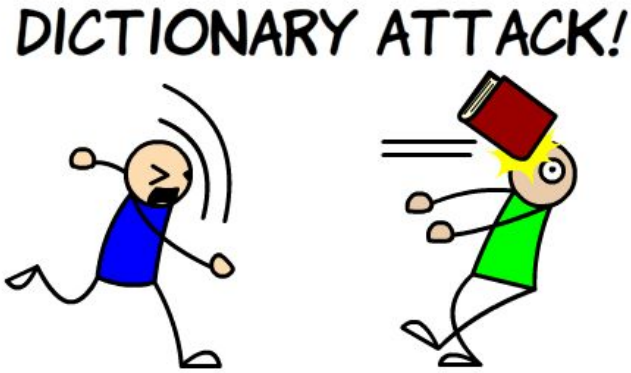
Cryptographie

Notes de cours

TLS-SEC 2015 - 2016



TLS-SEC

Cours : Carlos Aguilar Melchor

Notes de cours : Matthieu Vidal & Julie Demianenko

Cryptographie

Chapitre 1 - Le chiffrement symétrique

Rappels du cours de 2A

Propriétés associées à l'information (CID) :

- Confidentialité : Ne pas être accessible aux personnes non autorisées (associé à la notion d'authentification).
- Intégrité : Ne pas être modifiable par des personnes non autorisées. Peut être implicite ou explicite.
- Disponibilité : Peut aussi être associé à un service. Être accessible aux personnes autorisées (éviter par exemple les dénis de service).

Mécanismes associés :

– Authentification : Procédure permettant de valider une identité (nom, rôle, nationalité..)
– Autorisation : Politique d'accès à des ressources
– Traçabilité/Accounting : Log pour savoir ce qui a été consommer (tarification) ou fait (en cas de problème).
– Auditabilité : Analyse post mortem, grâce à la traçabilité.

Menaces dans les réseaux informatiques :

– Écoute : L'intérêt peut être le contenu ou l'obtention d'information pour mener une attaque ultérieurement (peut être un moyen ou une fin).
– Contournement des moyens d'authentification / d'autorisation : login/mdp faciles à obtenir par écoute, ou à deviner.
– Déni de service : Si le système d'authentification / autorisation est défaillant → fallback. Solution de repli peut être moins faible.
– Types d'attaquants : Souvent interne (80 % du temps). → Se protéger contre les attaquants internes.

I. Introduction

La cryptographie n'est pas le saint Graal de la sécurité. En effet, la cryptographie n'est qu'un outil pour la sécurité mais peut aussi avoir ses failles et surtout ses limites. La cryptographie n'apporte de la sécurité que si elle est bien utilisée : bon aléa, bien homogénéiser la sécurité sur l'ensemble du système pour ne pas avoir de maillon faible…Le problème de chiffrement viendra principalement de l'humain ou d'un autre maillon du système.

Ce que la cryptographie est :

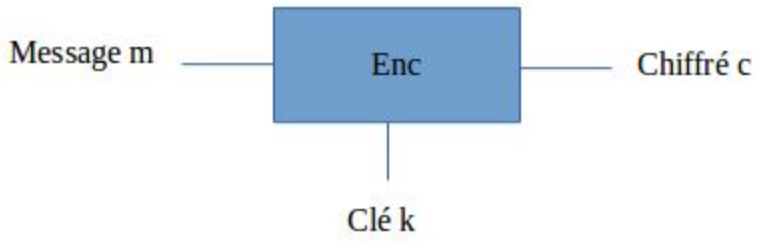
- Un outil indispensable.
- La base de nombreux protocoles de sécurité

Ce qu'elle n'est pas :

- Ne pas implémenter sa propre cryptographie, même les experts font des erreurs (des erreurs ont été trouvées dans openSSL il y a encore un an).
- Mettre de la cryptographie ne signifie pas que c'est sûr. En effet certaines attaques portant sur des systèmes cryptographiquement de haut niveau peuvent être menées. Un des cas les plus connus est l'attaque par canaux cachés.

Source : Wikipédia

Dans le domaine de la sécurité informatique, une attaque par canal auxiliaire (en anglais : Side channel attack) désigne une attaque informatique qui, sans remettre en cause la robustesse théorique des méthodes et procédures de sécurité, recherche et exploite des failles dans leur implémentation, logicielle ou matérielle. En effet, une sécurité « mathématique » ne garantit pas forcément une sécurité lors de l'utilisation en «pratique».



Si on a accès au temps que la boite « Enc » met à calculer ~~alors l'attaquant peut avoir~~ le chiffré. On pourrait aussi avoir la consommation de courant d'un terminal, etc.. Souvent, quand on a

le temps, la consommation, etc.. on peut obtenir des informations sur k et surtout m. Il faut donc faire attention à ce que le dispositif se comporte de manière constante, quel que soit les calculs effectués.

- Dans tous les cas il faut utiliser les librairies standards.
- Elle doit être utilisée correctement : aléa correct, fonction de chiffrement correcte, nombre de bits de clé adapté à la sécurité des autres composants (pas de maillon faible), etc..

Quiz du Dr Evans : La cryptologie est la science du secret quel qu'il soit (réponse : 1234).

- Clé pour la porte : le secret est « Quelle est cette forme ? ». La clé est reproductible à partir d'une photo, brute-forçable et surtout facilement crochetable → 5 positions possibles sur chacun des 5 barillets par exemple. Donc au lieu d'essayer 5^5 clés, on peut faire 25 essais de crochetage au total. Beaucoup de possibilités (5^5) mais on peut énormément réduire le champ de recherche.
- Poker : secret.
- Site internet : Mot de passe. Le logiciel Pwgen, prend une entrée et la transforme en un truc illisible. Les mots de passe en cryptologie sont faciles pour un individu, plus difficile en groupe. Comment on le gère dans une entreprise ? Peut vite devenir difficile.
- Recherche Google : Mots clés secrets. Utilisation de https partout. Le protocole https ~~permet de~~ consiste à faire passer l ensemble des communications en chiffré à travers un tunnel TLS. ~~xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx~~ le traffic HTTP

II. Notations

Tirages

Définition : Une distribution χ est une fonction qui à chaque élément de l'ensemble S associe une probabilité.
La somme des éléments de l'ensemble S vaut 1 et le couple (S, χ) forme un ensemble probabilisé.(simplification)

On note $x \xleftarrow{\chi} S$ ou $x \leftarrow (S, \chi)$ la définition de x comme le tirage d'un élément de S en suivant la distribution de probabilité χ

NB : x est tiré de l'ensemble S en utilisant la loi de probabilité de χ. Dans certains cas on simplifie et on ne note pas la distribution.

Attention pour l'union de deux ensembles
Pour A = {a, b} et B = {b, c}
Généralement : $\chi(A \sqcup B) \neq \chi(A) + \chi(B)$
Mais $\chi(A \sqcup B) = \chi(a) + \chi(b) + \chi(c)$
On fait la somme des éléments, pas des ensembles

Fonctions et algorithmes

Algorithmes déterministes

Pour une entrée, soit il existe une unique sortie, soit il n'y en a pas . On utilise la notation fonctionnelle f(x) = y.

Par exemple :

```
In x,y
Out x+y
```

Pour une entrée, il existe une seule sortie. On peut noter A(x) = y ou A(x) = {y}. La seconde notation se comprend comme la probabilité sur une entrée est certaine.

Algorithmes randomisés

Pour une entrée x il existe un ensemble de résultats possibles (S, appelé ensemble des sorties) et une distribution (χ, appelée distribution de sortie) associant à chaque élément une probabilité.

Exemple :

```
In n
Out Rand(1,n)
```

A l'inverse des algorithmes déterministes, les algorithmes randomisés associent à une entrée x un ensemble probabilisé A(x). Ainsi, alors que précédemment A(x)={y} et donc que la probabilité pour une entrée est certaine pour la sortie, ici une meme entrée donne un ensemble de résultats qui suit la loi de distribution χ. Ainsi, « y appartient à A(x) » signifie que y est une sortie possible de A(x), c'est une affirmation, pas une définition.

Composition :
Appliquer g à l'ensemble obtenu par f : Dans ce cours on ne tient pas compte des problème d'ensembles infinis, discrets, continues...

III. Bases sur le chiffrement symétrique

Définitions : Important

Il faut faire très attention à bien maîtriser la terminologie :

- Clair ↔ Plaintext : message non chiffré.
- Clé symétrique (ou clé de session) != clé secrète. L'expression "clé secrète" signifie juste qu'elle n'est pas publique puisque de toute façon, une clé doit généralement être secrète.
- On dit chiffrer / Déchiffrer et **PAS** crypter / décrypter.
- Termes anglais : symetric key, encryption, decryption, cipher, decipher
- Canal sûr : Un canal est un *moyen* de communication. On le considèrera comme sûr si un attaquant ne peut pas écouter ou modifier les clés qui passent. Par exemple : bouche à oreille, clé USB en main à main, communication quantique (toute écoute est détectable), carte SIM…Un canal peut donc etre quelque chose qui ne bouge pas et le transit de l'information n'est pas dans l'espace mais dans le temps. Il peut aussi bien être un canal de transmission "classique" que le dépot d'une donnée sur un disque dur (auquel on accède à des périodes différentes, qu'on déplace, …). Quand on pense à un canal il faut aussi penser au stockage.
- Canal non sûr : pour envoyer/stocker les données (wi-fi, internet, téléphone, etc...).On peut communiquer de manière sûre dans un canal non sur. Il suffit de s'échanger une clé par un canal sur, et on peut ensuite communiquer via n'importe quel canal

La cryptographie sert à communiquer de façon sûre sur un canal non sûr, en utilisant une clé.
Le transit de l'information ne se fait pas forcement dans l'espace, il peut aussi se faire dans le temps (je met un document sur un disque accessible, mon ami y accède plus tard).

Le chiffrement

Il est très difficile de cacher l'existence d'une communication. Une technique extrêmement coûteuse revient à émettre de très petits bouts de communication sur des fréquences porteuses différentes et de manière très brève. Elle est réservée à des messages courts (positionnement de personnel dans le domaine militaire par exemple,...) et pour des situations très particulières.

Source : Wikipédia

*L'**étalement de spectre par saut de fréquence** est une méthode de transmission de signaux par ondes radio qui utilise plusieurs sous-porteuses réparties dans une bande de fréquence selon une séquence pseudo-aléatoire connue de l'émetteur et du récepteur. Cette technique a trois avantages :*

1. *Rendre le signal transmis très résistant aux interférences,*
2. *Le signal est plus difficile à intercepter,*
3. *Les signaux transmis de cette manière peuvent partager des bandes de fréquence avec d'autres types de transmission, ce qui permet d'utiliser plus efficacement la bande passante*

Il est aussi parfois difficile mais nécessaire de cacher la taille et le moment de l'émission. Pour cela une méhode de padding peut être mise en place. Cette méthode fut utilisée au cours de la seconde guerre mondiale: les résistants faisaient des communications permanentes et fixes pour cacher les pics de communication.

Par exemple on peut faire envoyer des mails tous les matins (chaque mail avec la taille maximum). Pour répondre réellement on doit toujours attendre le matin, même si on envoie rien on envoie que du padding avec la taille maximum. Mais dans certains cas (bande passante couteuse, etc) il peut etre impossible de cacher ces communications. Les envois peuvent aussi être brouillés à l'aide de décalage dans le temps des paquets émis.

Le padding peut être fait avec juste des 0 mais on peut aussi en profiter pour mettre de l'aléatoire, comme ça même si la fonction de chiffrement est déterministe, on a un peu d'aléatoire quand même. Cet aléa sera très destructeur même pour un message très structuré.

Attention, le chiffrement n'assure pas que le message n'a pas été modifié. Il faut donc ajouter par dessus un contrôle d'intégrité. La maléabilité d'un chiffrement permet de modifier le sens d'un clair sans connaître la clé secrète, en modifiant le chiffré. En effet si on connaît la structure du chiffré, on peut faire des collages entre plusieurs chiffrés sans qu'on ai besoin de les déchiffrer.

Mais même sans toucher au chiffré, des attaques sont possibles. On n'est par exemple pas protégé contre le rejeu (l'attaquant renvoie un message qu'il a déjà enregistré) ou encore le changement de destinataire (si il y a une communication chiffré entre A et B, alors l'attaquant va récupérer les messages et les envoyer a C qui connaît aussi la clé secrète, sans déchiffrer le message et à l'insu de A et B). Les rejeux sont très dangereux et facilement utilisés.

Si on veut être sûr que l'interlocuteur à dit quelque chose, il ne faut donc pas de chiffrement mais du contrôle d'intégrité (marquage d'horloge, numéro de séquence,…autant de mécanismes qui doivent venir compléter les méthodes de chiffrement)

.

De plus on ne peut juste pas tout chiffrer ! Meme si j'envoie un code hyper secret à un ami par la poste, je ne peut pas chiffrer son adresse car les intermédiaires doivent pouvoir la lire. De même, on peut chiffrer une boite mail intégralement, mais si on veut faire une recherche dedans, il faudra tout télécharger (les 30 Go éventuels), déchiffrer puis chercher. Ce n'est pas très pratique. En

plus de ne pas pouvoir chiffrer les adresses de destination, le chiffrement n'est pas adapté à toutes les applications.

Principe de Kerckhoff

Plus compliqué le secret résidait dans le secret des rotors et c'est le boitier des rotors qui était toujours détruit avant l'arrivée des alliés. En fait ça ressemble à une clé.

Les machines Enigma utilisées par les nazis pendant la guerre étaient très protégées , on a jamais mis la main sur une machine Enigma. Le secret résidait dans l'algorithme de chiffrement. En revanche, avec des clés, le secret réside dans une clé, pas dans un algorithme. Quand une clé est dévoilée, ça n'affecte que la personne pour qui la clé est dévoilée.

Aussi, on s'est dit, pourquoi avoir besoin de la même clé pour chiffrer et déchiffrer. On s'est rendu compte que ce n'était pas une obligation. Dans les années 70, on a réalisé qu'on pouvait faire des algorithmes pour générer deux clés, une privée et une publique. On pouvait ainsi dévoiler l'algorithme de chiffrement et la clé publique sans que pour autant on puisse avoir des renseignement sur la clée privé. C'est l'émergence du chiffrement asymétrique.

Le chiffrement symétrique

- Chiffrement d'un clair m : $c \leftarrow Enc(sk, m)$
- Déchiffrement d'un chiffré c : $m \leftarrow Dec(sk, c)$
- Consistance : $\forall m, sk : Dec(sk, Enc(sk, m)) = \{m\}$
- Sécurité : les chiffrés ne dévoilent rien sur les messages ou la clé

Le chiffrement symétrique a deux propriétés : la **consistance** et la **sécurité**

- **Consistance** : Quand on applique le fonction de déchiffrement, il doit y avoir un seul résultat possible. Quelque soit le chiffré qu'on obtient à partir du clair m, lorsque l'on applique l'algorithme de déchiffrement, on obtient un et un seul résultat qui est m.

- **Sécurité** : Le chiffré ne doivent rien dévoiler sur le clair associé ou la clé. En pratique : ce qu'un attaquant peut déduire des chiffrés n'est pas exploitable en un temps raisonnable pour obtenir quelque chose d'intéressant.

Le quiz du Dr Evans : 12

NB : On peut randomiser la fonction de chiffrement en ajoutant du padding aléatoire après le clair. Dans ce cas un clair peut donner différents chiffrés.

IV. Le One Time Pad

Cours de 2ème année :

OTP consiste à utiliser un masque à usage unique, de la taille du message et fonctionne tel

que : message XOR masque = chiffré. Le masque suit une distribution uniforme, le chiffré aussi (résultat = bruit blanc).

Preuve :

P(c=1) = P(n=1)P(k=0) + P(n=0)P(k=1) = 1/2 [P(n=1) + P(0=1)] = 1⁄2

Le message codé ne donne aucune information sur le message initial, tous les messages sont possibles et équiprobables. Le problème est que le masque doit être de la taille du message. De plus si on code deux messages avec la même clé : c1 XOR c2 = ... = m1 XOR m2. Ceci donne plein d'information sur les messages. Ce chiffrement est quand même utilisé.

OTP utilise la propriété du modulo 2 grâce à laquelle y+x = y-x. Finalement, c'est le seul chiffrement symétrique parfaitement sûr, même avec une puissance de calcul énorme, c'est incassable.

Quiz : Quelle propriété on peut souhaiter ? 2.

Y chiffré vu par l'attaquant, X clair associé

1. $\forall m, c : P[X = m | Y = c] = 1/|M|$
2. $\forall m, c : P[X = m | Y = c] = P[X = m]$
3. $\forall m, c : P[X = m] = P[Y = c]$
4. $\forall m, c : P[X = m | Y = c] = 1/|K|$

1. Dépend du générateur de messages (M = espace des messages)
2. Idée d'apprentissage. En regardant le chiffré, l'attaquant n'apprend rien. Evenement indépendants (d'après la définition mathématique).

Question :

Combien de clés sont possibles pour $X = i \cap Y = c$?

Un nombre 1,2,3,4,..
Une gamme 0-1, 0-K, 1-K

Réponse : 1.
En effet, en connaissant c et i (i xor i = 0)

c = i + k

i + c = k

Autre question :

L'OTP est-il parfaitement sûr ?

$P[A|B] = P[A \cap B]/P[B]$
A, B indépendants $\Leftrightarrow P[A] = P[A] * P[B] \Leftrightarrow P[A|B] = P[A]$
$P[B] = \sum P[A_i \cap B]$ pour A_i un partitionnement de Ω

$P[X = m|Y = c] = P[X = m \cap Y = c]/P[Y = c]$
$P[Y = c] = \sum_{i \in M} P[X = i \cap Y = c]$
$P[Y = c] = \sum_{i \in M} P[X = i \cap K = c \oplus i] = \sum_{i \in M} P[X = i]/|K| = 1/|K|$

Que vaut $P[X = i \cap K = c \oplus i]$?

1. $1/|K|$
2. $1/(|M| * |K|)$
3. $P[X = i]/|K|$
4. $P[X = i]$

Il faut bien comprendre que la cryptographie revient a dire « je fais confiance à ce mode de chiffrement car si quelqu'un arrivait à casser cet algorithme, alors il résoudrait un problème sur lequel des milliers de personnes se sont penchées sans y arriver ».

L'OTP possède tout de même des limites telles que la **malléabilité**. En effet, un attaquant peut intercepter un message chiffré et l'altérer en y ajoutant un masque arbitraire et donc, modifier précisément certains bits et ainsi donner un autre sens au message (transformer un oui en non par exemple) ou tout simplement rendre le message irrécupérable.

De plus, la **clé peut se réveler** par elle même : en effet, si on envoie un message qui est une suite de 0, alors on envoie la clé en clair. Aussi, la clé ne doit surtout être utilisée qu'une seule fois ! En effet, si une clé K est réutilisée, on peut utiliser les cryptés issus de ces deux émissions pour retrouver le clair (m1+k+m2+k=m1+m2).

chiffrés

Notons aussi que la clé doit avoir la **même taille que les données** à chiffrer et que l'**aléa est coûteux** autant à générer qu'à partager. En effet il est contraignant de devoir utiliser un canal sûr pour se transmettre la clé, qui fait la même taille que les données.

Le One Time Pad est donc utilisé en pratique mais de manière limitée (un CD dans un satellite en cas de secours par exemple).

Théorème de Shannon

Si on veut que le système de chiffrement soit parfaitement sûr, si on veut que la connaissance du chiffré n'apporte rien sur la connaissance du message alors il faut K ≥ M (espace des clés plus grand que l'espace des messages).

Démontrons le par l'absurde : Supposons un système où K ≤ M (l'espace des clés est plus petit que l'espace des messages). Supposons un attaquant qui voit passer un chiffré et qui essaye toutes les clés (K clés). En utilisant le fait que les fonctions de chiffrement / déchiffrement donnent un unique résultat au final, après K itérations, il récupère K possibilités. Or K ≤ M donc l'attaquant apprend que le message n'est pas à trouver parmi M mais parmi K, c'est à dire que l'ensemble des M messages possibles pour ce chiffré à été réduit (certains messages sont impossibles). Le chiffré donne donc des informations à l'attaquant ! Par itérations, il pourra à long terme casser le système.

(un peu plus compliqué ça dépend du système)

Plus formellement :

Supposons par l'absurde que $|K| < |M|$ et qu'un attaquant étudie $Y = c$
Si il calcule $M_0 = \cup_{k \in K} Dec(k, c)$ on aura $|M_0| < |M|$
$\Rightarrow$ pour $m \notin M_0$, $P[X = m \mid Y = c] = 0$ même si $P[X = m] = 1/|M|$

Conséquence :
Soient | | | | | | | | | ... | | | | → M messages équiprobables de probabilité 1/|M|
Si certains sont impossibles, la probabilité des autres montent. De plus très souvent on peut itérer et converger vers le bon clair.

(trop peu d'informaion pour être mis comme ça ici)

Nous pouvons aussi introduire une propriété qui transmet bien la notion de sécurité : l'analyse d'un chiffré ne doit pas donner d'informations sur le clair.

L'uniformité écrase la singularité

Soit Z, la somme de deux variables aléatoires Y et X, avec X suivant une loi de probabilité uniforme et Y suivant une loi de probabilité quelconque. Alors, si et seulement si Y est indépendant de X, peut importe la distribution de X (sa singularité), Z sera uniforme.

```
p(Z=1)      =     P(x=1 ∩ y=0) + p(x=0 ∩ y = 1)
            =     P(x=1)p(y=0) + p(x=0)p(y=1)
            =     1/2 [p(x=1)  + p(x=0)]
```

Il faut s'assurer que le générateur d'aléa peut donner deux séries d'aléa indépendantes. Ainsi si il en est pas capable, il faut utiliser deux générateurs différents pour générer X et Y. Mais ce n'est pas forcément suffisant.Il y a déjà eu des attaques importantes il y a 2/3 ans sur RSA. L'un des créateurs s'est rendu compte que 13% des clés RSA utilisées dans le monde étaient non sûres. Tous les générateurs d'aléa Intel viennent de la même usine par exemple. En cherchant des

suivent un processus de fabrication similaire
.2% (j'avais dit un % complètement faux à l'oral)

corrélations entre les aléa générés par les différents générateurs d'Intel, ils en ont trouvé. Ça a créé des problèmes. Ils retrouvaient des chaînes d'aléa semblables.

xxxxxxxxxxxxxxxxx xxxxxxx
on peut trouver qu'ils sont pas indépendants

En pratique sans regarder d'où vient le problème si on regarde les clés RSA générées dans le monde on voit qu'il y a des similarités (facteurs communs) qui permettent de casser des clés.

Négociation

Supposons A et B qui veulent négocier une valeur commune uniforme, mais ils ne se font pas confiance car l'un d'entre eux peut pervertir le choix. Ils échangent des messages et à un moment ils veulent arriver à un numéro aléatoire commun pour commencer à séquencer les paquets. On peut pas dire toi vas y choisi un nombre aléatoire.

Solution : A tire d'un espace d'aléatoire un nombre en supposant une distribution quelconque. De même pour B, la même distribution ou pas on s'en fout. Si A à choisit une valeur uniforme ou si B l'a fait, le résultat est uniforme (après XOR). Peut importe ce que fait l'autre, le résultat sera uniforme si on envoie un truc uniforme. Ce mécanisme est bien dimensionné pour des échanges entre un opérateurs et un système: l'opérateur, qu'il soit attaquant ou non, pourra choisir ce qu'il veut, le système est dimensionner pour toujours choisir de l'uniforme et donc, le résultat sera uniforme.

Attention ! Ceci n'est vrai que si les choix sont indépendants donc si A donne son choix à B et B choisit en fonction de ce qu'à dit A ça ne marche pas. Comment faire que B ne choisisse pas en fonction du choix de A (ou viceversa) ? Voir plus loin les engagements avec une fonction de hachage.

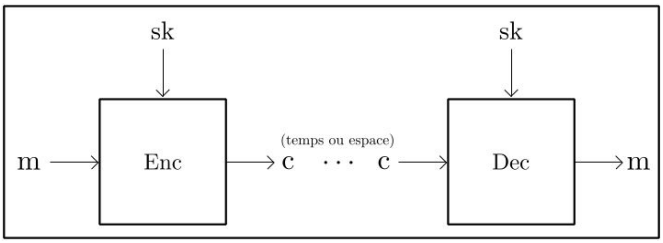
Cryptographie

Chapitre 2 - Le chiffrement par blocs

I. Introduction

Dans un algorithme de chiffrement symétrique la clé va peut être donner des informations sur le message, mais on se dit que ce n'est pas grave si ça en révèle très peu (pas d'infos utiles) et en demandant un temps de calcul énorme. De plus, le OTP impose que la clé soit uniforme, de même taille que le message et que la clé soit utilisée qu'une seule fois.

On peut donc décider de changer d'optique et d'accepter que C donne des informations sur M, mais que ces informations soient inutilisable. On part quand mémé du principe que cet attaquant est humain et a donc des moyens très limités. On souhaite alors ne pas avoir besoin que la clé trop contraignant : elle ne nous limite pas sur la quantité de message à envoyer tout en gardant cependant un lien entre la taille de la clé et le niveau de sécurité.

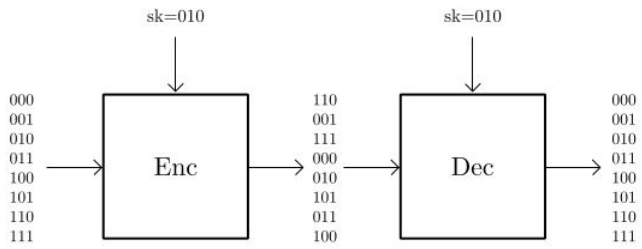


II. Le chiffrement pas blocs

Généralités

Le principe du chiffrement par bloc est d'avoir une clé de taille fixe K qui nous permettra de coder des blocs de messages. De nos jours k et au moins égal à 100 et généralement égale à 128 ou 256. On choisit ces valeurs pour éviter les attaques de Shannon, aussi appelées attaques par

force brute, qui reviennent à essayer toute les clés soit, pour les valeurs précédentes, en 2^128 ou 2^256 opérations.



Avec le système de chiffrement par blocs suivant, 2³ opérations sont nécessaires pour casser l'algorithme.

Prenons quelques unités de référence pour comprendre la complxité d'une telle attaque sur un chiffrement par bloc.

2^10 = 1024 → De l'ordre de 10³
2^128 → De l'ordre de 10^39
k = 30 → Un ordinateur fait environ 2^30 opérations par seconde
k = 55 → Ce que peut faire un ordinateur en 1 an
k = 85 → Un milliard d'ordinateur tournant pendant un an (un an Ordinateur Sur Terre)
k = 285 → Résiste à un calcul correspondant à une opération par atome dans l'univers depuis le début de l'univers. Il y a 2^40 atomes dans l'univers…

Ainsi pour une clé de 100 bits, il faudrait 32000 ans pour que tout les ordis sur terre fassent le calcul.

Fonctionnement

Padding :

En pratique comment utilise-t-on un chiffrement par bloc ? Soit un message en entrée de 4 bits et une clé de 128 bits. On va padder le message avec un padding que l'on sait reconnaître (en utilisant des standards tel que le Public Key Cryptographic Standards). Si le message est plus grand que la taille de la clé, on va le couper par bloc de 128 bits `M1,...,Mn` avec le dernier bloc potentiellement paddé. On va chiffrer des blocs et on obtiendra C1...Cn.

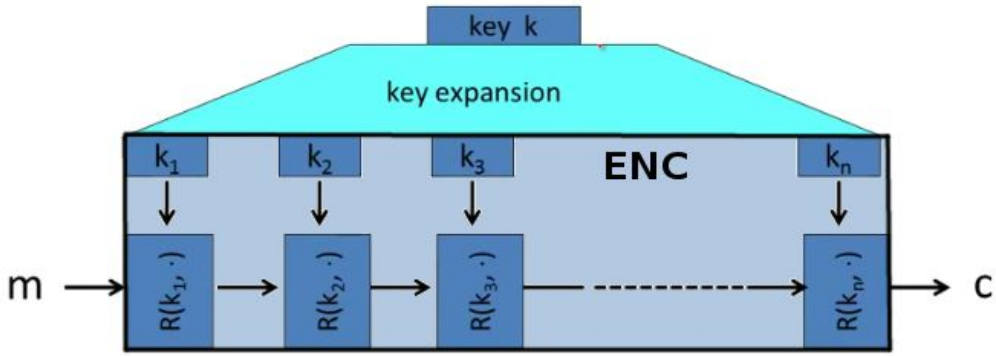
Structure :

Le chiffrement par bloc va se dérouler par étapes, appelés rounds. Chaque round i sera définit avec une sous clé Si. Cette sous-clé de k bits sera générée à partir de la clé (k bits aussi) et

d'une fonction de dérivation qui va fournir N sous-clés. Dans ce cas, on fera N rounds. Pour le chiffrement, on va récupérer le résultat du round i-1, le xorer avec une partie de la clé Ski,.
Fonction de chiffrement F :

```
Round i pour i de 1 à 20(nb de sous clés)
     Input   :     Output of round i-1 OU tout simplement le message si c'est le premier
round
    output :    k bits
```

Deux éléments importants sont donc la **fonction de dérivation** (pas dans ce cours) et la **fonction de chiffrement**.



Partout:
citer avec une URL
tous les bouts de texte
et les images prises
ailleurs

La fonction ENC

La fonction ENC est une permutation de $\{0,1\}^n$ inversible (⇔ une bijection dans un espace fini). On veut prendre une clé au hasard et que la permutation soit le plus aléatoire possible. Mais c'est impossible, car pour x éléments dans l'ensemble des messages, il y a 2^x permutations possibles et seulement k bits de clé pour les définir. Les permutations ne sont donc pas faites réellement au hasard. La fonction Enc, définit un petit ensemble de permutation possibles. Cependant il reste très difficile de savoir quelle permutation a effectuer la fonction.

La fonction ENC est construite en itérant une fonction de chiffrement (exemple avec AES)

La fonction de chiffrement est composée d'une étape permettant d'éviter les attaques algébriques (autrement les équations qui décrivent ça sont des équations de petit degré permettant des attaques) et d'étapes linéaires (permettant de "diffuser" l'aléa créé). On cherche ici à casser la linéarité qui pourrait potentiellement exister entre les octets. Ces cette linéarité qui rend la méthode de chiffrement sensible. Sans rentrer dans les détails :

- 1er Round : $m_i \oplus k_1$
- Puis on itère :
  - $C_{i-1} \oplus k_i$ pas nécessairement ça peut être plus compliqué "On mélange c_i avec des bits de la clé avec des opérations de ty type XOR"
  - Opération de substitution (substitution avec une S-boite : table de substitution inversible)

- Opérations linéaires : permutation

Plus d'information sur la S-boite : C'est une table de substitution inversible et publique. Cette table fait correspondre un octet avec un autre. La table prend par exemple en entré un bloc d'un octet. On regarde les 4 premiers bits, ils nous donne la ligne, les 4 derniers nous donnent la colonne. On prend l'octet situé aux coordonnées (ligne, colonne) obtenues et on remplace l'entré. Cette table doit être inversible. Dans AES on utilise plutôt une généralisation des S-boites : au lieu de choisir dans une table un octet correspondant, on choisit et on applique une matrice.

Pour rappel :

Définition d'une permutation pseudo-aléatoire (PRP)

C'est une famille de fonctions telle que le choix uniforme d'un élément est indistingable du choix uniforme d'une permutation parmi toutes les permutations possibles

Déchiffrer

Pour déchiffrer on va faire l'inverse de la permutation (par définition une permutation est inversible), en se référant à la table on remplace les octets par les bons et ensuite on xor avec la clé Ski.

Fonction de déchiffrement (fait des rounds à l'envers)

```
Round : pour i de 1 à 20
    Input : Sortie du round précédent (i+1) OU le chiffré initial de k bits
    Output : k bits
```

Propriétés

- Changer un bit dans la clé va intégralement changer les sous clés qui lui sont dérivées et donc intégralement changer le chiffré.
- Changer un bit dans le message change intégralement un octet du chiffré (utilisation de la S-boite). Dans un premier temps uniquement ensuite par le jeu des permutations et substitutions ultérieures on change tout autant l'ignorer
- Si on change un bit du chiffré, le déchiffrement ne marche pas du tout (globalement tout les systèmes de chiffrement ressemblent à ça. Le moindre changement dans le message provoque un changement radical dans le chiffré).
- Le nombre de round que l'on doit effectué est aussi très important : l'algorithme de chiffrement AES est cassable si il est fait avec moins de 10 tours (full AES). pour full AES il y a toujours plus de 10 tours

Sécurité

Il y a deux types d'attaquants :

- Celui qui dispose seulement de données chiffrées et essaie de deviner la clé.

- Celui qui dispose de texte clair / texte chiffré (soit par chance soit en s'étant fait envoyer un message connu exprès). Ce cas la est très probable !

Nous allons généraliser le cas de cet attaquant en supposant qu'au dela d'avoir accès à des couples clair/chiffré, il peut choisir ce couple. Ainsi un attaquant va choisir un ensemble de clair et va pouvoir observer les chiffrés associés.
Ainsi on considère qu'un bon système de chiffrement doit résister à ce dernier type d'attaquant.

Mais que cherche l'attaquant ?

- Avoir la clé ! Mais du coup est-ce une attaque de récupérer seulement quelques informations sur la clé ? Souvent en itérant, en cherchant un peu, on peut arriver à récupérer la clés entière. Typiquement, des algorithmes avec 128 bits de sécurité ont été énormément réduit.
- Distinguer un chiffrement d'une distribution aléatoire. En effet si on ne peut pas faire la différence entre un plaintext chiffré et un flux aléatoire alors les flux sont indistingables (rappel : le but d'un bon algo de chiffrement est de ressembler à une *permutation aléatoire*).

La notion d'**indistingabilité** est importante : cette notion permet de dire à quel point ce qu'on a créé est proche du modèle idéal. Si notre chiffré ne se comporte pas comme une permutation aléatoire alors notre système peut avoir des failles. On verra plus tard le lien entre indistingabilité et sécurité des applications.

On rappelle qu'un système de chiffrement est considéré comme sûr si toute attaque, même a chiffrés choisis, à une complexité de l'ordre 2^k pour récupérer la clé.

Algorithmes à connaître

- **DES** – Data Encryption Standard, proposé par le NIST (National Institute of Standard and Technology) en 1980. Il est complètement cassé. DES utilise des clés de 64 bits mais la fonction de dérivation de clé se fait toujours avec 56 des 64 bits. On peut énumérer relativement rapidement toutes les clés. Certaines attaques sont possibles en 2^48. Aujourd'hui c'est très rapide avec du bon matériel. DES n'est cependant pas complètement mauvais non plus, des idées ont été reprises plus tard.

- **3DES** – Prend 3 clés DES, chiffre avec k1, déchiffre avec k2, re-chiffre avec k3. 3DES est trois fois plus lent que DES. En pratique, 3DES a 112 bits de sécurité (et pas 3*56). La sécurtié a été réduite comme si il n'y avait que deux clés. Ce système a été très utilisé car DES était déjà implanté partout, du hardware spécialisé avait été déployé pour DES, on a pu le réutiliser.

- **AES** – Un concours du NIST a été ouvert à tous pour développer un nouvel algorithme. Les 15 premières soumissions ont été étudiées pendant des années, puis 5 ont été retenues, testées et mises à l'épreuve par la communauté. Les autres critères étaient : la facilité d'implantation software, hardware, etc. Le gagnant était Rjindael. Le système généralise les

S-Boites et utilise une taille de bloc constante (128bits, mais la clé peut avoir des tailles différentes) pour qu'une même implémentation software ou hardware puisse fonctionner pour tout. La meilleure attaque à ce jour est en 2^{k-2} donc théoriquement AES est cassé, en pratique non. En effet cette attaque requiert 2^{56} bits de mémoire ce qui est énorme (alors qu'une attaque par force brut prend 2^{k} opérations mais ne prend pas de mémoire.

en étant jusqu'au boutiste,

- RC5, Blowfish, IDEA…

Modes de fonctionnement

Dans le cas du chiffrement par bloc, au delà du choix de l'algorithme il est aussi nécessaire de choisir un mode de chiffrement. Le mode de chiffrement est la manière de traiter les blocs de texte clairs et chiffrés au sein d'un algorithme.
Jusqu'à maintenant implicitement on disait ECB (livre de code) :

Mode ECB (Electronic Code Book)

> ***Cours de 2A – Mode ECB***
>
> *Il s'agit du mode le plus simple. Le message à chiffrer est subdivisé en plusieurs blocs qui sont chiffrés séparément les uns après les autres. Le gros défaut de cette méthode est que deux blocs avec le même contenu seront chiffrés de la même manière. Le seul avantage qu'il peut procurer est un accès rapide à une zone quelconque du texte chiffré et la possibilité de déchiffrer une partie seulement des données.*

Le fait que deux blocs de clairs identiques soient chiffrés de la même manière pose un problème. En effet, la répétion d'un bloc n'est pas rare. Ceci est dû au paradoxe des anniversaire et aussi au fait que le clair est généralement un message structuré. Le début peut être des en-têtes TCP qui se ressemblent toutes entre elles par exemple. Ce n'est pas du tout une bonne propriété. Le mode ECB possède de plus des problèmes de malléabilité.

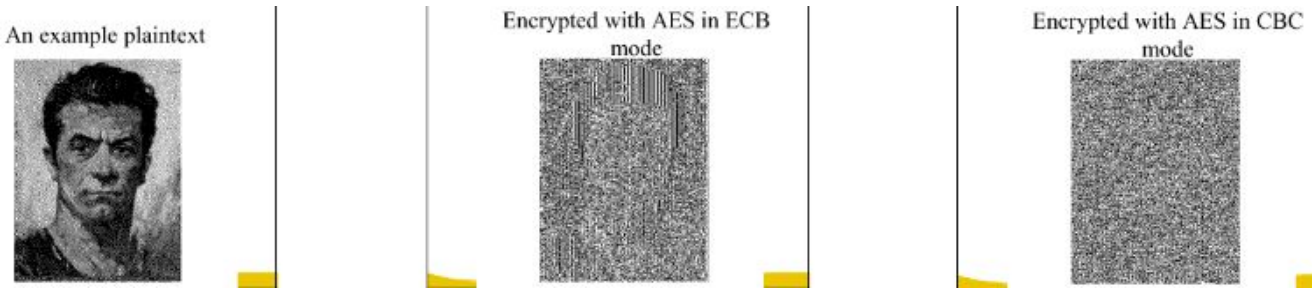
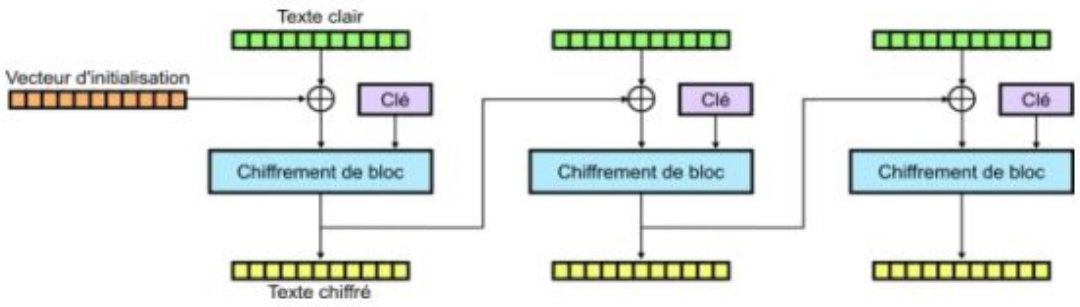


Image chiffrée avec AES en mode ECB : on distingue les cheveux qui sont tous de la même couleur. On reconnaît des choses. Par défaut on utilise donc plutot le mode CBC.

Mode CBC (Cipher Bloc Chain)

> ***Cours de 2A : mode CBC***
>
> *Dans ce mode, on applique sur chaque bloc un XOR avec le chiffrement du bloc précédent avant qu'il ne soit lui-même chiffré. De plus, afin de rendre chaque message unique, un vecteur d'initialisation (VI) est utilisé.*



Partout:
citer avec une URL
tous les bouts de texte et les images prises ailleurs

Dans ce mode ci, nous allons enchainer les blocs de chiffrés les uns aux autres de manière à créer une interdépendance entre ce qui a été dit dans le passé et ce qui sera dit dans le futur. Cette interdépendance nous permet de randomiser l'émission des chiffrés.

Le VI, ou vecteur d'initialisation, est un bloc de 128 bits généré à partir d'aléa pur ou d'un générateur pseudo aléatoire. C'est le premier bloc qu'on émet. Ensuite on XOR ce VI avec le premier bloc de 128 bits du message qu'on veut émettre. Ça donne un résultat qu'on va appeler R0 et on chiffre (AES 128) ce résultat. Le tout est appelé C0.

A ce stade on a dû émettre 2 blocs de 128 bits pour chiffrer 128 bits, c'est pas top. On peut donc envoyer un seul VI pour toute une série de blocs. Chaque chiffré est lié au précédent (mais on peut quand même déchiffrer n'importe quelle partie sans repartir du début). On utilise de l'aléa une seule fois, au début. Ainsi, si je fais exactement la même communication deux fois, ça donnera des chiffrés complètement différents car les VI seront différents (aléatoire). En effet, l'image d'un ensemble choisit uniformément au hasard (la communication est randomisée) par une permutation donne une distribution uniforme. Donc même si les messages sont structurés, puisque IV est choisi uniformément au hasard, le résultat est uniforme au hasard. Ceci peut permettre de faire un tunnel de communication (envoyer que des zéros en permanence et un jour mettre des vrais messages).

NB: Cet IV doit pouvoir être échangé entre les deux interlocuteurs. De plus, le chiffrement utilise tout de même une clé qui devra être partagée. Il faudra donc utiliser à un moment donné un canal sûr.

III. Formalisation

Pour une clé donnée, un chiffrement par bloc ressemble à une permutation aléatoire de l'espace des clairs dans l'espace des chiffrés. L'idée est de montrer qu'il n'y a pas de test qui permet de distinguer une permutation aléatoire d'un chiffrement par bloc : tout test utilisable et permettant de différencier deux sorties de l'algorithme prend un temps de test prohibitif. Cependant, quand on dit que deux choses sont indistingables il faut définir ce que ça veut dire.

La **complexité** d'un **algorithme efficace** : c'est la notion de temps polynomial. Un algorithme est polynomial (en temps polynomial) si son temps d'exécution suit un polynôme en la taille de l'entrée. C'est à dire que si pour une entrée de taille N, le temps d'exécution est N, l'algorithme a un temps d'exécution linéaire (donc polynomial). De même N^3 est cubique donc polynomiale et $10^9 * N^{(10)^6}$ est aussi un polynôme.Il faut s'intéresser aux grandes valeurs de N pour vérifier qu'il n'y ai pas une explosion du temps de calcul à partir d'une certaine valeur. Même des algorithmes couteux peuvent être considérés comme polynomial si le temps de calcul reste raisonnable lorsqu'on augmente N. On veut éviter les temps exponentiels (très mauvaise propriété). Les algorithmes qui ont un temps d'exécution exponentiel sont dits **prohibitifs** ou **super-polynomial**. Ils s'opposent aux algorithmes standards.

exécutables
en temps raisonnable

Quiz : quelles opérations sont réalisables en temps polynomial ?

1. chiffrer/dechiffrer par OTP un message de taille n
2. essayer toutes les clés de taille n pour un déchiffrement
3. multiplier deux nombres de taille n
4. calculer a^b pour a, b de taille n
5. calculer $a^b \mod c$ pour a, b, c de taille n

L'algorithme trivial (multiplier a par lui même b fois) est

1. OTP en temps linéaire. Pour un algorithme on peut pas s'empêcher de lire l'entrée et d'écrire la sortie. Si on écrit N bit il faut déjà au moins N opérations.
2. Essayer toutes les clés de taille N : 2^N opérations, temps exponentiel.
3. Possible en temps polynomial.
4. Prend un temps exponentiel.
5. C'est un algorithme inefficace qui est exponentiel. Cependant il y a de l'espoir, le résultat est en N bits donc il peut exister un algorithme qui fait ce calcul en un temps polynomial. Ce sont les algorithmes de type square and multiply.

Cependant c'est une chose de savoir utiliser et de connaître la complexité d'un algorithme mais s'en est une autre de connaître la complexité intrinsèque d'un problème (complexité du meilleur algorithme pour le résoudre). Trouver cette complexité est souvent difficile et trouver un algorithme exponentiel ne signifie pas que le problème est exponentiel, on peut peut être trouver un algorithme polynomial.

Pour rappel, la sécurité du chiffrement par bloc définit qu'il n'y a pas de test clairement discriminant pour différencier une permutation aléatoire et un chiffrement par bloc. Cette notion est fortement liée à la notion d'indistingabilité

Définitions :

- Un **algorithme utile** est un algorithme qui a un pourcentage de réussite non négligeable. Un algorithme polynomial peut être non utile si sa réussite est négligeable, c'est à dire exponentiellement petite.

- Quelque chose est **vrai asymptotiquement** (pour une fonction ou un algorithme) si au-delà d'une certaine limite cela reste vrai. Par exemple x^2 est inférieur à $x^{12}+x$ asymptotiquement. On rechercher cette propriété pour la négligeabilité: il faut qu'à partir d'un certain N, l'algorithme soit inexploitable.

- Une fonction est dite **non négligeable** si il existe un polynome p tel que la valeur absolue de la fonction est minorée asymptotiquement par $\frac{1}{p(n)}$ où n est l'entrée de la fonction.

- **Avantage non négligeable** : Si l'étude d'un algorithme nous permet d'avoir un avantage ϵ qui ne chute pas trop vite quand la taille des entrées augmente et qui est assez grand, on le définit comme avantage non négligeable. Dans ce cas, l'algorithme nous permet de trouver le bon résultat avec une probabilité supérieure à $\frac{1}{m}+\epsilon$. L'avantage d'un algorithme est dit non-négligeable si il suit une fonction non-négligeable en n pour des entrées de taille n. Notons qu'un algorithme réalisable en temps polynomial peut ne pas être utilisable si il a un avantage négligeable: cet avantage peut etre tout petit, ou encore chuter exponentiellement lorsque l'entrée augmente.

  Si on étudie une piece de monnaie pendant beaucoup de temps (on regarde le lancer, la rotation, etc..). On peut avoir une chance sur deux + ϵ de trouver si c'est pile ou face, c'est à dire plus de chance que celui qui n'a pas regardé et dit au hasard avec une chance sur deux.

Ainsi, pour qu'un algorithme soit reellement utilisable, il faut qu'il s'exécute en temps polynomial et qu'il apporte un avantage non négligeable. Si c'est le cas alors il existe une attaque efficace.

Quels algorithmes ont un avantage polynomial ?

1. pour un disque chiffré avec une clé de n bits, essayer n^{10} clés puis si on a pas trouvé la bonne tenter une au hasard parmi les restantes
2. même algo mais en essayant $2^{n/2}$ clés

1. Si on ne réussit pas à la première phase il reste $2^N - N^{10}$ clés, si alors on en prend une au hasard on a une chance de succès de $\frac{1}{2^N - N^{10}}$. Or un truc exponentiellement petit plus un truc polynomialement petit est un truc polynomialement petit. Donc en mettant sous la forme 1/2^n + epsilone, … donc l'avantage est négligeable. Cet algorithme reste cependant en temps polynomiale.
2. Avantage négligeable . Pas compris. Temps exponentiel.

1/(2^N-2^(N/2)) = 1/(2^(N/2)(2^(N/2)-1)) = 1/2^(N/2) * 1/(2^(N/2)-1)
c'est le produit de deux fonction exponentiellement petites
donc l'avantage est négligeable.

Les distingueurs

On va utiliser des algorithmes appelés jeux ou défis. Le but du jeu est de distinguer deux algorithmes A et B juste en regardant les sorties. Si il n'existe pas d'algorithme en temps polynomial qui va distinguer A et B avec un avantage non négligeable, alors les deux algorithmes sont **indistingables**. On va donc dire qu'un algorithme de chiffrement par bloc est sûr si il est indistingable d'une permutation aléatoire. C'est à dire qu'il n'existe pas d'algorithme en temps polynomial qui va distinguer (la distribution de sortie de) A d'une distribution aléatoire avec un avantage non négligeable.

Supposons un algo polynomial efficace, qui permet de distinguer A de B avec un avantage non négligeable mais tout de meme assez petit ($1/n^3$). Alors il existe aussi un algorithme polynomiale qui va distinguer A et B avec une probabilité très proche du premier algorithme. En faisant une récursion sur ce premier algorithme (appel polynomial) et en faisant des statistiques sur les resultat de ce premier algorithme, on peut définir un algorithme polynomial avec un avantage constant et très fort !

qui n'admet de des attaques exponentielles

Ce qui est bon pour un algorithme de chiffrement, c'est qu'à chaque ajout d'un bit, on double le coût de l'attaque tout en gardant un coût de chiffrement raisonnable. A ce stade, il nous faut de plus introduire la notion de **bit de sécurité** : on dit qu'un algorithme a k bits de sécurité s'il n'existe pas d'attaque en moins de 2^k opérations. La sécurité d'un algorithme ne peut pas généralement dépasser sa longueur de clé (étant donné que tout algorithme peut être cassé par force brute), mais il peut être plus petit. Par exemple le 3DES a une longueur de clé de 168 bits mais fournit au plus 112 bits de sécurité, depuis qu'une attaque de complexité 2^{112} est connue. La plupart des algorithmes à clé symétrique sont conçus pour avoir une sécurité égale à leur longueur de clé. Aucun algorithme à clé asymétrique de ce type n'est connu.

en fait c'est depuis son début, on a compris que face à certaines attaques (dites meet-in-the-middle) 2DES n'apporterait pas plus de sécurité que DES et 3DES n'apporterait que 112 bits

Cryptographie

Chapitre 3 - Les fonctions de hachage

I. Fonctions de hachage cryptographique

Propriétés

Une fonction de hachage est une fonction déterministe et publique qui à partir d'une entrée de taille arbitraire va calculer une empreinte de taille fixe et représentative de ces données. Cette fonction doit être uniforme, i.e elle doit donner des messages uniformément distribués (deux messages proches auront un haché très différent) et il faut que la probabilité de retomber sur le même message soit de très faible, ($\frac{1}{2^{128}}$ pour une entrée de 128 bits par exemple).

- **First pre-image resistance** : Pour qu'une fonction de hachage soit cryptologiquement sûre, il faut qu'il n'existe pas d'algorithme efficace (tournant en temps polynomial) permettant, à partir d'une empreinte, de trouver un antécédent.

- **Second pre-image resistance (ou weak collision resistance)** : Il faut qu'à partir du hashé d'un antécédent, il soit extrèmement difficile de trouver un antécédent différent de l'antécédent qui a génererait ce hashé. Dans ce cas l'attaquant ne choisit pas m1 et doit trouver un m2 qui donne le même haché. Elle est plus facile à obtenir que la Strong CR.

- **Strong collision resistance :** Il faut qu'il soit difficile de trouver deux antécédents (quelconques) ayant une même image. Un attaquant peut choisir m1 et m2. La SCR implique donc la WCR.

Ces trois types de resistance ont pour objectif d'assurer la non inversibilité de la fonction de hachage et aussi d'assurer que personne ne doit pouvoir modifier le message sans que le hashé soit lui aussi modifié. Cette propriété s'applique aussi au « créateur » du message et du hashé.

Différence entre Weak et Strong collision resistance (exemples)

- WCR : On a stocké les hachés de mots de passe dans une base de données. Quand un

utilisateur rentre son mot de passe, m1, on compare h(m1) au haché présent dans la base de données. Si ils sont égaux ont conclu que l'utilisateur à entré le bon mot de passe. Il faut donc s'assurer que si quelqu'un a accès aux hachés de la base de données, il ne pourra pas trouver un autre mot de passe m2 qui donne le même haché.

- SCR : On a une base de données donc chaque entrée à une taille important. Quand on fait une recherche, au lieu de la faire sur les données elles même on la fait sur le haché de ces données (moins couteux). Dans ce cas on ne veut pas avoir m1 et m2 (quelconques) qui donnent le même haché, sinon on perd l'unicité des identifiants des entrées de la base.

Objectifs de sécurité

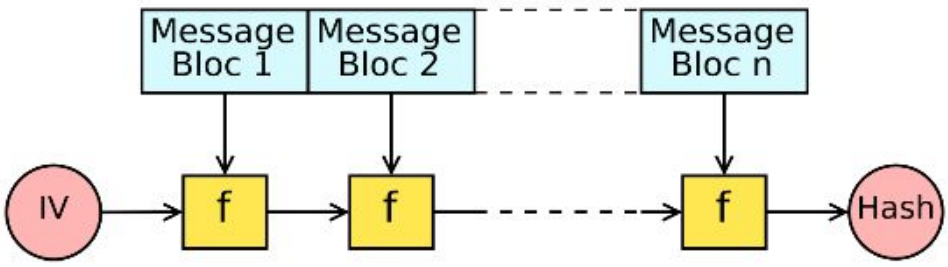
- Si la propriété de **non inversibilité** est respectée, il faut 2^k tentatives pour trouver un antécédent.
- Si la **weak collision resistance** est respectée, il faut 2^k tentatives pour trouver un autre antécédent.
- Si la propriété de strong collision resistance est respectée, il faut $2^{\frac{k}{2}}$ tentatives pour trouver un deux antécédents quelconques qui ont la même image. Ceci est dû au paradoxe des anniversaires.

> ***Paradoxe des anniversaires :***
> *Quand on tire uniformément dans un ensemble de N éléments, dès qu'on a tiré de l'ordre de racine de N éléments, on a une chance de l'ordre de 1/2 d'avoir deux éléments identiques.*

Exemple: si il y a une classe de 21 personnes, le 21eme étudiant à 20/365 chance qu'un autre étudiant soit né le meme jour que lui.

Construction classique

Une construction classique pour les fonctions de hachage est le modèle de Merkle-Damgard.



Partout:
citer avec une URL
tous les bouts de texte et les images prises ailleurs

La construction de Merkle-Damgard a servi de référence pour les constructions de type MDx. Cette construction a subi de nombreuses critiques et notamment à cause de la profondeur car cela devient mauvais pour des gros messages. C'est assez difficile d'optimiser cela électroniquement.

xxxxxxxxxxxxxxxx
avec du matériel dédié car la profondeur des circuits associés est importante

un bloc jaune == un paquet de rounds

On coupe le message en blocs de taille fixe et on le passe à la moulinette bloc par bloc en itérant. Chaque round des blocs (jaunes) est très simple mais il y en a beaucoup (à l'inverse des fonctions de chiffrement qui ont des rounds complexes mais relativement peu nombreux).

Les fonctions standards: SHA

- **SHA0** – Proposé initialement par le NIST, SHA0 n'est plus utilisée du tout. Cassé en 2^{60}.

- **SHA1** – SHA1, comme MD5, est par défaut utilisé dans tous les systèmes. Elle donne 160 bits en sortie (car la sécurité maximale qu'on peut attendre est de 80 bits de sécurité), ce qui était bien à l'époque. Mais SHA1 est (comme MD5) cassé par attaque par collision. Dans le contexte de la signature électronique, un attaquant qui peut choisir les collisions peut faire des trucs vraiment pas sympa du tout. Il ne faut donc plus utilisé SHA1 et MD5. SHA1 n'a cependant pas encore de fragilité sur l'inversion et peut donc être utilisé pour les protocoles qui ne nécessitent que la non inversion des fonctions de hachage.

- **SHA 2** – En cryptographie on utilise plutôt SHA2 en version en 224 bits (soit 112 bits de sécurité) et en 256 (128 bits de sécurité, comme AES128), etc.. SHA2 n'est fragilisé (comme AES) que pour des versions d'attaque qui font tourner un nombre de rounds inférieur à celui de la norme (31).

- **SHA 3** – Le NIST a poussé à l'organisation d'un nouveau standard, SHA3, qui n'a pas vocation a remplacer SHA2. SHA3 utilise juste un nouveau modèle, différent de Merkel-Vangard au cas où un jour on trouve une faille structurelle à ce modèle. On pourrait se rabattre sur SHA3. Il existe une version classique et d'autres versions qui permettent par exemple de choisir le nombre de bits « d » que l'on veut. en sortie

- **MD5** – Donne 128 bits en sortie soit 64 bits de sécurité mais est cassé (donne des collisions en 2^{18}). MD5 reste parfois utilisé par défaut. MD5 n'a cependant pas encore de fragilité sur l'inversion. Comme SHA1 on peut donc l'utiliser pour les protocoles qui ne nécessitent que la non inversion des fonctions de hachage. MD5 est de plus sensible aux attaques sur la résistance forte aux collisions. en revanche

- **Autres** : Whirpool, RIPEMD,

De façon générale, on essaie d'aligner la sécurité des différents protocoles de cryptographie. On connait le nombre de bits de sécurité apportés par les différentes fonctions de hachage selon leur version, on peut donc utiliser cette information pour ne pas introduire de maillon faible.

II. Cas d'usage

Les fonctions de hâchage peuvent être utilisées dans différents contextes :

- Vérifier l'**intégrité** : si le hashé de la version sûre et celui de la version téléchargée sont différents, cela peut signifier que le fichier est corrompu.

Exemple avec le téléchargement de logiciels (proFTP dans les slides) : Je veux faire une version du logiciel non vérolée et la partager. Je veux aussi créer une autre version, celle ci vérolée, qui aurait le même hashé (on peut jouer sur plein de choses dans m et m' pour qu'ils donnent le même hashé). On peut distribuer une version vérolée d'un logiciel dont le haché est correct, on peut même choisir qui aura la version vérolée et qui ne l'aura pas. Cet attaquant peut être n'importe qui, un employé, etc..

Pour ne pas se faire avoir dans ce cas il faut considèrer le bon attaquant et utiliser un modèle de clé de hâchage non obsolète. Ce type d'attaque met à l'épreuve la SCR car l'attaquant peut prendre m1 et m2 comme il le souhaite.

- **Stockage de mots de passe** : Voir un exemple donné précedemment. Autre exemple avec le stockage des mots de passe dans les OS. Un OS va stocker les mots de passe sous forme de hachés. En pratique aucun OS ne les garde en clair ou chiffré. C'est mieux puisque les mots de passe sont souvent réutilisés tels quels ou sous forme de variations évidentes. Si c'est stocké sous forme de haché, alors c'est mieux que de chiffrer puisque le hashage est non inversible (à la différence du chiffrement). Les attaques mettent à l'épreuve la WCR.

Attaques possibles pour le stockage des mots de passe :

- Attaques par **dictionnaire** : Je veux voir si le mot de passe appartient à un sous ensemble qu'on appelle généralement un dictionnaire (Larousse, ou Larousse+chiffre, etc..) à l'intérieur d'un espace beaucoup plus grand (l'ensemble de toutes les possibilités). On va donc chercher des valeurs qui sont structurées, probables. La ressources primordiale pour ce type d'attaque est la mémoire (plutôt que la puissance de calcul). Cette méthode dépend donc de l'habilité de l'attaquant à créer le bon dictionnaire (essai de choses structurées).

- **Force brute** : Il y a k bits et on essaie toutes les clés. Quand on sait qu'on a des mots de passe de 8 caractères alphanumériques et qu'ils ont été choisis uniformément parmi les caractères alphanumériques on teste tout.

En l'occurrence, entre l'ensemble des options laissées pas l'OS (128 caractères différents possibles environ) et les mots de passe que les gens choisissent, il y a un énorme écart. L'attaque par dictionnaire, moins complexe est donc particulièrement possible si le hashé vient d'un mot de passe qui a peu d'entropie. Pour un mot de K bits la complexité vaut $min(2^k, 2^{entropie})$.

les deux sont importants

de façon générale attention on dit haché et pas hashé

Il y a plusieurs manières de faire des attaques par dictionnaire :

- On connaît le haché de A. D'ici on énumération les entrées x d'un dictionnaire et on fait CSHF(x) (Cryptographic Secure *Hash* Function) jusqu'a ce qu'on tombe sur le haché qui convient.

- On utilise des **tables précalculée**. Quand on veut trouver plusieurs mots de passe par exemple, on se rend compte qu'on fait la même chose à chaque fois (on calcule le hashé, on compare, on continue). Il peut donc être avantageux d'utiliséer des tables précalculée qui permettent de réduire le nombre d'opérations nécessaire.On peut prendre le hashé de A, le chercher dans la table et traduire. La complexité plus faible puisqu'on ne passe plus de temps à calculer les hashés. Cependant ça prend pas mal de mémoire.

- On utilise une **Rainbow Table.** C'est une table qui contient 2^{35} lignes et 2^{35} colonnes. Une ruse permet de ne garder que la première et la dernière colonne. Ainsi, la recherche en temps maximum est de l'ordre de 2^35 (~heure) avec une mémoire de l'ordre de 2^35 (~Go). Le problème est que la mise en place ces tables prend beaucoup de temps de calcul.

Utilisation de sels

On peut souhaiter que les hachés soient différents à mots de passe égaux. On peut alors insérer de l'aléatoire en utilisant des « sels ».Ce sel, qui sera stocké en clair à coté du mot de passe, à la caractéristique de posséder beaucoup d'entropie. L'idée est de la concaténer au mot de passe de l'utilisateur et de hasher le tout. Ainsi, à l'aide de ce sel, deux utilisateurs qui ont le même mot de passe n'enregistreront pas le même hashé. De plus, même si le mot de passe est très structuré et qu'il possède une très faible entropie, l'ajout du sel va assurer, après hashage, une meilleure entropie totale.

Le but du salage est de lutter contre les attaques par analyse fréquentielle et les attaques utilisant des rainbow tables. En effet, il est impensable de calculer toutes les rainbow table associées à tous les sels possibles. Cependant, le sel n'évite pas les attaques par dictionnaire et les attaques par force brute, mais les rend plus couteuses car l'attaquant doit créer ses propres tables ce qui le rend lent et gourmand en mémoire. Cependant, le sel est stocké en clair sur le système, il peut donc être piraté. Aussi, on est toujours sensible aux attaques en 2^k avec k l'entropie de [mot de passe + sel]. L'objectif du sel est comparable à l'objectif des IV.

Une utilité des fonctions de hachage est le hachage d'un mot de passe pour générer une clé de chiffrement. Il est alors indispensable d'utiliser un sel.

Ainsi, les fonctions de hachage servent à: Assurer l'intégriter, stocker des mots de passe et générer des clés de chiffrement.

Une autre application importante des fonctions de hashage est l'**engagement,** ou commitment. Si on reprend l'exemple du chapitre 1 concernant le choix d'une chaine d'aléa

commune à Alice et Bob. L'un d'entre eux peut tricher car on n'a pas pris en compte la temporisation : en effet, l'un d'eux va recevoir le message de l'autre en premier et pourra donc faire un tirage adaptatif. Or il faut que un des deux au moins soit uniforme (celui d'alice) et qu'ils soient indépendants. Or si Bob triche on perd le fait que les aléas sont indépendants et le résultat ne peut donc pas être uniforme. Comment faire pour qu'aucun des deux ne puisse tricher ?

Solution : Alice va envoyer le hashé de son aléa à Bob. Ici elle ne veut pas que Bob choisisse un aléa adaptatif. Bob envoie son aléa à Alice, pas besoin que celui ci soit haché car Alice de toute facon s'est déjà engagé. Ensuite Alice renvoie son aléa en clair. Bob va alors comparer cet aléa avec l'engagement qu'elle avait fait.

Autre exemple impliquant la notion d'engagement. Comment faire pour jouer à pile ou face sans qu'il y ait de triche possible ? On pourrait dire que Bob lance la pièce, Alice envoie sa réponse sous forme de haché, puis sa véritable réponse et Bob vérifie. Or dans ce cas Bob peut tricher en testant les mots "pile" et "face" sur le haché de Alice. On peut résoudre ce problème en utilisant le salage, ainsi ni Bob, ni Alice ne peuvent anticiper la réponse de l'autre. Alice envoit le hashé d'un bit avec un sel. Bob enverra par la suite son aléa. Pour finir Alice enverra son bit et son sel pour que Bob vérifie.
Bob peut cependant tout de même faire une attaque la dessus : en effet, si on prend un sel de K bits et un haché salé de N bits, Bob pourra soit regarder tout les mots de K bits pour chercher un sel (il test les deux valeurs 0 ou 1 avec tous les sels possibles) soit il essaye toutes les valeurs de N bits (il cherche le complet). L'attaque sera donc en min(2^k, 2^N).
Mais Alice peut aussi faire une attaque : en effet il lui suffit de trouver deux aléas et deux sels tels que CSHF(Aléa1, Sel1)=CSHF(Aléa2, Sel2). Ainsi, elle pourra choisir l'aléa qu'elle veut envoyer à bob après que celui ci ai envoyé son aléa. La sécurité d'un engagement et donc basé sur le SCR.

Le système d'enchère enchères aveugles sans tiers de confiance se base sur le même principe. Il y a N clients qui se connectent à un serveur d'enchère. Ils envoient chacun un engagement du prix qu'ils sont prêt à payer. Ce prix sera salé pour ne pas que les autres participants voient le prix. Dans un second temps, ils dévoilent leur prix. On peut se passer des engagements si il y a un tier de confiance mais c'est risqué. Pour blinder le protocole, il faudrait que le serveur envoie des signatures qui accusent réception des engagements des clients, car il pourrait très bien dire à la fin qu'il n'a pas recu l'engagement.
Il faut considérer un modèle d'attaquant assez précis. ~~xxxxxx~~ suffisemment ce qui n'est généralement pas simple du tout (car il faut pour cela prévoir toutes les choses qui peuvent permettre de monter une attaque et les mettre dans le modèle, mais comment tout prévoir ?).
toujours

Un dernier cas d'usage des fonctions de hâchage est la lutte contre le spam. Sur internet, les MTA (Mail Transfer Agent) sont des ordinateurs qui permettent de relayer les mails. Le problème est qu'un client a besoin de peu de ressources (une ligne internet à haut débit suffit) pour envoyer un nombre très important de mails (10 à 100k par seconde). Pour limiter cette possibilité, les MTA peuvent demander une "preuve de travail" au client pour limiter le nombre d'envois possibles à 10 mails par secondes. Ainsi, on part du principe que si tu écris bien tes mails et que tu es pas un spammeur, il utilises au moins une demi-seconde entre l'envoie de deux mails.
Le fonctionnement est le suivant : le client qui souhaite envoyer un mail doit former un entete avant l'envoi. L'en-tete est de la forme : 1:20:date:@mail:nombreAléatoire (la date est synchronisé avec le MTA sinon le mail est refusé). Ensuite il calcule le hashé (SHA1) du header. Si les 20 premiers bits du haché sont des 0 c'est bon, sinon il incrémente le nombre aléatoire jusqu'à tomber sur un haché commençant par 20 zéros. Tant que cela n'est pas fait, on ne peut pas envoyer le message. On a

2^20 tests à faire pour trouver cette configuration. Ainsi, le MTA saura que l'expéditeur aura bien fait au maximum les 2^20 tests et donc qu'il a “passé du temps” et à en quelque sorte payé cet envoi d’email.

Ainsi pour éviter de se balader avec un carnet complet de clé, on va choisir un Master Secret Key, que l'on va dériver pour donner des Master Encryption Key. On fera le hachage salé de cette clé et on l'utilisera pendant une période donnée.

Conclusion sur les cas d’usage :

- Assurer l’intégrité
- Stocker des mots de passe
- Générer des clés de chiffrement
- Engagement
- Anti-spam

III. Le controle d’intégrité

Définition (Wikipédia)

*Un **code d'authentification de message** (**MAC**, Message Authentication Code) est un code accompagnant des données dans le but d'assurer l'intégrité de ces dernières, en permettant de vérifier qu'elles n'ont subi aucune modification, après une transmission par exemple.*

Le concept est relativement semblable aux fonctions de hachage. Il s’agit ici aussi d’algorithmes qui créent un petit bloc authentificateur de taille fixe. La grande différence est que ce bloc authentificateur ne se base plus uniquement sur le message, mais aussi sur une clé secrète.

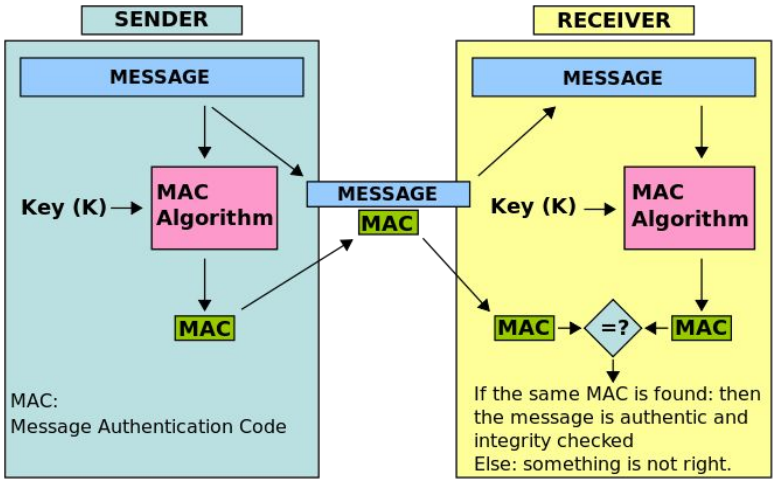
*Tout comme les fonctions de hachage, les MAC n’ont pas besoin d’être réversibles. En effet, le récepteur exécutera le même calcul sur le message et le comparera avec le MAC reçu. Le MAC assure non seulement une fonction de vérification de l'**intégrité** du message, comme le permettrait une simple fonction de hachage mais de plus **authentifie** l’expéditeur, détenteur de la clé secrète.*

Ainsi un MAC :

- Assure l'intégrité et l'authenticité
- Ne protège pas contre les rejeux

Un MAC est donc un petit tag de quelques bits qui permet d’authentifier un message, c’est à dire garantir sa provenance et de garantir son intégrité (aussi appelé MIC). Ce tag permettra de garantir que ce message est bien généré, sans modification (un MAC est fonction de tout le bits d'un message et la moindre modification le rend invalide).

Pour faire ce MAC, on a besoin du message et d'une clé symétrique. A la réception, Alice va récupérer le message. Disposant de la clé privée, elle va générer un MAC et comparer avec celui présent dans le message : si ils sont équivalent alors c'est bien Bob qui a envoyé le message.



On peut se demander, quand Alice envoie un message à Bob, pourquoi on insère pas plutôt un CRC. Un CRC n'est pas fait pour un usage cryptographique. Il n'a pas les propriétés de type non inversibilité, résistance aux collisions, etc.. C'est très facile de rajouter un petit bout au message (un virus) à un endroit, un petit bout ailleurs, sans faire changer le CRC. Il ne donne aucun type de résistance aux collisions (forte ou faible), on peut passer de m à m' sans problème en gardant le même CRC. Les authentications via adresse MAC ou adresse IP sont tout aussi nulles.

L'idée de base est qu'on peut pas authentifier quelqu'un ou quelque chose de façon fiable si on a pas un secret commun. En effet, on partage une clé secrète avec la personne qui à généré ce message (et ca peut etre moi meme… dans le passé). Si je laisse mon pc et qu'ensuite je me demande si c'est bien moi qui ai fait telle chose, ben oui c'est moi car le tag est bon et il n'y a que moi qui a la clé.

Mais attention, les MAC introduisent aussi un **problème de rejeu** ! Il n'y a pas nécessairement une dimension temporelle. En effet, le message que je reçois de Bob à l'instant T est peut etre un rejeu d'il y a très longtemps. On est donc certain du destinataire mais pas de la période d'envoi.

Propriétés

Pour un MAC (couple m,t valide) :

- Il ne faut pas que l'attaquant puisse générer des MAC même en ayant vu passer plein d'autres couples.

- Il ne faut pas que l'attaquant puisse forger des MAC même s'il à la possibilité de nous faire forger des MAC avec les couples message / tag qu'il choisit.

Ces hypothèses sont vérifiées si tout algorithme polynomial à une probabilité négligeable de succès.

Il y a deux tailles qui influent sur la sécurité, la taille de la clé symétrique et la taille de la sortie. Le problème est qu'il y a un certains nombre réseaux dans lesquels les messages qu'on veut envoyer sont petits et on a pas forcément envie de le coller 128 bits de tag à chacun (des ACK, etc..). Ca ferait une surcharge très significative. Par exemple, dans le cas de la téléphonie cellulaire, on ajoute des tags de 32 bits uniquement.

Question :

Soit un MAC prenant en entrée une clé de 128 bits et donnant en sortie 30 bits. Un attaquant peut :

- créer un *t* valide pour un *m* en 2^{30} opérations
- créer un *t* valide pour un *m* en temps constant avec une probabilité de succès d'au plus $1/2^{30}$
- créer un *t* valide pour un *m* en 2^{128} opérations
- créer un *t* valide pour un *m* en temps constant avec une probabilité de succès d'au plus $1/2^{128}$

1	Qu'est ce qu'une opération? Si "opération" = "calcul" alors à chaque fois pour vérifier si c'est bon, il manque à l'attaquant la clé secrète. Il ne peut pas tester en local 2^{30} tags. Si "opération" = "calcul + envoi", alors la ça peut marcher, avec 2^{30} envois. Tout dépend donc de si opération = calcul ou opération = calcul + envoi. C'est problématique si l'attaquant à assez d'informations pour faire les calculs en offline. Quand on tape un login sur un serveur, il faut faire des tests en ligne et donc le serveur peut réagir (délai entre les requêtes, dire stoppe, etc..).
2	Il peut aussi profiter du fait qu'il n'y a que 30 bits en sortie pour deviner t. Il prend un m et va le concaténer avec {0,1}^30. Mais dans ce cas il ne peut pas le vérifier. Ainsi ces 30bits ne nous protège pas de sa capacité computationnelle mais de sa chance ! Il peut donc créer un t valide pour un M donné en temps constant avec une probabilité de succès d'au plus $\frac{1}{2^{30}}$.
3	Il va essayer de voir des messages qui sont déjà passés et donc trouver Sk: il peut faire cette attaque en 2^138 opérations. Ainsi pour un M donné, il pourra toujours générer un tag grâce à la clé qu'il à trouver.
4	Voir 2.

Supposons qu'un attaquant essaye d'insérer du trafic : il va y arriver une fois sur un milliard. Mais je ne veux pas qu'il arrive à trouve la clé secrète en regardant des messages et fasse 2^30 opérations. Les 30 bits ont un rapport avec la probabilité et les 128bits à la puissance de calcul.

Bits de clé, bits de MAC et sécurité

Avec le hâchage, il y a une sécurité computationnelle qui doit être telle que 2^k opérations sont nécessaires pour casser la fonction complètement (k/2 pour la strong collision resistance). Ici, le nombre de bits de tag détermine plutôt la probabilité qu'un attaquant a de forger un tag correct.

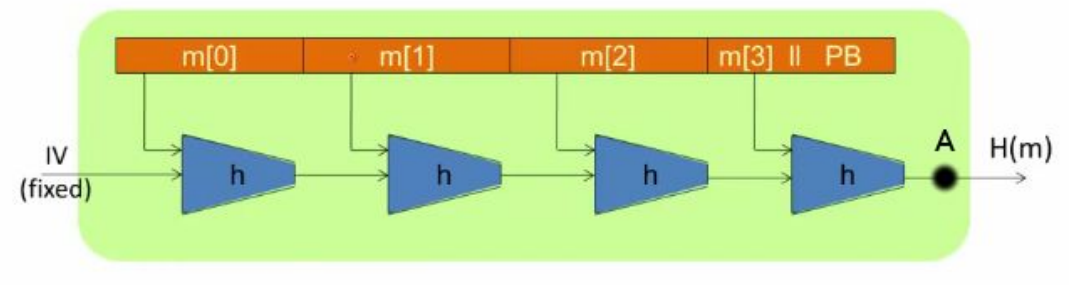
Plus on raccourci le tag plus il faut faire des renouvellement de la clé qui est utilisé. ~~Le nombre de bit du tag correspond plus à une probabilité de chance de l'attaquant.~~ L'idée est que si le tag fait k bits alors dès qu'on a généré 2^{k/2} messages tagés il commence à y avoir des collisions (paradoxe des anniversaires) au niveau du tag (i.e. plusieurs messages avec un même tag). Ceci n'est pas bon donc pour un tag de 30 bits il faudrait changer la clé tous les 2^15 messages.

Il faut que l'attaquant ne puisse pas faire d'attaques offline (attaques de Shannon). Si on utilise un tag peu de fois, ça, va, mais dès qu'on envoie un nombre important de messages avec une même clé, la sécurité n'est plus garantie (on peut prouver qu'après avoir envoyé un nombre de messages de l'ordre du carré des valeurs possibles dans un bloc, elle n'est plus assurée).

Architecture

Il y a deux modèles standardisés et répandus : HMAC et CBC-MAC :

- **H-MAC** est la construction la plus présente dans le monde de l'internet (SSL/TLS, IPSec, SSH…). Elle est souvent basée sur SHA1 de nos jours même si les recommandations disent de passer à SHA256. En effet, la sécurité d'un code MAC est basée plutôt sur la notion d'inversibilité et de préimage de la fonction h utilisée, or SHA1 a un très fort niveau inversibilité. Si SHA1 est utilisé, la condition est d'avoir assez de bits de sortie de SHA1.
  Pour comprendre comment fonctionne les HMAC, on rappelle la construction de Merkle Damgard :



URL

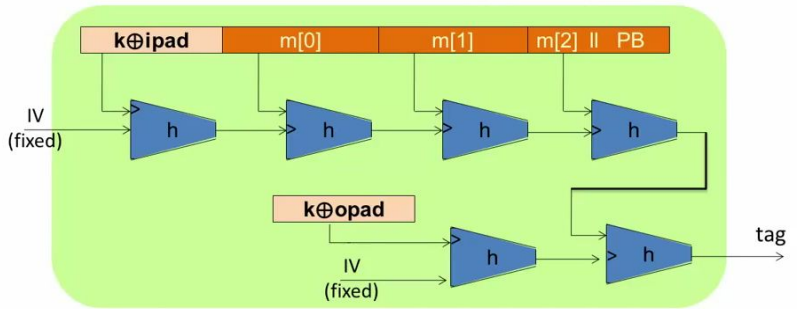
Ici h est une fonction de compression qui doit être résistante aux collisions pour que H, la fonction de hâchage en sortie soit résistante aux collisions. PB signifie Padding Blob et IV vecteur d'initialisation. On rappelle que le hashage est plus cher généralement que les fonctions de chiffrement. AES a 10, 12 ou 14 tours, dans les fonctions de hachage, on est plutot autour de 60 tours. Donc c'est relativement lent.

On peut se demander dans un premier temps pourquoi on ne se contente pas de construire notre HMAC en faisant HMAC(sk, m) = CSHF(sk || m) ? Si on le faisait alors un attaquant pourrait exploiter la construction de Merkle-Damgard pour faire une attaque par extension. En effet, si l'attaquant connait l'état du système en A (s'il connait H(m)), alors il peut ajouter ce qu'il veut à la suite c'est à dire utiliser à nouveau h avec en entrée m4 et H(m). Il peut construire un tag correcte pour sk, m, w avec w de son choix. Ce n'est évidemment pas souhaitable. On aurait donc pu inverser m et sk (ça règle pas tout mais c'est mieux). Si un attaquant voit passer un message avec des tags, il n'est pas sensé pouvoir re-générer du trafic. On veut que si on reçoit un message de Bob, on soit sûr que ça a été taggé par Bob et pas une extension de quelqu'un d'autre.

HMAC est donc un moyen d'utiliser la construction de Merkle-Damgard pour calculer des Message Authentication Code de façon sûre. Le calcul réalisé pour les HMAC est le suivant :

$HMAC\ :\ S(k,m)\ =\ H[k \oplus C_1,\ H(k \oplus C_2 || m)]$ avec C1 et C2 des constantes

Autrement dit, l'algorithme est appliqué une fois sur $H(k \oplus C_2 || m)$ qui à lui seul serait non sûr. Une fois la sortie obtenue, on la concatène avec la clé k (xor une constante) et c'est la sortie de ceci qu'on utilise comme HMAC. Voici finalement le schéma correspondant au fonctionnement des HMAC :

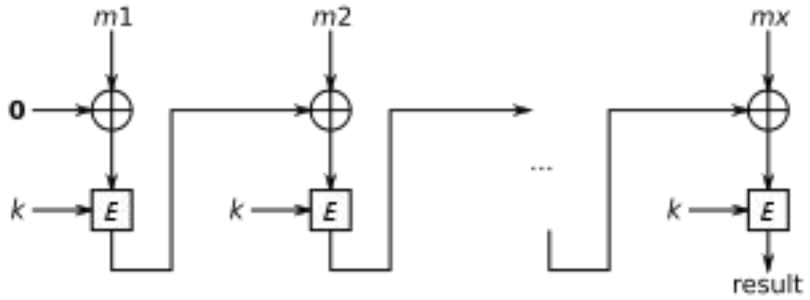


URL

Ceci revient a faire un post traitement qui est la finalisation ou on prend le premier hashage et on le concatène à une autre clé sk'. Ce qui est important donc, c'est qu'à la fin du message on utilise cette autre clé. Si ces clés étaient indépendantes la preuve de sécurité serait simple. En pratique elles ne sont pas indépendantes. Mais en tout cas, personne n'a réussi à attaquer ça jusqu'à maintenant.

Les HMAC sont utilisées dans SSL-TLS (HTTPS, IMAPS, …), IPSEC, SSH, etc.. Les MAC ne protègent pas du rejeu, c'est juste un contrôle d'intégrité. « Je suis sûre que c'est un message d'Alice».

- Les **CBC MAC** sont plus rapides mais les algos de chiffrement ont une histoire plus turbulente que les fonctions de hachage. Cette construction est basée sur un fonctionnement de chiffrement par blocs CBC. Elle a comme propriété d'être plus rapide (les fonctions de hachage sont lentes en générale et le chiffrement par bloc est plus rapide…). Du coup cette technique de hachage est par exemple utilisée dans les banques, le wifi, la téléphonie, …



hacher

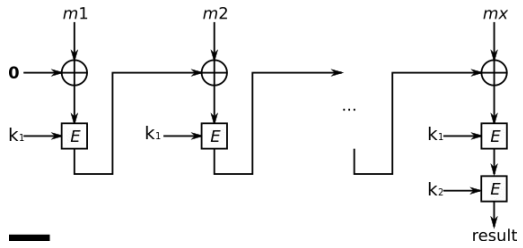
128 bits

Pour chiffrer, on découpe les données en blocs de taille adéquate (selon le chiffrement par bloc utilisé, au minimum un chiffrement par bloc de 32 bits). Les blocs sont chiffrés les uns après les autres, le résultat chiffré du bloc précédent est transmis au bloc suivant. Dans CBC MAC on ne donne pas les sorties intermédiaires et donc à la fin on a CBC-MAC(sk, m1|m2..|mn). Comme indiqué sur le schéma, il faut utiliser sk à chaque étape, si l'attaquant ne connaît pas sk, c'est fichu pour lui. Ici on est dans les 200 cycles par blocs pour la compression.

On peut pas faire les même attaques que pour les HMAC, de type extension. Ici cette attaque est plus compliquée et a un impact un petit peu moindre.

- Dans **ECBC MAC** on utilise le même principe que pour CBC-MAC, avec une petite différence cependant. Le résulat obtenu en sortie est à nouveau chiffré, avec la même fonction mais avec une clé sk' différente de la première . Il est important que les deux clés soient indépendantes. On les dérive d'une clé maitre. Pourquoi ajouter une étape avec sk' à la fin ? Car des attaques sont possibles si on utilise que sk. Si on apprend une collision, après on

peut créer plein d'autres collisions. Si on ne scelle pas derrière on peut faire plein de choses en étendant les messages. En pratique ça veut dire qu'un attaquant pourrait générer des tags pour d'autres messages. Dans des réseaux temps réel, les messages ont une taille fixe, du coup on peut utiliser simplement CBC MAC.



NB : Si on se place par exemple dans un cas où E=AES128 avec deux fois la même clé, on se retrouve alors dans la situation : AES-128-CBC-MAC(sk, m) || AES-128-CBC-MAC(sk, m) avec donc les propriétés de confidentialité, intégrité. Puisque les clés sont les mêmes il y a des attaques possibles. De façon générale, il ne faut **jamais** rien **réutiliser** en cryptographie.

Avant de conclure : L'Oracle aléatoire

Source : Wikipédia

Random oracles are typically used as an ideal replacement for cryptographic hash functions in schemes where strong randomness assumptions are needed of the hash function's output. Such a proof generally shows that a system or a protocol is secure by showing that an attacker must require impossible behavior from the oracle, or solve some mathematical problem believed hard in order to break it.

Se placer dans le modèle de l'oracle aléatoire, cela signifie que l'on se place dans une situation idéale, c'est à dire qu'on considère que la fonction utilisée est indistinguable de l'oracle aléatoire. Ce modèle nous permet d'élaborer des démonstrations là où les mathématiques simples ne pourraient être utilisées. On a pas réussi à contredire le modèle de l'oracle aléatoire. Il tombe un peu du ciel mais jusqu'à preuve du contraire, ça marche. On peut utiliser ce modèle sans rien comprendre à ce qu'ont fait les cryptographes (dans SHA256 etc). En particulier, le modèle de l'oracle aléatoire permet de donner une bonne intuition de si un protocole est sûre ou non. Si c'est cassable dans le cas du modèle de l'oracle aléatoire alors c'est plutôt le système est faible.
Comment fonctionne t il?

A chaque fois qu'on demande pour un x quel est le haché, l'oracle aléatoire fait :

1. Est-ce qu'on m'a déjà demandé x dans le passé ?

a. Si oui, l'image est stockée dans la liste d'état. Alors l'oracle aléatoire retourne y. Ceci assure le coté déterministe.
b. Sinon, l'image n'est pas dans la liste alors l'oracle aléatoire tire un Y de manière uniforme dans l'ensemble des hachés. Il note ça dans ma mémoire le couple (X,Y) et renvoie Y.

Il peut donc être utile est de se placer dans une situation ou une fonction de hachage est 100 % sûre c'est à dire qu'on peut la remplacer par un oracle aléatoire. Si on veut se mettre dans ce modèle pour SHA256 par exemple, il ne faut pas supposer qu'on appelle un programme utilisant une fonction déterministe. Quand on l'utilise une seule fois, on peut remplacer SHA256 par un programme qui lui utilise de l'aléa et si SHA256 est bien codé on ne voit pas la différence.

Plus généralement si un prédicat P est vrai dans le modèle de l'oracle aléatoire, alors P est aussi vrai quand on passe au modèle avec une vraie fonction de hachage. Donc au lieu de s'embêter à prouver des choses dans le modèle standard (par exemple avec SHA256 qui ne fournit pas de l'aléa pur, il faut établir un modèle etc..) on fait la preuve dans le modèle de l'oracle aléatoire. Par exemple, dire qu'un attaquant ne peut pas obtenir d'informations sur le clair n connaissant le chiffré peut etre vrai dans le modèle de l'oracle aléatoire. Or personne n'a prouvé que quelque chose de vrai dans le modèle de l'OA était faux dans le modèle standard.

Le passage du modèle de l'oracle aléatoire à une fonction de hachage à toutefois permis de casser des fonctions de hachage. Or c'était toujours de fonctions de hachage fragiles au départ. Personne n'arrive à casser SHA1 mais des choses qu'on a montrées avec l'oracle aléatoire ont permit de trouver d'autres failles.

Cryptographie

Chapitre 4 - Aléa, Pseudo-aléa et Chiffrement à flots

I. Introduction

En cryptographie on a besoin d'aléa pour beaucoup de choses. Dans les algorithmes de chiffrement symétrique, on suppose que la fonction de chiffrement garantit une sécurité mais il faut que la clé qu'on lui donne en entrée soit aléatoire et que l'algorithme soit sûr. Comment est-ce qu'on génère ces clés aléatoires ? Comment peut-on garantir qu'un algorithme est sûr ? Avoir des preuves de sécurité n'est pas évident. En pratique, les algorithmes sûrs aujourd'hui ont été standardisés longtemps après leur sortie (éprouvés par la communauté). On peut aussi avoir des preuves mais ce n'est pas sans coût.

Une dernière hypothèse dans laquelle on ne rentrera pas ici : on suppose quand même que Alice et bob partagent une clé. C'est une question extrêmement importante en cryptographie.

Pour générer une clé on peut utiliser de l'**aléa pur**. Un modèle tient : utiliser des phénomènes physiques. C'est parfaitement sûr si le modèle physique tient et si le hardware utilisé pour mesurer le phénomène physique est bien fait. Ce dernier point n'est absolument pas garantit. Celui qui l'a fait peut être malveillant, négligeant ou manquer de chance (ça commence à dévier et je ne m'en rends pas compte). L'aléa pur est donc un modèle théorique. On ne peut pas distinguer une source hardware et un modèle théorique si c'est vraiment aléatoire. On autre problème lié à cet aléa pur est qu'il est non reproduisible: la série de bits qui sera générée à l'instant T par un générateur, ne pourra être généré dans un autre système distant et décorrelé au même moment.

On peut essayer de créer un mécanisme qui ressemble beaucoup à de l'aléatoire, du **pseudo aléa** (ne veut pas dire déterministe et reproductible). On arrive à faire des mécanismes globalement proches de la distribution cible, ou alors des trucs cryptographiquement sûrs. On dit que ce système est cryptographiquement sûr si il est éprouvé par la communauté et calculatoirement indistingable d'un générateur pur (si on le met dans un jeu de distinction de deux algorithme dont un aléatoire, il n'y a pas d'algorithme polynomial qui pourrait distinguer les deux algorithmes avec une probabilité non négligeable). De plus, on sait rendre ces générateurs pseudo-aléatoire reproductibles.

II. Générateur d'aléatoire

On peut gérer la génération d'aléatoire avec des moyens physiques :

- Utiliser les émissions photoélectriques : quelqu'un qui regarde le composant ne sait pas quel est le prochain bit qui va sortir (physique quantique)
- Périodicité des radiation..
- Avec d'autres sources comme les bruits physiques qui sont des phénomènes chaotiques. Par exemple les trajectoires des planètes autour du soleil, il y a un phénomène chaotique qui fait que dans 50 millions d'années par exemple, à long terme, on ne peut pas prévoir leurs positions. De même avec le bruit thermique, etc..

Donc, soit on a accès à une centrale nucléaire soit on fait des choses plus modestes. Par exemple il y a un composant dans les processeur qui permet de mesurer le bruit thermique. Cependant, des backdoor sont créé par les fabricants de ses produits, sous ordre d'agences telles que la NSA. C'est backdoor peuvent prendre la forme d'un biai statistique qui sera connu par l'intéressé et qui lui permettra de déchiffrer les communications si besoin.

Exemple parano : Il existait une backdoor dans Windows NT, un mot de passe particulier qui marchait toujours. Cette histoire a fait beaucoup de bruit. De même une backdoor existait chez les autorités de certification qui étaient obligées par les instances de défense type NSA de leur permettre de générer de faux certificats.

Dans les systemes de Unix, on peut acceder à un device, programmé au niveau du noyau qui prend plusieurs sources d'entropie (bruit du CPU, mouvements de la souris, frappes au clavier, interruptions faites par la carte réseau..). Il mélange ces sources d'entropie et créer un pool d'entropie :

```
tls-sec@b301-35:~$ cat /proc/sys/kernel/random/entropy_avail
1250
```

Sur cet exemple on voit qu'on a 1250 bits d'entropie disponibles actuellement (on a pu récupérer des mouvement clavier etc..). On demande de générer de l'aléa avec la commande `dd`, on re-regarde. On constate que l'entropie est descendue.

Si jamais les sources d'entropie sont mauvaises (pas de mouvement souris, pas de ci, pas de ça..) ça pourrait être risqué, on aurait une mauvaise entropie. Du coup le processeur prend le résultat qu'il obtient et le passe par une fonction de hachage. Quand tout le pool est utilisé, il le bloque. Donc si on demande beaucoup d'aléa d'un coup, ça demande pas mal de temps. Ca donne la garantie que ce qui est généré est bien uniforme, mais c'est long. Donc cette méthode est recommandée pour les clés très long terme (clés maîtres, clés asymétriques, ..). Cette technique utilisée dans les systèmes type Unix est donc uniquement basée sur le hardware et une fonction de hachage. Un reproche qu'on peut faire à cette méthode est que des générateurs sont déjà tombés parce que certains composants hardware étaient mal désignés.

Il existe une alternative : les générateurs pseudo aléatoires cryptographiquement sûrs non déterministes (utilisés par BSD, OSX..). Au lieu de générer un pool d'entropie, on utilise l'entropie pour générer une graine qui dès qu'elle peut être générée va aller alimenter un générateur pseudo aléatoire. On essaie de maintenir le pool pour qu'il y ait toujours une graine pour le générateur pseudo aléatoire. Dans la famille BSD ça fait longtemps que c'est comme ça. On a tendance à se diriger vers ces CSPRNG (cryptographically secure pseudo-random number generator) non déterministes.

NB : Un PRNG déterministe permet de générer une même séquence d'aléa avec une même graine. Un PRNG non déterministe ne permet la génération d'une séquence d'aléa qu'une seule fois, c'est à dire de façon non reproductible.

III. Motivation : le chiffrement à flot

Introduction

L'idée du **chiffrement à flot** est de généraliser OTP en n'utilisant pas un masque aléatoire mais un masque pseudo aléatoire généré par un CSPRNG (Cryptographically Safe Pseudo Random Number Generator). En effet, puisqu'on ne peut pas réutiliser le masque aléatoire dans OTP, dans le cadre du chiffrement à flot on peut choisir de prendre un secret aléatoire et de générer un masque avec une même fonction déterministe (CSPRNG) des deux côtés (Alice et Bob). Du coup si cette fonction est indistingable d'un générateur aléatoire, les masques seront de très bonne qualité. On va donc faire la premiere preuve qui montre que si quelqu'un arrive à casser un système de chiffrement basé sur un générateur pseudo aléa, ça veut dire que la fonction n'est pas cryptographiquement sûr.

Problèmes potentiels :

- **Malléabilité** : Un problème qui sera toujours là avec OTP, c'est le problème de la malléabilité. Il y a des applications pour lesquelles OTP n'est pas bien. Quand on utilise le chiffrement à flot, il faut rajouter un contrôle d'intégrité. Ainsi, le message est modifiable mais rendu inutilisable.

- **Délai dûs à la génération et à l'application du masque** : La graine du PRNG est partagée entre A te B, donc par exemple dans le téléphone de A on utilise cette graine pour générer 1Mo de masque, de même pour B. Le masque est généré pendant les périodes de non envoi et lors de l'envoi on utilise le masque. Ainsi le processus entre le temps d'envoi et le temps de chiffrement n'ajoute pas de délai. Toute personne ayant la graine (générateur standard) peut déchiffrer une communication avec chiffrement symétrique. Cette caractéristique est très importante dans le monde des télécommunications où la notion de délai est importante.

- **Désynchronisation** : Si un message est perdu il y a désynchronisation. En pratique on utilise des numéros de séquence.

- **Réutilisation du masque** : Dans tout les cas B doit noter tous les bits qu'il a consommé et il ne faut surtout pas qu'il utilise deux fois le même masque. Il existe un danger, comme avec OTP si on le réutilise. Il serait possible de déterminer le xor des messages. Il ne faut donc jamais retourner à un index inférieur du masque (pour se re-synchroniser ou quoi que ce soit). Voici un exemple de mauvaise implantation dans PPTP sur Windows NT :

  - Messages client-serveur : c_1, c_2, c_3 masqués par $PRG(k)$
  - Messages serveur-client : s_1, s_2, s_3 masqués par $PRG(k)$

  $\Rightarrow$ on peut obtenir $(c_1 \| c_2 \| c_3) \oplus (s_1 \| s_2 \| s_3)$

  Ici, il n'y avait pas une personne qui utilisait deux fois le même masque mais deux personnes qui utilisaient le même masque et ça c'est pareil, c'est mauvais. Pour communiquer dans un sens et communiquer dans l'autre, il faut des clés différentes. Il faut les considérer indépendamment.

Quelqu'un qui veut simuler notre générateur, s'il ne connaît pas sk, doit tester 2^{128} possibilités de graine vu qu'il ne distingue pas la communication chiffrée d'une source parfaite.

Notion de réduction

La **notion de réduction** est importante : P ≠ NP

Source : Wikipédia

*En mathématiques, et plus précisément en informatique théorique, le **problème P = NP** est un problème non résolu, et est considéré par de nombreux chercheurs comme un des plus importants problèmes du domaine, et même des mathématiques en général.*

Très schématiquement, il s'agit de déterminer si le fait de pouvoir vérifier rapidement une solution à un problème implique de pouvoir la trouver rapidement ; ou encore, si ce que nous pouvons trouver rapidement lorsque nous avons de la chance peut être trouvé aussi vite par un calcul intelligent.

Preuve de sécurité par réduction

L'idée est de dire : je pense que le problème P' est difficile à résoudre parce que je peux réduire P à P'. Si A résout P' (en un temps polynomiale, avec une probabilité non négligeable de

succès) alors il existe un algorithme C simple (temps polynomial, probabilité non négligeable de succès) utilisant A qui résout P. Or résoudre P releverait de l'exploit. Si quelqu'un trouve A qui résoud mon problème P' alors il donne un moyen de résoudre P : c'est peu probable. Puisque le problème de factorisation est très difficile on peut l'utiliser comme problème de référence P.

NB : Si l'algorithme A existait, une série d'appels à A qui résout P' permettrait avec la série de réponses de A de résoudre P. Je ne suis pas certain qu'un algorithme A n'existe pas et en même temps, c'est peu probable que ça existe.

Pour SHA, la seule garantie c'est le fait que ça n'a pas été cassé. Dans d'autres cas on a confiance par « propagation de la confiance » qu'on a dans SHA256 ou la factorisation. Si tu casses ce qui dérive de ces systèmes sûrs, de la factorisation etc.. Alors tu casses (temps polynomiale, probabilité non négligeable de succès) SHA, tu résouds la factorisation.. Donc on a confiance dans les dérivés.

Un chiffrement à flot sûr

Si on utilise un CSPRNG alors le chiffrement à flots associé est sûr. Il n'y a pas d'autre moyen de casser un chiffrement à flot que de casser le PRNG sous jacent. Pour comprendre ça, on va revenir à ce que « **PRNG est cryptographiquement sûr** » signifie. Ca veut dire "**indistingable d'une source d'aléa**".

Qu'est qu'un chiffrement à flot qui est sûr ? Idéalement pour le définir il faudrait définir la sécurité sémantique. Mais on ne le fait pas dans ce cours. Informellement, on va dire que qu'un chiffrement à flot est sûr si il est « aussi sûr » que le One Time Pad. On sait que le One Time Pad à de bonnes propriétés. Si on veut des informations sur la sécurité sémanique et le fait que ça implique de la sécurité on peut lire l'article de recherche ???.

Indistingabilité

Situation :

a. L'idée de base, c'est qu'on suppose qu'un attaquant peut résoudre un problème de sécurité sur le chiffrement à flot. On sous entend là qu'il dispose d'un algorithme polynomial A qui casse le chiffrement ie. qu'il apprend quelque chose sur le plaintext en lisant le chiffré avec une probabilité non négligeable.
b. On prend alors B, un algorithme polynomial qui distingue le PRNG de l'aléa avec une probabilité non négligeable.
c. Mon algorithme B va interagir avec le challenger C. Le challenger a une source A0 = PRNG et une source A1 = générateur d'aléa, b={0,1}. C tire b avec probabilité $\frac{1}{2}$.

Déroulement :

a. Le challenger C choisit un bit au hasard soit en sortie de A0, soit en sortie de A1 et va l'envoyer à l'attaquant B.
b. L'attaquant va xorer cette sortie avec un message M et, si l'algorithme A existe, il va lui donner le résultat de cette opération.
c. A voit ce chiffré et est capable de dire quelque chose sur M (parité, taille, n'importe quelle propriété) et renvoie donc f(M).
d. B (le distingueur) vérifie que cette propriété est vraie et si c'est le cas, B dit qu'on était dans le cas du PRNG (et pas de l'aléa).

Pourquoi B a-t-il un avantage non négligeable ? Pourquoi A va-t-il forcément voir que ça marche dans un des deux cas ? Si b = 1 (générateur d'aléa), la probabilité de succès de A est de $\frac{1}{2}$ (OTP), A ne peut rien deviner avec probabilité supérieure à $\frac{1}{2}$. En revanche avec le PRNG (b=0), A peut étudier le chiffré (P[succès de A] = ½+ε). Son avantage est au moins polynomialement petit parce qu'on avait précisé « si A existe » et on essaie de prouver « alors B existe ».

L'indistingabilité est donc une propriété importante pour un PRNG car si une attaque marche plus d'une fois sur deux, alors l'attaquant peut profiter du biais de notre fonction savoir qu'il est face à un PRNG.

On est en train de montrer que si A existe ("chiffrement à flot faible") alors B existe ("PRNG faible"). Dans nos hypotheses, A a un avantage non négligeable donc B a un avantage non négligeable. L'idée est d'exploiter ça, de façon couteuse comme décrit ici pour ne pas faire trop de probabilités. Le principe des distingueurs est assez général.

IV. Générateurs déterministes de pseudo-aléa

Introduction

Les générateurs déterministes de pseudo aléa sont pratiques pour de nombreuses raisons. On peut citer la pré génération des masques (on peut générer des tonnes de bits rapidement) et la reproductibilité. En revanche ils utilisent des fonctions dans lesquels les bits générés dans le futur dépendent des bits déjà générés. La prédictibilité est donc un problème (si je connaîs les n premiers

bits d'un générateur est-ce que je peux connaître le suivant ?). La plupart de PRNG ne sont pas cryptographiquement sûrs (prédictibles et distingables d'une source aléa).

Vocabulaire :

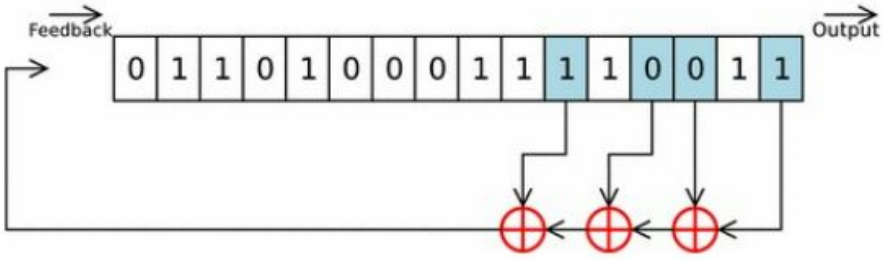
PRNG est un terme utilisé pour les PRNG déterministes (même clés, même masques) et non déterministes (changer la graine avec de l'aléa constamment). Par défaut on parle généralement de PRNG déterministes, car les PRNG non déterministes sont rares. Le NIST appelles les PRNG déterministes des DRBG (Deterministic Random Bit Generators). Ils sont utilisés dans plein de choses, dans des algos de base, au niveau système, des jeux, en cryptographie...

Exemples de PRNG

PRNG non sûrs classiques :

- Congruentiel linéaire : $X_{n+1} = (aX_n + c) \,.\, mod(m)$ avec X_0 la graine et typiquement $m = 2^{32}$. A chaque cycle on sort 32 bits (~32Gbps). Ces générateurs sont assez prédictibles et ont de mauvaises propriétés statistiques. C'est la fonction `rand` de C/C++/Java/.. Ne pas utiliser.
- Mersenne Twister : Plus compliqué, a de meilleures propriétés statistiques, mais ne pas utiliser.Il peut cependant être utilisé dans des fonctions de test.

Les registres à décalage (LFSR) :



Ce sont des générateurs pseudo aléatoire non cryptographiquement sûrs mais qui ont été très utilisés en cryptographie, notamment par les militaires. Il a un design idéal pour le hardware. En effet, il a une très faible profondeur de circuit et donc on peut utiliser une horloge très fortement cadensée. Le fait que la génération du pseudo aléa se fasse à très très haut débit permet de l'utiliser pour générer la séquence pour les saut de fréquence par exemple.

L'idée de base est que l'on met une graine sous forme binaire dans les cases mémoire puis on le fait tourner une fois complètement sans faire de sortie. On a un état sur 128 bits, on prend le dernier bits de cet état, il sort, de même pour le suivant et on re remplit petit à petit au niveau de "feedback" avec les XOR de certains bits (les bits xorés représentent un polynome dans un corps de Gallois).

Mise à part la vitesse, ces générateurs ont un autre avantage : ils ont de fortes garanties de non périodicité. En effet, la période est plus grande que 2^{128}.

En revanche ils ont un principal inconvénient : la linéarité donne lieu à de nombreuses attaques algébriques. Donc une solution est d'utiliser plein de LFSR en parallèle et de passer la sortie dans un truc non linéaire (ayant une faible profondeur de circuit), pour casser cette linéarité. On fait ça dans CSS (chiffrement pour les DVD par exemple), A5/1 (techno GSM) et E0 (bluetooth)..

Y a t il un moyen automatique de dire ceci est un bon ou un mauvais PRNG ? On peut effectivement dire si il est bon et s'il est indistingable d'un source d'aléa (cf. tests fournis par le NIST), mais il n'y a pas de moyen automatique de savoir si il est cryptographiquement sûr. Attention, un systeme peut avoir de bonnes propriétés statistiques et pour autant être cassable très facilement. Un attaquant va cibler le générateur A0 et trouver un moyen de faire la distinction.

Algorithmes à connaître

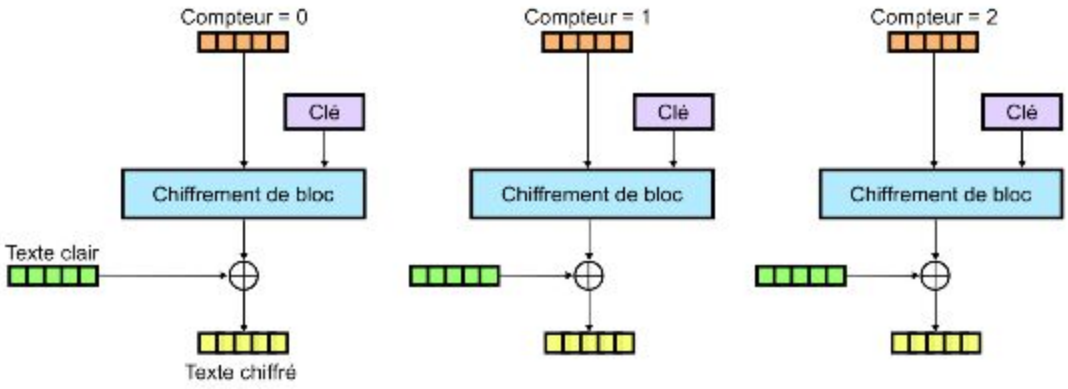
- **Utilisant le LFSR** :
  - CSS (pour les DVD) : cassé
  - A5/1 – A5/2 (pour le GSM) : 1 cassé, 2 cassé
  - E0 (bluetooth) : cassé
  - SNOW : cassé (avant standardisation), SNOW2 fragilisé et comporte des erreurs de design, SNOW3G abandonné et remplacé par AES.
- **Utilisant RC4** :
  - WEP
  - TKIP
  - HTTPS
- **Standards de cryptographie :**
  - Salsa20
  - SOSEMANUK
- **Chiffrement par bloc en mode CTR :**
  - AES-CTR : leader absolu, le plus utilisé de nos jours, utilisé pour A5/3

Remarque sur RC4 : RC4 est fortement fragilisé et a quelques très gros défauts. Le système est très sensible à l'utilisation de graines qui ont un lien entre elles (related keys). Aussi RC4 produit du très mauvais aléa sur le premier Mo, il ne faut donc pas l'utiliser à partir du bit 0 et pourtant c'est ce qui se fait. Ce système a des biais statistiques qui sont suffisamment faibles pour que quand on veut les exploiter il faille beaucoup de données. WEP cumule tous ces défauts. TKIP (WPA1) jette les premiers bits et s'assure que la clé est utilisée très peu de temps (temp key.. protocol). Donc WPA1 utilise bien RC4. C'est en ce sens que RC4 n'est pas complètement cassé. Pour WPA2 le standard est AES. HTTPS utilise RC4 car il est peu coûteux et au motif que les données sont confidentielles mais pas non plus vitales.

Remarque sur Salsa20 et SOSEMANUK : Un gros projet européen a été lancé pour faire des PRNG. On peut les faire avec de bonnes propriétés et rapides si on ne s'impose pas l'utilisation d'une fonction de chiffrement par bloc (où il doit y avoir l'inversibilité et qui sont généralement plus difficiles à faire) ou l'utilisation de fonction de hachage (coûteuses en temps et doivent résister aux collision). On veut donc faire des PRNG sans utiliser de chiffrement par blocs ou des fonctions de hachage . On a pas besoin pour un PRNG d'inversibilité, de résistance aux collisions. Ça prend du temps, ce n'est pas utile. Donc l'Europe a organisé un concours qui a donné : Salsa20 et SOSEMANUK par exemple. Salsa20 est le PRNG rapide de référence, il a une bonne capacité a être implémenté en hardware. SOSEMANUK ne s'implémente pas aussi bien en hardware mais est 2 fois plus rapide que salsa20 en software (et il est francais!).

AES - CTR

Principe (wikipédia) :



Ce qui est bien avec AES c'est qu'on développe le bloc bleu et on gagne un chiffrement par bloc et un chiffrement à flot, selon le mode. On a une plus grande compacité du code. De plus si AES est une PRP (Pseudo Random Permutation) alors AES-CTR est un CSPRNG et est donc sûr.

NB: Le standard est d'utiliser LTE avec AES

L'idée de base est qu'on a la clé, une fonction de chiffrement par bloc, et un compteur. Le compteur est un nombre sur 128 bits et est de la forme : 128 '0' pour '0ème', 127 '0' et un '1er' pour le '1', etc etc. On fait appel à la fonction de chiffrement par bloc avec en entrée le compteur et la clé. A chaque fois on obtient un masque qui va pouvoir être utilisé (XORé) avec bloc de clair. Autrement dit, les blocs de sortie sont utilisés comme des masques pour les blocs de clair. Le fait qu'on fasse un XOR à la fin pour chiffrer n'est pas trop le problème ici, on s'intéresse juste à la génération de pseudo aléa.

On aurait très bien pu imaginer un système qui aurait fait le XOR du message avec le compteur. Cependant, on pourrait dans ce cas tomber deux fois sur le même chiffré ce qui nous ferait perdre la propriété de permutation (on perd la bijection car plusieurs antécédents donnent le même résultat).

Le paradoxe des anniversaires nous donne aussi un résultat important. En effet, si l'on concatène N masques de 128 bits (masque1 || masque2 || … || masqueN), alors il faut seulement $\frac{2^{128}}{2}$ masques pour que l'on ait une réelle probabilité d'avoir deux fois le même masque, ce qui est mauvais ! **AES n'est donc pas un PRNG cryptologiquement sûr. Mais cela ne veut pas dire qu'il va rendre cassable un chiffrement à flot**. Car mis à part le fait de reconnaitre que ce n'est pas de l'aléa, un attaquant ne pourra pas, en temps raisonnable, apprendre des choses sur le message en clair. Ainsi AES-CTR possède de très bonnes propriétés statistiques. Il n'y a aucune attaque en temps polynomiale ou proche (il en existe bien une qui enlève deux bits de sécurité).

NB : On peut avoir un PRNG distingable de l'aléa et pourtant avec de très bonnes propriétés statistiques qui fait que le PRNG est utilisable.

Le mode CTR possède, au même titre que les PRNG, un problème de **malléabilité**. Il est donc fortement déconseillé de chiffrer un disque dur avec ce mode de chiffrement, même si il n'y a pas de rétroaction nécessaire (le mode CTR permet de déchiffrer un bloc sans avoir besoin de déchiffrer le reste).

En mode CTR, il faut aussi faire attention de ne jamais **réutiliser** une même clé avec une même valeur de compteur. Le faire mettrait en péril la confidentialité des blocs concernés. Ne jamais utiliser deux fois la même valeur de compteur est donc très important. Une statégie peut etre d'utiliser les 64 premiers bits pour une valeur qu'on incrémente à chaque bloc (type compteur donc) et d'utiliser les 64 derniers bits pour mettre un vecteur aléatoire. De cette façon, on s'assure la non réutilisation d'une valeur de compteur.

Si on utilise le mode CTR pour ensuite XORer avec des clairs alors le chiffrement est sûr au vu de la propriété précédente : "Si AES est une PRP alors AES-CTR est un CSPRNG". En pratique : admettons que l'on veuille 100 bits de sécurité. Si AES est une PRP avec une sécurité de k bits

alors AES-CTR est un CSPRNG pour des sorties de taille inférieure à $2^{\frac{k}{2}}$ (si on veut éviter les collisions). On prend donc 2^{30} à 2^{40} sorties puis on change la clé pour changer les permutations.

NB : Si le chiffrement par bloc est sûr alors le chiffrement à flot l'est aussi, si on ne chiffre pas trop longtemps.

Rappel sur quelques modes existants (cours 2A TR)

- Electronic Code Book, (**ECB**)

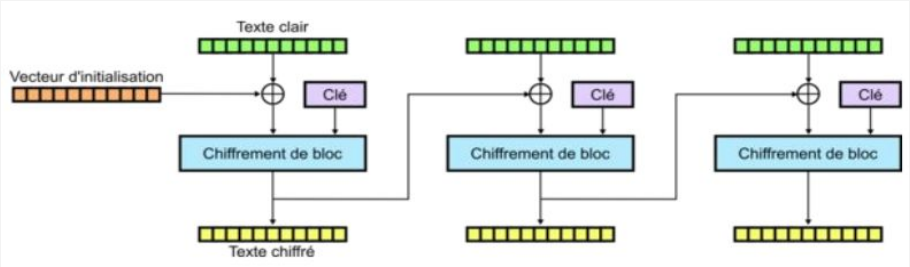
Il s'agit du mode le plus simple. Le message à chiffrer est subdivisé en plusieurs blocs qui sont chiffrés séparément les uns après les autres. Le gros défaut de cette méthode est que deux blocs avec le même contenu seront chiffrés de la même manière. Le seul avantage qu'il peut procurer un accès rapide à une zone quelconque du texte chiffré et la possibilité de déchiffrer une partie seulement des données.

- CounTeR (**CTR**)

Dans ce cas, c'est la valeur d'un compteur qui est chiffrée et qui produit un bloc pseudo - aléatoire qui sera ensuite x-orer avec les blocs de message clair pour produire le chiffré correspondant : cette méthode correspond au célèbre "one-time-pad". Le compteur est une fonction simple mais garantissant que la séquence utilisée pour chiffrer ne sera pas réutilisée. Ce mode peut également être utilisé avec un vecteur d'initialisation.

- Cypher Block Chaining (**CBC**)

Dans ce mode, on applique sur chaque bloc un ‘OU exclusif’ avec le chiffrement du bloc précédent avant qu’il soit lui-même chiffré. De plus, afin de rendre chaque message unique, un vecteur d'initialisation (IV) est utilisé.



OFM → permet aussi générer PRNG, mais un peu plus complexe que CTR donc pas trop utilisé. Il ne permet pas un déchiffrement parallèle.

Tout ce qu'on a vu jusqu'à présent possède une sécurité relativement ad-hoc. Depuis trente ans, la communauté à acquis beaucoup de savoir faire et a créé et testé beaucoup de protocoles. Ainsi la confiance qu'on dans ces modèles repose sur la confiance que l'on a en la communauté et l'expérience, leur savoir faire et les tests.

On a vu qu'il y a tout de même une notion de preuve mathématique qui est assez décevante finalement. En effet, on aurait espéré pouvoir utiliser les avancées du formalisme actuel pour prouver plus de choses. Mais il faut toujours partir d'une base en laquelle on a confiance. A partir de cette primitive de base, on peut construire des preuves formelles pour prouver la sécurité, mais elles se basent sur des primitives de confiance. On a jamais vu pour l'instant des primitives qui possèdent en elle même des preuves de sécurité.

Si on veut un système à sécurité prouvée, alors on utilise des structures algébriques sous jacentes bien plus complexes que de simples xor ou autres opérations utilisées jusqu'à présent. Des nouvelles structures algébriques donnent des propriétés très intéressantes, meme si elles impliquent des implémentations soft et hard plus couteuses. De plus, l'automatisation est très compliquée : on remplace les opérations faciles en informatique par des opérations classiques en mathématiques mais pas naturelles pour un ordinateur. Le problème est aussi qu'un ordinateur peut faire facilement des choses en taille finie, mais pas à partir de tailles trop importantes.

Les générateurs prouvés

Blum Blum Shub :

La communauté a proposé un générateur de pseudo aléa qui est particulièrement lent mais à sécurité prouvée. Son état interne est représenté par la variable x. A chaque évolution de cet état, il sort un bit (bit d'information sur l'état : parité, plus significatif,...). Il y a toujours un compromis dans un PRNG a sécurité prouvé entre révéler quelques bits de l'état interne et le fait de vouloir aller vite et de produire beaucoup de bits.
On prend un grand entier M qui a plus de 1024 bits et est appelé entier sûr. Il est le produit de deux nombres premiers de même taille. M est un nombre qui va être difficile à factoriser. On va choisir x au hasard et on vérifie que x0 n'est pas divisible ni par P ni par Q (M=PQ). Mais il y a 1/2^1024 chances que X0 soit dans ce cas la. On va donner le premier bit de parité de X0 (premier bit du PRNG). Puis au fait X0 au carré modulo N. Ici on est dans la centaine voir le millier de cycles pour un bit en sortie. A titre de comparaison, AES-CTR est de l'ordre de 30 cycles/127 bits. Ce générateur peut sortir environ 1Mb par seconde, ce qui est très bien pour générer des clés (long terme).

Pour choisir un modelé de chiffrement, il faut regarder la résistance de l'algorithme dans le temps face aux avancées des mathématiques. Par exemple, on peut estimer que AES sera cassé dans 10 ans, donc il faut que dans 10 ans, le message chiffré soit obsolète. Blum Blum shub est certes lent, mais peut etre employé à long terme pour générer une clé car c'est un prng quand meme assez sur ! Il servira donc à généré des clés importantes et qui doivent avoir une longue durée de vie (clés maitres,...).
Preuve formelle qui montre que si QR est difficile Alors BBS indistingable de l'aléatoire.
QR représente le problème de la residuosité quadratique : il consiste à retrouver Y a l'aide de $X=Y^2$ mod(N). C'est hyper difficile mais reste plus facile que la factorisation. Ni prouvé, ni infirmé.
Est-ce que 3 est un carré modulo 15 ? Il faut prendre tous les carrés et calculer les modulos. Est-ce que 7 est un carré modulo 15 ? On sait pas. Il faut faire des essais.

Un contre exemple : Dual_EC_DRBG

Générateur de nombres aléatoires déterministes basé sur l'utilisation de courbes elliptiques. Il fait partie du standard du NIST SP800-90A auquel on se réfère pour construire des PRNG déterministes. C'est le standard que ces générateurs doivent respecter. Il y avait 3 types de générateurs proposés. Le dernier était le DRBG qui était une backdoor de la NSA. Un biais statistique suffisamment fort avait été introduit, il permettait de retrouver la clé puis de déchiffrer toute une communication. Pour contrer, des chercheurs on breveté Dual_EC_DRBG, ils sont alors devenus propriétaires de cette backdoor. Ça n'a donc pas été révélé au grand publique. La NSA s'est retournée vers le Bullrun program. La NSA s'y aussi intéressée pour que le standard utilise Dual_EC_DRBG (car le générateur était troué, bien sûr). Le NIST a fini par recommander l'abandon de Dual_EC_DRBG.

On a ici un exemple du fait que le passage de la preuve au standard peut introduire des failles, la preuve formelle n'est toujours pas suffisante. En effet, mêmem si formellement le chiffrement est sur, l'implémentation peut le rendre très fragile. Des coefficients mal choisis, quelques tours en plus ou en moins, et on introduit un biai important. Bien évidemment, cela peut aussi être recherché (NSA).

Cryptographie

Chapitre 5 - La gestion des clés

I. Introduction

L'expression "gestion de clé" recouvre tout un ensemble d'étapes de la vie d'une clé. On rencontre en particulier deux problèmes majeurs de cette gestion de clé : la distribution et la dérivation.

Le premier problème touche donc à l'échange de clé : on a pas de canal sûr si on ne peut pas se voir physiquement pour échanger un clé, en tout cas pour le commun des mortels (peut etre différent pour l'armée, ..). Le problème est encore pire quand on est un groupe de personnes. Un solution peut être de tous se réunir et de s'échanger des clés un à un. Dans ce cas, le nombre de clés va grandir quadratiquement (N^2 clés pour N personnes). Souvent, ce n'est pas possible de voir tout le monde, ni même de prévoir avec qui on peut potentiellement avoir besoin de chiffrement et donc d'échanger une clé commune dans le futur.

On peut de plus avoir un groupe dynamique, avec des gens qui entrent et d'autre qui partent, donc le temps de mettre en place des clés, le groupe a changé. Des systèmes de trousseaux peuvent être mis en place mais cela reste extrèmement couteux et complexe.

Une autre solution serait d'avoir un secret partagé. Le problème est qu'il y a beaucoup d'entrées / sorties de gens et ce n'est donc pas terrible pour la confidentialité de la clé. Il y a des systèmes qui existent pour gérer les groupes dynamiques.

Exemple avec Thales : Les gens abonnés devaient revenir tous les ans avec leur carte pour la changer. Ils voulaient un système plus pratique. Ils avaient donc besoin d'un autre canal sécurisé.

Utilisation d'une tierce partie de confiance

Principe :

- Toute nouvelle personne s'enregistre auprès de la tierce partie de confiance (TTP) et échange par un canal sécurisé une clé unique.
- Toute communication avec une TTP est sécurisée (confidentialité + intégrité) par cette clé

Ainsi on peut imaginer deux modes d'utilisation de la TTP:

- Quand un utilisateur A veut communiquer avec B, il demande à TTP de transférer un secret k_{ab} à B. Cette clé de session permettra à A et B de communiquer directement sans passer par la TTP. Il n'y a ainsi que les couples qui ont voulu communiquer un jour qui auront une clé commune. Mais cette clé de session peut aussi être éphèmère.
- Les communications peuvent toutes passer par la TTP qui servira de relai entre A et B.

Le défaut principal ici est qu'il faut que A, B, C, D.. fassent tous confiance à un même tiers. Une telle unité entre pays, entreprises, etc.. N'est pas crédible. De plus il y a un besoin de passer par le tiers de confiance même si A et B sont a portée directe. Ainsi, si ils sont cote à cote mais coupé du réseau, ils ne pourront pas communiquer de manière sure ensemble. Pour finir, la TTP est un point central dont dépendent beaucoup de machines, potentiellement. Ainsi, si elle tombe, toute la structure s'effondre.
Pour pallier à ce dernier point, une entreprise de l'envergure de THalès peut mettre en place un système de TTP hiérarchisé géographiquement. Ça concentre tout de même la menace au niveau régionale par exemple. Dans ce cas, unr TTP gère plusieurs TTP, etc etc

De plus Kerberos peut mettre en place des systèmes de ticket qui fait des systèmes d'autorisation. Il permet d'échanger des clés ou encore avoir une entité qui dit que telle ou telle personne peut faire une certaine action en lui collant un Hmac vérifiable dessus. En effet, Kerberos partage une clé symétrique avec tout le monde dans le réseau.

On peut faire des choses en gestion de clé en symétrique, en passant par un tiers. C'est possible en interne dans une entreprise. Dès que c'est inter entreprise c'est plus compliqué.

II. Les premiers échanges de clé

Les **puzzles de Merkle** constituent la première construction à clé asymétrique (1974). Elle permet à deux partis de se mettre d'accord sur un secret en commun par l'échange de messages, et sans que ces partis n'aient préalablement de secret commun.

1. Alice génère 2^{k1} (k1=30) puzzles de la forme Enc(W_{ki}, id_i || s_{ki}). W_{ki} correspond à la wrapping key et contient k2 bits aléatoires , id_i et sk_i ont k1 + k2 bits aléatoires. Ici id_i est un identificateur de message (l'identifiant du puzzle) et sk_i est la clé du puzzle. L'identifiant est généré aléatoirement.

2. Alice envoie à Bob les 2^{k1} puzzles. Bob reçoit tous les puzzles et en choisit un au **hasard**. Il résout le puzzle dans un temps raisonnable (tests les 2^{k2} clés de wrapping possibles) et obtient une clef sk_i et un identifiant de puzzle id_i. Il envoie id_i à Alice.
3. Alice retrouve sk_i correspondant à l'identifiant reçu. Alice et Bob peuvent ensuite utiliser sk_i comme clé commune pour leurs communications.

Que se passe t'il pour Eve? Imaginons qu'elle capte tout ce qui est écrit dans le canal. Elle reçoit elle aussi les 2^{k1} puzzles envoyés à l'étape 2. Elle ignore par contre le choix de Bob en 3. Elle n'a d'autre solution que de résoudre tous les puzzles, et tenter les 2^{k1} clés qu'elle en a extrait. Cependant, attention, il peut y avoir un écart dans les possibilités de résolution de Bob et d'Eve. Il est tout à fait possible qu'elle sache résoudre le million de puzzles plus vite que le temps nécessaire à Bob pour le puzzle qu'il a choisi.

En résumé, le coût pour l'émetteur est de 2^{k1}, le coût pour de récepteur de 2^{k2} et celui pour l'attaquant est de 2^{k1+k2}. C'est donc plutôt bien tenté même si cette méthode ne permet pas d'avoir un coût polynomial pour A et B et exponentiel pour l'attaquant. Ca rend quand même le travail assez lourd pour ce dernier.

Cette techniques peut être implémentée dans des communications entre deux capteurs.

Diffie Hellman

Généralités

Justement, l'algorithme de Diffie-Hellman (1976) permet d'avoir un coût polynomiale pour l'utilisateur et exponentiel pour l'attaquant. Un attaquant qui écoute voit qu'un secret est échangé mais ne peut rien en apprendre. En revanche un attaquant qui peut modifier les échanges peut tout casser. En particulier, ce protocole est vulnérable à « l'attaque de l'homme du milieu », qui implique un attaquant capable de lire et de modifier tous les messages échangés entre Alice et Bob. Pour comprendre le proto de DH, il y a quelques bases de maths à comprendre.

Un peu de maths

On travaille modulo un nombre premier p (supérieur à 1024 bits, généralement pris à 2048 bits mais aussi 3072 ou 4096). Pour des raisons qui seront évidentes, on ne prend plus p = 1024 bits sauf pour des applications obsolètes, ce n'est plus recommandé. Au lieu de le choisir de plus en plus grand, on verra qu'on choisit une autre alternative.

On cherche ce nombre premier de la forme p = 1 + Kq avec q de 160, 224 ou 256 bits. On fait varier K de la bonne taille jusqu'à tomber sur un nombre premier.

> On note $\mathbb{Z}_p$ (ou $\mathbb{Z}/p\mathbb{Z}$) l'ensemble $\{0, \ldots, p-1\}$ muni de l'addition et de la multiplication réduites modulo p (simplification)
>
> Il existe (et on peut trouver facilement) un élément g de $\mathbb{Z}_p$ générant q éléments distincts : c'est-à-dire $g \in \{0, \ldots, p-1\}$ tq $\{g \bmod p, g^2 \bmod p, \ldots, g^q \bmod p\}$ soit de taille q

Les q éléments sont différents. On va trouver ce g tel que le reste g^q mod(p) = 1. On dit que g est un élément d'ordre q.

Deux choses à retenir jusque là :

- Quand je somme et multiplie je travaille modulo un grand nombre
- J'ai un nombre g qui multiplié par lui même donne plein d'autres nombres et quand je le multiplie q fois par lui meme ça donne 1. Autrement dit : g engendre un ensemble de taille q.

On peut prendre un exemple :

p = 11
p-1 = 10
Je prend par exemple g = 2. On regarde groupe engendré par g

<g> = {2, 4, 8, 5, 10, 9, 7, 3, 6, 1}
en effet : 2^1 mod(11) = 2
2^2 mod(11) = 4
2^3 mod(11) = 8
2^4 mod(11) = 5 etc...
g permet de générer un groupe de taille 10.

Déroulement de l'algorithme

En notant :
– g, un générateur de nombres modulo p | Informations connues de Alice et
– p un nombre premier de 2048 bits | Bob (echangé en clair)

Et
– a, un nombre au hasard, modulo p, dans <g> connu d'Alice
– b, un nombre au hasard, modulo p, dans <g> connu de Bob

Avec de chaque coté la logique :

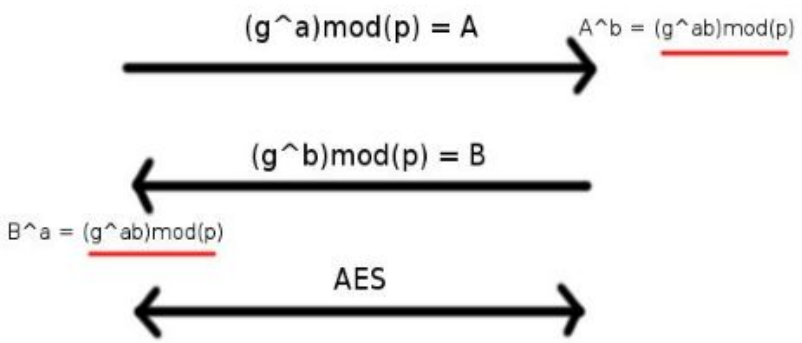
- Index de l'élément = clé privée
- Ce que vaut l'élément = clé publique ($A = g^a mod(p)$ et $B = g^b mod(p)$)

A tire son secret $priv_A$ dans {0, .., q-1}. Il va ensuite calculer sa clé publique comme étant : $pubA = g^{privA} mod(P)$. Il a donc pris un élement au hasard de l'ensemble <g>. A envoie pub_A à B. B fait pareil de son coté. A récupère pub_B et fait $skA = pubB^{privA}.mod(p)$ et B va faire pareil de son côté $skB = pubA^{privB}.mod(p)$. Ils obtiennent donc au final le secret commun $sk = g^{privA.privB}.mod(p)$

La preuve montrant que $sk_A = sk_B$ est :

$A^b \bmod p = (g^a \bmod p)^b \bmod p = (g^a)^b \bmod p = (g^b)^a \bmod p = (g^b \bmod p)^a \bmod p = B^a \bmod p$

On peut illustrer le protocole de Diffie Hellman de cette façon :



Complexité N = $\log_2(p)$? [...]
A chaque bit je fais soit une soit deux multiplications modulaire. Donc complexité entre ...
Algo du square and multiply très important en cryptographie

L'algorithme du square and multiply est très important en cryptographie, en voici une application :

Enoncé :

- Choisir nombre petit entre 10 et 50 qui va être p. On prend p=29.
- Choisir g uniformément entre 0 et p-1. On prend g = 14
- Calculer $g^{42} mod(p)$ avec l'algorithme du square and multiply.

NB : 41=101010

Résolution :

- Initialisation :
  - On initialise la variable résultat à 1.

- Itération 1 :
  - Le 1er bit (en partant du LSB) de 42_2 = 101010 vaut 0 donc on garde $résultat = 0$
  - On calcule : $14^2 mod(29) = 22$

- Itération 2 :
  - Le bit suivant est 1 donc on fait $résultat = 1x22.mod(29) = 22$
  - On calcule : $22^2 mod(29) = 20$

- Itération 3 :
  - Le bit suivant est 0 donc on garde $résultat = 22$
  - On calcule : $20^2 mod(29) = 23$

- Itération 4 :
  - Le bit suivant est 1 donc on fait $résultat = 23x22.mod(29) = 13$
  - On calcule : $23^2 mod(29) = 7$

- Itération 5 :
  - Le bit suivant est 0 donc on garde $résultat = 13$
  - On calcule : $7^2 mod(29) = 20$

- Itération 6 :
  - Le dernier bit est 1 donc on fait $résultat = 20x13.mod(29) = 28$

- Résultat : $14^{42} mod(29) = 28$

Le problème du log discret

Problème du log discret, pour comprendre :
Je vois pub_A, je sais que c'est g^x, je vais trouver x qui correspond à la clé privée de A. Le problème du log discret est facilement résolu quand p a peu de bits et quand le sous groupe engendré par g est petit.

La réciproque est fausse : Rien ne dit que quelqu'un qui casserait Diffie Hellman casserait le log discret (jusqu'à maintenant en tout cas). De même rien ne prouve que casser Diffie Hellman est plus

difficile ou plus facile que de casse le problème du log discret. On a donc confiance en Diffie Hellman car son fonctionnement est basé sur un problème difficile et aussi car Diffie Hellman n'a pas été cassé en 40 ans.

En pratique Diffie Hellman est vulnérables aux attaques de l'homme du milieu :

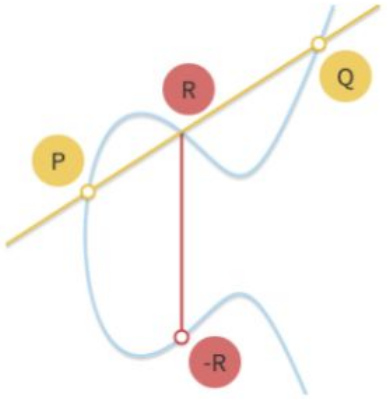
> **Source : Wikipédia**
>
> *Un attaquant peut se placer entre Alice et Bob, intercepter la clé g^a envoyée par Alice et envoyer à Bob une autre clé $g^{a'}$, se faisant passer pour Alice. De même, il peut remplacer la clé g^b envoyée par Bob à Alice par une clé $g^{b'}$, se faisant passer pour Bob. L'attaquant communique ainsi avec Alice en utilisant la clé partagée $g^{ab'}$ et communique avec Bob en utilisant la clé partagée $g^{a'b}$, Alice et Bob croient communiquer directement. C'est ce que l'on appelle « attaque de l'homme du milieu ».*
> *Alice et Bob croient ainsi avoir échangé une clé secrète alors qu'en réalité ils ont chacun échangé une clé secrète avec l'attaquant, l'homme du milieu.*

De plus comme dit en introduction, on espère avoir un système d'échange de clés de complexité polynomiale pour A et B et de complexité exponentielle pour un attaquant. Or dans le cas de Diffie Hellman dans un corps fini (comme décrit précedemment), on peut avoir des attaques de complexité allant jusqu'à $n^{\frac{1}{2}}$. Or quand on a une attaque en $n^{\frac{1}{3}}$ ou $n^{\frac{1}{2}}$ etc.. ça veut dire que pour que l'attaque ait un coût au carré il fait multiplier par 2² ou 2³ etc.. la taille des clés. C'est la raison pour laquelle on aussi la possibilité d'utiliser Diffie Hellman non pas sur des corps finis mais sur les courbes elliptiques.

Pour Diffie Hellman dans un groupe générique, l'idée générale est « je me met dans un endroit ou le log discret est difficile et je prend une base d'un groupe g de grande taille». On calcule (g OP g OP g .. OP g) = pubA et on fait ça PrivA fois au total avec PrivA grand et OP la multiplication modulaire.

Sur les courbes elliptiques : ECDH

Courbes elliptiques de Wierstrass : ensemble des points de coordonnées entières (a,b) tel qu'ils vérifient l'équation $y^2 = x^3 + ax + b$. Si on trace la tangente entre deux points aux coordonnées entières, cette courbe va toucher un troisième point qui lui aussi a des coordonnées entières. On définit ainsi un groupe avec comme opération somme la propriété de la tangente (P+Q = R). Dans ce groupe, le log est difficile.



Pour passer des corps finis aux courbes elliptiques : multiplier dans un corps fini reviens à sommer deux points dans une courbe elliptique. Le Square and Multiply devient le Double and Add.

$$A = g^a,\ B = g^b,\ sk = g^{ab} \rightarrow A = aG,\ B = bG,\ sk = abG$$

Ainsi, sur les courbes elliptiques, le meilleur algorithme pour résoudre le log discret n'est plus en $n^{\frac{1}{2}}$ comme dans les corps finis mais en $\sqrt{p}$ qui n'est donc pas sous exponentiel. On va pouvoir prendre des plus petits paramètres. Un autre avantage qu'on peut trouver est qu'on a des sommes à la place des multiplications... Est-ce que c'est bien de faire des sommes ? Moyennement car au passage on a aussi rajouté un calcul de pente de la courbe, etc..

Suite au passage sur les courbes elliptiques, combien d'opérations supplémentaire pour l'ordinateur ce changement représente-t-il ? et surtout est-ce que des opérations sont chères ? Le meilleur algorithme pour résoudre le log discret est en $\sqrt{p}$ qui est bien exponentiel (en le nombre de bits de p), c'est à dire qu'on grandit linéairement en faisant grandir les exposants.

Concernant la sécurité, sur les courbes elliptiques, pour avoir une attaque en X bits il fait avoir 2X bits de sécurité à cause de l'attaque en $\sqrt{p}$. C'est vrai sauf pour le dernier 512→ 521 qui n'est pas in rata car il a de bonnes propriétés. En ECC on grandit linéairement la sécurité mais en FFC ca grandit cubiquement. On parle de géométrie projective quand on parle de la sécurité en courbe elliptique.

Jusqu'à 112 on peut utiliser la cryptographie sur les corps finis, après non

Voici l'algorithme utilisé pour réaliser P + Q = R

```
Input: P = (X1,Y1,Z1), Q = (X2,Y2,Z2)
Y1Z2 = Y1*Z2; X1Z2 = X1*Z2; Z1Z2 = Z1*Z2
u = Y2*Z1-Y1Z2; uu = u2
v = X2*Z1-X1Z2; vv = v2; vvv = v*vv
R = vv*X1Z2; A = uu*Z1Z2-vvv-2*R
X3 = v*A
Y3 = u*(R-A)-vvv*Y1Z2
Z3 = vvv*Z1Z2
Output: P+Q = (X3, Y3, Z3)
```

Le Z est fixé à 1 au début mais il ne reste pas à 1, il grandit. Sa présence permet de ne jamais faire de division qui coûtent horriblement cher. On peut se débrouiller, avec Z pour ne faire des multiplications et des additions.

Exemple pour k=128 :

Sur les corps finis, log(p) est 12 fois plus grand. Donc le coût par multiplication est de 12x12 c'est à dire quadratique par la taille. Donc le coût par multiplication est multiplié par 12*12 = 144.
En revanche dans les courbes elliptiques, le nombre d'opérations à faire est multiplié par 14. Pour un coût de base de une multiplication de 256n par rapport à un coût de base pour une multi de 256, FFC va être ça (Cmulti256) fois 144 et ECC va être 14 multi de 256bits donc ça fois 4. Donc le rapport est environ de 10.
FFC = C(mult256)*144
EEC=C(mult256) *14 ==> FFC/ECC=10

Avantage de Diffie Hellman

Ce qui est avantageux avec le protocole de Diffie Hellman, c'est que A et B peuvent s'envoyer les clé publiques dans un ordre qui est indifférent. Cette possibilité de non synchronisation est importante dans beaucoup de domaines, par exemple là ou il y a beaucoup de latence (satellite). Exemple : On suppose qu'Alice publie sa clé publique sur Facebook. On peut généraliser, on a 200 amis et on a la clé publique de chacun. Si Alice veut communiquer avec C, combien de messages doivent-ils s'envoyer pour arriver à un secret commun ? Aucun. Car quand on veut contacter quelqu'un, on peut deviner quel sera le secret commun et directement envoyer un chiffré vers lui (si on a accès à sa clé publique). On suppose qu'on utilise le standard du NIST P256 qui fournit p, g, etc.. Ce non besoin d'échange est valable pour DH uniquement. Il ne faut utiliser que le modèle des courbes de Weierstrass car les autres sont cassés.

Fragilité : Le Man in the middle sur Diffie Hellman

Voici un exemple avec le principe des puzzles de Merkle. L'idée de base est que quand on voit Alice envoyer g^{privA} et Bob envoyer g^{privB}, l'hypothèse de Diffie Hellman computationnelle nous dit " qu'il est difficile de passer de {p, g, g^a, g^b} à g^{ab}". A et B ont comme objectif d'obtenir un secret commun sans que personne d'autre ne l'apprenne. L'attaquant peut vouloir s'arranger pour

que A obtienne un secret commun avec lui, C, et pas B. En effet, l'hypothese précédente ne dit pas que qu'on ne peut pas usurper l'identité de B.

La première manière de gruger B est de tuer B, de venir voir A et de lui dire “bonjour je suis B”. L'autre méthode si on ne veut pas tuer B est de se mettre entre A et B. Dans ce cas l'attaquant s'est placé en Man in the Middle. Il a la capacité d'écouter et de modifier.
Cette attaque est possible car A et B ne se connaissent pas et n'ont pas de secret commun. C'est facile pour C de se faire passer pour A auprès de B. C peut très bien envoyer à B un nombre de 2048 bits qui servirait de clé publique. B réagirait et établirait un secret commun avec C en croyant que c'est A. Comment y remédier ?

- Être dans un groupe relativement fermé de N personnes où tout le monde connaît les clés de tout le monde.
- Avoir créé cette liste en se réunissant ou éventuellement en échangeant par un canal sûr. Le problème reste l'intégrité et la confidentialité, il fait utiliser un canal sûr pour garantir l'intégrité. On peut donner un hash. Si on connaît un hash de la clé publique de B, quand on nous donne sa clé publique, on peut vérifier que c'est bien la sienne. On peut donc mémoriser des hashs et aller les verifier (par téléphone ou n'importe) quand on reçoit une clé.

Au final ce que l'on veut c'est pouvoir récupérer des clés publiques via des canaux non sûrs et pouvoir être certain qu’elles appartiennent à la bonne personne, donc on va utiliser des **certificats signés**.

Si A s'est fait voler sa clé privée, alors quel que soit l’interlocuteur de A plus tard, C pourra toujours reconstituer les échanges car il aura acces à la clé publique de l’interlocuteur, échangée en clair. De plus, quelqu'un qui vole les clés un an apres avoir enregistré les communications pourra les déchiffrer ! Cependant, il existe une version de Diffie Hellman qui permet de résoudre ce problème. En effet, Diffie-Hellman a une version avec des clé éphémères « DHE » (ou « ECSHE » dans le cas avec des courbes elliptiques). L'idée c'est que A et B vont échanger leurs clés publiques (dîtes statiques ou long terme). On ne fait pas de changement régulier de ce couple de clé, ça réglerait le problème mais cela serait pénible à gérer. Un correspondant voyant que notre clé publique a changé pourrait penser qu’on s’est fait pirater par exemple. Ainsi, on fait plutôt de la génération de clé occasionnelle, juste pour une communication. Juste après l'obtention du secret commun, on l'efface immédiatement, et le secret utilisé sera en réalité un hash de la concaténation des deux secrets (le secret commun et le secret éphémère). De cette façon le secret commun obtenu a deux garanties. Alice se dit qu’elle reconnu la clé publique de Bob et donc elle sait qu’elle parle à Bob ou à quelqu’un qui a sa clé privée. Elle dispose de plus d’un autre secret éphémère qui assure la confidentialité persistante.

NB sur les clés éphémères : On ne concatène pas n’importe quoi à la clé privée habituelle. On fait attention à ce qu'il y ait quelques chaines de caractères qui donnent des infos sur la connexion (comme l’adresse MAC de l'interlocuteur, la date, etc…). Ceci permet d'éviter la répétition du même contexte: en effet, le bout de message rajouté dépend intrinsèquement de la communication et changera à chaque fois. Ces caractères dépendent de l'application (même s’il y a des idées dans les standards). Ce bout de message, appelé bind, joue le rôle d’anti replay mais permet aussi de faire passer un bon nombre d’informations pratiques entre A et B. A l’aide de ce segment, A et B pourront se mettre d’accord sur certains paramètres de la communication. Ainsi une bonne pratique est de mettre le plus d’informations possibles dans le bind.

Si on utilise EDH (ou ECSHE) alors pour toutes les communications futures, si l'attaquant veut mettre en péril la confidentialité de l'échange, il devra faire une attaque active en se plaçant en homme du milieu pour récupérer les clés éphémères à chaque fois.

> ***Source : Wikipédia***
>
> *La **confidentialité persistante** (forward secrecy), est une propriété en cryptographie qui garantit que la découverte par un adversaire de la clé privée d'un correspondant (secret à long terme) ne compromet pas la confidentialité des communications passées.*

Standards pour DH

Dans le standard, il y a plusieurs protocoles qui définissent l'utilisation des clés statiques ou éphémères. Par exemple on note C(2s,2e) pour l'utilisation de deux clés statiques et deux clés éphémères. L'utilisation de la clé statique va permettre de s'authentifier (comme la clé est liée à notre identité) et le clés éphemères permettent d'assurer la confidentialité persistente (protection si la clé statique est révélée).

Petit tour sur les standards:

C(2s) : Permet l'authentification mutuelle
C(2e) : Confidentialité persistante
C(2s,1 e) :

A → pubA B
A pubB ← B
A → pubA' B
A pubB← B

On obtient un secret statique avec le protocole de Diffie Hellman normal. Ensuite A utilise sa clé éphémère et B son unique clé (privée). Si jamais la clé statique de B est révélée, on pourra déchiffrer une communication à postériori. Par exemple un site qui héberge beaucoup de clients utilise une clé statique et une clé éphémère, et les clients peuvent n'utiliser qu'une clé statique. On pourra déchiffrer les communications du client si on obtient sa clé statique. On ne pourra pas déchiffrer les communications du serveur par contre car il utilise des clés éphémères.Dans ce cas la A peut bien s'authentifier et permet la confidentialité persistante si privA est dévoilée. Du coté de B par contre, il n'y a pas de confidentialité persistante !! Mais B s'est bien authetifié

C(1s,2 e) : A utilise une clé statique et une clé éphémere et B utilise qu'une clé ephemere. A va etre identifié et va avoir de la confidentialité persistante si sa clé privée est dévoilée. Par contre B est non

authentifié mais possède de la confidentialité persistance. Ca correspond bien à ce qui se passe dans un site web : un client veut etre sur qu'il se connecte au site X et voudrait que si un jour quelqu'un se connecte à X, ce qui se passe dans le passé soit pas dévoilable. Par contre le site X, il s 'en fou que le client soit authentifié car de toute facon, tout le monde peut se connecter à lui. Donc pas la peine de demander au client de fournir une clé statique. On gagne donc des échanges.
Pour un site de banque par exemple, on veut les memes caractéristiques que précédemment mais en plus on veut que le client s'authentifie (on veut être sûr d'être connecté à la banque ET qu'il y ait de la confidentialité persistante). On peut effectuer le premier échange avec ECDHE (HTTPS) et on obtient un canal sur. Mais de toute facon, le client va s'authentifier à postériori (login/mdp) donc pas besoin d'utiliser une clé statique pour le client (on peut omettre l'authentification du client). On reste en C(1s,2 e).

C(1s,1 e) : Ce qui se passe en pratique lorsque l'on utilise ECDH ou DH. La banque n'associe pas à notre identité une clé, donc elle s'en fou de la clé statique.

Protocole MQV : protocole à 4 échanges qu'il voit comme un unique échange et va directement faire un calcul sur les variables. Il est plus rapide que faire deux fois DH par exemple.

III. Les fonctions à trappe

Introduction

Dans le contexte du chiffrement symétrique, nous étions obligé de mettre en place des mécanismes coûteux pour gérer des groupes de N personnes : d'avoir une clé partagée totale, une clé par couple (N² clés) ou un tiers de confiance. Ce modèle de clé partagé est très faible car il faut pour chacun un canal sur et cela augmente le risque de fuitage de la clé. Dans le contexte de la crypto à clé publique, on remplace le problème de la confidentialité des clés par l'intégrité des clés (bien moins compliqué à géer). De plus le partage de clé est clairement résolu.

Source : EPFL

Une fonction trappe est en fait une famille de fonctions bijectives f_t, indexée par un paramètre t (la clé de la « trappe »), telle que chaque fonction est à sens unique mais telle que, lorsque t est connu, f_t^{-1} est facile à calculer.

L'utilité en cryptographie d'une fonction à trappe est la suivante : chaque utilisateur choisit au hasard (et en secret) une clé, disons t, et publie f_t (mais pas t lui-même !). Habituellement f_t est prise dans une famille de fonctions de manière à ce que seuls quelques paramètres doivent être publiés. Ces paramètres sont appelés « clé publique ».

Si quelqu'un veut communiquer un message m aux personnes dont la fonction à trappe publiée est f_t , il envoie simplement f_t (m), qui est facile à calculer comme f_t est à sens unique. Pour obtenir le message correct, le destinataire calcule f_t^{-1} qu'il lui est facile de calculer comme il connaît la clé t. Ce calcul est toutefois difficile pour toute personne qui ne dispose pas de la clé.

Diffie Hellman sert à faire potentiellement des échanges de clés (potentiellement car si on utilise que des clés publiques, pas besoin de se les échanger), pas à chiffrer. On utilise après cet échange des chiffrements de type AES, etc.. Les fonctions à trappe sont la partie “magie” du chiffrement asymétrique.

Le chiffrement asymétrique

Une fonction de chiffrement asymétrique fonctionne avec une de paire de clés (ou bi-clé) qui est composée d’une clé publique qui servira à chiffrer, et d’une clé privé qui servira à déchiffer (et réciproquement). Cette bi-clé est générée par un algorithme keyGen(1^k). Ici, le 1^k correspond à l ‘écriture de k en unaire. Quand on veut représenter un nombre k, on écrit simplement k fois ‘1’. C’est une norme de représentation. Cette notation est utilisée pour mettre en avant l’une des (mauvaise) propriétés de l’algorithme KeyGen: c’est un algorithme exponentiel.

Un algorithme est intéressant si il est polynomial en la taille des entrées pour les deux parties et l'algorithme de l'attaquant exponentiel en la taille de l'entrée. Il y a donc 3 algorithmes : KeyGen, Enc et Dec. Avec ces algorithme on doit avoir de la consistance (on retombe sur le même message si on chiffre puis déchiffre le même message avec les même paramètres de sécurité pour toute paire de clé identique. Il que les deux clés soient différentes et que la connaissance de la cé publique ne nous donne pas de renseignements sur la clé privée). Comment faire un système de chiffrement qui est sûr ?

Propriété du chiffrement asymétrique : Si je prends un message quelconque au hasard (tiré uniformément) et je donne un chiffré C de ce message à un attaquant, tout algorithme polynomial prenant en entrée le couple (C,PubA) a une chance négligeable de trouver le message ou la clé privé.

Bilan : C'est pénible de ne pas avoir le droit de choisir son message, si il faut qu'il soit tiré uniformément. De plus d'après cet énoncé, l'attaquant ne trouvera peut etre pas m' = m mais il pourrait trouver des informations comme sa parité, 1000 bits sur 1024, … Vu comme ça, RSA n'a pas l'air très intéressant. On peut souhaiter des propriétés plus fortes.

La **sécurité sémantique** est l'équivalent de la sécurité parfaite dans Shannon (le chiffré C ne nous apprend rien sur M, ce qui revient à dire que C et M sont indépendants). La sécurité sémantique dit : soit un algorithme polynomial prenant en entrée C et les paramètres publics, alors cet algorithme nous donne des informations négligeables sur M. Autrement dit, en un temps polynomial on apprend presque rien sur m.

L'**indistingabilité** dit que pour tout couple de messages potentiellement choisis par l'attaquant l'ensemble des chiffrés possibles pour M0 est indistinguable des chiffrés d'un M1. L'objectif est d'atteindre une indistinguabilité entre les clairs qui nous permet d'obtenir la sécurité sémantique. On propose un défi : J'ai envoyé "Bonjour" ou "Au revoir", je donne le chiffré avec les deux options : l'attaquant n'apprend rien en un temps polynomial et ne peut pas dire si j'ai dit "bonjour" ou "au revoir".

Face à quel attaquant veut on avoir indistingabilité ?

- **Indistinguishability under chosen-plaintext attack (IND-CPA)** : On donne la clé publique à l'attaquant, il génère ce qu'il veut, de notre côté on génère un chiffré qu'on envoie. L'attaquant doit trouver notre message. Il est important que le système de chiffrement ne soit pas déterministe et que la fonction de chiffrement soit randomisée. Ainsi pour un message M, il y a un ensemble de chiffrés possibles. Sinon dans le cas contraire, l'attaquant peut brute forcer.

- **Indistinguishability under chosen ciphertext attack (IND CCA1)**. Le challenger envoie à l'attaquant la clé publique et l'attaquant va pouvoir choisir des chiffrés qu'il veut et demander au challenger de les déchiffrer. On dit que l'attaquant à accès à un oracle de déchiffrement (il peut demander au challenger de lui déchiffrer des choses). Ensuite l'attaquant est confronté au challenge : il envoie deux clairs m0 et m1 (ou un ensemble E de clairs possibles) au challenger, qui les chiffre et lui renvoie le chiffré d'un des deux clairs (choisi aléatoirement). L'attaquant ne peut plus utiliser l'oracle de déchiffrement et doit retrouver le clair.

- **Indistinguishability under adaptative chosen ciphertext attack (IND CCA2).** Après avoir été confronté au challenge, l'attaquant peut continuer à utiliser l'oracle. Cependant l'attaquant ne peut pas demander de déchiffrer le message m sur lequel porte le challenge. L'attaquant, de la meme façon que précédemment donne un ensemble E de clair, le challenger en chiffre un et il le renvoie. L'attaquant doit retrouver le clair.

Si un système ne donne pas de protection IND CCA2 (contre chiffrés choisis adaptatif) alors, on peut casser la sécurité. Comment fait l'attaquant pour accéder à un oracle de déchiffrement ? Si on est pas face à un humain mais un programme on peut se débrouiller pour obtenir des bits d'informations progressivement. De plus, il faut bien faire attention de ne pas sortir des modèles définis, même un tout petit peu.

Authentification

Le serveur va prendre un nombre au hasard, de K bits et il va m'envoyer un défi, qui correspond au chiffré de ce nombre. Il me suffira de déchiffer ce message et de renvoyer le clair au serveur. Ainsi je me suis bien authentifié car je suis le seul détenteur de la clé privée (et donc le seul à pouvoir déchiffrer). Ainsi, ce schéma apporte de la sécurité sémantique, à partir de ce chiffré, un attaquant ne peut rien apprendre du clair et donc ne peut pas usurper mon identité. Mais on peut se dire qu'avec les K bits on peut toujours chercher parmi les 2^k combinaisons possibles ce qui permettrait une attaque par force brute. Or il faudrait mener une attaque online dans ce cas ce qui complique la tache de l'attaquant donc ce k peut être petit.

Quel est l'avantage de cette méthode par rapport à un serveur avec un mot de passe ? Ici la sécurité ne dépend pas de la configuration du serveur ! La sécurité est du côté client. Mais dans ce cas là, on devient pour le serveur un oracle de déchiffrement. Donc il est important qu'on soit IND-CC1 ou 2. L'autre bon point c'est que le serveur ne stocke que la clé publique, c'est à dire rien de sensible, donc on obtient rien si on le hack.

Le principe de cryptographie hybride est de passer d'un chiffrement qui est seulement à trappe à un système qui donne de l'indistingabilité.

Obtention d'un cryptosystème à clé publique IND-CCA2

Enoncé du théorème (prouvé dans le modèle de l'oracle aléatoire)

Avec : (KeyGen, Enc, Dec) une fonction à trappe
H un CSHF
(E_s , D_s) un système de chiffrement symétrique authentifié (Authenticated Encryption)

Pour :

$$Enc'(pub, m) = \{x \xleftarrow{U} M, sk = CSHF(x), y \leftarrow Enc(pub, x), \text{ return } y \| E_S(sk, m)\}$$
$$Dec'(priv, y\|c) = \{x = Dec(priv, y), sk = CSHF(x), \text{ return } D_S(sk, c)\}$$
$$KeyGen' = KeyGen$$

Alors : (KeyGen', Enc', Dec') est un chiffrement à clé publique IND-CCA2

Commentaires :

- Une fois que le clair est chiffré, on prend le résultat et on lui colle un MAC. Par exemple un CBC-MAC basé sur AES128. Si on fait ça avec une fonction de chiffrement sûre comme AES, alors on se rapproche très fortement de la propriété d'IND-CCA2.

- On peut montrer que c'est optimal de chiffrer d'abord et mettre le MAC ensuite (et pas l'inverse). Puisqu'il y a une trapdoor function, on sait que l'attaquant ne peut pas obtenir X dans sa totalité. Mais dans le cas particulier ou X est Sk, alors l'attaquant pourra avoir des renseignements sur Sk. Dans le modèle de l'oracle aléatoire : si l'attaquant n'a pas

exactement x, et fait appel à la fonction de hachage, il tombera sur un truc complètement différent (propriété des fonctions de hachage).

- Une fonction à trappe impose l'utilisation d'aléa en entrée (premier défaut, contourné) et ne nous dit pas si l'attaquant apprend quelque chose sur le clair à partir du chiffré ou pas. Elle nous dit qu'il ne l'apprend pas exactement (2eme défaut contourné grâce à la fonction de hachage).

Un chiffrement symétrique comme AES à une rapidité de l'ordre du multigigabit en chiffrement et en déchiffrement. Avec un chiffrement asymétrique comme RSA, on va avoir un chiffrement qui est de l'ordre des 500 Mbits/sec en chiffrement et beaucoup plus long en déchiffrement. Meme si on avait des chiffrements à clé publique qui permettent d'êre IND CCA2, ils resteraient vraiment trop lents. Il existe un moyen de tirer profit des deux types de chiffrement.

Dans la littérature industrielle, on ne parle pas de cryptographie hybride mais du paradigme de KEM-DEV. Avec le KEM (Key Encryption Mecanism), on chiffre une clé symétrique avec un système de chiffrement à clé publique (Public Key Crypto System, PKCS). Le DEM (Data Encryption Mecanism) est un système de chiffrement symétrique pour chiffrer les données. En pratique on rajoute un MAC sur le chiffé.

On peut aussi trouver le KEM-KWS car ce qu'on trouvera dans le 56B, comme c'est un standard pour les echanges de clés, ce sera du matériel cryptographique (secret commun).
En crypto quand on chiffre une clé avec un autre clé, on appelle ca du key wrapping.
Comment passer d'un chiffremeent à trappe vers un IND-CCA2 sans passer par du KEM-DEM.
Le très bon point du KEM-DEM c'est la rapiditè de traitement.

Exemple d'utilisation du KEM-DEM :
Avec RSA (RSA KEM-KWS) KWS car dans ce standard par exemple ce ne sont pas des données qu'on transporte mais du matériel cryptographique (secret commun, …)

Optimal Assimetric Encryption Padding (OAEP, 1994)

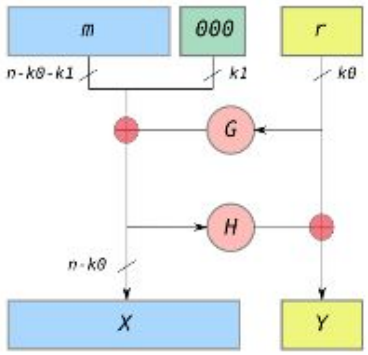
Si le message M est petit, on va le concatétner avec beaucoup d'aléa et donc avoir de très bonnes propriétés. Le problème est si M est grand et surtout si il est structuré. On ne peut mettre que très peu d'aléa et dans le reste du message, on aura peu d'entropie. Donc une bonne méthode, serait de positionner un masque sur tout ca mais pourquoi masquer l'aléa et comment regénérer ce masque ? Et bien ce qu'on fait c'est qu'on prend une graine aléatoire, on la met au début du message. Avec cette graine, on génère un masque uniformement aléatoire et on le xor avec le message (on laisse la graine en clair au début). Ceci marcherait mais en pratique on ne va pas laisser la graine comme ca, on va faire un feed back sur la graine pour la cacher aussi.

Dans le modèle de l'oracle aléatoire, si RSA est un chiffrement à trappe, à l'aide la construction OAEP, on peut passer à une caractéristique IND-CCA2. Dans la structure de l'OAEP, l'attaquant est obligé de connaître plusieurs données sur le message pour espérer le déchiffrer. Par contre ATTENTION, ca ne marche qu'avec RSA ! Avec les autres systèmes de chiffrement ce n'est que IND-CCA1. Mais avec l'OAEP+, on peut passer de n'importe qu'elle fonction générique à trappe, à de l'IND-CCA2.

```
| taille | Aléa | Message |
|          MASQUE           |
```

Mais comment retrouver le masque pour déchiffrer ? On utilise plutot une graine (aléatoire) et on génère un masque avec un PRNG.

```
| Graine | Message |
|      Masque      |
--------------------
      + Chiffrement
```



- n taille en bits des clairs
- G et H fonctions de type SHA récentes
- $k0$ longueur de sortie de la fonction H
- m paddé avec des zéros jusqu'à une taille $n - k0$
- G utilisée avec un compteur de 32 bits pour avoir une sortie assez grande $G(r||0)||G(r||1)\ldots$ (puis on tronque)

Source https ://en.wikipedia.org/wiki/Optimal_asymmetric_encryption_padding

Il reste un problème, on fait un dernier XOR pour masquer la graine. Notations :

- r graine
- 000 masque
- m message
- G PRNG
- H SHA256

Etapes :

- XOR : pour masquer la graine
- On chiffre X.Y

Tout ceci est inversible. Dans ce cas, l'attaquant ne peut pas découvrir de sous partie (r, X, …). Même si le message de départ était structuré, l'attaquant voit de l'aléatoire.

NB : RSA est à trappe
standard PKCS : Public Key Cryptographic Standard. Le numéro 1 dit comment bien faire du chiffrement et de la signature, et il y en a plein.
Revenons sur les utilisations. Si on utilise RSA avec un KEM-Dem ou OAEP, tout envoi de message peut être un envoi de clé.

Chiffrement à clé publique : Utilisations [..] déjà vu

RSA : Bases

La fonction PHI d'Euler donne le nombre d'éléments inversibles modulo M. Mais ce qui est surtout très important, c'est le théorème d'Euler.

Modules et fonction phi

Pour un module m la fonction $\phi(m)$ définit la taille du groupe multiplicatif
Th d'Euler : $\forall x \in \mathbb{Z}_m, x^{\phi(m)} = 1 \bmod m$

Je prend X aléatoire et je connaît phi(m). J'élève à la puissance phi(m) avec l'algorithme du Square and Multiply. Une fois fait j'obtiens 1mod(m).

Fonction phi pour un module RSA

Si $N = p * q$ avec p, q premiers, on a $\phi(N) = (p-1)(q-1)$

Pour M un nombre qui est le multiple de deux nombres premiers on a l'égalité donnée. A partir d'un grand nombre supérieur à N, nombre étant le produit de deux premiers de même taille p et q, alors trouver p et q (factoriser N) est très difficile. On peut assez facilement montrer que à partir de N, trouver phi(N) est équivalent à factoriser en terme de difficulté. Mais par contre, si on connaît N et Phi(N), alors en cherchant les racines d'un polynomes de second degré, on peut trouver P et Q.

$\phi(N) = p * q - (p+q) + 1$
Si on connait $\phi(N)$ et $N = p * q$ on connait $p + q$
Quelles sont les racines de $X^2 - (p+q) * X + (p * q)$?

Techniquement un attaquant, à partir d'un N, ne peut ni obtenir P et Q, ni obtenir Phi(N).
Dans RSA, pour définir keygen, on définit une taille de module en suivant les recommandations du NIST ou ANSSI (K et log2(N)) :

Taille N	K
1024	70
2048	112
3072	128

On itère pour trouver deux entiers de 1027 bits par exemple et je fais la multiplication des deux une fois trouvé. Je définit e = 2^16+1 et on va utiliser l'algorithme d'Euclide étendu pour trouver d (l'exposant de déchiffrement) et une constante tel que e*d = 1+const*phi(N). On retourne donc une clé pub/priv qui correspond au couple (e,N)/(d,N).

Pour chiffrer on fait M^emod(n) et déchiffrer c^dmod(N)

On peut montrer la consistance :

$$c^d = m^{e*d} = m^{1+const*\phi(N)} = m * m^{const*\phi(N)} = m * (m^{\phi(N)})^{const} = m$$

Pour chiffrer, avec SaM, on fait 17 opérations. Pour déchiffrer on fait 3072 opérations.
C^dmod V avec d de 2048bits => on fait 2048square + 1024multiply (uniforme donc 1/2 d'avoir un '1') = 3072

Exemple avec e=3
N = 15 = 3*5
phi(N) = 2*4 = 8 → Il y a 8 nombres qui sont inversibles (?)
Enlevé tous les nombres qui sont des multiples de 3 ou 5 car sinon ça complique les choses.

1^3mod(15) = 1
2^3mod(15) = 8
4^3mod(15) = 4
7^3mod(15) = 13
8^3mod(15) = 2
11^3mod(15) = 8
13^3mod(15) = 7
14^3mod(15) = 14

RSA sécurité (1/2)

$N = 15$
$? \longleftarrow 1$
$? \longleftarrow 2$
$? \longleftarrow 4$
$? \longleftarrow 7$
$? \longleftarrow 8$
$? \longleftarrow 11$
$? \longleftarrow 13$
$? \longleftarrow 14$

Facile à trouver pour 1 et 8 → Il n'y a que 1^3 qui donne 1 et de même $2^3 = 8$, on peut trouver l'antécédent. Cependant plus e est grand, moins c'est possible d'être confronté à ça. Le seul problème qui peut rester est 1.

Le chiffrement est à (log2N) carre et le déchiffremnt (log2N) au cube : M est en log2N, le square fait log2n * log2N donc du coup le carré du chiffrement et pour déchiffre on fait ca multiplier le 1/2 chance d'avoir un multiply donc au cube.
Si on est un attaquant, on peut essayer toutes les possibilités et ca a un coup exponentiel. Ou alors on prend un algorithme de factorisation.
Le RSA nu, ou appelle RSA RAW, est un système de chiffrement déterministe. Donc on a pas d'indistinguabilité avec RSA. Un attaquant pourra facilement résoudre le jeu d'indistinguabilité avec M0 et M1 car il lui suffit d'avoir la clé publique, qu'il a forcément.
L'hypothèse de RSA, c'est que c'est une fonction à trappe, cad qu'un attaquant peut apprendre des choses sur le X mais ne peut pas le trouver.

RSA sécurité (2/2)
cf Matthieu

Attention à RSA sans IND-CCA2 !
Utilisation d'une trapdoor function sans entrée uniforme : pose problème.
- Tout message de moins de k bits va être fragile. Un m petit est brute forçable. Si l'entropie est inférieure à k (ce n'est pas un m uniforme), on ne respecte pas les hypothèses nécessaires pour la trapdoor function, et on est vulnérable à des attaques en moins de 2^k. On peut avoir des attaques plus efficaces que la sécurité supposées.
- Autre attaque : Meet in the Middle. Laissée en exercice. Si un serveur envoie un challenge de k bits où je dois trouver le r, alors il y a une probabilité assez forte que r soit le produit de deux nombres de k/2 bits chacun. Dans ce 20 % de cas on va trouver une attaque qui est plus efficace que la précédente, en 2^(k/2).

Ainsi, les paramètres de sécurité sont respectés si on est dans le cas de la fonction à trappe.

Autres attaques à prendre en compte
- Petit e ou petit d :Mauvais.
- Analyse de temps ou de courant (attaque par canaux auxiliaires): L'algorithme de base pour beaucoup de systèmes de chiffrement est le SaM, ou DaA. Cette méthode de calcul va traiter de manière différente le cas ou on est face à un ‘1’ et face à un ‘0’. Ainsi, en faisant une analyse de courant, on peu deviner si le HW vient de traiter un bit à un ou à zéro et ainsi en déduire les parmètres. De plus, les algorithmes de SaM sont connus et leur signature “énergétique” ‘est de même. On peut de plus utiliser une mesure de temps, complémentaire ou substitutive à l’analyse de courant. Du coup on utilisera plutôt des algorithmes moins efficace que le SaM, mais qui se font en temps et opérations constantes (faire toujours un square et toujours un multiply → On perd 30 % d'efficacité mais au moins c'est pas vulnérable à cette attaque).
- Manque d'aléa pour Keygen → Factorisation massive : Sur les serveurs openSSL, le pool d'entropie trop petit pour p, donc p vivait dans un espace petit. Cependant le pool grandissait assez ensuite et q était généré dans un espace plus grand. De ce fait, faire un PGCD sur N permettait de trouver p, puis ainsi q.
- Faute avec le CRT : […]

→ Être très prudent avec la cryptographie à clé publique.

Et les échanges de clés? (1/2)

On va appelé RSA-SVE pour prendre un z uniformément au hasard dans l'espace des messages, on le chiffre et on l'envoie à B. Et B retourne à A de l'aléa. Ainsi le secret commun sera A concaténé avec cet aléa. Le problème et que du coup, on a une partie connue et une partie secrete. Don con met une règle : tout échange de clé doit se finir par une dérivation de clé.

Finalisation d'un échange de clés

Suite

Si on veut faire une confirmation de clé, il faut faire une étape en plus dans le protocole d'échange

Cryptographie

Chapitre 6 - Signatures, certificats et PKI

I. La signature

Introduction

L'objectif d'une signature est d'assurer l'intégrité d'un message et garantir l'authentification. Il est peut être important (dans un contexte légal par exemple) d'assurer qu'une personne et bien la seule à avoir réaliséune certaine action sur un document. De même, si on appose notre signature sur un document mais que celui ci est modifiable, cela pose un problème. Il est donc indispensable d'assurer l'intégrité quand on appose une signature.

On peut alors se demander si une signature électronique et un MAC sont la même chose. Au final avec le MAC on fait à peut près la meme chose. On va poser un tag de façon volontaire qui signifie qu'on valide le message. On veut que le message ne puisse pas être modifier : si il l'est, le tag n'est plus valide.
Alors qu'apporte en plus une signature ?

Alice va créer une bi-clé avec un algorithme à clé publique (de type RSA, Diffie-Helllan ou ECDH) et elle va publier sa clé publique. Cette clé, tout un tas de personnes vont pouvoir la recevoir. Un jour Alice va vouloir contacter quelqu'un, Bob par exemple. Elle lui envoie un message m signé : sigma = SIGN(priv, m). L'algorithme SIGN est pour le moment non définit mais il est très important de notre qu'Alice utilise sa lé privé dans cette action. Ainsi, en utilisant son secret, ce qui la définit car il n'est connu que par elle, Alice va s'authentifier. Ainsi, on dit qu'Alice signe le message m. Cette biclé est utilisée pour la signature uniquement et ne sera surtout pas réutilisé pour d'autres usages tel que le chiffrement (en cryptographie on utilise une clé différente pour chaque usage). Ainsi Alice aura un autre ensemble de clé pour réaliser le chiffrement. Dans les signature, c'est la personne qui à la bi clé qui génère le message. C'est tout l'inverse du chiffrement : c'est Bob qui chiffre avec la clé publique et qui l'envoie à Alice. Le sens du message et la clé utilisée sont opposés !

Bob reçoit le message, le lit et se demande si la signature est bonne. Il utilise la clé publique d'Alice, la signature et le message et demande à son algorithme une vérification. Si l'algorithme VERIFY dit que le triplet (m, pub, sigma) est valable, c'est bon.

En pratique cet algorithme VERIFY est très sensible. Dans le cas d'une clé de type Diffie Hellman par exemple, il faut vérifier que g et p sont valides, que la clé publique a le bon format, etc.. On considère ici toute entrée comme quelque chose de potentiellement malveillant. Cependant on va plutôt se concentrer sur la partie cryptographie ici. Quand l'algorithme dit que le triplet est valide donc, de quoi est-ce qu'on peut être sûr ? Il n'y a pas de notion de temps, donc on n'est pas protégé contre un rejeu. **La seule chose dont on est certain c'est que c'est la clé privée associée à la clé publique qui a été utilisée.**

Meme si le test est vérifié, la seule chose dont Bob est certain c'est que c'est la clé privée associée à pub qui a été utilisé. Mais il y a potentiellement deux problèmes. D'abord je ne suis pas certain que cette clé publique soit bien celle d'Alice. Il faut donc trouver un canal sur pour vérifier cette clé à postériori. Mais ce problème n'est pas résolu de façon très satisfaisante. Il faut pouvoir etre certain du lien entre clé publique et l'identité d'Alice.
Le second problème c'est que n'importe quel attaquant qui connaît la clé privé (a-t-elle été volé ? A t'elle été révoquée ?…) peut faire cette signature. Ce problème engendre le problème de la révocation.

Concernant la sécurité des signature, une chose importante est qu'un attaquant ne doit pas pouvoir faire d'*existential forgery*, c'est à dire qu'un attaquant, ne connaissant pas la clé privé, ne peut pas créer un couple message-signature si il n'a pas déjà vu passer un couple avec ce meme message avant.

Attention : Pour des utilisations différentes, on utilise des clés privées différentes. La bi-clé utilisée pour la signature ne doit pas être la même utilisée pour l'échange de clé ou autre chose.

Exemple d'utilisation

On dispose d'un logiciel, disons un OS. Un certain nombre d'utilisateurs disposent de cet OS. Ainsi chacune de cette personne dispose d'une clé publique notée en dur, dans son code. On appelle A l'éditeur de l'OS. Si A veut déployer un patch, il l'envoie, concaténé avec une signature pour ce patch. Ceci, par l'intermédiaire d'un système multi agent se diffuse et chaque ordinateur peut voir qu'il y a un patch disponible. L'OS vérifie que la signature est valable, et si c'est le cas il installe le patch.

C'est plus compliqué si c'est Adobe par exemple qui veut envoyer un patch pour sa version sur l'OS. Le plus simple serait qu'il l'envoie à A qui diffuserait, mais à grande échelle c'est impossible. En pratique, les éditeurs de logiciel s'enregistrent auprès de A (protocole à définir) et peuvent ensuite diffuser leurs patchs.

A l'aide d'une Intergrity Key, on pourrait créer un MAC qu'on pourrait concaténer au message. Mais du coup soit chaque utilisateur connaît IK en dur dans son système et peut donc vérifier que le tag correspond bien au message. Mais cette technique se base sur une clé symétrique. Donc tout le monde peut créer un MAC avec Ik ! Il n'y a pas de dissymétrie, l'éditeur est pas plus fort que n'importe quel utilisateur.

Une autre option est aussi d'éviter que tout le monde connaisse lk. L'éditeur peut donc décider de partager une clé avec chaque utilisateur dans le monde. Ainsi l'éditeur possédera des millions de clés, il devra insérer une clé dans chaque pc, et il ne pourra pas distribuer les patchs car il devra faire un tag spécifique à chaque utilisateur, i.e des millions d'envoie unicast.

Rappel sur les clés symétriques partagées :

En symétrique, lorsque l'on doit gérer un groupe, on peut utiliser une clé partagée commune, mais cela n'est généralement pas sûr. On peut aussi faire le choix d'utiliser des clés symétriques deux à deux, ce qui est horriblement cher. Ainsi, on pourrait réaliser l'exemple précédent (sur les patchs) avec des MAC mais ça serait horriblement cher.

Il existe une autre différence entre les MAC et les signature. Si on utilise des MAC, en cas de soucis, il est impossible de savoir qui en est l'origine. En effet, dans ce cas, A et B aurait le même secret, il n'y a pas de dissymétrie, on a aucune preuve permettant de prouver si c'est A ou B qui a causé le problème. La signature possède donc, à la différence des MAC, la notion de non répudiation pour l'éditeur de signature. La non répudiation de l'expéditeur de la signature : c'est la base de la justification légale de la signature électronique. On ne peut attribuer à autrui les méfaits qu'on a commis.

La non-répudiation signifie la possibilité de vérifier que l'envoyeur et le destinataire sont bien les parties qui disent avoir respectivement envoyé ou reçu le message. Autrement dit, la non-répudiation de l'origine prouve que les données ont été envoyées, et la non-répudiation de l'arrivée prouve qu'elles ont été reçues.

Schéma de signature

Un cryptosystème à clé publique (PKC) est composé de trois algorithmes

- KeyGen(1^k) : algorithme randomisé donnant en sortie une bi-clé (pub/priv)
- Sign(priv, m) : algorithme randomisé donnant en sortie une signature de m
- Verify(pub, m, sigma) : algorithme randomisé donnant en sortie true ou false.

Consistance : Pour toute bi-clé (et tout k) générée par KeyGen(1^k) on a

$$Verify(pub, m, Sign(priv, m)) = true$$

Et si...on faisait une signature avec RSA Raw. Si on utilise comme signature la fct de decipher de RSA (M^d mod N) et comme onction de vérification la fonctio de chiffrement (Enc(Sign,pub)==M=Sign^e mod N).
Inverser l'action de chiffrement et déchiffrement (cipher puis decipher ou decipher puis cipher) nous permet toujours d'obtenir le message en résultat. Sign^e mod N= (M^d)^e modN=M^(d*e) modN = M^(e*d) modN

Mais ce système est clairement très faible face à la Existential forgery : on peut créer de nombreux couples. Dans le cas général si on prend le cas ((sign^e modN)/M , sign) alors en faisant la méthode de vérification, elle valide à tout les coup.
Supposons de plus que l'on voit passer (M1, σ1) et (M2,σ2). Alors en multipliant les deux, si les messages sont structurés, on peut avoir des informations.
Tout cela vient d'une mauvaise propriété de RSA qui s'appelle la malléabilité. Mais on peut la casser grace à une fonction de hashage. Ce qu'on a fait mal, c'est qu'on a laissé le choix à l'attaquant du message. Donc si l'attaquand peut mettre ce qu'il veut dans un message, on a pas nécesseraiement des garantit qu'il ne puisse pas le déchiffre (RSA est une trapdoor function). L'attaquant ici ne sait pas faire l'opération de déchiffrement, il récupère juste l'image et la manipule un peu pour faire un tour de passe passe. C'est du au fait que l'on laisse un grans espace à l'attaquant.

[Matthieu → Calcul consistance]
[Matthieu → Suite]
Je choisis une signature au hasard.
Je peux créer des milliards de couple message / signature mais pas créer une signature pour un message donné.
[Matthieu]
Ce qu'on a mal fait c'est qu'on a laissé le choix de l'attaquant du message. Si il peut mettre ce qu'il veut dans le message. [Matthieu].

RSA- FDH Crash course

Il existe deux raisons d'utiliser RSA FDH (Full Domain Hash) plutôt que RSA-RAW. C'est l'utilisation de RSA avec une fonction de hachage qui nous fait tomber dans tout l'espace des messages de RSA.
Algorithme :
KeyGen peut être SHA1 avec un compteur. On peut facilement faire des fonctions de hachage qui ont la bonne taille de sortie par rapport au module RSA qu'on utilise.
La génération d'une signature n'est pas faite par l'utilisation directe de la methode de chiffrement sur M, mais il a un prétraitement sur le message : on le hashe.

Verify : Vérifier que sigma est bien un déchiffré de H(m)

H fonction de hachage $\{0,1\}^* \to \{0,1\}^n$ pour $n = \log_2(N)$

- $KeyGen_{RSA-FDH}(1^k) = KeyGen_{PKC}(1^k)$
- $Sign_{RSA-FDH}(priv, m) = Dec_{PKC}(priv, H(m))$
- $Verify_{RSA-FDH}(pub, m, \sigma) = (H(m) == Enc_{PKC}(pub, \sigma))$

Ca a un avantage pratique car comme le fonction de hashage prend en entrée des messages de taille arbitraire, donc ce schéma permettra de signer tout et n'importe quoi, il faut juste que n soit entre 0 et N-1. Le second avantage est que c'est plus rapide sur des gros messages: on le prend, on le traite a toute vitesse par la fonction de hachage et on applique la fonction à clé publique dessus. On aurait du couper le message et faire des trucs compliqué et couteux alors que la, la fonction de hashage est rapide.
Prendre un nombre au hasard, chiffrer, on sait pas déchiffrer
Donner un nombre au hasard, pareil on sait pas déchiffrer
La trapdoor function, assure qu'on saura pas déchiffrer.

II. Les infrastructures

Éditeurs d'OS et de logiciels : retour sur l'exemple précédent
Pour chaque patch m, un éditeur distribue (patch, sigma) avec

$$\sigma_{patch} = Sign(priv_{Éditeur}, patch)$$

Le problème est que les utilisateurs ne peuvent pas connaître les clés publiques de tous les éditeurs. De plus, même s'ils les connaissaient, comment les utilisateurs feraient-ils pour s'assurer qu'ils disposent bien de la bonne clé publique ? N'importe quel attaquant pourrait fournir un patch en utilisant sa clé privée et en faisant utiliser aux clients la clé publique associée.

C'est là qu'interviennent les racines de confiance, pour régler le problème de l'authenticité de la clé publique de l'éditeur. En l'appliquant sur notre exemple : on peut partir de la base en constatant que tous les utilisateurs disposent de la clé publique de leur éditeur d'OS. L'éditeur de logiciels envoie sa clé publique à l'éditeur d'OS. Ainsi l'éditeur va s'engager légalement uniquement pour bien faire le lien entre l'identité de l'entreprise et la clé publique. Il va approuver le certificat de Adobe. L'éditeur génère le message suivant « Je suis l'éditeur d'OS et voici la clé publique de l'éditeur de logiciel ». Il renvoie à l'éditeur de logiciel son certificat qui est ce message signé par la clé privée de l'éditeur de logiciel c'est à dire :

$$certif_{Éditeur} = (m, \sigma_m) \text{ avec } \sigma_m = Sign(priv_{OS_Y}, m)$$

Par exemple Microsoft, fait payer l'éditeur qui vient lui livrer son logiciel et sa clé publique. Microsoft assure la correspondance logiciel / clé publique de l'éditeur en apposant sa signature dessus. Ainsi l'utilisateur recevra le triplet (patch, certificat, signature de l'éditeur). L'utilisateur va récupérer le certificat et va le déchiffrer à l'aide de la clé publique de l'éditeur d'OS (qu'il possède de toute facon) et on va faire une vérification. L'utilisateur pourra donc être certain que l'éditeur de logiciel est un éditeur connu, existant, que son identité est réelle et que la clé publique qui est donné

est bien la sienne. On va donc faire une transmission de confiance : on fait confiance à la racine qui est notre OS et du coup on fait confiance à Adobe.
Maintenant que l'utilisateur fait confiance à l'éditeur, il lui suffit de vérifier que le logiciel est bon à l'aide de la signature..

Exemple
Certificat de Adobe = Clé publique d'Adobe / logiciel / Signature de Microsoft.
Si on fait confiance à l'éditeur d'OS alors on sait qu'on a la bonne clé publique pour Adobe par exemple. Il y a donc une transmission de confiance : « Je connais la bonne clé publique pour mon éditeur d'OS donc je connais la bonne clé publique pour Adobe ». Il peut y avoir plusieurs niveaux de hiérarchie de clé publique en clé publique.

Passage à l'échelle

Public Key Infrastructure : exemple hiérarchique

$cert_{CNRS}, cert_{INRIA}, cert_{INPT}, \ldots$ signés par la clé privée du MESR
$cert_{IRIT}, cert_{LAAS}, cert_{LIP6}, cert_{LIX}, cert_{LIENS}, \ldots$ signés par la clé privée du CNRS
$cert_{CarlosAguilar}, cert_{EmmanuelChaput}, \ldots$ signés par la clé privée de l'IRIT

Chaque $cert_X$ signé par Y sous la forme :
(m="Clé de X signée par Y"$||\sigma_m = Sign(priv_Y, m)$)

Vérification hiérarchique

pub_{MESR} dans tous les ordinateurs des fonctionnaires (racine de confiance)

Emmanuel envoie à Carlos $(m, \sigma_m, cert_{EmmanuelChaput}, cert_{IRIT}, cert_{CNRS})$
Carlos vérifie que $cert_{CNRS}$ est bien signé par le MESR, puis utilise pub_{CNRS} pour vérifier que $cert_{IRIT}$ est bien signé par le CNRS, puis utiliser pub_{IRIT} pour vérifier que $cert_{EmmanuelChaput}$ est bien signé par l'IRIT, puis utilise $pub_{EmmanuelChaput}$ pour vérifier que (m, σ_m) est bien signé par Emmanuel Chaput

Tous les fonctionnaire ont la clé publique du Ministère de l'Enseignement Supérieur et de la Recherche (MESR). Le MESR génère des certificats pour le CNRS, l'INPT, etc... Ces certificats contiennent : « Je suis le MESR et ceci est la clé publique du CNRS ». Le MESR signe ce message avec sa clé privée, ce qui donne sigma_m. Ainsi toute personne qui a la clé publique du CNRS peut vérifier ce message. Un adversaire ne peut donc pas envoyer un message avec une fausse clé du CNRS.

Je ne connais que la clé publique du MESR donc je reçois d'abord le certificat du CNRS signé par le MESR. J'obtiens, la clé publique du CNRS. J'ai confiance car c'est signé par le CNRS. De même jusqu'à obtenir la clé publique d'Emmanuel. Les PKI hiérarchique peuvent être utilisés pour une hiérarchie nationale mais aussi pour avoir une hiérarchie de sécurité.

L'**enregistrement est une étape très importante**, la confiance qu'on a dans une clé publique est fortement liée à la qualité de l'enregistrement. Il est légalement important que la clé privée ne soit connue que de la personne qui l'a générée car elle engage sa responsabilité pour tout ce qui est fait avec.

Exemple concernant l'enregistrement : Verisign
[…] Quelques mots pour présenter Verisign

La certification peut etre sur une infrastructure hiérarchique nationale. Verisign est une entrerpise qui possède LA clé secrète qui est à la base de tout une hiérarchie de confiance. Elle paye très cher et fait mettre sa clé publique dans absolument tous les appareils ! Ensuite elle possède des certificats de classe 2 par continent, puis de classe 3 par pays, puis de classe 4 dans le pays, etc etc. En bas de l'échelle, il y a les clients qui payent pour avoir une signature de verisign. Ainsi, Verisign a mis en place une structure hiérarchique tentaculaire. C'est une hiérarchie de sécurité !!
Une cérémonie de clé c'est le changement du couple clé prive et publique. Du coup, cette cérémonie pour une boite comme verisign coute des millions.
Il y a beaucoup d'autorites de certification du type de verisign ;

Infrastructure HTTPS de l'internet

Quand on se connecte à une page web, on utilise un échange TLS qui commence par une vérification de certificat. Dans chaque navigateur, il y a une liste de toutes les autorités de certification. Le problème principal c'est que toute autorité peut signer tout lien . La plus faible des autorités de certification presente dans un navigateur, peut fournir un certificat à n'importe quel serveur web meme ceux qui paye pour des authorités qui sont vraiment très cheres et sures. Si il y en avait peu, avec un mecanisme mondial d'audit, on pourrait avoir beaucoup plus de confiance dans les infrastructures à clé publique du monde de l'internet. Malheureusement, tout pays veut participer, tout le monde veut s'en meler, et ca créé des trous immenses utilisés par des hackeurs qui generent des masses de faux certificats.

Certificate pinning:
On peut dire qu'on ne fait pas confiance à toutes les autorités de certification. Par exemple je peut dire que pour un update je ne veux que la PKI de Microsoft. Chome n'accepte des certificats de google que si ils viennent à l'authorité de Komodo ! En effet, chrome sait que google à payer pour komodo donc on refuse les autres.

III. Les standards

FIPS 186-4 : Digital Signature Standard (DSS)
Il propose en pratique 3 algos : un RSA en mode FDH probabilisé qui s'appelle RSA PSS.
Au lieu de faire juste un hachage du message, on fait un hachage puis un padding randomisé.La randomisation permet de montrer très clairement ..

Autres :
- DSA : type DH
- ECDSA : type ECDH pour les signatures

FIPS 186-4 : C'est le standard de signature. En pratique il propose trois algo : un RSA en mode FDH probabilisé qui s'appelle RSA-PSS. Ca a juste une meilleure preuve de sécurité que RSA-FDH qui permet d'assurer que la securité de la signature est equivalente à celle du chiffrement. Au lieu de faire juste un hash du message, on fait un hash du message plus un padding randomise, exactement comme OAEP. La sécurité est moins lié a celle du chiffrement, il est un petit peu plus facile de casser la signature que le chiffrement. Puis il y a DSA, qui est l'utilisation d'une construction de type DH. Et ECDSA qui se base sur ECDH.

Certificats X.509. Le format Pem est le format lisible pour les certificat car il est fait pour etre envoyé par mail. Le format DER est généralement binaire et donc illisible.
Chaque certificat possède un numéro de serie qui l'authentifier uniquement.
Les certificats ont des contraintes sur eux : des durées de validités, … Un certificat est de plus utilisée que pour un usage (web ok, mais pas de WIFI).

Comment fait on pour que Alice prévienne tout le monde si elle se fait voler la clé ?Dans les champs du certificat, il y a l'adresse d'un hôte qui possèdes de CRL qui et la liste des certificats non révoqués. Donc si je reçois un certificat et je vois que la CRL est valide, je lui fais confiance, sinon on.
Si on me vole ma clé privé, qu'advient t'il des documents que j'ai signé dans le passé ? On ne saura pas quels sont les certificats fait avant et apres vol. On peut faire des certificats avancés. On signe et on envoie à un service de time storm. Ainsi on pourra récupérer le contrat mais qui aura une certification de la date, meme malgré la certification.

Rappel sur les echanges de clés: On a deux actions possibles

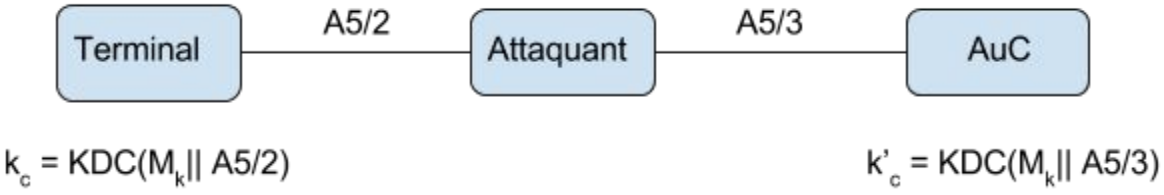
- Un échange de type DH : les deux personnes font la meme action (envoie de la clé publique et dérivation des clés)
- Utiliser un cryptosysteme a clé publique qui soit au moins une fonction à trappe et cette fonction a trappe on va l'utiliser de facon naturelle c'est a dire en tirant un message aléatoire et en le chiffrant et en l'envoyant à l'autre chiffrée. Et ensuite B va contribuer d'une façon ou d'une autre (en envoyant un bout d'aléa par exemple pour contribuer à l'obtention du secret)
  On peut faire des accords ou les deux contribuent a l'obtention du secret ou du transport de clé. C'est une approche beaucoup plus assymétrique au partage de clé.

NB: les fonctions a trappes ne sont sures que si on sélectionne un clair au hasard!

Du coup on trantransforme RSA pour qu"'il soit sur. Soit on utilise un masquage randomisé bien fait soit on utilise un chiffrement symétrique authentifié (keywrapping) avec un transport de la clé sym authentifié.

Après cette obtention de secret commun, on a pas de clé de session a utiliser directement, on a juste du materiel de cryptographie.

Vu en reseau cellulaire : avoir des secrets non attachés à une notion de session est dangereux donc quand on fait la dérivation d'un secret, on le rattache à un contexte pour obtenir une clé de session (date, chaine qui decrit A, décrit B, décrit les algos utilisés, sens d'utilisation, etc..) on concatène ceci au secret et on dérive pour obtenir la clé K_c = KDF(M_k || "Contexte").
Si il n'y a pas de contexte pour rattacher le secret ça pose un problème.



- Dans le sens "→" : envoi de l'IMSI (en clair)
- Dans le sens "<--" : AuC dit qu'il fait A5/3 et l'attaquant dit qu'il fait A5/2. (en clair)
- L'attaquant pourra casser K_c car il a demandé l'utilisation d'A5/2 au terminal
- Si on utilise pas de contexte, alors, l'attaquant connait aussi K'_c.

Comme vu précedemment, un moyen de faire un accord de protocole, ou plus généralement de rattacher le secret à un contexte particulier, est de concaténer le secret à une chaine de caractère qui correspond au contexte.
Il faut donc etre sur que si on s'est mis d'accord sur un secret commun, il faut aussi se mettre d'accord sur la manière de l'utiliser. Si un attaquant qui se positionnerait entre le client et le serveur, et qui voudrait faire des choses différentes sur chaque coté, il faut que l'on aboutisse à des clés de session différentes et donc qu'on ne puisse pas communiquer.
A la fin d'un accord de clé, on ne sait pas si on parle avec la bonne personne ou non, car elle ne m'a envoyé que des éléments publics. L'échange de clé ne nous garantie pas qui est notre interlocuteur. Cela est garantie par le fait que apres, lorsqu'on a la communication chiffré, ca marche.
Une possibilité, c'est que des qu'on s'est échangé les clés, on s'envoie un mac d'une chaine de caractère, par exemple le contexte. On va utiliser le début de la clé échangée pour généré ce mac. Ce mac va controler l'ensemble des communications qui ont été envoyées et recues. On va signer la transcription des communications : onl va concaténer tous les envois qui on été fait, tous les envois qu'onl a recu, et les éléments définis dans la norme (DestA||DestB). Ainsi, si un attaquant modifie ne serait ce qu'un bit, on saura qu'il y a eu une attaque ! Mais attention, jusqu'à cette confirmation de clé, on ne sait pas.

But de la dérivation de clé :

- Extraction : Dans le cadre d'un échange de clé, le but est de réaliser une extraction d'entropie. Par exemple, on a un résultat de DH modulo un nombre de 2048bits P. MK=g^(privA*PrivB)mod P.
  Mais DH ne nous garantit pas que P est uniforme parmi les nombres de 2018bits. DH nous assure juste 128bits de sécurité, ca veut dire que l'entropie est au moins de 128bits. La distribution nous donne la sécurité d'une distribution uniforme de 128bits. On a une chaine de caractère de 2048bits mais on a pas autant de possibilités. Ce qu'on veut donc c'est extraire l'entropie de ces 2048bits.
  NB = g^(privA.PrivB.mod q) → g^q=1

- Binding (rattachementà un contexte) : En meme temps qu'on va prendre le grand nombre M_k et on va utiliser un fonction de dérivation pour concaténer des bits autres et s'assurer l'unicité pour une utilisation donnée.

- Expansion : A partir d'un petit nombre avec beaucoup d'entropie, on peut le dériver pour obtenir un nombre plus long qui a l'air tiré uniformément. Une fois on a un secret uniforme bien concentré autour un certain nombre de bit, on va pouvoir utiliser ces bits comme graine pour un PRNG pour dériver et avoir une plus longue chaine. On appelle ca l'extension. Ca nous permet de dériver pour avoir tout un nombre de clés pour pleins d'usages différents.

- Extraction + renforcement : Au lieu d'utiliser un mdp pour chiffrer on utilise une clé dérivée de ce mot de passe. A chaque essai de déchiffrement il faut d'abord dériver la clé du mot de passe et ensuite essayer de déchiffrer. On utilise ici une KDF très lente ce qui va nettement ralentir les attaques possibles. On va en fait renforcer la clé. C'est une extraction mais avec l'option de la KDF lente

Comment est-ce qu'on dérive ?
On a un MK qui est grand et qui a peu d'entropie, qui a des propriétés structurelles fortes à certains endroits : des séries de '0', des propriétés mathématiques sur le nombre, etc.. On va changer ca en un autre nombre qui est peut etre plus long, qui ne possède pas forcement plus d'entropie, mais qui va avoir de bien meilleures propriétés statistiques.

Le PKCS1 enseigne comment utiliser RSA pour du chiffrement ou de la signature. On définit dans ce standard une focntion KDF.
Le principe des KDF1,2,3 est identique : on a un secret maitre Z, on va avoir un contexte qui va définir ce qu'il faut mettre dans une application donné, et un compteur. La différence entre les KDF, c'est la manière dont on implémente le compteur. On génère ensuite un hashé de cette concaténation. Et on va générer autant d'hachés que nécessaire pour obtenir en sortie une chaine suffisamment longue. A chaque tour, on changera juste le compteur.
La fonction de hashage n'est pas définie dans les standards, il faut utiliser ceux recommendés par le NIST: SHA2 et SHA3.

KDF3 donne la possibilité d'utiliser une HMAC basée sur n'importe quelle fonction de hachage standard. On va reprendre le meme format de message que précedemment (contexte, compteur,...) et on va appliquer du padding avec des zero ou un sel, qu'on peut transmettre en clair, qui permet d'intégrer de l'aléa dans le haché même si notre contexte est faible (en effet il peut se répeter). Ainsi, meme si on fait un replay, le nouveau sel va assurer que le secret qu'on va dériver sera tout de même différent.
NB : Quelque soit la taille de la clé en AES, les blocs de sortie et d'entre sont de 128bits !!! Une grosse lcé augmente juste le nombre de round**.**

L'extraction est faire en prenant le présecret maitre et on recommande d'utiliser un sel aléatoire. Pour faire cette extraction on va utiliser le MAC qu'on utilise (HMAC ou AES-CMAC avec comme clé le sel obtenu et comme message le pre secret maitre). A partir de cela, on obtient une petite clé. On voulait une sécurité de K bits et on a K bits assez uniforme. On va la concaténer à un contexte. On peut donc utiliser un PRNG cryptographiquement sur qui va utiliser Key Derivation Key comme graine, pour générer l'aléa qui est demandé. On veut dériver un ou plusieurs secrets avec au moins k bits de sécurtié.
Si on fait extraction puis expansion la sécurité dépend de la non inversibilité. Le paradoxe des anniversaires n'intervient pas. Dans tous les cas l'expansion va assurer que la sortie a de tres bonnes propriétés

KDF pour les mots de passe

La base, c'est d'utiliser un sel. On va stocker dans le pc le hashé du concatènement du mdp et du sel. Ca évite qu'un attaquant utilise une table de mot préhachée (rainbow table). Du coup quand quelqu'un va vouloir faire une attaque spécifique sur un mdp, il va devoir tout tester !
PBKDF : fonction de hachage lente pour ralentir l'attaquant.

NB: SHA1:160bits

L'utilisation hiérarchique

On l'a vu avec les eNode. Il y a un chef qui a une clé, qu'il dérive et qu'il transmet aux subalternes. Sur un meme niveau, les équivalent ne connaissent pas les mots de passe.