutes deux ont été reçues avec un scepticisme similaire quant à la faisabilité et à l'absence de fondements théoriques. La recherche théorique sur l'offuscation de code a pris de l'ampleur avec le papier fondateur de Barak *et al.* **[Barak01]** qui a démontré qu'il est impossible de construire un offuscateur générique, c'est-à-dire un offuscateur qui peut protéger n'importe quel programme. Barak *et al.* ont ainsi construit une famille de fonctions qui ne peuvent pas être offusquées en exploitant le fait que le logiciel peut toujours être copié tout en préservant sa fonctionnalité. Néanmoins, ce résultat n'exclut pas l'existence d'offuscateurs de code sûrs : Wee **[Wee05]** a présenté un offuscateur prouvé sûr pour une fonction de point, qui peut être exploité dans la pratique pour construire des fonctionnalités d'authentification.

Des approches théoriques similaires à l'offuscation de code ont été conçues pour la cryptographie en boîte blanche **[Sax09]**. La principale différence est que la sécurité des implémentations en boîte blanche doit être validée par rapport à des notions de sécurité. Une notion de sécurité est une description formelle de la sécurité souhaitée pour un système cryptographique. Ainsi, un système est considéré comme sûr du point de vue CPA (attaque à clair choisi ; *Chosen Plaintext Attack* en anglais) s'il n'est pas possible pour un attaquant ayant accès à un oracle de chiffrement de déterminer le message clair correspondant à un message chiffré donné sans connaître la clé ; ou encore un système est dit KR-sûr (*Key Recovery)* s'il n'est pas possible de retrouver la clé secrète.

Il est sain de définir la cryptographie en boîte blanche avec la même approche, car celle-ci correspond à la réalité. En effet, en pratique, il n'est pas suffisant de s'assurer qu'il ne soit pas possible d'extraire des secrets cachés dans une application. Par exemple, pour créer en logiciel l'équivalent d'une fonction AES dans une carte à puce, il ne suffit pas que l'implémentation en boîte blanche résiste à l'extraction de sa clé, mais elle doit aussi être difficile à inverser. Saxena et Wyseur ont démontré que certaines notions de sécurité ne peuvent pas être satisfaites avec une implémentation *software* (par exemple la notion IND-CCA2), et ils ont proposé une construction dite à « securité prouvable » par rapport à la notion de sécurité IND-CPA **[Sax09]**.

Il résulte d'une analyse théorique que la cryptographie en boîte blanche peut être perçue comme un pont entre la cryptographie symétrique et la cryptographie asymétrique. Un système à clé publique d'un nouveau type peut ainsi être élaboré, où l'opération privée est effectuée de manière efficace par un chiffrement par bloc instancié avec une clé symétrique, alors que l'opération publique est l'implémentation en boîte blanche de la fonction de chiffrement avec la même clé symétrique incorporée dans son code. Notez que ceci est plus difficile à réaliser que la seule protection contre l'extraction de la clé secrète. Par exemple, sans considérer les résultats de cryptanalyse, l'implémentation en boîte blanche originale de l'AES peut facilement être inversée puisque chaque itération peut être décrite comme quatre opérations parallèles de 32 bits dans 32 bits ; chacune d'elles peut être inversée facilement.

3 Cryptographie en boîte blanche dans la pratique

La cryptographie en boîte blanche est utilisée dans plusieurs produits. Des entreprises telles que Microsoft, Apple, Irdeto, NAGRA, Sony, Arxan, et de nombreuses autres ont annoncé le déploiement de techniques en boîte blanche, ont déposé des brevets et/ou ont montré qu'ils ont déployé cette technologie. Jusqu'à présent, nous avons surtout discuté d'implémentations en boîte blanche à clés fixes, dans lesquelles la clé secrète est codée en dur. Dans la pratique cependant, des implémentations en boîte blanche dynamiques sont plus adaptées. Ce sont des implémentations qui ne sont instanciées avec une clé qu'au moment où elles sont appelées. Dans ce cas, ce n'est pas la clé originale qui est présentée (ce qui conduirait à une divulgation facile), mais une version protégée de la clé. L'implémentation en boîte blanche dynamique effectue ensuite une opération de chiffrement/déchiffrement en exploitant la version protégée de la clé de telle sorte qu'aucune information sur cette clé ne soit exposée.

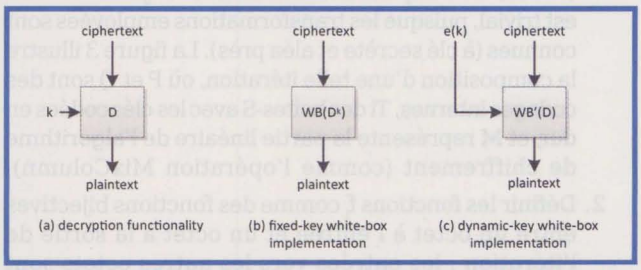


Figure 4 : Types d'implémentations en boîte blanche

Les principales préoccupations avec la cryptographie en boîte blanche sont d'une part la pénalité en termes de performance et de taille, et d'autre part la sécurité.

Les problèmes de performance limitent son exploitabilité dans les usages à haut débit ou très contraints, tels que les systèmes mobiles. Cela peut être résolu en utilisant des implémentations en boîte blanche spéciales qui peuvent exploiter des accélérateurs matériels.

En ce qui concerne la sécurité, pour autant que nous le sachions, aucune implémentation en boîte blanche dans un produit réel n'a souffert d'une attaque de récupération de clé, en dépit des résultats de cryptanalyse qui ont été publiés. Cela montre qu'il existe un écart évident entre la théorie et la pratique, et que même des implémentations en boîte blanche faibles peuvent avoir leur utilité. Pour avoir une idée de la difficulté de monter une telle attaque dans la pratique, nous avons lancé quelques défis publics sur des implémentations en boîte blanche faibles. Ils ont récemment résulté en deux *exploits* de récupération des clés. La conclusion de ces attaques est que casser une implémentation en boîte blanche en pratique est un travail dédié et très coûteux en temps. Les attaques sont très dépendantes de la construction de l'implémentation en boîte blanche et des propriétés de l'algorithme de chiffrement sous-jacent. Par conséquent, des attaques largement applicables sont difficiles à déployer et il n'existe pas d'outils automatiques pour casser systématiquement des implémentations en boîte blanche.

3.1 Code lifting

Le principal problème dont souffrent les implémentations en boîte blanche dans la pratique est le « code lifting ». Dans cette attaque, l'attaquant ne cherche pas à extraire la clé secrète à partir de l'implémentation, mais utilise l'ensemble de l'application comme si elle était une grosse clé. De telles attaques ont été utilisées dans la pratique à plusieurs reprises, où une bibliothèque de sécurité n'est pas rétroconçue, mais où sa fonctionnalité de déchiffrement est directement exploitée.

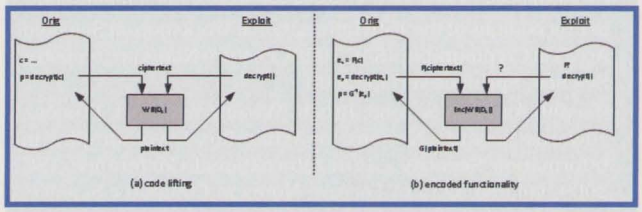


Figure 5 : Code lifting

Le *code lifting* peut être combattu en poussant à la limite le principe de l'implémentation en boîte blanche. Ceci peut être réalisé en appliquant des codages externes au système original, et notamment les codages annihilants à d'autres endroits dans l'application principale. Par exemple, des codages annihilants peuvent être incorporés dans la fonction qui appelle l'opération de déchiffrement. Ainsi, la fonctionnalité ($G\ D_{\_k}\ F^{-1}$) est implémentée, et sans la connaissance de G et F, l'attaquant sera incapable d'utiliser l'opération de déchiffrement en dehors du contexte prévu. Il faudra alors utiliser des techniques d'offuscation ou d'isolation de processus dans un environnement sécurisé afin de s'assurer que les fonctions G et F ne peuvent pas être extraites de la fonction appelante. Par conséquent, nous avons aussi une méthode pour associer une opération cryptographique à un composant sécurisé ; nous pouvons donc faire du *node-locking*. Ceci est un exemple de technique de protection préventive des logiciels.

3.2 Software fingerprinting

Une autre direction consiste à déployer des stratégies réactives : être capable de détecter qui a partagé le code au moyen d'une clé embarquée. Dans **[Mich07]**, Michiels et Gorissen présentent une technique pour insérer des *fingerprints* dans des implémentations en boîte blanche. L'idée principale repose sur l'observation que les codages annihilants internes peuvent être choisis de façon que les tables en boîte blanche qui en résultent contiennent une valeur arbitraire prédéfinie. Cette valeur peut être une empreinte qui sera utilisée pour identifier le logiciel ou introduire un droit d'auteur via des clés cryptographiques. Cette même approche peut également être utilisée pour permettre une forme de résistance aux modifications des logiciels : lorsque les tables contiennent des valeurs qui représentent du *bytecode* ou des instructions assembleur, toute modification de cette représentation conduirait à une modification des tables en boîte blanche, et donc de rendre la fonctionnalité de l'algorithme de chiffrement sous-jacent incorrecte. Ainsi, le logiciel dispose d'une double représentation. La sécurité de cette approche est cependant contestable. Tout d'abord, cela dépend de la sécurité des implémentations en boîte blanche, ce qui est déjà discutable. D'autre part, une telle approche peut être assez facilement contrée puisque l'attaquant peut insérer de nouveaux encodages internes qui détruisent l'empreinte, il peut exploiter une attaque par clonage qui distingue l'exécution du code de l'exécution de l'implémentation en boîte blanche sous-jacent.

Une meilleure stratégie réactive a été présentée par Billet et Gilbert **[Bil03]** qui ont proposé une implémentation en boîte blanche intrinsèquement traçable. Ils ont ainsi présenté une nouvelle construction d'un chiffrement par bloc, à partir de laquelle plusieurs instances peuvent être générées ; chaque instance est fonctionnellement équivalente, mais identifiable par inspection du code. Malheureusement, le bloc de base sur lequel ce chiffrement a été construit a été cassé. En introduisant des perturbations, Bringer *et al.* **[Bri06trace]** ont renforcé ce chiffrement traçable.

Conclusion

La cryptographie en boîte blanche est un élément nécessaire à toute stratégie saine de sécurisation globale du logiciel. Elle est la pierre angulaire de la protection des primitives cryptographiques des applications qui s'exécutent sur des plates-formes hostiles. Les techniques

en boîte blanche initiales ont été présentées en 2002, avec des implémentations du DES et de l'AES. Depuis, la cryptographie en boîte blanche a pris de l'ampleur : plusieurs résultats de cryptanalyse et de nouvelles constructions ont été publiés et de véritables fondations théoriques ont été formulées.

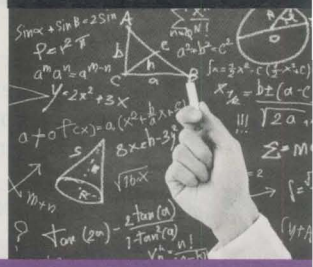
Aujourd'hui, la cryptographie en boîte blanche est utilisée dans des applications réelles, principalement des applications de DRM. Malgré la publication d'attaques académiques, aucune attaque contre les implémentations commerciales en boîte blanche n'a été observée à notre connaissance. Les attaquants se concentrent plutôt sur d'autres parties du système et exploitent la fonctionnalité cryptographique sans l'attaquer. Mais, la cryptographie en boîte blanche offre des possibilités de combattre également sur ce front : les attaques par code lifting peuvent être évitées par le verrouillage des implémentations dans l'application et d'autres fonctions de sécurité telles que la traçabilité peuvent être ajoutées aux implémentations en boîte blanche. ■

■ REMERCIEMENTS

La version originale de cet article a été écrite en anglais, et peut être téléchargée à l'adresse suivante : http://www.whiteboxcrypto.com. Nous tenons à remercier Jean-Bernard Fischer et André Nicoulin pour la traduction de l'article.

■ RÉFÉRENCES

- [Barak01] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yag, On the (Im)possibility of Obfuscating Progras. Iin Advances in Cryptology - CRYPTO 2001, volume 2139 of Lecture Notes in Computer Science, pages 1–8, Springer-Verlag, 2001.
- [Bil03] Olivier Billet and Henri Gilbet, A Traceable Block Ciphr. Iin Advances in Cryptology - ASIACRYPT 2003, volume 2894 of Lecture Notes in Computer Science, pages 331–36, Springer-Verlag, 2003.
- [Bil04] Olivier Billet, Henri Gilbert, and Charaf Ech-Chati, Cryptanalysis of a White Box AES Implementatin. Iin Proceedings of the 11th International Workshop on Selected Areas in Cryptography (SAC 2004), volume 3357 of Lecture Notes in Computer Science, pages 227–20, Springer-Verlag, 2004.
- [Bri06trace] Julien Bringer, Hervé Chabanne, and Emmanuelle Dottx, Perturbing and Protecting a Traceable Block Cipher, in Proceedings of the 10th Communications and Multimedia Security (CMS 2006), volume 4237 of Lecture Notes in Computer Science, pages 109–19, Springer-Verlag, 2006.
- [Bri06wbc] Julien Bringer, Hervé Chabanne, and Emmanuelle Dottx, White box cryptography: Another attemt, Cryptology ePrint Archive, Report 2006/468, 2006.
- [Chow02DES] Stanley Chow, Philip A. Eisen, Harold Johnson, and Paul C. van Oorscht, A white-box DES implementation for DRM applications, in Proceedings of the ACM Workshop on Security and Privacy in Digital Rights Management (DRM 2002), volume 2696 of Lecture Notes in Computer Science, pages 1–5, Springer, 2002.
- [Chow02AEQ] Stanley Chow, Philip A. Eisen, Harold Johnson, and Paul C. van Oorscht, White-Box Cryptography and an AES Implementation, in Proceedings of the 9th International Workshop on Selected Areas in Cryptography (SAC 2002), volume 2595 of Lecture Notes in Computer Science, pages 250–270, Springer, 2002
- [Dem10] Yoni De Mulder, Brecht Wyseur, and Bart Preneel, Cryptanalysis of a Perturbated White-box AES Implémentation, in Progress in Cryptology - INDOCRYPT 2010, volume 6498 of Lecture Notes in Computer Sciene, Springer-Verlag,pp. 292-310, 2010
- [Ding04] Jintai Dig, A New Variant of the Matsumoto-Imai Cryptosystem through Perturbatin, in Proceedings of the 7th International Workshop on Theory and Practice in Public Key Cryptography (PKC 2004), volume 2947 of Lecture Notes in Computer Science, pages 305–318. Springer-Verlag, 2004
- [Goubin07] Louis Goubin, Jean-Michel Masereel, and Michaël Quisquater, Cryptanalysis of White Box DES Implementations. In Proceedings of the 14th International Workshop on Selected Areas in Cryptography (SAC 2007), volume 4876 of Lecture Notes in Computer Science, pages 278–295, Springer-Verlag, 2007
- [Jacob02] Matthias Jacob, Dan Boneh, and Edward W. Felten, Attacking an Obfuscated Cipher by Injecting Faults. In Proceedings of the ACM Workshop on Security and Privacy in Digital Rights Management (DRM 2002), volume 2696 of Lecture Notes in Computer Science, pages 16–31, Springer, 2002
- [Kerins06] Tim Kerins and Klaus Kursawe, A cautionary note on weak implementations of block ciphers, in 1st Benelux Workshop on Information and System Security (WISSec 2006), page 12, Antwerp, BE, 2006
- [Link05] Hamilton E. Link and William D. Neuman., Clarifying Obfuscation: Improving the Security of White-Box DES, in Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC 2005), volume 1, pages 679–684, Washington, DC, USA, 2005, IEEE Computer Society
- [Mich07] Wil Michiels and Paul Gorissen, Mechanism for software tamper resistance: an application of white-box cryptography, in Proceedings of 7th ACM Workshop on Digital Rights Management (DRM 2007), pages 82–89, ACM Press, 2007
- [Mich08] Wil Michiels, Paul Gorissen, and Henk D.L. Hollmann, Cryptanalysis of a Generic Class of White-Box Implementations, in Proceedings of the 15th International Workshop on Selected Areas in Cryptography (SAC 2008), Lecture Notes in Computer Science, Springer-Verlag, 2008
- [Sax09] Amitabh Saxena, Brecht Wyseur, and Bart. Preneel, Towards Security Notions for White-Box Cryptography, in Information Security - 12th International Conference, ISC 2009, volume 5735 of Lecture Notes in Computer Science, pages 49–58, Springer-Verlag, 2009
- [Wee05] Hoeteck Wee, On Obfuscating Point Functions, in Proceedings of the 37th ACM Symposium on Theory of Computing (STOC 2005), pages 523–532, New York, NY, USA, 2005, ACM Press
- [Wyseur07] Brecht Wyseur, Wil Michiels, Paul Gorissen, and Bart Preneel, Cryptanalysis of White-Box DES Implementations with Arbitrary External Encodings, in Proceedings of the 14th International Workshop on Selected Areas in Cryptography (SAC 2007), volume 4876 of Lecture Notes in Computer Science, pages 264–277, Springer-Verlag, 2007
- [Wyseur09] Brecht Wyseur, White-Box Cryptography, PhD thesis, Katholieke Universiteit Leuven, Bart Preneel (promotor), 169+32 pages, 2009



INTRODUCTION AU REVERSE CRYPTO

Jean-Philippe Luyten - jp@r-3-t.org - DGA Maîtrise de l'Information

mots-clés : REVERSE ENGINEERING / HASH / HMAC / PBKDF2 / BLOCK CIPHER

La rétroconception d'algorithmes cryptographiques est une discipline à part dans le monde des reversers. La difficulté réside dans le fait de ne pas se « noyer » dans des parties inutiles à l'analyse (dans une fonction de compression d'un algorithme de hachage par exemple). La connaissance des concepts mathématiques sous-jacents à chaque algorithme est inutile. Par contre, la connaissance des grandes familles de fonctions cryptographiques et de leurs implémentations est indispensable : ce sera l'objet de la première partie de cet article. Puis, nous verrons comment appliquer ces résultats sur un exemple concret.

1 Familles de fonctions cryptographiques

1.1 Fonction de hachage

Une fonction de hachage est une fonction à sens unique : son but est d'obtenir un identifiant à partir de données binaires. Par exemple, CRC32 peut être considéré comme un algorithme de hachage bien qu'il soit faible d'un point de vue cryptographique. Ce type de fonction peut avoir plusieurs rôles : stocker une empreinte de mot de passe, être utilisé comme somme de contrôle ou encore dans des processus de signature numérique. Lorsque la fonction utilisée est une fonction standard (md5, sha1, ripemd160, ...), elle s'identifie facilement dans un binaire. Cela provient de sa forme spécifique et de l'utilisation de constantes propres à chaque algorithme.

Une fonction de hachage s'utilise en trois phases :

- Initialisation de la fonction (fonction **hash_init**) : cette phase initialise un contexte qui sera utilisé comme support pour la suite des calculs. Ce contexte a quelques particularités : il contient le tableau stockant l'empreinte de sortie et un accumulateur. L'empreinte de sortie est initialisée avec des constantes propres à chaque algorithme. L'accumulateur a une taille fixe qui dépend de la fonction de « compression » que nous allons détailler (64 octets pour **md5** et **sha1**, 128 octets pour **sha512**). Un contexte possède donc une taille d'environ **BLOCK_SIZE + DIGEST_SIZE**, soit un peu plus de 80 octets pour **md5**. Cette taille pourra être un indicateur sur l'algorithme utilisé.
- Mise à jour du contexte en intégrant de nouvelles données (fonction **hash_update**). Cette phase a besoin de deux entrées : le contexte à mettre à jour et les données. Lorsque la fonction intègre de nouvelles données, elle les stocke dans un accumulateur. Une fois l'accumulateur rempli, la fonction **hash_update** fait appel à la fonction de compression souvent appelée **hash_transform**. Cette fonction a la particularité d'utiliser des constantes spécifiques qui permettent de facilement l'identifier dans un code binaire.
- Récupération de l'empreinte (fonction **hash_final**). Cette phase prend le contexte en entrée et renvoie l'empreinte. Cette fonction fait appel à la fonction **hash_update** pour terminer l'intégration de données (entre autres, si les données présentes dans l'accumulateur ne sont pas multiples de **BLOCK_SIZE** et n'ont donc pas encore été intégrées).

Nous avons donc une séquence d'appel caractéristique avec deux fonctions possédant des constantes marquantes (**hash_init** et **hash_transform**).

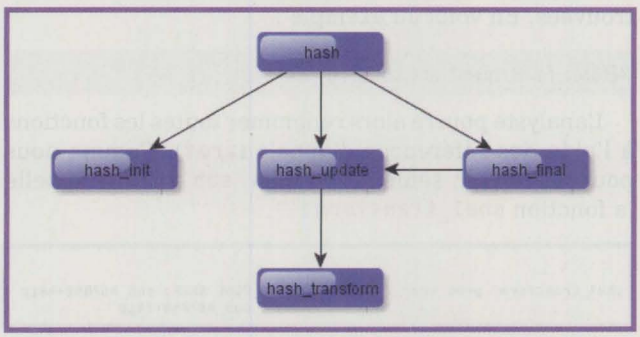


Figure 1 : Arbre d'appel d'une fonction de hachage

Prenons par exemple la fonction **sha1**. Comme nous l'avons vu, il est possible de retrouver des constantes dans les fonctions **sha1_init** et **sha1_transform**. À partir de ces deux fonctions, il est très facile de retrouver les fonctions **sha1_update** et **sha1_final** en suivant l'arbre d'appel. Regardons le code C de la fonction **sha1_init** :

```
void sha1_init(sha1_context* ctx)
{
    ctx->total[0] = 0;
    ctx->total[1] = 0;

    ctx->state[0] = 0x67452301;
    ctx->state[1] = 0xEFCDAB89;
    ctx->state[2] = 0x98BADCFE;
    ctx->state[3] = 0x10325476;
    ctx->state[4] = 0xC3D2E1F0;
}
```

Nous pouvons observer cinq constantes de 32 bits initialisant l'empreinte de 160 bits. Ces constantes peuvent se retrouver dans d'autres algorithmes mais la taille du contexte et de l'empreinte est un bon indicateur.

Regardons maintenant la fonction **sha1_transform**. Elle peut être décomposée en quatre boucles qui utilisent une constante différente : **0x5A827999**, **0x6ED9EBA1**, **0x8F1BBCDC** puis **0xCA62C1D6**. Le fait de retrouver ces constantes dans le code permet d'identifier la fonction à laquelle elles appartiennent.

```
lea     edi, [edi+eax+5A827999h]
mov     eax, [ebp+var_28]
xor     eax, [ebp+var_40]
ror     ebx, 2
xor     eax, [ebp+var_48]
mov     [ebp+var_C], edi
xor     eax, [ebp+var_54]
mov     [ebp+var_14], ebx
rol     eax, 1
mov     [ebp+var_48], eax
mov     eax, [ebp+var_10]
xor     eax, ebx
xor     eax, [ebp+var_8]
mov     ebx, [ebp+var_8]
rol     edi, 5
add     edi, [ebp+var_48]
add     eax, edi
mov     edi, [ebp+var_4]
lea     edi, [edi+eax+6ED9EBA1h]
```

Figure 2 : Constantes spécifiques à la fonction sha1_transform

Le plugin **FindCrypt** pour IDA est une aide précieuse : il indique les emplacements mémoire des constantes trouvées. En voici un exemple :

```
4068B6: found sparse constants for SHA-1
```

L'analyste pourra alors renommer toutes les fonctions à l'aide des références d'appels (**xref**). Comme nous pouvons le voir, seule la fonction **sub_407B40** appelle la fonction **sha1_transform** :

```
sha1_transform  proc near               ; CODE XREF: sub_407B40+4B↓p
                                        ; sub_407B40+74↓p
```

Figure 3 : cross_ref de la fonction sha1_transform

La fonction **sub_407B40** peut donc être renommée en **sha1_update**. En utilisant la même démarche, nous pouvons voir :

```
; int __cdecl sha1_update(int, void *Src, size_t Size)
sha1_update     proc near               ; CODE XREF: _main+E5↑p
                                        ; sub_407BF0+83↓p
                                        ; sub_407BF0+8F↓p
```

Figure 4 : cross_ref de la fonction sha1_update

Cette fonction est appelée dans la fonction **main** mais également deux fois par la fonction **sub_407BF0**. Cette dernière est donc la fonction **sha1_final**.

Nous venons de voir qu'en se basant simplement sur les constantes et la connaissance de l'implémentation de **sha1**, il a été possible d'identifier toutes les fonctions élémentaires. Bien entendu, cette identification devra être validée : rien n'empêche le développeur d'avoir légèrement modifié l'algorithme tout en conservant les constantes.

1.2 HMAC

Un **HMAC** est une fonction permettant de vérifier l'intégrité et l'authenticité d'un message (MAC signifie code d'authentification de message). Ce calcul est réalisé en combinant une fonction de hachage avec une clé secrète. La forme particulière de cette fonction (RFC 2104) permet de l'identifier rapidement. Elle est définie ainsi :

```
HMAC(K,m) = H((K ^ OPAD) || H((K ^ IPAD) || m)) où :
- H est un algorithme de hachage
- ^ l'opérateur ou exclusif
- || l'opérateur de concaténation.
- IPAD est un buffer de taille BLOCK_SIZE rempli de 0x36
- OPAD est un buffer de taille BLOCK_SIZE rempli de 0x5C
```

Cette fonction ressemble à un algorithme de hachage dans son utilisation. Les étapes sont quasiment identiques : initialisation, compression des données puis récupération de l'empreinte. Seule diffère la phase d'initialisation qui prend en paramètre une clé secrète. L'arbre d'appel de la fonction est (la plupart du temps) :

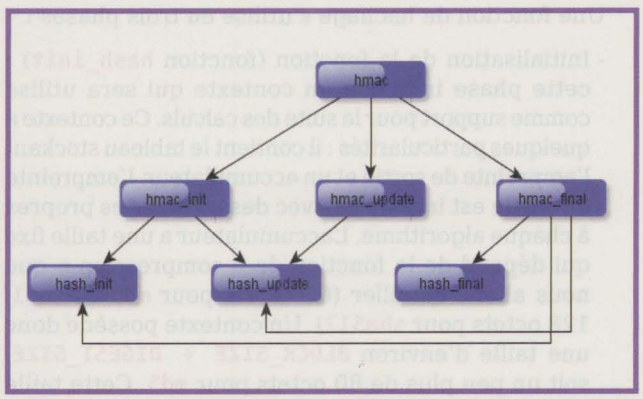


Figure 5 : Arbre d'appel d'une fonction de HMAC

Si la clé secrète est plus grande que la taille d'un bloc (**BLOCK_SIZE**), alors elle est auparavant hachée par la fonction de hachage : c'est l'empreinte de la clé qui sera alors « xorée » avec **ipad** et **opad**. La structure de la fonction **HMAC** fait apparaître une boucle où les valeurs particulières **0x36** et **0x5C** sont présentes. À noter que la taille d'un bloc étant la plupart du temps multiple de 32 bits, on retrouve régulièrement les valeurs **0x36363636** et **0x5C5C5C5C** par souci d'optimisation.

Regardons le code d'initialisation suivant :

```
memset (internalKey, 0, SHA1_BLOCK_SIZE);
if (key_len < SHA256_BLOCK_SIZE)
  memcpy (internalKey, key, key_len);
else
  sha1(key, key_len, internalKey);
memcpy (ctx->ipad, internalKey, SHA1_BLOCK_SIZE);
memcpy (ctx->opad, internalKey, SHA1_BLOCK_SIZE);
for (i = 0; i < SHA256_BLOCK_SIZE; i++)
{
  ctx->ipad[i] ^= 0x36;
  ctx->opad[i] ^= 0x5C;
}
```

Il se dégage une boucle où l'opération « xor » avec les constantes 0x36 et 0x5C apparaît. Cette boucle se retrouvera dans le code assembleur généré :

```
.text:00401422 create_ipad_and_opad_buffer:            ; CODE XREF: sub_4013A0+BB↓j
.text:00401422                 mov     edx, [ebp+i]
.text:00401425                 add     edx, 1
.text:00401428                 mov     [ebp+i], edx    ; i++
.text:0040142B
.text:0040142B loc_40142B:                             ; CODE XREF: sub_4013A0+80↑j
.text:0040142B                 cmp     [ebp+i], BLOCK_SIZE
.text:0040142F                 jnb     short loc_401450 ; i < BLOCK_SIZE ?
.text:00401431                 mov     eax, [ebp+hmac_ctx]
.text:00401434                 add     eax, [ebp+i]
.text:00401437                 movzx   ecx, [eax+hmac.ipad]
.text:0040143A                 xor     ecx, 36h
.text:0040143D                 mov     edx, [ebp+hmac_ctx]
.text:00401440                 add     edx, [ebp+i]
.text:00401443                 mov     [edx+hmac.ipad], cl ; hmac_ctx.ipad[i] ^= 0x36
.text:00401445                 mov     eax, [ebp+hmac_ctx]
.text:00401448                 add     eax, [ebp+i]
.text:0040144B                 movzx   ecx, [eax+hmac.opad]
.text:0040144F                 xor     ecx, 5Ch
.text:00401452                 mov     edx, [ebp+hmac_ctx]
.text:00401455                 add     edx, [ebp+i]
.text:00401458                 mov     [edx+hmac.opad], cl ; hmac_ctx.opad[i] ^= 0x5C
.text:00401458                 jmp     short create_ipad_and_opad_buffer
```

Figure 6 : Initialisation des buffers ipad et opad pour un HMAC

Tout comme les algorithmes de hachage, un **HMAC** possède un contexte qui est initialisé dans la fonction d'initialisation. Ce contexte possède la taille du contexte de la fonction de hachage utilisée, auquel on ajoute la taille d'un bloc correspondant à **opad**. Ce bloc (ou éventuellement la clé secrète dans certaines implémentations) doit être conservé en mémoire car il sera nécessaire lors de la phase de finalisation.

1.3 Dérivation de clés

Les fonctions de dérivation de clés se retrouvent dans toutes les applications qui transforment un mot passe en une ou plusieurs clés. Un algorithme de hachage offre cette fonctionnalité, mais il existe plusieurs standards qui permettent de générer une clé de longueur variable à partir d'une graine (un mot de passe par exemple). Ces fonctions sont construites pour être robustes face aux attaques par force brute.

Aujourd'hui, la fonction la plus répandue est **PBKDF2** (*Password Based Key Derivation Function* - RFC 2898). Cette fonction possède quatre paramètres :

- un mot de passe ;
- une graine (sel) qui permet de s'affranchir des attaques par dictionnaire précalculé ;
- un nombre d'itérations ;
- une taille de sortie.

Si l'utilisateur souhaite dériver son mot de passe en plusieurs clés, il pourra additionner les longueurs de clé nécessaires et utiliser la sortie suivant son besoin. Par exemple, s'il a besoin d'une clé de chiffrement de 256 bits et d'une clé d'authentification de 256 bits, il pourra générer une clé de 512 bits : les 256 premiers bits serviront comme clé de chiffrement et les 256 derniers bits comme clé d'authentification.

Cette fonction est très répandue : nous la retrouvons aussi bien dans des logiciels bureautiques comme TrueCrypt que dans des protocoles réseau comme WPA2.

PBDKF2 s'appuie sur une fonction dite pseudo-aléatoire (**PRF**), généralement définie par une fonction **HMAC**. À chaque itération, l'algorithme permet de calculer une clé de la taille de sortie de la fonction **PRF**. S'il s'agit d'une fonction **Hmac-sha256**, 32 octets sont générés à chaque tour. L'algorithme sera donc itéré autant de fois que nécessaire afin d'obtenir la taille de sortie désirée. Voici un pseudo-code réalisant cette fonction :

```
NumLoop = ceil(OutputSize / BLOCK_SIZE)
for (i = 0 ; i < NumLoop; i++)
  K = S = PRF (Password, Salt || i)
  For (j = 1 ; j < NB_ITERATION ; j++)
    S = PRF (Password, S)
    K ^= S

  DK ||= K
```

Le code généré présente deux boucles **for** et l'appel à la fonction **PRF** (**HMAC** dans ce cas) sera présent deux fois. Voici un exemple :

```
.text:004011E0                 push    ecx             ; data_size
.text:004011E1                 mov     edx, [ebp+data]
.text:004011E4                 push    edx             ; data
.text:004011E5                 call    hmac
.text:004011F4                 add     esp, 14h
.text:004011F7                 push    20h             ; Size
.text:004011F9                 lea     eax, [ebp+Dst]
.text:004011FC                 push    eax             ; Src
.text:004011FD                 lea     ecx, [ebp+var_2C]
.text:00401200                 push    ecx             ; Dst
.text:00401201                 call    memcpy
.text:00401206                 add     esp, 0Ch
.text:00401209                 mov     [ebp+num_loop], 1
.text:00401210                 jmp     short loc_40121B
.text:00401212 ; ---------------------------------------------------------------------------
.text:00401212
.text:00401212 pbkdf_loop:                             ; CODE XREF: sub_401100:loc_401282↓j
.text:00401212                 mov     edx, [ebp+num_loop]
.text:00401215                 add     edx, 1
.text:00401218                 mov     [ebp+num_loop], edx
.text:00401218
.text:00401218 loc_40121B:                             ; CODE XREF: sub_401100+110↑j
.text:00401218                 mov     eax, [ebp+num_loop]
.text:0040121E                 cmp     eax, [ebp+arg_18]
.text:00401221                 jnb     short loc_401284
.text:00401223                 lea     ecx, [ebp+digest]
.text:00401226                 push    ecx             ; digest
.text:00401227                 mov     edx, [ebp+password_size]
.text:0040122A                 push    edx             ; password_size
.text:0040122B                 mov     eax, [ebp+password]
.text:0040122E                 push    eax             ; password
.text:0040122F                 push    20h             ; data_size
.text:00401231                 lea     ecx, [ebp+Dst]
.text:00401234                 push    ecx             ; data
.text:00401235                 call    hmac
```

Figure 7 : Boucle principale de PBKDF2

1.4 Cryptographie symétrique

Deux types d'algorithmes symétriques (aussi appelés algorithmes à clé secrète) existent : les algorithmes de chiffrement par bloc (*block ciphers*) et les algorithmes de chiffrement par flot (*stream ciphers*). Dans cet article, nous ne retiendrons que les algorithmes par bloc.

En effet, les algorithmes par flot sont aujourd'hui moins rencontrés. Seul rc4 se retrouve couramment, mais sa simplicité d'implémentation en fait un algorithme difficile à identifier dans un code binaire. Le seul élément qui peut mettre l'analyste sur la bonne voie est qu'il utilise comme contexte un tableau de 256 octets initialisés de 0 à 255.

Le reverse d'algorithme de chiffrement par bloc se divise en deux parties : le chiffrement de blocs élémentaires (réalisé à l'aide d'un algorithme comme AES) et le chaînage de ce chiffrement par bloc réalisé à l'aide d'un mode de chiffrement (par exemple CBC).

1.4.1 Algorithmes de chiffrement par bloc

Comme pour les **HMAC**, ces algorithmes débutent par un e phase d'initialisation prenant une clé en paramètre. Cette phase est souvent appelée « key_schedule ». Elle consiste à prendre la clé secrète en entrée pour produire les sous-clés utilisées lors du chiffrement (ou déchiffrement). Ces sous-clés sont stockées sous la forme d'un contexte. Les fonctions de chiffrement (respectivement déchiffrement) transforment un bloc de texte clair (chiffré) en un bloc de texte chiffré (clair). La taille de bloc est fixée par l'algorithme, à titre d'exemple DES traite des blocs de 64 bits et AES des blocs de 128 bits.

Les algorithmes de chiffrement par bloc utilisent des constantes reconnaissables facilement dans le code. Comme pour les algorithmes de hachage, une fois les constantes localisées, il est facile de retrouver les différentes fonctions composant l'algorithme.

Prenons AES comme exemple. L'algorithme repose sur 10 tables : Te0 à Te4 pour le chiffrement et Td0 à Td4 pour le déchiffrement **[1]**. En regardant le code source, nous nous apercevons que la fonction **AES_BlockEncrypt** est la seule à utiliser les tables Te0 à Te3. L'analyse peut donc commencer par rechercher ces tables. Pour cela, il est possible de s'appuyer, une fois encore, sur le plugin **FindCrypt**. En voici un exemple de sortie :

```
4090F8: found const array Rijndael_Te0 (used in Rijndael)
4094F8: found const array Rijndael_Te1 (used in Rijndael)
4098F8: found const array Rijndael_Te2 (used in Rijndael)
409CF8: found const array Rijndael_Te3 (used in Rijndael)
40A0F8: found const array Rijndael_Te4 (used in Rijndael)
40A4F8: found const array Rijndael_Td0 (used in Rijndael)
40A8F8: found const array Rijndael_Td1 (used in Rijndael)
40ACF8: found const array Rijndael_Td2 (used in Rijndael)
40B0F8: found const array Rijndael_Td3 (used in Rijndael)
40B4F8: found const array Rijndael_Td4 (used in Rijndael)
```

```
Rijndael_Te3    dd 6363A5C6h, 7C7C84F8h, 777799EEh, 7B7B8DF6h, 0F2F20DFFh
                                        ; DATA XREF: sub_4020E0+C4↑r
                                        ; sub_4020E0+108↑r
                                        ; sub_4020E0+14C↑r
                                        ; sub_4020E0+190↑r
                                        ; sub_4020E0+1ED↑r
                                        ; sub_4020E0+230↑r
                                        ; sub_4020E0+274↑r
                                        ; sub_4020E0+288↑r
                dd 6B6BBDD6h, 6F6FB1DEh, 0C5C55491h, 30305060h, 1010302h
                dd 6767A9CEh, 2B2B7D56h, 0FEFE19E7h, 0D7D762B5h, 0ABABE64Dh
```

Figure 8 : Liste des références à la table Te3

Il est donc possible de renommer la fonction **sub_4020E0** en **AES_BlockEncrypt**. Afin de retrouver la fonction **SetEncryptKey**, il faut regarder les références à la table Te4 et chercher quelle fonction n'a pas encore été identifiée :

```
Rijndael_Te4    dd 63636363h, 7C7C7C7Ch, 77777777h, 7B7B7B7Bh, 0F2F2F2F2h
                                        ; DATA XREF: sub_4019A0+B8↑r
                                        ; sub_4019A0+D3↑r
                                        ...
                                        ; sub_4019A0+717↑r
                                        ; AES_BlockEncrypt+2D6↑r
```

Figure 9 : Liste des références à la table Te4

Nous pouvons voir ici que la fonction **sub_4019A0** n'a pas été renommée : il s'agit très certainement de la fonction **AES_SetEncryptKey**. Il est ensuite possible d'appliquer le même raisonnement pour retrouver les fonctions **SetDecryptKey** et **AES_BlockDecrypt**. La fonction **AES_BLockDecrypt** est la seule à utiliser la table Td4 alors que la fonction **SetDecryptKey** utilise les autres.

Une fois encore, en s'appuyant sur les constantes rencontrées et en ayant une bonne connaissance des implémentations, il est possible de retrouver les primitives cryptographiques sans reverser un code ligne par ligne.

1.4.2 Mode de chiffrement

Lors de la rétroconception d'algorithmes cryptographiques, nous retrouvons (presque) toujours les mêmes modes de chiffrement : il s'agit des plus courants, c'est-à-dire **CBC**, **CFB**, **OFB**, **CTR**, ... La page Wikipédia traitant de ces modes est un excellent support (http://en.wikipedia.org/wiki/Cipher_block_chaining). Les schémas les décrivant permettent de les reconnaître presque instantanément. Par exemple, sur le code assembleur suivant, nous pouvons voir une boucle qui réalise des « xor » puis un appel vers la fonction **AES_BlockEncrypt**.

```
.text:004018D8 xor_loop_cbc:                           ; CODE XREF: _main+DB↓j
.text:004018D8                 mov     ecx, [ebp+i]
.text:004018DE                 add     ecx, 1
.text:004018E1                 mov     [ebp+i], ecx
.text:004018E7
.text:004018E7 loop_first_start:                       ; CODE XREF: _main+96↑j
.text:004018E7                 cmp     [ebp+i], 10h
.text:004018EE                 jge     short EncryptBuffer
.text:004018F0                 mov     edx, [ebp+i]
.text:004018F6                 movsx   eax, byte ptr [ebp+edx+PlainText]
.text:004018FE                 mov     ecx, [ebp+i]
.text:00401904                 movsx   edx, [ebp+ecx+IV]
.text:0040190C                 xor     edx, eax
.text:0040190E                 mov     eax, [ebp+i]
.text:00401914                 mov     [ebp+eax+IV], dl
.text:0040191B                 jmp     short xor_loop_cbc
.text:0040191D ; ---------------------------------------------------------------------------
.text:0040191D
.text:0040191D EncryptBuffer:                          ; CODE XREF: _main+AE↑j
.text:0040191D                 lea     ecx, [ebp+CipherText]
.text:00401923                 push    ecx
.text:00401924                 lea     edx, [ebp+IV]
.text:0040192A                 push    edx
.text:0040192B                 lea     eax, [ebp+AES_Context]
.text:00401931                 push    eax
.text:00401932                 call    AES_BlockEncrypt
```

Figure 10 : Exemple d'implémentation du mode CBC

En comparant cette structure avec les différents modes de chiffrement, CBC paraît tout de suite un bon candidat :

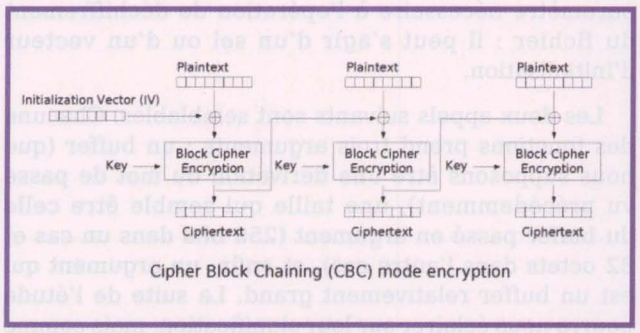


Figure 11 : Schéma décrivant le mode CBC sur Wikipédia (source : http://upload.wikimedia.org/wikipedia/commons/d/d3/Cbc_encryption.png)

1.5 Cryptographie asymétrique

La cryptographie asymétrique est assez difficile à appréhender : l'analyste ne peut pas se baser sur des constantes. Il doit commencer par identifier les fonctions composant la bibliothèque de manipulation des grands nombres utilisée. Une bonne pratique est de tenter de retrouver ces fonctions en boîte noire. Que ce soit pour RSA, Diffie-Hellman ou encore dans El Gamal, la principale fonction est l'exponentiation modulaire. Une solution peut être d'utiliser une calculatrice grands nombres (comme **http://world.std.com/~reinhold/BigNumCalc.html**) et d'essayer d'identifier les fonctions mathématiques d'après les entrées/sorties.

Il est également important d'avoir une idée de la représentation mémoire des grands nombres. La majeure partie des bibliothèques utilisent le même principe : une structure contenant un buffer stockant le grand nombre accompagné d'un membre indiquant la taille utilisée pour la représentation. Par exemple, voici la structure définie dans **libgmp** :

```
typedef struct
{
   int _mp_alloc;    /* Number of *limbs* allocated
      and pointed to by the _mp_d field. */
   int _mp_size;     /* abs(_mp_size) is the number of limbs
      the last field points to. If _mp_size is negative
      this is a negative number. */
   mp_limb_t *_mp_d;  /* Pointer to the limbs. */
} __mpz_struct;
```

À noter que les recommandations concernant l'utilisation de RSA et le *padding* à appliquer peuvent être un bon point d'entrée. Par exemple, OAEP (*Optimal Asymmetric Encryption Padding*) s'appuie sur une fonction de hachage qui, comme nous l'avons vu, est souvent facilement identifiable.

2 Étude de cas

Nous allons utiliser ces principes pour reverser un programme similaire à celui présenté par Olivier Tuchon dans ce même numéro. Imaginons que l'on souhaite retrouver l'architecture de ce logiciel sans rentrer précisément dans les détails d'implémentations de chaque brique cryptographique.

L'étude doit débuter par la recherche d'un point d'entrée pour l'analyse. L'expérience et le type de logiciel étudié peuvent permettre de formuler des hypothèses. Par exemple, pour un logiciel de chiffrement de fichier, on peut raisonnablement supposer que les traitements sur les données s'effectuent entre les fonctions de lecture et d'écriture de fichiers.

Appel de fonction sur architecture x86

Avant de rentrer dans les détails, rappelons le fonctionnement d'un appel de fonction (de type stdcall) sur architecture x86. Les arguments sont empilés sur la pile à l'aide de l'instruction `push` puis la fonction est appelée à l'aide de l'instruction `call`. Ainsi, si nous avons une fonction ayant pour prototype : `void MyFunction(int arg1, int arg2, int arg3)`, un appel à cette fonction sera traduit par :

```
push arg3
push arg2
push arg1
call MyFunction
```

Le lecteur curieux pourra aller voir http://en.wikipedia.org/wiki/X86_calling_conventions.

Une fois les fonctions cryptographiques isolées, il faut rester, autant que possible, au même niveau dans le code (en quelque sorte dans la fonction principale des traitements cryptographiques). Pour chaque appel de fonction, il s'agit d'identifier les entrées et les sorties. Cette étape permettra d'identifier les grandes familles de

fonctions et de ne pas se lancer dans la rétroconception d'une partie sans intérêt par rapport aux objectifs fixés. Si une analyse dynamique est possible, elle peut aider à identifier les arguments : il ne faut surtout pas s'en priver.

Commençons par regarder les arguments et variables locales de la fonction réalisant les opérations cryptographiques et considérons qu'une analyse statique ou dynamique a permis d'identifier les deux arguments de cette fonction :

```
; int __cdecl MainCryptoFunction(char *InputFile, char *Password)
MainCryptoFunction proc near            ; CODE XREF: _main+5D↓p

Handle_Src_File = dword ptr -528h
Buffer6_16_Bytes= byte ptr -524h
Buffer1_32_Bytes= byte ptr -514h
Buffer8_32_Bytes= byte ptr -4F4h
Handle_Dest_File= dword ptr -4D4h
Buffer7_16_Bytes= byte ptr -4D0h
Buffer2_32_Bytes= byte ptr -4C0h
Buffer3_16_Bytes= byte ptr -4A0h
Filename        = byte ptr -490h
Buffer4_284_Bytes= byte ptr -298h
NumBytesRead    = dword ptr -17Ch
Buffer5_372_Bytes= byte ptr -178h
var_4           = dword ptr -4
InputFile       = dword ptr  8
Password        = dword ptr  0Ch
```

Figure 12 : En-tête de la fonction réalisant tous les traitements cryptographiques

Le premier argument est le nom du fichier source. Le second est le mot de passe utilisé pour l'opération de chiffrement. Les variables locales ont été renommées par ordre d'apparition dans le code (ce que nous allons détailler plus loin). La taille de chaque variable a été ajoutée à la fin de chaque nom par souci de lisibilité.

Regardons maintenant l'enchaînement des appels afin de déduire l'architecture :

Le premier appel de fonction est le suivant :

```
lea     ecx, [ebp+Buffer2_32_Bytes]
push    ecx             ; Buffer2
lea     edx, [ebp+Buffer1_32_Bytes]
push    edx             ; Buffer1
mov     eax, [ebp+Password]
push    eax             ; Password
call    sub_401540
```

Figure 13 : Appel à la fonction de dérivation de mot de passe

Comme nous l'avons vu, **Password** est un argument donné à la fonction. Les deux autres arguments sont des variables non initialisées de 32 octets. Nous avons donc :

```
(Buffer1_32_Bytes, Buffer2_32_Bytes) = sub_401540(Password)
```

Il s'agit très certainement de la fonction s'occupant de la dérivation du mot de passe. Cette hypothèse pourra être validée (ou non) par la suite.

L'appel à cette fonction est suivi par :

```
push    10h
lea     ecx, [ebp+Buffer3_16_Bytes]
push    ecx
call    sub_4010A0
```

Figure 14 : Initialisation d'un buffer de 16 octets

Ici, la fonction prend deux arguments. Le second, **push 0x10**, est très certainement la taille du buffer passé en premier paramètre. Ce buffer est également non initialisé avant l'appel. À ce moment de l'analyse, il est impossible de formuler une hypothèse quant à son utilitéCet appel est suivi de :

```
mov     edx, [ebp+Handle_Dest_File]
push    edx             ; File
push    10h             ; Count
push    1               ; Size
lea     eax, [ebp+Buffer3_16_Bytes]
push    eax             ; Str
call    ds:fwrite
```

Figure 15 : Écriture d'un buffer de 16 octets

Nous avons donc une séquence qui revient à écrire dans un fichier un buffer de 16 octets qui ne dépend d'aucun paramètre. Grâce à ce second appel, nous pouvons supposer que **Buffer3_16_Bytes** est un paramètre nécessaire à l'opération de déchiffrement du fichier : il peut s'agir d'un sel ou d'un vecteur d'initialisation.

Les deux appels suivants sont semblables. Chacune des fonctions prend trois arguments : un buffer (que nous supposons être une dérivation du mot de passe vu précédemment), une taille qui semble être celle du buffer passé en argument (256 bits dans un cas et 32 octets dans l'autre cas), et enfin, un argument qui est un buffer relativement grand. La suite de l'étude pourra nous éclairer sur leur signification, mais comme nous l'avons vu dans la première partie, il pourrait s'agir d'une fonction d'initialisation de contexte. Ce contexte peut correspondre à un algorithme symétrique ou à une fonction d'authentification.

```
push    256
lea     ecx, [ebp+Buffer1_32_Bytes]
push    ecx
lea     edx, [ebp+Buffer4_284_Bytes]
push    edx
call    sub_4056B0
```

Figure 16 : Initialisation d'un contexte

```
push    32              ; Size
lea     eax, [ebp+Buffer2_32_Bytes]
push    eax             ; Src
lea     ecx, [ebp+Buffer5_372_Bytes]
push    ecx             ; int
call    sub_4013A0
```

Figure 17 : Initialisation d'un contexte

Vient ensuite la boucle principale. Son rôle est de lire 16 octets (si possible) dans le fichier source, d'ajouter du padding si nécessaire, d'appeler deux fonctions que nous allons détailler, puis d'écrire 16 octets (données chiffrées) dans le fichier destination.

```
.text:00401714 loc_401714:                             ; CODE XREF: MainCryptoFunctions+1F4↓j
.text:00401714                 mov     edx, [ebp+Handle_Src_File]
.text:0040171A                 push    edx             ; File
.text:0040171B                 push    10h             ; Count
.text:0040171D                 push    1               ; ElementSize
.text:0040171F                 lea     eax, [ebp+Buffer6_16_Bytes]
.text:00401725                 push    eax             ; DstBuf
.text:00401726                 call    ds:fread        ; read 16 bytes from InputFile
.text:0040172C                 add     esp, 10h
.text:0040172F                 mov     [ebp+NumBytesRead], eax
.text:00401735                 cmp     [ebp+NumBytesRead], 0
.text:0040173C                 jz      no_more_data    ; if no more bytes are available, exit loop
.text:00401742                 cmp     [ebp+NumBytesRead], 10h
.text:00401749                 jz      short _16_bytes_buffer
.text:0040174B ; add_padding
.text:00401779
.text:00401779 _16_bytes_buffer:                       ; CODE XREF: MainCryptoFunctions+169↑j
.text:00401779                 lea     edx, [ebp+Buffer7_16_Bytes]
.text:0040177F                 push    edx
.text:00401780                 lea     eax, [ebp+Buffer6_16_Bytes]
.text:00401786                 push    eax
.text:00401787                 lea     ecx, [ebp+Buffer3_16_Bytes]
.text:0040178D                 push    ecx
.text:0040178E                 push    10h
.text:00401790                 push    1
.text:00401792                 lea     edx, [ebp+Buffer4_284_Bytes]
.text:00401798                 push    edx
.text:00401799                 call    sub_4066B0
.text:0040179E                 add     esp, 18h
.text:004017A1                 push    10h             ; Size
.text:004017A3                 lea     eax, [ebp+Buffer7_16_Bytes]
.text:004017A9                 push    eax             ; Src
.text:004017AA                 lea     ecx, [ebp+Buffer5_372_Bytes]
.text:004017B0                 push    ecx             ; int
.text:004017B1                 call    sub_4014A0
.text:004017B6                 add     esp, 0Ch
.text:004017B9                 mov     edx, [ebp+Handle_Dest_File]
.text:004017BF                 push    edx             ; File
.text:004017C0                 push    10h             ; Count
.text:004017C2                 push    1               ; Size
.text:004017C4                 lea     eax, [ebp+Buffer7_16_Bytes]
.text:004017CA                 push    eax             ; Str
.text:004017CB                 call    ds:fwrite
.text:004017D1                 add     esp, 10h
.text:004017D4                 jmp     loc_401714
```

Figure 18 : Boucle de chiffrement

Regardons la première fonction : elle prend 6 arguments. Le premier ressemble à un contexte (**Buffer4_284_bytes**), le second est toujours à 1, le troisième est le buffer qui a été écrit dans le fichier avant de commencer, le quatrième correspond aux données claires, et enfin, le dernier va être sauvegardé plus loin : il s'agit des données chiffrées. Cette fonction réalise donc le chiffrement proprement dit.

La seconde fonction prend trois arguments : un contexte, puis les 16 octets qui viennent d'être chiffrés et la taille de ce buffer. Il s'agit sans doute d'une fonction type **HMAC**. En effet, nous avons vu que **Buffer5_372_Bytes** était initialisé par une fonction prenant un pointeur sur ce buffer ainsi qu'une dérivation du mot de passe. Ceci va être vérifié par les deux derniers appels qui se trouvent après la boucle de chiffrement :

```
lea     ecx, [ebp+Buffer8_32_Bytes]
push    ecx             ; Src
lea     edx, [ebp+Buffer5_372_Bytes]
push    edx             ; int
call    sub_4014C0
```

Figure 19 : Récupération de l'empreinte d'authentification

```
mov     eax, [ebp+Handle_Dest_File]
push    eax             ; File
push    32              ; Count
push    1               ; Size
lea     ecx, [ebp+Buffer8_32_Bytes]
push    ecx             ; Str
call    ds:fwrite
```

Figure 20 : Enregistrement de l'empreinte dans un fichier

Buffer8_32_Bytes correspond aux données chiffrées authentifiées par le mot de passe.

Arrivés à ce point, nous avons pu identifier l'architecture du logiciel (voir Figure 21).

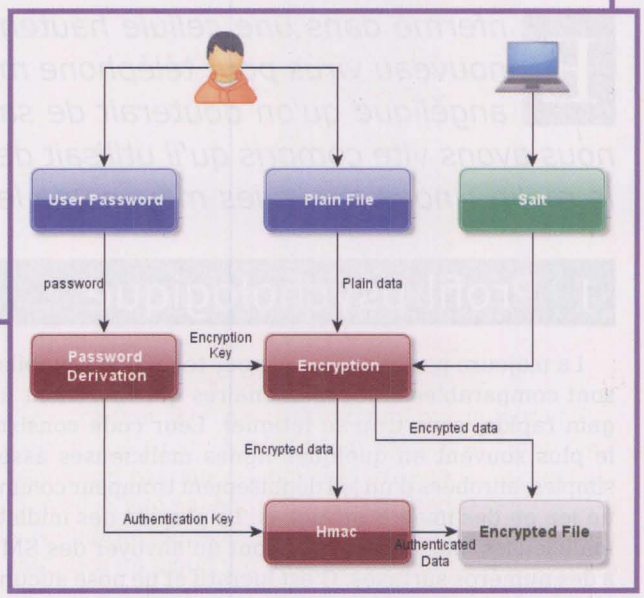


Figure 21 : Architecture du logiciel étudié

L'étude approfondie d'une des briques élémentaires nous aurait ramenés au cas vu dans la première partie de l'article.

Conclusion

La bonne connaissance des entrées/sorties des différents types d'algorithmes cryptographiques rend possible la rapide identification du rôle de chaque fonction. De même, appréhender les différentes implémentations des briques cryptographiques permet de déduire l'algorithme utilisé. Un analyste ne doit jamais hésiter à investir du temps pour comprendre les choix possibles effectués par le développeur de l'application qu'il reverse. Cet investissement se soldera la plupart du temps par un gain de temps. ■

NOTE

[1] Certaines implémentations reconstruisent ces tables à partir d'autres tables, la démarche est alors un peu différente, mais le principe reste le même.

DÉTENU VIRUS MOBILE :

NOUS AVONS LES MOYENS DE VOUS FAIRE PARLER !

Axelle Apvrille - Aapvrille@fortinet.com - Analyste Antivirus

mots-clés : *VIRUS / TÉLÉPHONE MOBILE / CRYPTOGRAPHIE / REVERSE ENGINEERING*

Enfermé dans une cellule hautement gardée, on vient de nous transférer un nouveau virus pour téléphone mobile. À première vue, il a une tête tellement angélique qu'on douterait de sa culpabilité. Mais derrière la vitre sans tain, nous avons vite compris qu'il utilisait de la... cryptographie. Il ne fera pas longtemps le malin : nous avons les moyens de le faire parler. Et sans torture matérielle.

1 Profil psychologique

La majeure partie des virus pour téléphones mobiles sont comparables à des mercenaires qui cherchent un gain rapide, sans trop se fatiguer. Leur code consiste le plus souvent en quelques lignes malicieuses assez simples, enrobées d'un joli déguisement trompeur comme un jeu ou des images érotiques. La plupart des midlets malicieuses, par exemple, ne font qu'envoyer des SMS à des numéros surtaxés. C'est lucratif et ne pose aucune difficulté de programmation :

```
String address = "sms://" + telNumber;
MessageConnection smsconn = null;
try {
      smsconn = (MessageConnection)Connector.open(address);

      TextMessage txtmessage = (TextMessage)smsconn.newMessage("text");

      txtmessage.setAddress(address);
      txtmessage.setPayloadText(text);
      smsconn.send(txtmessage);
}
```

L'envoi de SMS est le cœur de la partie malicieuse de Java/Konov.B!tr.

Un bon petit butin est vite ramassé, et sans trop d'efforts. Bien sûr, il existe quelques souches plus complexes, des sortes de tireurs d'élite dont on entend parler dans la presse, à l'instar d'Android/DroidKungFu **[1]**, qui embarque 1 ou 2 exploits pour « rooter » le téléphone victime. Ceux-ci sont heureusement encore relativement peu fréquents.

Que ce soit de petits mercenaires ou des tireurs d'élite, ils ont une caractéristique commune : ils ont un penchant pour les secrets et n'aiment pas exposer leurs intentions à tout va. Certains développeurs considèrent que leur façon de coder sera suffisamment obscure en elle-même, d'autres – de l'ordre de 20% tout de même - ont recours à la cryptographie.

2 Arme blanche

Un nombre non négligeable de nos « mercenaires » emploient l'arme blanche, c'est-à-dire une arme simple, efficace, mais peu sophistiquée. Ils utilisent alors les bases de la cryptographie dite « classique », telles que l'algorithme de Jules César. Certes, l'algorithme n'est pas solide au sens cryptographique, mais il est « efficace » dans le sens où un analyste perdra un peu de temps à réaliser de quel algorithme il s'agit, puis à déchiffrer.

Par exemple, Android/FakeNotify.A!tr.dial, un virus découvert en janvier 2012 qui envoie des SMS à des numéros courts, obfusque certains des mots-clés qu'il utilise à l'aide d'une substitution. Si l'on décompile son byte-code, on obtient quelque chose de semblable au code ci-dessous :

```
table = new Hashtable();
Character c1 = new Character((char)0x70); // key
Character c2 = new Character((char)0x23); // value
table.put(c1, c2);
c1 = new Character((char)0x2a);
c2 = new Character((char)0x34);
table.put(c1, c2);
...
```

Ainsi, le caractère p (0x70) sera substitué à # (0x23) et le caractère * (0x2a) à 4 (0x34) et ainsi de suite.

Petit à petit, on déchiffre ainsi le texte chiffré :

```
V     v{u-jZgu nZ {u--     V
```

en

```
*     WHELCOME TO HELL     *
```

Sur Windows Mobile, le virus WinCE/Sejweek (2009), lui, utilise une substitution de plusieurs caractères. Cela lui permet de cacher les numéros courts auxquels il souhaite envoyer des SMS.

```
public: static void __gc* FillCodeTable()
{
    Parameters::codeTable = __gc new ShifrDataTable();
    Parameters::codeTable->AddShifrRow(S"YGL", S"1");
```

```
    Parameters::codeTable->AddShifrRow(S"HKR", S"2");
    Parameters::codeTable->AddShifrRow(S"DPO", S"3");
    Parameters::codeTable->AddShifrRow(S"WHR", S"4");
    Parameters::codeTable->AddShifrRow(S"MKT", S"5");
  ...
}
```

WinCE/Sejweek est configurable à l'aide d'un fichier XML. Avec la configuration suivante, il envoie un SMS à 1151 (YGL=1, MKT=5...).

```
<xml version= "1.0 ">
  <getxml>
    <phone> YGLYGLMKTYGL </phone
  </getxml>
</xml>
```

Quant au détenu Java/Konov.S!tr, membre d'une famille prolifique de midlets malicieuses, il a eu l'esprit plus compliqué et s'est inventé son propre algorithme de chiffrement. Drôle d'idée ! Un cryptologue rirait ouvertement de son invention, mais il a au moins atteint un objectif, celui de faire perdre un peu de temps aux analystes antivirus...

```
public String decodeCes(String param)  {
  String str = "";
  char [] paramString = param.toCharArray();
  int i = (encryptSFrom.toCharArray()).length - 1;
  char [] enc = encryptSFrom.toCharArray();
  int j = paramString.length - 1;
  for (int l = 0; l <= j; ++l)
     {
    int k = -1;
    for (int i1 = 0; i1 <= i; ++i1)
       if (enc[i1] == paramString[l])
     {
         k = i1;
         break;
     }
    if (k != -1)
       {
     if (k == 0)
         k = i;
     else
         k -= 1;
     paramString[l] = enc[k];
       }
    str = str + paramString[l];
     }
  return str;
    }
```

Comment déchiffre-t-on du texte produit avec cet algorithme ? C'est en fait relativement simple. La clé étant en dur dans le code (**encryptSFrom**), on ne cherche pas à comprendre l'algorithme : on le reproduit dans notre propre code Java, et on l'appelle avec le texte chiffré en entrée... et on obtient le texte en clair en résultat, en l'occurrence une liste de numéros courts à qui envoyer un SMS avec le corps correspondant.

3 L'arme préférée

Parmi tous les algorithmes simples, il y a très visiblement un petit préféré. C'est le XOR (ou exclusif). Des plus faciles à coder, efficace (si la clé est aléatoire et aussi longue que le texte en clair, c'est même un algorithme parfait), le XOR a de quoi plaire. Il est utilisé par Java/Espaw.D!tr, Java/Swapi.AF!tr, Java/Konov.K!tr, WinCE/PmCryptic, SymbOS/Yxes!worm, SymbOS/Shurufa, SymbOS/ShadowSrv, SymbOS/Zhaomiao, Android/DrdDream, etc.

Identifier l'utilisation du XOR pour des virus Android ou des midlets malicieuses n'est pas très compliqué : il faut juste lire attentivement le code décompilé pour ne pas passer à côté. Le code ci-dessous est pris d'Android/DrdDream, un des premiers virus à utiliser un exploit pour rooter un téléphone :

```
public static void crypt(byte[] paramArrayOfByte)  {
   int i = 0;
   int j = 0;
   while (true)
   {
     int k = paramArrayOfByte.length;
     if (j >= k)
       return;
     int m = paramArrayOfByte[j];
     int n = KEYVALUE[i];
     int i1 = (byte)(m ^ n);
     paramArrayOfByte[j] = i1;
     i += 1;
     int i2 = keylen;
     if (i == i2)
       i = 0;
     j += 1;
   }
}
```

Ce code est issu d'un décompilateur, d'où son aspect un peu étrange. Néanmoins, on identifie facilement le ou-exclusif entre chaque caractère de **paramArrayOfByte** et chaque caractère de la clé **KEYVALUE** (ligne 11).

Sur Symbian, c'est un peu plus compliqué, car on ne dispose que de désassembleurs, pas de bons décompilateurs (N.B : Hex Rays propose un décompilateur assez pratique – et payant – mais généralement il faut encore retravailler le résultat pour qu'il soit vraiment lisible). Il faut donc relire l'assembleur ARM. Dans SymbOS/Yxes.E!worm (2009), un des premiers vers pour les versions récentes de Symbian, le code suivant est appelé pour chaque caractère d'une chaîne à déchiffrer.

```
LDR  R0, [R11, #buffer]
LDR  R1, [R11, #position]
BL  analyst_AtC
MOV  R4, R0
LDR  R0, [R11, #buffer]
LDR  R1, [R11, #position]
BL  analyst_AtC
LDRB R2, [R0]
LDRB R3, [R11, #key]
EOR  R3, R2, R3
STRB R3, [R4]
LDR  R3, [R11, #position]
ADD  R3, R3, #1
STR  R3, [R11, #position]
B  test_Loop
```

Tout d'abord, le code lit l'adresse du caractère de la chaîne « buffer » qui se trouve à la position « position » (lignes 1 à 3). Le résultat est mémorisé dans un registre temporaire R4 (ligne 4). Ensuite, on relit à nouveau la même adresse (pas très optimisée) de la ligne 5 à 7. L'adresse se trouve dans le registre R0 (et dans R4). À la ligne 8, on lit le contenu de cette adresse et on le stocke dans R2. LDRB signifie *Load Byte*, c'est-à-dire la lecture d'un seul octet. Les crochets disent de lire le contenu d'une adresse. Donc, dans R2 se trouve maintenant le caractère qui est à la position « position ». À la ligne 9, on fait pareil pour charger la clé. Notons que dans ce virus, la clé n'est pas une chaîne mais un seul octet. À la ligne 10, EOR correspond à l'opération du ou-exclusif :

c'est donc là que s'effectue le déchiffrement. Puis on va écrire le résultat à la place du caractère déchiffré, dont on avait soigneusement conservé l'adresse (ligne 11). Enfin, on incrémente le compteur « position » (lignes 12 à 14) et on saute (B veut dire *Branch*) à la routine `test_Loop` qui va vérifier si on a tout déchiffré ou pas.

C'est donc un peu plus difficile à suivre pour celui qui ne connaît pas bien l'assembleur, mais tout s'explique. Pour faire « parler » le détenu SymbOS/Yxes.E !worm, il faut trouver la clé. Dans ce cas spécifique, une clé d'un seul caractère n'est pas bien difficile à casser et on peut utiliser l'outil XORSearch **[2]**. Si la clé est plus longue, il y a d'autres solutions. On peut examiner les routines qui appellent la routine de déchiffrement à la recherche d'une constante (la clé) qui serait passée en paramètre, ou on peut faire tourner le virus dans un débogueur pas à pas et inspecter la valeur contenue dans R3 à la ligne 9. Peu de détenus résistent au désassemblage patient !

4 L'artillerie lourde

Sur Symbian, le support de la cryptographie n'était pas évident, ce qui pourrait expliquer que les virus utilisent des algorithmes simples (arme blanche). En revanche, sous Android, on a directement accès aux API crypto de Java, et par conséquent, on voit fleurir toute une série de virus qui utilisent des algorithmes modernes tels que l'AES (arme « lourde »).

Cependant, à la surprise peut être des cryptologues, il n'est pour l'instant pas vraiment plus difficile de faire parler un virus qui utilise un algorithme simple qu'un autre qui utilise un algorithme moderne.

Pourquoi ? Parce que les auteurs de virus ont des difficultés à cacher la clé qu'ils utilisent d'une part, et l'algorithme de cryptographie d'autre part. Que l'algorithme soit simple ou compliqué, la façon de procéder est la même :

- Comprendre que le code de virus utilise de la cryptographie.
- Repérer et identifier l'algorithme utilisé.
- Parmi les appels à l'algorithme, chercher la clé et ce qui est chiffré. La clé est souvent en dur dans le code, ou cachée dans une ressource.
- Écrire un programme à part : copier-coller l'algorithme, la clé et le texte chiffré. Puis appeler l'algorithme pour déchiffrer.

Du moment que la clé est contenue quelque part dans le virus et n'est pas échangée avec un serveur distant, utiliser de l'AES, une simple substitution ou un algorithme « maison » est du même ordre de complexité. La substitution ou l'algorithme « maison » sont efficaces dans le sens où ils font perdre du temps à l'analyste. C'est comme si notre détenu avait un attaché-case verrouillé avec lui. Si la clé est cachée sur lui, nous aurons vite fait de la trouver, et que le verrou soit simple ou sophistiqué, cela ne fera aucune différence pour nous à l'ouverture. En revanche, si la clé est cachée ailleurs, il faudra sans doute interroger le détenu ou le filer pour découvrir la cachette.

Par exemple, dans Android/DroidKungFu.A!tr, on repère :

```
private static byte[] defPassword = { 70, 117, 99, 107, 95, 115, 69, 120, 121,
 45, 97, 76, 108, 33, 80, 119 };
   ...
public static byte[] decrypt(byte[] paramArrayOfByte)  throws Exception   {
    byte[] arrayOfByte = defPassword;
    SecretKeySpec localSecretKeySpec = new SecretKeySpec(arrayOfByte, "AES");
    Cipher localCipher = Cipher.getInstance("AES");
    localCipher.init(2, localSecretKeySpec);
    return localCipher.doFinal(paramArrayOfByte);
  }
```

Nul besoin d'être un expert Java pour comprendre que ce cheval de Troie utilise l'algorithme AES, et que la clé de chiffrement est `defPassword`.

Il suffit alors de récupérer toutes les données qu'on souhaite déchiffrer et d'écrire son propre code Java pour déchiffrer en utilisant le copier-coller du code de portions du code du virus. Bien sûr, il est à prévoir dans l'avenir que les auteurs de virus cacheront mieux leur clé...

5 Révélations

Les détenus parlent... mais que révèlent-ils ? Dans la plupart des cas, ils cachent les numéros courts et corps des SMS qu'ils envoient, ou l'URL du serveur distant qu'ils contactent. L'objectif est sans doute d'une part de cacher le caractère malveillant du code (si un analyste ne voit rien de suspicieux au premier abord, il est possible qu'il l'écarte et passe au suivant) et de rendre l'analyse un peu plus compliquée. Car, en effet, si l'artillerie lourde n'est guère plus difficile à déjouer que l'arme blanche, dans les deux cas cependant, l'analyste perd un peu de son précieux temps à comprendre ce qu'il se passe.

S'ils ne chiffrent pas les URL ou numéros courts, les détenus chiffrent parfois les noms des variables, mots-clés, noms de fichiers ou même les exploits qu'ils utilisent. Cela leur permet de passer plus inaperçus et de rendre la tâche plus longue pour l'analyste. Imaginons par exemple qu'on détecte le condensat `sha1` de l'exploit CVE-2009-1185, un exploit qui permet de passer localement `root` sur un téléphone Android. Alors, dès que ce condensat `sha1` est repéré, l'alerte est donnée. Mais avec Android/DroidKungFu.A!tr qui chiffre l'exploit, le condensat est différent, et une telle détection ne fonctionne pas.

D'autres détenus, moins nombreux, se servent de la cryptographie pour chiffrer leurs communications avec le(s) serveur(s) distant(s). Encore une fois, l'objectif est de rendre l'analyse un peu plus compliquée, mais il y a une autre utilité pour les *botnets* : l'auteur du botnet peut conserver le contrôle de son réseau sans qu'un concurrent vienne lui voler ses victimes ou trésors.

Sachant que la cryptographie est facile à utiliser sous Android (utilisation des API Java), que les auteurs de virus ont derrière eux l'expérience des virus pour PC, il y a fort à parier que l'utilisation de la cryptographie ne fera que s'amplifier sur les virus pour téléphones portables dans les années à venir. ■

■ RÉFÉRENCES

[1] Descriptions techniques des virus mentionnés dans cet article : http://www.fortiguard.com/antivirus/mobile_threats.html

[2] XOR Search, http://blog.didierstevens.com/programs/xorsearch/