Université de Lille 1 SCIENCES ET TECHNOLOGIES

Faculté des Sciences et Technologies

Notes de cours

Principes et Algorithmes Cryptographiques

M1S2

Table des matières

Première partie

Présentation

Chapitre 1

Mécanismes de chiffrement à clef secrète

La cryptographie est, étymologiquement, la « science du secret ». L'un de ses principaux objectifs, mais non le seul, est d'offrir la possibilité de communiquer des informations confidentielles sur des canaux de communication non-fiables.

À ce sujet, on peut remarquer qu'en 2019, la plupart des canaux de communications qui sont à notre disposition sont (assez) faciles à « espionner ». Nos ordinateurs et nos téléphones communiquent par radio : n'importe qui peut capter les ondes en étant suffisamment proche. Nous sommes connectés à internet par des fils de cuivre ou des fibres optiques qui sont sous le contrôle d'une myriade d'opérateurs et d'entreprises publiques ou privées qui ont la possibilité technique d'inspecter ce qui passe sur leur réseau. Nos boites mails sont hébergés chez des fournisseur d'accès ou de service qui ont tout le loisir d'examiner leurs contenus (c'est d'ailleurs parfois une source de revenu pour eux). Enfin, il est bien connu que les polices politiques de certains pays ouvraient les lettres « physiques » des gens qu'elles plaçaient sous surveillance [1]. Notez aussi que dans un certain nombre de cas, il est difficile d'être certain de l'identité de notre correspondant (dans le protocole UDP par exemple, l'expéditeur d'un paquet peut prétendre être qui il veut, c'est invérifiable).

Bref, à peu de choses près, le seul canal de communication « vraiment sûr » que des individus normaux ont a leur disposition consiste à se donner rendez-vous au milieu d'une forêt et à parler à voix basse.

L'un des principaux objectifs de la cryptographie consiste donc à garantir la *confidentialité* d'informations transmises sur des canaux non-fiables. Généralement, ceci s'accompagne aussi de garanties sur *l'authenticité* des données transmises et éventuellement aussi sur *l'identité* de l'exéditeur.

1.1 Chiffrement authentifié à clef secrète

Sans plus attendre, on va considérer qu'il existe des *mécanismes de chiffrement authentifié à clef secrète*. Il s'agit d'une paire d'algorithmes *déterministes* $\mathcal{E}$ et $\mathcal{D}$, qu'on va considérer comme des boites noires... pour l'instant. L'algorithme de chiffrement ("*encryption*") $\mathcal{E}$, prend en entrée :

- Un message [2] *clair* ("*plaintext*"), qui est une chaine de bits de taille arbitraire.
- Une *clef secrète*, qui est une chaine de bits de taille fixe (de 128 à 256 bits en 2019).
- Une chaine de bits *aléatoires* dont la taille (éventuellement nulle) dépend de l'algorithme.

Un mécanisme de chiffrement peut être un algorithme *randomisé*, c'est-à-dire capable de faire des choix aléatoires. C'est pour cette raison qu'on lui fournit explicitement des bits aléatoires : pour implémenter la fonction `random.getrandbits(k)`, il suffit de lire les k prochains bits de la séquence de bits aléatoires fournie. Comme la source de l'aléa est externe, il n'est pas de la responsabilité du mécanisme de chiffrement de se « produire » de l'aléa tout seul. En effet, la production de ces bits aléatoires est un sujet délicat qu'on abordera plus tard.

1. On voit ceci dans le très bon film « La vie des autres » (2007), avec la Stasi en ex-RDA
2. en cryptographie, un « message » désigne en principe des données quelconques, non spécifiées, mais sur lesquelles on a besoin d'appliquer les mécanismes cryptographiques

Sans rentrer dans les détails, le mécanisme de chiffrement « mélange » la clef et le message clair de manière inextricable. Cela produit un message *chiffré* ("*ciphertext*"). Il faut encore ajouter que le chiffré est généralement un tout petit peu plus gros que le clair et que de l'aléa est souvent injecté dans le chiffrement, ce qui le rend non-déterministe (si on chiffre plusieurs fois le même clair avec la même clef, on obtiendra des chiffrés différents).

Par exemple, si on chiffre :

```
Au début des années 1980, en Allemagne de l'Est, l'auteur à succès Georg Dreyman
et sa compagne, l'actrice Christa-Maria Sieland, sont considérés comme faisant
partie de l'élite des intellectuels de l'Etat communiste, même si, secrètement,
ils n'adhèrent pas aux idées du parti. Le Ministère de la Culture commence à
s'intéresser à Christa et dépêche un agent secret, nommé Wiesler, ayant pour
mission de l'observer. Tandis qu'il progresse dans l'enquête, le couple
d'intellectuels le fascine de plus en plus...
```

avec la clef `0x033e6101345b10da1a7f55df373dd5e832d1d673`, on pourrait obtenir le résultat suivant (encodé en Base64 pour la lisibilité) :

```
U2FsdGVkX19X+//oKPCtE3fttVFJMOFm9ywDMEhohxWY7bwvMXJIB/5HeYyscJcb
VJChqND84jp008wz8jmqSpmoEvsPytZtalzX9ImifO3jXmv5Jv4HuAPv0wWVZYB0
Bx2YvtVj4stNb8QM18wueADx9RkxHq+mJ5JSzKdPQGkIF9H44c6YifLw20UDgU5G
WALsXEn+bElmj7q2KKfraYeHC8viZgVFSOafG5kYspXVuqX6DUthG0QtYvqo0/xw
lyB+PJoMFdKV3txLKMU/TnS8KmzmVvJYJWJ+9SOG6KWQjdTnh+eRgAiwdh+VsuyJ
WUXWwgQOWeD2U/UKZlSgEGPU5DXi1IORscKpIKX2NHJk4jwRphe2MgOxbOxS3Qmj
Wnfam+A/Jd8nL1TUFsoi/9eN8CkS+hQxTJqpIVNynBjVD1wwEHzgwTFdrqQyWEip
3rIYFmH74lTY2Bdvvl0UlJmTxLaq3BN7Dnvum8uec2B3/ChRk2Lj0lbq4+KH19XR
QeBSO10pcZhTZFMucMqH+a/0aJLEkS8O5drphkG56upvSt7fdjm/R5NWVnJhBqXL
Dnxf6P4s0syNE+CrZkAhwiScgEEzPhoqTqPhF+uiiFmFg2QWLyZX/ezq+AOHCUC8
HV1NrcTVERJEcrPasts1SvsFJm4mIXC0Fd4znjgzD8t4x45V4VppL8+CItEQQdkv
39L5X+IEZYf/jUHLDpFjUPJuftJr0x0WI0+VsVk0dYs=
```

La procédure de déchiffrement $\mathcal{D}$ prend en argument le message chiffré et la clef utilisée lors du chiffrement, et renvoie :

— ou bien le message clair qui a été chiffré,
— ou bien $\perp$, ce qui signale qu'une erreur a eu lieu.

La même clef est nécessaire lors du chiffrement et du déchiffrement. Par conséquent, la confidentialité du message chiffré n'est garantie que tant que cette clef reste « secrète ». On parle aussi parfois de chiffrement *symétrique*. C'est un peu paradoxal : pour pouvoir établir un canal de communication sécurisé grâce au chiffrement, il faut *déjà* avoir eu, *dans le passé*, un canal sécurisé sur lequel la clef secrète a pu être transmise entre les participants.

La description du mécanisme de chiffrement, elle, n'est pas secrète. Il existe des implémentations libre, *open-source* de la totalité des mécanismes de chiffrement largement utilisés. D'ailleurs, les algorithmes en questions sont généralement standardisés par des institutions internationales, pour garantir l'interopérabilité du logiciel et du matériel. Ceci est en fait indispensable pour qu'un mécanisme de chiffrement puisse être utilisé à grande échelle dans le monde entier : son fonctionnement ne peut pas être secret. Seul le fait que les clefs soient secrètes garantit quoi que ce soit.

Au passage, la même situation prévaut pour les serrures qui équipent la plupart des portes. Tout le monde sait, ou peut savoir en consultant une encyclopédie, comment fonctionne une serrure. Mais ça ne permet pas directement d'ouvrir la porte de son voisin.

1.1.1 Spécification informelle

Si le mécanisme de chiffrement authentifié est de bonne qualité, alors :

1. Il est impossible de reconstituer le message clair à partir du chiffré si on ne connait pas la clef.
2. Il est impossible de produire une chaine de bits qui soit déchiffrée sans erreur avec une clef K si on ne connaît pas K.

Le premier point affirme que le mécanisme de chiffrement peut servir à garantir la confidentialité de messages qui doivent transiter par un canal non-fiable. Le deuxième point affirme qu'il garantit

l'authenticité du message et l'identité de l'expéditeur : si le déchiffrement est correct, c'est que le message a été produit par le bon expéditeur (qui connaît la clef secrète), et qu'il n'a pas été modifié entre temps.

Il existe aussi des systèmes de chiffrement non-authentifié, qui ne possèdent pas la seconde garantie, tout comme des mécanismes d'authentifications qui offrent la seconde garantie mais pas la première. Contentons-nous d'affirmer pour l'instant que de tels mécanismes de chiffrement authentifié existent.

L'exercice 1.1 démontre qu'il est nécessaire que le chiffré soit plus gros que le clair si on souhaite obtenir la propriété d'authentification. Dans OpenSSL, par défaut, le chiffrement n'est pas authentifié.

1.1.2 Adversaires

Si on se pose le problème d'utiliser des solutions cryptographiques, c'est bien qu'on veut garantir des propriétés de certaines communications (confidentialité, intégrité, etc.) face à un environnement supposé hostile. Pour raisonner, on suppose donc qu'il y a des *adversaires* qui essayent de contourner ou de briser les propriétés offertes par nos mécanismes cryptographiques. Une procédure mise en oeuvre à cet effet par un adversaire est une *attaque*.

Dans le fond, c'est l'hypothèse que de tels adversaires existent, ou peuvent exister, qui motive l'emploi de la cryptographie. On prend généralement une approche pessimiste, en supposant que les adversaires sont puissants : mieux vaut les surestimer que les sous-estimer.

On considère généralement qu'ils peuvent espionner à leur guise tous nos canaux de communication. Si c'est tout ce qui est en leur pouvoir, on dit qu'ils sont *passifs*. Mais si, en plus, ils peuvent modifier ce qui circule sur les canaux (retrait, modification ou injection de données), alors ils sont *actifs*. Sans surprise, les adversaires actifs sont plus problématiques.

Face à un mécanisme de chiffrement authentifié à clef secrète, les adversaires peuvent avoir plusieurs objectifs :

— Apprendre les clefs secrètes (ce qui leur permet de tout faire).
— Déchiffrer un message intercepté chiffré.
— Forger un chiffré valide (qui se déchiffre sans erreur).

En pratique, le premier objectif est le seul qui vaille vraiment le coup, et c'est bien souvent le seul moyen raisonnable de réaliser les deux autres.

1.1.3 Recherche exhaustive

Un adversaire qui intercepte un message chiffré a toujours à sa disposition une idée relativement naïve : l'attaque par « force brute », aussi appelée la *recherche exhaustive*. Il s'agit d'essayer de déchiffrer le message intercepté avec toutes les clefs possibles, jusqu'à ce que quelque chose de raisonnable soit obtenu.

```
1: procedure ExhaustiveSearch(C)
2:     Input : a ciphertext C
3:     Output : valid plaintext P and secret key K such that D(K, C) = P
4:
5:     for all possible key K do
6:         P ← D(K, C)
7:         if P ≠ ⊥ then
8:             Return (P, K)
9:         end if
10:    end for
11: end procedure
```

Cette procédure finit forcément par s'arrêter en renvoyant la bonne clef dans la plupart des cas. De toute façon, on peut toujours vérifier si le clair renvoyé a un sens ou pas, et continuer la procédure si ce n'est pas le cas.

Les pré-requis de cette attaque sont minimaux, et les adversaires peuvent toujours la tenter. La seule question qui reste est : « *combien de temps la recherche exhaustive nécessite-t-elle* » ? Si la

clef secrète occupe n bits, alors la boucle **for** de la procédure ci-dessus va faire au maximum 2^n itérations. On peut donc estimer, de manière assez conservatrice, que l'ensemble de la procédure nécessite environ 2^n opérations. La valeur de n est choisie pour qu'un tel calcul soit hors de portée de l'humanité.

Les exercices 1.2– 1.11 discutent de la taille des clefs et de la faisabilité de la recherche exhaustive.

1.1.4 Un peu d'histoire

La cryptographie a longtemps été l'apanage des services secrets et des militaires. Cependant, le développement de l'informatique à la fin des années 1960 et son adoption par certains secteurs de l'économie (banques, trafic aérien, comptabilité, etc.), le développement des réseaux sur lesquels des entreprises ou des services étatiques faisaient transiter de plus en plus de données éventuellement sensibles, tout cela rendait nécessaire l'emploi de mécanismes cryptographiques.

Alors que les connaissances cryptographiques étaient peu disponibles, et pour éviter que les entreprises américaines n'utilisent des algorithmes « maison » peu sûrs, la NSA a publié en 1977 la spécification du DES, le « *Data Encryption Standard* ». Il s'agissait à l'époque d'un algorithme moderne, sûr et novateur. Cependant, la NSA avait refusé d'expliquer quoi que ce soit à son sujet : ni les arguments qui justifiaient sa sécurité, ni la logique de sa conception. Elle en avait fourni le code source, et c'est tout, ce qui posait des problèmes de crédibilité, surtout à l'étranger, où la NSA était accusée de distribuer un algorithme qu'elle pouvait, elle, casser.

Les clefs secrète du DES occupent 56 bits. À l'époque, la recherche exhaustive sur des clefs de 56 bits n'était pas possible, en tout cas pas sans des moyens considérables. Cependant, la puissance de calcul de l'humanité double grosso-modo tous les 18 mois. En 1997, plusieurs recherches exhaustives ont pu être menées à bien par des projets du style « CrackDES@Home ». La première machine parallèle dédiée à la recherche exhaustive du DES a été construite en 1998. Elle fonctionnait en 48h. En 2019, on peut acheter, pour le prix d'une petite voiture, une machine formée de FPGAs qui effectue la recherche exhaustive en une dizaine d'heures.

Aujourd'hui, la puissance de calcul des plus gros supercalculateurs (connus du public...) se mesure en peta-flops, et bientôt en exa-flops, c'est-à-dire en multiples de 2^{50} ou 2^{60} opérations par seconde.

Pour résumer, l'évolution technologique a rendu caduc le DES, ne serait-ce que parce que ses clefs devenaient trop petites. Pour faire face à ce problème, le gouvernement américain s'est doté d'un nouveau standard en 1999, l'AES (« *Advanced Encryption Standard* »). Pour en maximiser l'adoption, le gouvernement avait organisé une compétition internationale entre les équipes de recherche en cryptographie [3] du monde entier. Chaque « équipe » devait justifier la conception de son algorithme... et essayer de casser ceux des autres. Celui qui est devenu l'AES a été conçu par deux chercheurs belges, V. Rijmen et J. Daemen.

L'AES a des clefs de 128, 192 ou 256 bits, ce qui est bien assez pour éviter la recherche exhaustive dans l'avenir prévisible. L'AES a été beaucoup étudié depuis 1999, sans que des problèmes de sécurité majeurs ne puissent être mis en évidence.

1.1.5 Génération des clefs

Toute la sécurité du chiffrement repose sur le fait que les clefs secrètes sont... vraiment secrètes, c'est-à-dire inaccessible à des tiers. En particulier, il ne faut pas les choisir d'une manière prévisible par les adversaires.

Le plus sûr, et le plus simple, consiste à les générer *uniformément au hasard* — ça les rend imprédictibles par définition. En pratique, générer des bits aléatoires imprévisibles est un problème délicat sur lequel on se penchera plus tard. Une séquence de 128 bits aléatoires est un *secret fort*, dans le sens qu'il n'est pas possible de faire une recherche exhaustive dessus et qu'aucune possibilité n'est plus fréquente que les autres.

Dans la pratique, certaines clefs sont générées aléatoirement, mais d'autres doivent être mémorisées par des utilisateurs. Il n'est pas commode de mémoriser un nombre de 128 bits du type :

`0x5756ec0de7936948a7865d63d1a82d5c`.

3. Alors qu'en 1977 il n'en existait presque pas, en 1999 il y en avait de nombreuses

Quand les utilisateurs doivent mémoriser des secrets, ils mémorisent généralement des *mots de passe*, c'est-à-dire des *secrets faibles* (certaines combinaisons sont plus fréquentes que d'autres, et il est envisageable de tester toutes les possibilités les plus probables). Il est possible de faire de la cryptographie fiable avec des secrets faibles, mais c'est plus délicat.

1.2 Protocoles basés sur le chiffrement à clef secrète

Dans la littérature sur les protocoles, on écrit souvent :

$$A \to B : \{M\}_K$$

pour dire : « A(lice) envoie à B(ob) le chiffrement du message M par la clef K », c'est-à-dire $\mathcal{E}(K, M, r)$ où r est une chaine de bits aléatoires de la bonne taille, générée exprès pour l'occasion et jetée à la poubelle ensuite. Le contexte indique en principe de quel mécanisme de chiffrement il s'agit, et ce que Bob doit faire avec.

1.2.1 Identification

Un mécanisme de chiffrement authentifié permet de vérifier l'identité d'un correspondant, via un mécanisme de « *challenge-response* ». Dans la vraie vie, comment Alice peut-elle s'assurer qu'elle est bien en train de discuter en ligne avec Bob, et pas avec un imposteur ? Elle peut demander à Bob de révéler une information qui lui seul est censé connaître (votre banque se contente de vos dates et lieux de naissance).

Il y a là une idée très générale : ce qui garantit l'identité des acteurs, c'est la *connaissance* d'un *secret*. C'est le principe de la connexion par login/password qui est très répandue : le login est public, le password est secret. La connaissance du password démontre votre identité. Le protocole HTTP prévoit un contrôle d'accès de cette nature, le « Basic Access Control » (cf. RFC 2617). Dans les années 1990, une bonne partie des connections internet s'établissaient avec le protocole PPP (« Point-to-Point Protocol »), qui prévoyait un mécanisme d'authentification où les utilisateurs envoient simplement leur mot de passe en clair : c'est le protocole PAP (« PPP Authentication Protocol »), cf. RFC 1334.

Cependant, on ne peut pas se contenter d'envoyer son mot de passe « en clair » sur le réseau pour démontrer son identité : si on le faisait en wifi depuis un lieu public, on permettrait à n'importe qui se trouvant à proximité de se faire passer pour nous. La cryptographie permet de démontrer qu'on connait un secret *sans le révéler*.

Dans le protocole suivant, Alice et Bob on un secret commun, et ce secret est une clef cryptographique K. Bob démontre à Alice qu'il connaît bien K, un adversaire ne peut pas apprendre K.

Protocole I (*Identification challenge-response*). Alice génère un nombre aléatoire N, le chiffre avec leur clef commune K et transmet le chiffré à Bob. Bob déchiffre ce qu'il a reçu d'Alice avec la clef K et lui envoie le résultat (en principe N). Alice vérifie enfin qu'elle a bien reçu son nombre aléatoire N : si c'est le cas, Bob a donc réussi à effectuer le déchiffrement, et a priori il connaît la clef secrète K.

I1. [Challenge.] $A \to B : \{N\}_K$ (N est un grand nombre aléatoire frais)

I2. [Response.] $B \to A : N$

On dit qu'une donnée est *fraiche* si elle générée spécialement pour l'occasion, et qu'elle n'a jamais servi avant. Un *nonce* est un *nombre (arbitraire) à usage unique*[4] La plupart du temps, les nonces sont aléatoires.

Le secret K ne transite jamais sur le canal. Un adversaire passif (qui ne ferait qu'espionner le canal de communication) apprend des chaines de bits aléatoires (N) —qui ne lui apportent pas d'information sur K—, ainsi que leurs chiffrés. Il apprend donc des *paires clair-chiffré*, qu'il ne choisit pas. Il ne faut pas qu'il puisse en déduire K (ce serait une attaque à *clairs connus*).

4. le barbarisme « nonce » vient certainement de Number et de ONCE.

Par contre, un adversaire actif pourrait se faire passer pour Alice auprès de Bob, et envoyer un message C *de son choix* à Bob ; Bob se fera un plaisir de le déchiffrer et de renvoyer le clair correspondant au faussaire. L'adversaire peut donc faire déchiffrer des messages de son choix par Bob, et il ne faut pas qu'il puisse en déduire K (ce serait une attaque à *chiffrés choisis*). Les mécanismes de chiffrement habituels résistent tous à ces scénario d'attaques.

Les exercices 1.12–1.15 discutent des détails de ces protocoles et de solutions qui ne marchent pas.

Variante : le stockage « historique » des mots de passe dans UNIX .

Dans la plupart des systèmes d'informations, des comptes utilisateurs sont protégés par mot de passe. Le système doit donc disposer d'une procédure pour vérifier si un mot de passe entré par un utilisateur est correct ou pas. Stocker les mots de passe *en clair* est une mauvaise idée : n'importe quel problème de gestion des droits d'accès au fichier stockant les mots de passe serait un gros problème (sans même parler des virus ou autres *malware*). Par exemple, on pourrait essayer de faire démarrer l'ordinateur d'un collègue avec une clef USB *bootable*, lire le contenu du disque dur, et voler les mots de passe.

Dans les systèmes UNIX et dérivés, les mots de passe des utilisateurs étaient traditionnellement stockés dans le fichier `/etc/passwd`, dont on ne pouvait pas garantir la confidentialité. Par conséquent, au lieu de stocker : `"user: password"`, on stockait : `"user: salt,` $\mathcal{E}$`(password‖salt, 0)"`, où `salt` est aléatoire, choisi une bonne fois pour toute lors de la création du mot de passe. Ceci se faisait avec le DES, donc le `password` devait faire 56 bits... ce qui correspond à 8 charactères ASCII « classiques » (dont le code est compris entre 0 et 127, donc sur 7 bits).

La vérification du mot de passe consiste à dechiffrer ce qui est stocké avec le mot de passe saisi par l'utilisateur, et vérifier si on récupère bien 0. On verra plus tard que, de nos jours, on utilise plutôt des fonctions de hachage pour cette tâche (cf. § 3.1.2).

1.2.2 Techniques d'Encapsulation de clef

L'*encapsulation de clef* est la technique qui consiste à chiffrer des données avec une clef... puis à chiffrer cette clef avec une autre clef.

Système anti-force-brute Imaginons un système informatique avec une identification par mot de passe, tel qu'un code d'accès sur un smartphone. On souhaite qu'au bout de 10 tentative d'accès infructueuses, toutes les données contenues dans la mémoire du système soient détruites (ceci avait posé problème au FBI américain dans une l'enquête sur la tuerie de San Bernardino en 2015).

On pourrait envisager de recouvrir les données à protéger de charabia après la saisie du 10-ème mot de passe erroné. Le problème c'est que cette opération va prendre un temps non-négligeable et qu'un éventuel adversaire pourrait couper le courant pour l'empêcher ou la limiter.

Une solution plus intéressante est la suivante : une clef cryptographique (forte) K est générée aléatoirement lors de la fabrication du périphérique. Toutes les données stockées sont chiffrées avec cette clef K. La clef K elle-même est stockée sur le périphérique. Autrement dit, le périphérique stocke :

$$\left(K, \{data\}_K\right) \qquad (K \text{ est une clef aléatoire spécifique au périphérique})$$

Lors de la mise en route du périphérique, si le bon mot de passe est fourni, K est chargée en mémoire et les données sont (dé)chiffrées à la volée. Par contre, si 10 tentatives infructueuses ont lieu, alors K est effacée. Cette opération, elle, est très rapide, puisqu'il n'y a que 128 bits à effacer. Une fois que K a disparu, il reste les données chiffrées... qu'on ne peut plus déchiffrer puisqu'on a perdu la clef.

Le chiffrement « révocable » des DVD Aussi surprenant que cela puisse paraître, les films contenus sur les DVDs (ainsi que leurs successeurs, les disques Blu-Ray) sont entièrement chiffrés. L'idée de départ semble avoir été d'empêcher les utilisateurs de faire quoi que ce soit avec leurs films, si ce n'est les regarder sur un lecteur "agréé". En particulier, il devait être impossible de copier les films, de les re-graver, de les stocker sur le disque dur, de les transférer sur internet, etc.

Nous savons aujourd'hui que cet objectif n'a pas été atteint. Voyons cependant comment les choses étaient censées se passer. Si le contenu des disques était chiffré, il fallait cependant que les lecteurs "normaux" puissent en extraire le contenu. Le système le plus simple aurait été que l'organisme de standardisation choisisse *une* clef secrète qui aurait servi à chiffrer le contenus de tous les disques, et qui serait connue par tous les lecteurs, permettant ainsi le déchiffrement. Cela aurait posé un problème de taille : il aurait suffit qu'une seule fois, un seul employé des différentes entreprises de fabrications de disques, de lecteurs, etc. rende publique la clef secrète du système (par exemple sur un forum internet anonyme) pour que la protection cryptographique ne serve plus à rien.

Un système plus sophistiqué a donc été mis en place, pour contourner ce problème. L'organisme de standardisation qui gère l'ensemble du système a généré (au hasard) une liste d'environ 400 clefs de chiffrement, les *clefs de lecteur* et en a attribué une à chaque entreprise produisant des platines de lecture. Les lecteurs doivent contenir la clef de lecteur attribuée à leur fabriquant dans une petite mémoire difficilement accessible, et s'en servir pour déchiffrer le contenu des disques.

L'idée était que si un fabriquant laissait fuir une clef de lecteur dans la nature, alors il serait possible de la *révoquer*, c'est-à-dire de fabriquer de nouveaux disques qui ne pourraient plus être lus par les lecteurs de la marque incriminée.

A cet effet, le contenu des disques est chiffré avec une clef secrète propre à chaque disque, la *clef de disque*, qui est choisie au hasard au moment de la mise sous presse. Cependant, comme il faut bien qu'au final on puisse voir le film, il faut que les lecteurs agréés puisse déchiffrer le contenu, en n'étant en possession que de leur propre clef de lecteur. Chaque disque contient donc, dans une zone spéciale (qui n'existe d'ailleurs pas sur les disques vierges), sa clef de disque, chiffrée par chacune des quelques 400 clefs de lecteurs. On peut en effet chiffer une clef de chiffrement, en s'y prenant exactement de la même manière que pour n'importe quelle autre séquence de données.

De la sorte, quand on veut regarder un film, le lecteur utilise sa clef de lecteur pour déchiffrer la clef du disque, puis utilise cette dernière pour déchiffrer le disque lui-même.

Ce système à deux étages permet à l'organisme qui gère le système de révoquer une clefs de lecteur donnée. Pour cela, il suffit de décider que les disques qui sortiront à l'avenir ne *contiendront plus* la clef de disque chiffrée par la clef de lecteur compromise. Les lecteurs de la marque dont la clef a été révoquée ne pourront donc plus déchiffrer ces nouveaux disques (tandis que ceux des autres marques le pourront comme avant).

Alors, qu'est-ce qui a raté dans cette belle construction ? Pourquoi a-t-on pu copier des DVD quasiment dès leur apparition ? Le problème le plus fondamental c'est qu'à l'époque de la création du standard, le gouvernement américain interdisait l'exportation de produits cryptographiques "forts", et limitait la taille des clefs. Aussi, toutes les clefs dont il a été question au-dessus ont une taille de... 40 bits, une taille qui rend possible la recherche exhaustive.

Pour effectuer cette recherche exhaustive, cependant, il faut connaître le fonctionnement du mécanisme de chiffrement. Or celui des DVDs était gardé secret. Il est bien présent dans les circuits électroniques des lecteurs de salon, mais leur fonctionnement est difficile à analyser par des pirates. Par contre, le fonctionnement des *programmes informatiques* de lecture de DVD est bien plus simple à analyser, et en 1999 les détails du procédé de chiffrement étaient publiés sur internet, dans la fameuse libdvdcss. C'en était fini de la protection des DVDs, d'autant plus que le chiffrement était très faible, et qu'en exploitant ses défauts quelques secondes suffisent à retrouver la clef d'un disque.

Les entreprises qui voulaient protéger leurs revenus contre la piraterie numérique ont eu une chance de mettre en place une cryptographie de meilleure qualité avec la sortie des disques Blu-Ray. Elles ont accouché d'un système comparable (un peu plus sophistiqué en réalité), mais cette fois avec des clefs d'une taille très respectable de 128 bits. Et ce qui devait arriver arriva : la première clef de lecteur, extraite d'un programme de lecture s'est rapidement trouvée sur internet. Elle a été révoquée, puis une seconde clef a fui, qui a été révoquée, puis une troisième clef a fui, etc.

L'exercice 1.16 présente une autre application utile de la technique d'encapsulation de clef.

1.2.3 Protocoles GSM – la téléphonie mobile « 2G »

Les communications des téléphones portables sont chiffrées et (partiellement) authentifiées. Chaque carte SIM contient un identifiant unique, l'IMSI (« International mobile subscriber identity »), sur

64 bits. L'IMSI indique au réseau l'identité de l'abonné (à qui il faut facturer, a-t-il le droit de passer des appels, etc.). Un des objectifs du protocole GSM était que l'IMSI soit envoyé le plus rarement possible, afin d'éviter la surveillance des utilisateurs. Ceci dit, il l'est forcément de temps en temps pour identifier le mobile vis-à-vis du réseau. La carte SIM contient aussi une clef de 128 bits K, qui lui est spécifique.

Outre les mobiles, le réseau (d'un opérateur) est constitué d'antennes-relais (les VLR, « Visited Location Registers ») et d'un serveur central (HLR, « Home Location Register »). Le serveur central connaît les numéros IMSI et les clefs K de tous les abonnés. Les antennes-relais, elles, ne les connaissent pas.

Deux protocoles sont utilisés, le premier lorsqu'un téléphone entre en contact avec une antenne-relais. Le réseau authentifie le téléphone, des clefs de sessions sont établies, et une identité temporaire (le TMSI) est (secrètement) affectée au téléphone. Ensuite, le téléphone se présente à l'antenne-relai par son TMSI, ce qui le rend anonyme.

Protocole Ga (*Prise de contact avec une antenne-relais*). Un téléphone mobile M entre en contact avec une antenne Relais R. Le mobile envoie son IMSI, qui est relayé au serveur central. Le serveur central, avec la clef K_i, génère des triplets (r_i, s_i, k_i), où r_i est aléatoire, et $(s_i, k_i) \leftarrow \mathcal{E}(K, r_i)$. L'antenne-relais se sert de ces données pour authentifier le téléphone : r_i sert de défi et s_i de réponse ; le téléphone qui connaît K peut calculer s_i et k_i à partir de r_i. Une fois le téléphone authentifié, les communications sont chiffrées avec k_i.

Ga1. [Identité.] $M \to R$: IMSI

Ga2. [Relai.] $R \to S$: IMSI

Ga3. [Identifiants.] $S \to R : (r_1, s_1, k_1), \ldots, (r_n, s_n, k_n)$,

Ga4. [Défi.] $R \to M : r_1$

Ga5. [Réponse.] $M \to S : s_1$

Ga6. [Identité temporaire.] $S \to M : \{\text{TMSI}\}_{k_1}$

Un deuxième protocole est effectué périodiquement par le téléphone chaque fois qu'il veut parler à une antenne-relais auprès de laquelle il s'est déjà authentifié. Ce second protocole est exécuté bien plus souvent que le premier.

Protocole Gb (*Communication après authentification*). Le mobile s'identifie (de manière anonyme) par son TMSI. L'identité du téléphone est vérifiée par le réseau. À l'issue du protocole, les communications sont chiffrées par la clef de session k_i. Chaque exécution de ce protocole « consomme » un triplet (r_i, s_i, k_i). Si l'antenne-relais tombe à court de triplets, il faut re-exécuter le protocole Ga.

Gb1. [Identité temporaire.] $M \to R$: TMSI

Gb2. [Défi.] $R \to M : r_i$

Gb3. [Réponse.] $M \to S : s_i$

Dans le protocole Ga, l'identité temporaire du téléphone (TMSI) est transmise de façon chiffrée. De la sorte, un adversaire qui espionne une session du protole Gb ne connaît pas l'IMSI du téléphone, donc ne sait pas qui est en train de téléphoner. Ceci dit, il existe des dispositifs utilisés par la police (les « IMSI-catchers ») qui se présentent comme des antennes-relais fictives et qui, du coup, apprennent les IMSI des téléphones à proximité en exécutant le protocole Ga. Il semble que les opérateurs stockent des logs d'exécution du protocole Ga, car la police affirme parfois que le téléphone mobile de tel ou tel suspect a « borné » à un endroit précis.

1.2.4 Protocole d'identification unique (« Single Sign-On »)

Imaginons un réseau où un grand nombre d'utilisateurs veulent communiquer de manière chiffrée. Pour garantir la confidentialité, il faudrait que chaque utilisateur partage une clef secrète différente avec chaque autre... et la gestion des clefs va vite devenir un problème

Les protocoles d'authentification unique ("*Single Sign-On*") permettent de résoudre en partie ce problème en faisant intervenir un *serveur d'authentification*. Dans ce contexte, chaque utilisateur partage un secret avec le serveur d'authentification. Celui-ci permet à une paire utilisateurs d'établir entre eux une *clef de session* fraiche. Une « clef de session » est une clef secrète « jetable », qui n'a vocation qu'à être utilisée pendant la prochaine session de communication, puis jetée après.

Quand Alice veut établir une liaison chiffrée avec Bob, elle contacte d'abord le serveur d'authentification. Ce dernier génère une clef de session fraiche K_{ab} pour Alice et Bob. Il reste à la communiquer de manière sûre aux deux utilisateurs concernés. Une solution potentielle serait :

Protocole P (*le serveur d'identification contacte les deux protagonistes*). Alice contacte le serveur, l'informe qu'elle veut établir une session avec Bob. Le serveur génère une clef de session fraiche, puis la renvoie chiffrée à Alice. Il contacte Bob, l'informe qu'Alice veut communiquer avec lui, et lui transmet la clef de session chiffrée. Alice entre finalement en contact avec Bob, et démontre la possession de la clef partagée.

P1. [Début.] $A \to S : A, B, N$ (N est un grand nombre aléatoire frais).

P2. [A reçoit K_{ab}.] $S \to A : \{N, A, B, K_{ab}\}_{K_a}$ (K_{ab} est une clef de session fraiche).

P3. [B reçoit K_{ab}.] $S \to B : \{N, A, B, K_{ab}\}_{K_b}$

P4. [A contacte B.] $A \to B : \{N, A, B\}_{K_{ab}}$

Dans ce protocole, la clef de session ne circule jamais en clair sur le réseau. L'inconvénient de cette solution c'est qu'elle nécessite que le serveur d'authentification initie des connexions vers les utilisateurs sans que ceux-ci n'aient rien demandé (lors de l'étape P3). En fait ce n'est pas indispensable. On peut modifier le protocole de la façon suivante, pour obtenir :

Protocole NS (*Needham-Schroeder, 1978*). Le serveur d'authentification ne contacte plus Bob directement, mais donne à Alice les moyens de le faire. Ensuite, Bob vérifie l'identité d'Alice. Comme précédemment, K_{ab} est une clef de session fraiche choisie par le serveur, et N_a, N_b sont des nonces aléatoires de taille convenable.

NS1. [Début.] $A \to S : A, B, N_a$

NS2. [Interaction avec S.] $S \to A : \left\{N_a, B, K_{ab}, \{A, K_{ab}\}_{K_b}\right\}_{K_a}$

NS3. [A Contacte B.] $A \to B : \{A, K_{ab}\}_{K_b}$

NS4. [B défie A.] $B \to A : \{N_b\}_{K_{ab}}$

NS5. [A répond.] $A \to B : \{N_b + 1\}_{K_{ab}}$

À l'étape NS3, Alice transmet la clef de session à Bob. Comme elle est chiffrée par la clef secrète de Bob, celui-ci sait que c'est le serveur qui l'a produite. Les deux dernières étapes servent à convaincre Bob qu'Alice possède elle aussi la clef de session.

Une variante légèrement plus compliquée de ce protocole a été/est largement déployé : il s'agit du protocole Kerberos.

Les exercices 1.17–1.21 montrent pourquoi chaque aspect du protocole de Needham-Schroeder est indispensable à la sécurité.

Exercices

Exercice 1.1 : Montrer que si chiffrement authentifié produit des chiffrés qui font k bits de plus que les clairs, alors un adversaire dont le temps d'exécution est modeste peut réussir à forger un chiffré valide avec probabilité supérieure ou égale à 2^{-k}.

▷ **Exercice 1.2 :** Le "Cost-Optimized Parallel COde Breaker" est une machine conçue par les universités de Kiel et de Bochum en Allemagne dans la perspective de casser des systèmes de chiffrement par recherche exhaustive. La machine coûte le même prix qu'une petite voiture. Une

COPACOBANA contient 16 cartes-support. Chacune de ces cartes contient 8 FPGAs (circuit reprogrammable). Chaque FPGA est capable d'executer 4 "coeurs" de chiffrement DES en parallèle. L'ensemble est cadencé à 140Mhz, et est capable d'effectuer un chiffrement DES par cycle d'horloge.

Combien de chiffrement DES cette machine est-elle capable d'effectuer par seconde ?

▷ **Exercice 1.3 :** En combien de temps une COPACOBANA peut-elle casser le DES (qui a des clefs de 56 bits) ?

▷ **Exercice 1.4 :** En combien de temps une COPACOBANA peut-elle casser SkipJack (qui a des clefs de 80 bits), en admettant qu'un chiffrement skipjack prend le même temps qu'un chiffrement DES ?

▷ **Exercice 1.5 :** Combien y a-t-il de mots de passe de 8 charactères, en autorisant minuscules, majuscules, chiffres et quelques signes de ponctuation ? Quelle est la taille de clef secrète correspondante ? (ceci est parfois nommé l'*entropie* du mot de passe).

▷ **Exercice 1.6 :** Un coeur d'un CPU contemporain, pas particulièrement puissant, est capable de faire 8 millions de chiffrements AES par seconde. Combien de coeurs faut-il pour casser un tel mot de passe en une semaine ?

▷ **Exercice 1.7 :** Le physicien Hans-Joachim Bremermann (1926–1996) a démontré, sur les bases de nos connaissances physiques actuelles, qu'un système matériel auto-suffisant (pas de source d'energie extérieure, pas d'échanges de matière avec l'exterieur) ne peut pas réaliser plus de $1.36 \cdot 10^{50}$ opérations par seconde et par kilo de matière (cette valeur est environ c^2/h, où c est la vitesse de la lumière et h est la constante de Planck).

Sachant que la planète terre pèse $5.972 \cdot 10^{24}$ kg, donner une borne inférieure sur le temps nécessaire pour casser une clef secrète de 512 bits par force brute.

Exercice 1.8 : Considérons l'adversaire suivant, qui prend en argument un message C, chiffré par une clef inconnue de n bits avec un chiffrement authentifié.

```
1: procedure SingleTrial(C, n)
2:     K ← Random bit string of size n
3:     P ← D(K, C)
4:     if P ≠ ⊥ then
5:         Return (P, K)
6:     else
7:         Return ⊥
8:     end if
9: end procedure
```

Sachant que K est une clef secrète de n bits générée aléatoirement, quelle est la probabilité que l'algorithme trouve la clef ?

Exercice 1.9 : À l'Euro-Millions, il faut choisir 5 bons nombres parmis 50, et cocher deux bonnes étoiles parmis 12. Il y a donc $\binom{50}{5}\binom{12}{2} = 139\,838\,160$ combinaisons possibles, et une seule rapporte le gros lot.

Qu'est-ce qui est le plus probable : « SingleTrial casse une clef de 128 bits » ou « Je joue 4 fois à l'euro-millions et je gagne les 4 fois ? »

▷ **Exercice 1.10 :** Considérons une version améliorée de l'algorithme précédent :

```
1: procedure ManyTrials(C, n, k)
2:     K ← Random bit string of size n
3:     P ← D(K, C)
4:     repeat
5:         if P ≠ ⊥ then
6:             Return (P, K)
7:         end if
8:     until k trials have failed
9:     Return ⊥
```

10: **end procedure**

Quelle est sa probabilité de succès, en fonction de n et k ?

▷ **Exercice 1.11 :** Qu'est-ce qui est le plus probable : « ManyTrial casse une clef de 128 bits avec un million d'essais » ou « Je joue 4 fois à l'euro-millions et je gagne les 4 fois » ?

Exercice 1.12 : Un adversaire actif essaye de se faire passer pour Alice auprès de Bob dans le protocole I. Peut-il obtenir quelque chose si le mécanisme de chiffrement est authentifié ?

Exercice 1.13 : Que risque-t-il de se passer si le nombre aléatoire N est trop petit dans les protocoles précédents ?

Exercice 1.14 : Voici un exemple de protocole d'identification qui ne marche pas :

Protocole J (*Identification challenge-response à un tour*). Le mécanisme de chiffrement utilisé est authentifié. Bob génère un nombre aléatoire N, le chiffre, puis l'envoie à Alice. Celle-ci tente le déchiffrement, et si elle ne récupère pas $\perp$, c'est que Bob a produit un chiffré valide, donc qu'il connaît K.

J1. [Response.] $B \to A : \{N\}_K$ (N est un grand nombre aléatoire frais)

Qu'est-ce qui ne va pas ?

Exercice 1.15 : On considère une variante du protocole I donné dans le texte :

Protocole I' (*Identification challenge-response*).

I'1. [Challenge.] $A \to B : N$ (N est un grand nombre aléatoire frais)

I'2. [Response.] $B \to A : \{N\}_K$

Est-il correct ? Quelle condition le mécanisme de chiffrement doit-il satisfaire ?

Exercice 1.16 : On souhaite mettre en place un système où le contenu entier d'un disque dur est chiffré avec un mot de passe. On souhaite que l'utilisateur puisse changer son mot de passe sans avoir à déchiffer-rechiffrer complètement son disque (ce qui prendrait des heures pour un gros disque moderne). Comment faire ?

Exercice 1.17 : Dans le protocole P, un adversaire essaye de se faire passer pour Alice auprès du serveur. Qu'est-ce qui l'empêche d'aboutir à établir une session avec Bob ?

Exercice 1.18 : Dans le protocole P, un adversaire essaye de se faire passer pour le serveur d'authentification auprès d'Alice. Qu'est-ce qui va mal se passer ?

Exercice 1.19 : On considère le protocole suivant :

Protocole $\mathbf{NSS_{bad}}$ (*Needham-Schroeder sans la confirmation*).

NS1. [Début.] $A \to S : A, B, N_a$

NS2. [Interaction avec S.] $S \to A : \left\{N_a, B, K_{ab}, \{A, K_{ab}\}_{K_b}\right\}_{K_a}$

NS3. [A Contacte B.] $A \to B : \{A, K_{ab}\}_{K_b}$

A et B entament ensuite leur conversation « normale ». Quel problème de sécurité se pose-t-il alors ?

Exercice 1.20 : On considère le protocole suivant :

Protocole $\mathbf{NS_{bad}}$ (*Needham-Schroeder sans les identifiants*).

NS1. [Début.] $A \to S : N_a, A, B$

NS2. [Interaction avec S.] $S \to A : \left\{N_a, K_{ab}, \{K_{ab}\}_{K_b}\right\}_{K_a}$

NS3. [A Contacte B.] $A \to B : \{K_{ab}\}_{K_b}$

NS4. [B défie A.] $B \to A : \{N_b\}_{K_{ab}}$

NS5. [A répond.] $A \to B : \{N_b + 1\}_{K_{ab}}$

Que peut-il se passer ?

Exercice 1.21 : On considère le protocole suivant :

Protocole $\mathbf{NS_{bad}}$ (*Needham-Schroeder sans le nonce N_a*).

NS1. [Début.] $A \to S : A, B$

NS2. [Interaction avec S.] $S \to A : \left\{B, K_{ab}, \{A, K_{ab}\}_{K_b}\right\}_{K_a}$

NS3. [A Contacte B.] $A \to B : \{A, K_{ab}\}_{K_b}$

NS4. [B défie A.] $B \to A : \{N_b\}_{K_{ab}}$

NS5. [A répond.] $A \to B : \{N_b + 1\}_{K_{ab}}$

Que peut-il se passer ?

Chapitre 2

Chiffrement à clef publique

2.1 Chiffrement à clef publique

On a vu dans le chapitre 1 qu'il existe depuis longtemps des mécanismes de chiffrement *à clef secrète*. Le paradoxe, c'est que pour pouvoir se transmettre des informations confidentielles sur un canal de communication non-fiable, il avait auparavant fallu pouvoir se transmettre une clef secrète de manière confidentielle.

Ceci rend largement impossible les communications chiffrées avec des interlocuteurs (personnes / machines) auquelles on n'a par exemple accès que par internet.

La révolution est venue, en 1976 et 1977 de l'invention de la cryptographie *à clef publique*. À chaque fois, il y a en fait *deux* clefs. Une qui doit rester secrète, et une qui peut/doit être rendue publique. L'idée de fond est remarquablement simple : la clef publique sert à chiffrer, la clef secrète à déchiffrer.

Ainsi, je peux envoyer ma clef publique par email à un dissident politique dans un pays repressif, et après cela lui peut m'envoyer des messages chiffrés que moi seul peut déchiffrer, car moi seul possède la clef secrète qui correspond à la clef publique. Du coup, *exit*, le problème de la distribution des clefs !

Plus précisément, un système de chiffrement à clef publique se compose :

- D'un algorithme de génération de clef (« *key-generation* »), qui prend en argument des bits aléatoires, un argument n désignant la taille de la clef à produire, et qui produit des paires de clefs publique-privée qui se correspondent.
- D'un algorithme de chiffrement, qui prend en argument un message clair, potentiellement des bits aléatoires, la clef publique et qui produit un message chiffré.
- D'un algorithme de déchiffrement, qui prend en argument un message chiffré, la clef publique, et qui recrache le message clair.
- (Dans certains cas) D'un algorithme de génération de paramètres (« *parameter generation* »). Les paramètres sont des données publiques qui servent à produire les clefs. Ils peuvent être utilisés par tout le monde, mais il faut les spécifier quand même.

Tous les schémas cryptographiques à clef publique reposent sur l'existence de problème mathématique difficiles. En effet, la clef publique contient suffisamment d'information pour qu'il soit possible (en théorie) d'en déduire la clef secrète. Seulement, on se débrouille pour que la complexité de ce calcul soit telle qu'il soit impossible en pratique avant de longues années.

Face à un schéma de chiffrement à clef publique, il faut toujours supposer que les adversaires possèdent la clef publique. Il faut donc toujours envisager la possibilité d'un attaque (en « *key-recovery* ») qui déduise la clef secrète de la clef publique. Le deuxième objectif des adversaires consiste à déchiffer un message, étant donné la clef publique seulement.

2.1.1 Schémas de chiffrement à clef publique les plus connus

RSA Le système RSA (des initialies de ses trois inventeurs), inventé en 1977, est non seulement l'un des tout premiers, mais c'est aujourd'hui le plus largement utilisé, notamment dans les cartes

de crédit. Il repose sur des idées mathématiques qui datent du XVIII$^{\text{ème}}$ siècle et avant.

L'idée de fond est remarquablement simple : la clef secrète est formée de grand deux nombres premiers [1] p et q (de 300 chiffres chacun, au moins, en 2019), et la clef publique est simplement leur produit $n = pq$. Calculer la décomposition est impossible. La magie, c'est que le produit suffit pour chiffrer, mais les deux facteurs séparés sont nécessaires pour déchiffrer.

On remarque tout de suite que n est un nombre de 600 chiffres au moins, soit ≈ 2000 bits. Ceci illustre la triste vérité : la cryptographie à clef publique est *toujours moins efficace* que la cryptographie à clef secrète. Les clefs sont plus grosses. Les algorithmes sont (bien) plus lents. Les chiffrés sont (souvent un peu) plus gros que les clairs, etc. Ceci explique pourquoi la cryptographie à clef secrète n'a pas disparu avec l'invention de la clef publique.

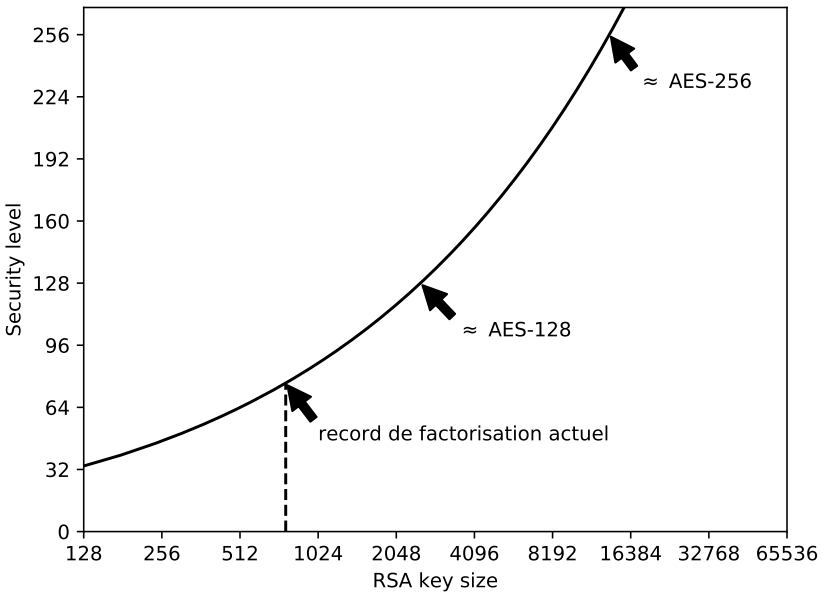
On a vu qu'en cryptographie symétrique, la taille des clefs était fixée principalement par le temps que mettrait la recherche exhaustive. Si un mécanisme peut être cassé plus vite que par la recherche exhaustive, il est considéré comme cassé. On fixe donc la taille des clefs symétriques pour qu'un adversaire qui entreprenne la recherche exhaustive avec un supercalculateur mette un temps T suffisamment grand pour que l'entreprise soit déraisonnable ($T = 10^9$ années est un bon début).

En cryptographie à clef publique c'est différent, car on peut généralement faire mieux que la recherche exhausive. Dans le cas de RSA, par exemple, on peut récupérer la clef secrète (p, q) à partir de la clef publique n si on parvient à *factoriser* n, c'est-à-dire à trouver un de ses diviseurs. Il existe de nombreux algorithmes pour cela. On choisit la taille de la clef publique de telle sorte qu'exécuter le plus rapide de ces algorithmes prennent un temps T aussi grand. Pour nous, cela signifie choisir un nombre n de plusieurs milliers de bits.

Le meilleur algorithme connu à ce jour pour factoriser de grands entiers de n bits, le crible algébrique (NFS) a une complexité de l'ordre de :

$$\exp\left(\left(\sqrt[3]{\ln 2 \frac{64}{9}} + o(1)\right) \sqrt[3]{n \ln^2(n)}\right).$$

En 2009, une coalition internationale d'équipes de recherche en cryptographie a battu un record de factorisation en cassant une clef RSA de 768 bits. Le calcul avait mis des mois sur de gros clusters de recherche.



On y reviendra en détail sur RSA dans § 10.1. Les exercices 2.1–2.7 discutent de la complexité des attaques sur RSA.

Elgamal Très vite, RSA a été protégé par un brevet qui rendait son utilisation libre impossible. Pour cette raison entre autres, d'autres mécanismes à clef publiques ont été développés, reposant

1. un nombre premier n'a pas d'autre diviseurs entiers que 1 et lui-même

sur d'autres problèmes mathématiques. En 1985, طاهر الجمل (Taher Elgamal) un chercheur égyptien né en 1955, propose une alternative à RSA. Sa sécurité est du même niveau que celle de RSA.

Le système Elgamal repose sur un problème mathématique différent, et il a connu un large succès dans la mesure où il a été utilisé par des logiciels libres tels que PGP/GPG. On y reviendra § 8.2.

2.1.2 Sécurité « Prouvée »

Comme la sécurité des mécanismes à clef publique repose sur la difficulté calculatoire de problèmes mathématiques, on peut envisager d'obtenir des « preuves de sécurité ». Cela revient à démontrer des théorèmes mathématiques de la forme :

> *Si on avait accès à un programme capable de casser le schéma (récupérer la clef secrète, déchiffrer un message, etc.) en temps T, alors on pourrait fabriquer un autre programme qui résoud le problème calculatoire sous-jascent en temps $T + ...$*

Il s'agit donc d'une *réduction* comme celles qui interviennent habituellement dans la théorie de la complexité. L'idée est que l'existence de telles réduction garantit que le schéma est sûr *tant que le problème mathématique sous-jascent est difficile*. Ces preuves ne sont donc pas absolues. Elles garantissent cependant que la sécurité du mécanisme repose *vraiment* sur la difficulté du problème mathématique (on ne peut pas casser le schéma sans résoudre le problème « en passant »).

Par exemple, bizarrement, il n'a jamais été possible d'exhiber une réduction qui démontre que la sécurité de RSA repose « vraiment » sur le problème de la factorisation des grands entiers. Il est donc concevable qu'un jour quelqu'un casse RSA sans qu'on puisse en déduire un algorithme de factorisation. Ceci semble cependant improbable, et le meilleur moyen de « casser » RSA aujourd'hui (en 2019) consiste à essayer de factoriser la clef publique.

2.2 Protocoles à clef publique

Le fait que tout le monde puisse chiffrer des messages qui me sont destinés change beaucoup de choses. En premier lieux, il est impossible de savoir qui est l'expéditeur. En effet, dans la cryptographie à clef secrète, une fois qu'on a réussi à déchiffrer le message qu'on vient de recevoir, on a la certitude que l'expéditeur possédait la clef secrète (ceci est certain avec un chiffrement authentifié, et probable avec un chiffrement non-authentifié). Ceci l'identifie dans une certaine mesure. En clef-publique, c'est assez différent.

2.2.1 Identification sans mot de passe dans SSH

Dans le protocole SSH, il est possible de se connecter à une machine distante sans taper de mot de passe, mais avec une identification cryptographique à la place. Si la machine à laquelle on se connecte possède notre clef publique SSH (ces clefs sont stockées dans un fichier dns un sous-repertoire « caché » de notre HOMEDIR, `~/.ssh/authorized_keys`), alors elle peut vérifier qu'on possède bien la clef secrète correspondante. Ceci nous identifie (on possède un secret que seul nous sommes sensés posséder).

Pour cela, le serveur SSH génère une suite de bits aléatoires, la chiffre avec notre clef publique et nous l'envoie. On effectue le déchiffrement avec la clef secrète, et on renvoie le nombre aléatoire.

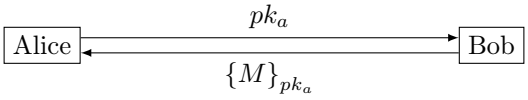
Notez qu'un observateur extérieur n'apprend *rien* : il voit passer un nombre aléatoire (qui ne contient aucune information), ainsi que son chiffré par notre clef publique. Or ceci, il aurait pu le calculer lui-même, puisque la clef qui sert à ça est publique.

Evidemment, ceci ne fonctionne bien que si on a eu un moyen *fiable* de transmettre notre clef publique sur la machine en question.

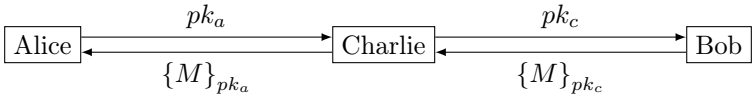
2.2.2 Attaques par le milieu

Cette caractéristique ouvre la porte à des « attaques par le milieu » dont il faut se méfier. Au passage, ceci illustre que les adversaires *actifs* sont beaucoup plus difficiles à gérer que les passifs.

Imaginons le protocole suivant : Alice envoie sa clef publique à Bob, et en retour Bob lui envoie un message chiffré.



L'attaque est la suivante : Charlie intercepte la communication, et envoie *sa propre clef publique* à Bob. Charlie peut alors déchiffer le message envoyé par Bob, puis le rechiffrer avec la clef publique d'Alice, qui ne se rend compte de rien !



Le problème est que Bob ne peut pas savoir si la clef publique qu'il a reçu est bien celle d'Alice.

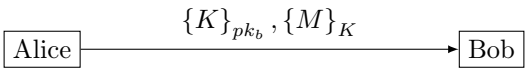
Des protocoles d'identification mutuelle, comme le protocole de Needham–Schroeder-Lowe à clef publique (cf. TD), permettent de déjouer ces attaques dans une certaine mesure.

2.2.3 Chiffrement hybride

Le chiffrement à clef publique est très lent comparé au chiffrement à clef secrète. Ce problème peut se contourner de deux façons :

1. En utilisant des techniques à clef publique pour établir une *clef de session* secrète.
2. Avec le chiffrement *hybride*.

L'idée du chiffrement hybride est simple. Pour envoyer un long message chiffré à Bob, Alice génère aléatoirement une clef symétrique fraiche K puis chiffre le message clair avec la clef K et un algorithme à clef secrète. Enfin, elle chiffre K avec la clef publique de Bob (cette fois avec un algorithme à clef publique).



Bob déchiffre la clef temporaire K avec sa clef secrète sk_b, puis déchiffre le message lui-même. Cette technique est largement employée, par exemple dans le logiciel PGP et dès qu'on envoie un email chiffré.

2.2.4 Authentification mutuelle

Le protocole de Needham-Schroeder *à clef publique* (ne pas confondre avec celui à clef secrète de § 1.2.4) est un protocole qui date de 1978. Alice et Bob l'exécutent pour s'assurer de leurs identités respectives (Alice veut être sûre qu'elle parle à Bob, et Bob veut être sûr qu'il parle à Alice). Alice et Bob utilisent tous les deux une méthode de chiffrement à clef publique, et ils possèdent chacun la clef publique de l'autre. L'un des deux participants (ici Alice) lance le protocole, qui fonctionne en trois temps :

Protocole NS$_{1978}$ (*protocole de Needham-Schroeder à clef publique, version 1978*). Bob démontre à Alice qu'il peut déchiffrer avec sk_b (donc qu'il est bien Bob), puis Alice démontre elle aussi la connaissance de sk_a en effectuant un déchiffrement.

NS1. [A défie B.] $A \to B : \{N_a\}_{pk_b}$ (N_a nonce aléatoire)

NS2. [B répond, défie A.] $B \to A : \{N_a, N_b\}_{pk_a}$ (N_b nonce aléatoire)

NS3. [A répond.] $A \to B : \{N_b\}_{pk_b}$

Alors que ce protocole paraissait sûr, Gavin Lowe a trouvé en 1995 (presque 20 ans après !) une faille en utilisant un programme de recherche automatique d'attaques qu'il avait conçu. L'attaque permet permet à Irène (l'*intrus*) de se faire passer pour Alice auprès de Bob si, au même moment, Alice exécute légitimement le protocole avec elle (ce qui permet à Irène de relayer des messages d'Alice à Bob, alors qu'ils ne lui étaient pas destinés).

On peut réparer le protocole de la façon suivante : chaque fois qu'un participant envoie un contenu chiffré, il ajoute son identité à l'intérieur du message chiffré. Le protocole devient donc :

Protocole NSL (*protocole de Needham-Schroeder-Lowe à clef publique*). Bob démontre à Alice qu'il peut déchiffrer avec sk_b (donc qu'il est bien Bob), puis Alice démontre elle aussi la connaissance de sk_a en effectuant un déchiffrement.

NSL1. [A défie B.] $A \to B : \{A, N_a\}_{pk_b}$ (N_a nonce aléatoire)

NSL2. [B répond, défie A.] $B \to A : \{B, N_a, N_b\}_{pk_a}$ (N_b nonce aléatoire)

NSL3. [A répond.] $A \to B : \{A, N_b\}_{pk_b}$

On suppose aussi que les participants au protocole vérifient que l'identité supposée des participants correspond à celle qui figure dans les message. Les exercices 2.8 et 2.9 discutent de l'attaque et de la réparation.

2.3 Signature numérique

Les signatures servent à garantir l'authenticité d'un document. En principe, quand on signe un papier, on exprime son accord avec le contenu du papier en question. Dans ce sens, une signature engage notre responsabilité.

Il est donc important qu'on ne puisse pas signer à la place de quelqu'un d'autre. L'authenticité des signatures est établie par comparaison avec un échantillon « de reférence » (la signature qui figure sur un passeport, par exemple).

Comme la signature est liée physiquement au document, il n'y a en principe pas d'ambiguité sur le contenu auquel le signataire a voulu apporter son accord. On est d'accord avec la feuille qu'on a signée ! Mais par contre, rien n'interdit qu'elle soit modifiée *après coup*... Enfin, cela laisse en principe des traces physiques. Enfin, dernière caractéristique : une signature doit être publiquement vérifiable. Des tierces personnes (l'administration, un tribunal, le destinataire de vos lettres d'amour) doivent pouvoir se convaincre qu'il s'agit bien de votre signature.

Si on voulait faire des signatures électroniques, les choses seraient forcément assez différentes. La signature électronique d'un document (ou d'un message) M va être une chaine de bits. Elle n'est plus liée physiquement au document de départ (qui est une autre chaine de bits).

Il faut donc qu'une signature de M par la personne P soit une chaine de bits qui dépende de M et que seule P puisse produire. Pour obtenir ce résultat, il faut que la production d'une signature au nom de P nécessite la connaissance d'une chaine de bits que seule P possède.

On se rend alors compte que la signature est alors, en quelque sorte, l'opération « inverse » du chiffrement :

chiffrement	**signature**
Tout le monde produit les chiffrés	Seul moi produit la signature
Seul moi peut déchiffrer	Tout le monde peut vérifier

De fait, un schéma de signature fonctionne avec deux clefs :
- Une *clef de signature* qui est secrète, et sert à signer.
- Une *clef de vérification* qui est publique, et sert à vérifier les signatures.

C'est le secret de la clef de signature qui garantit l'identité d'une personne, et qui rend les signatures non-répudiables : si on trouve dans la nature une signature de M qui est correctement vérifiée avec ma clef publique, c'est forcément qu'elle a été produite avec ma clef secrète.

Les signatures ont d'innombrables applications : qu'il s'agisse d'authentifier des logiciels à télécharger, d'authentifier des messages, de « signer » sa déclaration d'impôts en ligne...

Badges « Vigik » Par exemple, un procédé de signature électronique est utilisé dans les badges « Vigik » qui permettent d'accéder à de nombreux halls d'immeubles.

Le problème des hall d'immeubles, c'est que les facteurs doivent y avoir accès pour déposer le courrier. Dans le passé, il y avait une *clef PTT*, physique, la même partout en France, que tous les facteurs possédaient. Mais elle a « fuit » progressivement, et le dispositif de fermeture ne servait plus à grand-chose.

A la place, les badges Vigik sont des mémoires passives, incapables de faire des calculs (c'est une solution « *low-cost* »). Quand on « badge », ils ne font que transmettre leur contenu. Ils contiennent une chaine de bits A qui décrit les autorisations d'ouverture du badge (quelle rue, pendant quelle période), ainsi qu'une signature de A par La Poste ou par le syndic de l'immeuble.

Les habitants ont une autorisation uniquement pour leur immeuble, sans date limite de validité. Les facteurs ont une autorisation pour des secteurs entiers, mais valide uniquement une journée. Ils font reprogrammer leurs badges chaque jour en principe. Ceci limite les dégats lorsqu'un badge est perdu.

Les lecteurs muraux, qui sont très largement répandus, ne contiennent que la clef de vérification (publique) de la poste et du syndic. Voler un lecteur mural n'a donc aucun intérêt. Par contre, comme les badges sont des mémoires passives, ils sont clonables. Cette technique n'est donc pas adaptée, par exemple, à des titres de transport.

2.3.1 Algorithmes de signature

Un schéma de signature se compose :
- D'un algorithme (randomizé) de génération de clefs.
- D'un algorithme de signature, qui prend en argument un message, éventuellement des bits aléatoires, la clef secrète et qui produit une signature.
- D'un algorithme de vérification, qui prend en argument un message, une chaine de bits S, la clef publique, et qui renvoie « Vrai » si S est une signature valide du message.
- (Dans certains cas) D'un algorithme de génération de paramètres.

Si on a sous la main un algorithme de chiffrement à clef secrète qui s'y prête, on peut le « convertir » en algorithme de signature. En gros, il suffit de dire que la signature S d'un message M est son *déchiffrement* :

$$\textsc{Sign}(sk, M) = \mathcal{D}(sk, M).$$

Seul le propriétaire de la clef secrète peut le faire. Pour vérifier la signature, on vérifie si on retombe bien sur le message en *re-chiffrant* la signature :

$$\textsc{Check}(pk, M, S) = \begin{cases} 1 & \text{si } \mathcal{E}(pk, M) = S \\ 0 & \text{sinon} \end{cases}$$

Tout le monde peut effectuer la vérification.

Les principaux algorithmes de signatures correspondent aux principaux algortihmes de chiffrement : on trouve la signature RSA, la signature Elgamal, standardisée par le gouvernement américain sous le nom de DSA (« Digital Signature Algorithm »), plus récemment on trouve aussi l'algorithme ECDSA (« Elliptic Curves Digital Signature Algorithm ») utilisé dans la PlayStation 3 et dans les versions récentes du protocole SSH

Aujourd'hui, la plupart des logiciels de courrier électroniques ont des « *plugins* » qui permet de gérer du courrier chiffré et signé. Quant à l'auteur, vous pouvez récupérer sa clef publique sur sa page web. Vous êtes invités à envoyer du courrier chiffré et signé !

2.4 Infrastructure de gestion des clefs publiques (PKI)

Le problème, avec la cryptographie à clef publique en général (chiffrement, signature, identification, etc.) est de s'assurer qu'on a bien les bonnes clefs publiques.

Dès qu'on obtient les clefs publiques par des canaux non-fiables, on est à la merci d'adversaires actifs qui subtituent leurs propres clef à celles des interlocuteurs légitimes. Si jamais le serveur de

l'université se fait pirater, les pirates pourraient remplacer la clef publique de l'auteur de ces lignes par une autre sur sa page web.

2.4.1 Certificats

Une solution partielle à ce problème consiste à utiliser des *certificat*. Un certificat contient :

— Une identité (adresse email, URL d'un serveur, adresse IP d'une machine, ...).
— Une clef publique associée à cette identité.
— Une signature des deux éléments ci-dessus par une *autorité de certification*.

Quand quelqu'un vous donne un certificat, il vous dit « l'autorité de certification MachinChose garantit que ceci est bien ma clef publique ». Et effectivement seule l'autorité de certification peut produire la signature.

Le problème, c'est que pour vérifier si un certificat est valide, il faut posséder... la clef publique (la vraie !) de l'autorité de certification. Pour cela, l'autorité de certification peut fournir son propre certificat, qu'on doit vérifier... avec la clef publique d'une autre autorité de certification. On ne s'en sort plus ! En fait, les clefs publiques des plus importantes sont fournies avec votre navigateur web [2].

Exemple : protocole SSL/TLS Le portail de l'université (`https://portail.univ-lille1.fr/`) est accessible via une connexion sécurisée. Version courte : un protocole à clef publique est utilisé pour a) établir un tunnel chiffré et authentifié jusqu'au serveur HTTPS de l'université et b) garantir à votre navigateur web que la machine à laquelle il est connecté est bel et bien le serveur HTTPS de l'université. Pour démontrer son identité lors de la connection, le serveur web de l'université effectue une signature avec l'algorithme de signature RSA. Le serveur web envoie également un certificat, qui permet au navigateur web de vérifier la correction de la signature.

En fait, le certificat de l'université a été émis par une entreprise hollandaise, Terena. Mon navigateur web ne contient pas le certificat de Terena. Mais le serveur HTTPS de l'université fournit aussi un certificat pour Terena, émis par l'entreprise américaine DigiCert Inc. Mon navigateur web contient d'avance la clef publique de DigiCert Inc. La vérification de la chaine de certificats est automatique. Cf. exercices 2.10–2.11.

Exemple : le système des cartes bleues Une norme internationale nommée EMV spécifie le fonctionnement de la cryptographie dans les cartes bancaires contenant un microprocesseur (les plus répandues en Europe). On va admettre que ce microprocesseur contient une mémoire qui n'est pas facilement accessible, et qui permet donc de stocker des données secrètes. La carte bleue contient aussi des données non-secrètes, qui ont vocation à être communiquées au terminal de paiement : le numéro du compte bancaire à débiter, par exemple, ainsi que des clefs publiques et des certificats. Le système RSA est utilisé en toute circonstance.

L'usage de la cryptographie vise principalement à empêcher la fraude, par les clients (qui possèdent une carte bleue) ou par les marchands (qui possèdent un terminal de paiement). L'hypothèse de travail est que des fraudeurs peuvent espionner les communications entre la carte et le terminal de paiement. Dans ce système, chaque banque possède une ou des clef(s) publique(s). Une autorité de certification spécifique au système EMV fournit des certificats aux banques. Les terminaux de paiement possèdent tous la clef publique de cette autorité.

Les cartes bleues EMV peuvent fonctionner dans deux modes : le mode « statique » (moins sûr) et le mode « dynamique ». Dans le mode statique, la carte contient seulement une mémoire et n'a pas besoin de faire des calculs. La mémoire contient une signature des données spécifiques à la carte (le numéro du compte, etc.) par la banque, ainsi que le certificat de la banque.

Dans le mode dynamique, la carte possède un CPU. Sa mémoire contient des données bancaires, ainsi qu'une paire de clefs RSA. Pour authentifier la carte, le terminal lui envoie des données (le numéro de compte du marchand, la date, ainsi qu'un nombre aléatoire imprévisible), et la carte en renvoie une signature. Les exercices 2.12– 2.16 discutent des détails.

2. enfin, *des* clefs publiques sont fournies. De la à dire que ce sont les bonnes, il y a un pas que tout bon paranoïaque digne de ce nom hésiterait à franchir...

2.4.2 « Web-of-Trust »

La technique des certificats est bien adaptés, par exemple, aux sites institutionnels (banques, sites marchands, etc.) ou aux grandes entreprises (elles peuvent obtenir un certificat, puis certifier elles-mêmes les clefs de leurs employés, etc.).

Mais ce n'est pas vraiment adapté à toutes les situations (communautés décentralisés, hackers, geeks, etc.) [3]. Une autre solution, partielle elle aussi, à ce problème, est la technique du *réseau de confiance*. Je sais que *ma* clef est authentique. Celles des autres, je ne sais pas. Mais si je rencontre Alice physiquement, nous pouvons *authentifier nos clefs* mutuellement : Alice signe ma clef, et je signe celle d'Alice. Après, je pourrais montrer ma clef à des tiers, avec la signature d'Alice : cela garantit qu'Alice, au moins, pense que ma clef est valide.

Par exemple, si j'envoie ma clef à Bob, et que Bob pense qu'il a la « vraie » clef d'Alice, il peut vérifier la signature d'Alice et s'assurer que ma clef est authentique.

Ceci définit un graphe orienté (si A signe la clef de B on met une arête $A \to B$). C'est le « *Web-of-trust* ». S'il y a un chemin de X vers Y, ça veut dire qu'il y a une « chaine de confiance » ininterrompue entre X et Y. Madame X peut donc raisonnablement se convaincre que la clef de Monsieur Y qu'elle a entre les mains est légitime.

Ceci n'est pas la solution miracle. D'une part ça fait grossir la taille des clefs, d'autre part les chaines n'existent pas toujours. Le mécanisme a été implémenté dans le logiciel PGP.

Exercices

Exercice 2.1 : Casser une clef RSA de 1024 bits demande combien de fois plus de calculs qu'une clef de 768 bits ?

▷ **Exercice 2.2 :** Quel niveau de sécurité (=taille de clef secrète équivalente) offrent les clefs de 2048 bits recommandées aujourd'hui ?

▷ **Exercice 2.3 :** Quelle taille de clef RSA faut-il choisir pour s'assurer un niveau de sécurité équivalent à des clefs secrètes de 128 bits ? 256 bits ?

▷ **Exercice 2.4 :** Le meilleur code publiquement disponible de factorisation, CADO-NFS, a besoin de 90 jours pour factoriser un nombre de 512 bits sur un coeur à 2Ghz. On peut estimer qu'un serveur qui contient 36 coeurs comparables coûte 4000 euros, et consomme 400W. Quel budget est nécessaire pour casser RSA-1024 en 3 mois ?

▷ **Exercice 2.5 :** Quelle puissance electrique est nécessaire ?

▷ **Exercice 2.6 :** Ceci est-il a la portée des Etats des pays riches ? (note : en 2016, l'Etat français a commandé 6 sous-marins nucléaires pour $8.5 \cdot 10^9$€. Chaque sous-marin embarquerait un réacteur nucléaire de 150MW.)

▷ **Exercice 2.7 :** Casser une clef de 2048 bits dans les mêmes délais demandera combien de fois plus de calculs (donc d'argent et de puissance électrique) ?

Exercice 2.8 : Décrivez une attaque contre le protocole de Needham-Schroeder à clef publique de 1978. Alice initie légitimement le protocole avec Irène, et Irène en profite pour se faire passer pour Alice auprès de Bob.

Exercice 2.9 : Pourquoi le protocole modifié évite-t-il l'attaque ?

▷ **Exercice 2.10 :** Détaillez les opérations effectuées par mon navigateur web pour effectuer la vérification de la signature du serveur HTTPS de l'université.

3. Au passage, jusqu'à un passé récent, les autorités de certifications commerciales facturaient la délivrance de certificats. Aujourd'hui, il existe (au moins une) autorité de certification gratuite.

▷ **Exercice 2.11 :** La clef publique RSA de DigiCert Inc est de taille 2048 bits, tout comme celle de Terena. Celle de l'université est de taille 4096 bits. Ceci est-il judicieux ?

▷ **Exercice 2.12 :** Lorsqu'on lui présente une carte EMV « statique », que doit vérifier le terminal de paiement ? Que doit-il contenir à l'avance pour cela ?

▷ **Exercice 2.13 :** Quelle manoeuvre de fraude est rendue impossible par la vérification de l'exercice précédent ?

▷ **Exercice 2.14 :** Quelle autre manoeuvre de fraude n'est *pas* rendue impossible par cette vérification ?

▷ **Exercice 2.15 :** Lorsqu'on lui présente une carte EMV « dynamique », que doit récupérer le terminal de paiement sur la carte ? Que doit-il vérifier ?

▷ **Exercice 2.16 :** Pourquoi le mode dynamique améliore-t-il la sécurité par rapport au mode statique ?

Exercice 2.17 : Soit $\mathcal{E}$ un mécanisme de chiffrement à clef publique. Considérons le problème algorithmique suivant :

INPUT — Une clef publique pk.
— Une chaine de bits c.

QUESTION c est-il un chiffré valide ? Autrement dit, existe-t-il un message m et un aléa r tel que $c = \mathcal{E}(pk, m, r)$?

Démontrer que ce problème est dans $NP \cap coNP$ (coNP est la classe des problèmes pour lesquels il existe un certificat verifiable en temps polynomial lorsque la réponse est « NON »). Ceci semble suggérer que ce problème n'est pas NP-complet (car on ne connait aucun problème NP-complet qui soit dans coNP).

Chapitre 3

Fonctions de hachage cryptographiques

Il n'est pas facile de définir ce qu'est une fonction de hachage, et encore moins une fonction de hachage cryptographique. Le problème de fond est qu'on souhaiterait que les fonctions de hachage possèdent des propriétés impossibles à réaliser et/ou contradictoires entre elles. D'après Wikipédia, une *fonction de hachage* est « *[...] une fonction particulière qui, à partir d'une donnée fournie en entrée, calcule une empreinte servant à identifier rapidement, bien qu'incomplètement, la donnée initiale.u* ».

Le mot-clef est *empreinte*. Une fonction de hachage prend en entrée une chaine de bits de taille arbitraire et produit en sortie une chaine de bits de taille fixée et réduite : la fameuse empreinte. C'est donc une fonction $H : \{0,1\}^* \to \{0,1\}^n$. La taille des empreintes (la valeur de n) dépend de ce qu'on veut faire avec. S'il s'agit de calculer des index dans un tableau, $= 32$ est presque suffisant (ça fait déjà un tableau à $2^{32} \approx 4$ milliards d'éléments qui occupe donc au minimum 4Go...).

Une première remarque qui s'impose, c'est qu'une fonction de hachage ne peut pas être *injective*[1]. En effet, son domaine d'entrée contient une infinité d'éléments, tandis que son domaine de sortie en contient un nombre fini, qui est précisément 2^n. Il y a donc forcément des *collisions*, c'est-à-dire des paires d'entrées $x \neq y$ telles que $H(x) = H(y)$.

La plupart du temps, on essaye cependant de faire en sorte que la sortie soit *uniformément répartie*, c'est-à-dire que si on choisit l'entrée à peu près au hasard, alors on a autant de chance de produire n'importe laquelle des 2^n sorties possibles (mais ce n'est pas un objectif facile à atteindre, et puis tout dépend de ce qu'on veut dire par « choisir l'entrée à peu près au hasard »). Dans le cadre de leur usage en algorithmique ou dans des base de données, l'uniformité des fonctions de hachage sert à assurer l'efficacité des opérations, en « dispersant » les données à stocker dans toutes les alvéoles (« *buckets* ») d'une table de hachage, en fonction de leurs empreintes.

Cependant, les fonctions de hachages cryptographiques doivent avoir des propriétés supplémentaires. L'idée générale, légèrement contradictoire, est la suivante : il faut d'une part que l'empreinte d'une donnée M (éventuellement secrète) contienne assez d'information sur M pour qu'on puisse la distinguer d'une empreinte de $M' \neq M$ avec très forte probabilité. Mais il faut d'autre part que l'empreinte de M ne révèle *aucune information exploitable* sur M. Pour poser le problème, voici un catalogue de cinq situations où des fonctions de hachage cryptographiques peuvent être utilisées pour résoudre des problèmes apparement insolubles.

3.1 Exemples d'utilisation

3.1.1 Dérivation d'une clef secrète à partir d'un mot de passe

On a vu qu'il y a de nombreuses situations où des mots de passe doivent être utilisés pour chiffrer des données avec un mécanisme de chiffrement symétrique. Le problème, c'est que ces mécanismes

1. Une fonction f est injective si $x \neq y \Rightarrow f(x) \neq f(y)$, autrement dit si deux entrées différentes produisent toujours deux sorties différentes.

de chiffrement attendent des clefs d'une taille fixée (par exemple 128 bits). Un mot de passe ne fait pas forcément 128 bits. Il peut être plus long, ou plus court.

La solution est simple : il suffit de *hacher* le mot de passe en posant

$$K \leftarrow H(\texttt{password})$$

avec une fonction de hachage H qui produit des empreintes de la bonne taille. En fait, si H produit des empreintes trop grande, ce n'est pas grave, il suffit de les tronquer. Ceci nécessite que la sortie de la fonction de hachage « ait l'air aléatoire » même quand les entrées possèdent une distribution particulière.

3.1.2 Stockage des mots de passe

On a déjà discuté de ce problème et présenté eune première solution au § 1.2.1. Une solution raisonnable consiste à stocker, l'empreinte du mot de passe $H(\texttt{pwd})$. Le système peut toujours tester si un mot de passe donné x est correct en calculant $H(x)$ et en comparant le résultat avec la valeur stockée.

Pour que ce système offre une certaine sécurité, nous voyons apparaître une première propriété désirable des fonctions de hachage cryptographiques : il faut que H soit à « sens unique », c'est-à-dire qu'il soit calculatoirement impossible de trouver une entrée qui produise une sortie donnée.

Pour éviter que des adversaires utilisent des tables précalculées contenant les empreintes de tous les mots de passe les plus courants, on utilise parfois une technique de *salage* : on stocke la paire $\big(\texttt{salt}, H(\texttt{salt}\|\texttt{pwd})\big)$, où `salt` est aléatoire. Du coup, le précalcul n'est plus possible.

Salage et chiffrement non-déterministe sont deux choses différentes. Un mécanisme de chiffrement déterministe $\mathcal{E}$ peut être rendu non-déterministe par l'idée suivante :

$$\mathcal{E}'(K, M, r) = \mathcal{E}(K, M \parallel r).$$

Lors du déchiffrement, il suffit de jeter à la poubelle la partie correspondant à r après avoir appliqué $\mathcal{D}$. En tout cas, la différence est manifeste : dans le salage, l'aléa utilisé est fourni, alors que dans le chiffrement non-déterministe, il ne l'est pas.

L'exercice 3.2 présente un système alternatif, plus sophistiqué.

3.1.3 Signatures

Les algorithmes de signatures ne peuvent signer que des messages d'une taille fixée, qui est toujours inférieure à la taille de la clef. Par exemple, les signatures RSA avec des clefs de 2048 bits ne peuvent signer que des messages de moins de 2048 bits. Comment faire pour signer un fichier d'un Giga-octet (par exemple : l'enregistrement vidéo d'un évenement officiel, tel qu'un procès) ? Générer des clefs RSA de cette taille-là est impossible en pratique.

La solution est simple : il suffit de signer une empreinte du message, donc de signer $H(M)$ à la place de M. Mais la sécurité du système de signature dépend alors aussi de celle de la fonction de hachage. Si Alice trouve une collision ($M \neq M'$ et $H(M) = H(M')$), alors elle peut produire une signature de M, et ensuite prétendre que non, pas du tout, il s'agissait d'une signature de M' (très pratique si M est une reconnaissance de dettes et M' du charabia aléatoire). Du coup les signatures ne garantissent plus rien. Il faut donc qu'il soit calculatoirement impossible de produire des collisions.

En pratique, la méthode « hash-and-sign » est universellement utilisée.

3.1.4 Mise en gage

« Pierre-Feuille-Ciseau » en ligne

Imaginons que nous voulions mettre au point un site permettant de jouer en ligne à une variante généralisée de *shifumi* (« pierre-feuille-ciseau »). L'idée est la suivante : à chaque tour les deux

joueurs choisissent en secret une chaine de n bits. Il y a une fonction de gain F, que tout le monde connaît. Le calcul de $F(x, y)$ renvoie 1 si la chaine de n bits x « bat » y, et -1 si c'est l'inverse (on peut prendre par exemple $F(x, y) = 1$ si le XOR de x et y contient une majorité de bits à zéros). Comme dans le jeu classique, chaque choix a autant de chance de gagner que n'importe quelle autre contre un joueur aléatoire. Si on perd, on donne 1 euro au gagnant.

Pour jouer, il suffit de se connecter sur le site, de choisir un adversaire, puis d'entrer sa chaine de n bits. Une fois que l'adversaire à fait de même, le serveur envoie à l'un la chaîne de n bits de l'autre (très important pour élaborer une stratégie), puis détermine qui a gagné et enfin effectue le transfert financier.

Le problème c'est que celui qui fait tourner le serveur pourrait tricher : il pourrait y avoir un (ou même un énorme paquet) d'utilisateur(s) fictif(s) sur le site de jeu qui sont en fait des bots. Si vous jouez contre un bot, vous envoyez à un moment donné votre chaine de bits au serveur. Celui-ci, qui participe à la fraude, la transmet au bot qui peut calculer une réponse lui permettant de gagner face à votre choix.

Pour éviter ce problème, on peut se servir d'une fonction de hachage pour effectuer une sorte de « mise en gage ». On joue alors de la façon suivante :

1. Alice et Bob choisissent leurs chaines de bits, respectivement x et y.
2. Alice et Bob envoient respectivement $H(x)$ et $H(y)$ au serveur
3. Le serveur envoie $H(y)$ à Alice et $H(x)$ à Bob
4. Alice et Bob envoient respectivement x et y au serveur
5. Le serveur redispatche y et x, annonce le résultat, etc.

Le schéma de triche précédent est impossible, car le bot est obligé de *s'engager* sur son choix en envoyant l'empreinte. Il ne peut plus attendre de connaître ce que joue l'autre pour jouer lui-même, à moins de réussir à « contourner » la fonction de hachage.

S'il est capable de fabriquer des *collisions* pour H, alors l'administrateur du serveur peut mettre au point un bot qui a 66% de chances de gagner (contre 50% si on joue au hasard). Il n'en faut pas plus pour amasser une fortune !

Pour cela, le bot précalcule une collision $x \neq x'$ et $H(x) = H(x')$. Ensuite, face à un joueur normal, il envoie $H(x)$. Après avoir reçu le y de son adversaire, il a le choix entre jouer x ou x'. Ceci double ses chances de gagner. En plus, il suffit de connaître *une seule* collision pour pouvoir monter la fraude ; tant que le bot joue contre des gens différents, ils ne peuvent pas détecter le problème.

Comme dans le cas des schémas de signatures, il faut que la fonction de hachage soit « *résistante aux collision* », c'est-à-dire qu'il n'existe pas d'algorithme capable de produire des collisions sans procéder à une quantité astronomique de calculs.

Définition formelle de la mise en gage Un *schéma*[2] *de mise en gage* (« *commitment scheme* ») est un algorithme $\textsc{Commit}(r, b)$ qui reçoit en argument un nombre spécifié de bits aléatoires r et un bit b. Il produit une « mise en gage » (appelons-la $\mathcal{C}$) de b, c'est-à-dire une chaine de bits dans $\{0, 1\}^k$.

Pour mettre en gage un bit b, Alice envoie $\textsc{Commit}(r, b)$ à Bob, où r est un nonce aléatoire. Plus tard, Alice va « ouvrir » la mise en gage pour démontrer qu'elle l'avait effectuée correctement. Pour cela, Alice envoie b et r à Bob. Celui-ci peut alors calculer $\textsc{Commit}(r, b)$ et vérifier que la valeur transmise auparavant était correcte.

Un schéma de mise en gage doit avoir deux propriétés :

— Il doit être « *binding* » : une mise en gage de 0 ne doit pas pouvoir être ouverte comme une mise en gage de 1 (et vice-versa).

— Il doit être « *hiding* » : les mises en gage de 0 sont indistinguables des mises en gage de 1.

Ces deux propriétés peuvent être obtenue de manière *calculatoire* (briser la propriété nécessite un énorme calcul) ou bien *absolue* (*statisticaly*, *information-theoretically*) (briser la propriété n'est pas possible même avec une puissance de calcul illimitée). Malheureusement, on ne peut pas avoir les deux à la fois. Les exercices 3.7–3.10 discutent des détails.

2. « schéma », ici est utilisé dans le sens de « mécanisme ».

Horodatage

Imaginons maintenant que nous voulions mettre en place un webservice d'horodatage notarié. En gros il s'agit de certifier qu'un document a été produit à un moment donné — c'est une autre forme de mise en gage. Les utilisateurs du service veulent obtenir une preuve qu'ils étaient en possession d'un document M à l'instant t. Par exemple, un inventeur peut vouloir faire certifier qu'il était en possession d'une description de son invention avant de la rendre publique. En cas de plagiat, dans un procès il doit pouvoir exhiber cette preuve et démontrer qu'il « avait eu l'idée avant l'autre ». Mais les utilisateurs paranoïaques ne tiennent pas à confier le contenu de leurs documents au webservice : ils craignent, à juste titre, que le webservice ne récupère leur document, puis renvoie une erreur du type « le serveur est saturé, revenez plus tard » — et entre temps leur pique leur invention si elle vaut le coup.

La solution est simple : il suffit que les utilisateurs du webservice envoient l'empreinte de leurs documents. Le webservice leur renvoie un certificat (par exemple avec une signature electronique) qu'ils étaient bien en possession du fichier dont l'empreinte est $H(File)$, à l'instant t.

Mais pour que le webservice serve à quelque chose et qu'il puisse faire foi en cas de litige, il ne faut pas qu'il soit possible de tricher. En l'occurence, tricher signifierait, pour les utilisateurs, donner un haché h au webservice, puis fabriquer un document dont le haché est h *après coup*.

Par exemple, un escroc pourrait prétendre qu'il est capable de prédire le futur. Pour démontrer son allégation, il essaye de faire horodater un fichier contenant une description d'un évenement qui va se produire (l'évolution cours de la bourse par exemple, le résultat de la prochaine élection, ...). Après un gros calcul, il envoye un haché h_{fraud} au webservice. Maintenant, l'évenement qu'il avait soit-disant prédit se réalise : l'ecroc écrit sa description dans un fichier M_{event}. Il tente ensuite de trouver un « suffixe » M_{suffix} tel que

$$H\left(M_{\text{event}} \parallel M_{\text{suffix}}\right) = h_{\text{fraud}}.$$

S'il y parvient, l'escroc pourrait réléver son fichier de « prédiction » $M_{\text{event}} \parallel M_{\text{suffix}}$. Bien sûr, il y aurait un peu de données incompréhensibles à la fin, mais il pourrait toujours dire qu'il les a tapées pendant sa transe divinatoire.

La propriété que les fonctions de hachage doivent avoir pour résister à ce problème spécifique s'appelle la « *Chosen-Target Forced-Prefix Collision Resistance* ». En effet, l'adversaire peut choisir l'empreinte cible (h_{fraud}), mais on lui impose le début (M_{event}) de la préimage qu'il doit trouver.

3.1.5 Intégrité de messages

Imaginons que l'on doive mettre en place le système de guidage radio d'une fusée. Sauf dans des cas bien précis (satellites espion...), le contenu des messages envoyés à la fusée n'a rien de secret : allume le moteur, éteint le moteur, correction de trajectoire, etc. Ca ne sert donc à rien de le chiffrer. Par contre, ce qui est important, c'est que nous, et nous seuls, puissions envoyer des messages à notre fusée [3]. Pour ça on pourrait envoyer des messages chiffrés avec une clef secrète (aussi stockée dans la fusée) : une tierce personne ne possédant pas la clef ne pourrait pas chiffrer, et donc ne pourrait qu'envoyer des message qui, une fois déchiffrés par la fusée, seront du charabia aléatoire.

Ce qui est plus problématique, c'est la possibilité qu'une tierce personne perturbe nos messages à nous. Imaginons qu'on envoie :

```
Correction trajectoire : 0x01 degré à gauche pendant 0x001A secondes
```

En envoyant un paquet d'ondes radio pile au bon moment, nos ennemis pourraient altérer le message de manière imprévisible, et vous imaginez la catastrophe si la fusée reçoit :

```
Correction trajectoire : 0x01 degré à gauche pendant 0x6B2F secondes
```

En plus, on voudrait éviter de chiffrer les messages, parce que le déchiffrement demande de l'énergie, qui est une chose rare dans une fusée [4]. On décide donc d'utiliser un *Code d'authentification de message* (« Message Authentication Code », ou MAC). Après avoir envoyé le message M, on envoie aussi $t \leftarrow \mathcal{E}(K, H(M))$, où K est une clef symétrique partagée entre la fusée et la base. La fusée, après réception d'un message M, ne le considère comme légitime que si $H(M) = \mathcal{D}(K, t)$.

3. Toute ressemblance avec le film « James Bond 007 contre dr. No » est purement accidentelle
4. Toute ressemblance avec le film « Apollo 13 » est purement accidentelle

Un MAC est donc une sorte de signature à clef symétrique : il faut la même clef pour produire le *tag* (la « signature ») et pour le vérifier. Les MAC garantissent que l'expéditeur du message connaît la clef de MAC utilisée. Ils ne garantissent pas la non-répudiation comme les signatures, car la clef est partagée par plusieurs utilisateurs.

Envoyer un message pirate est difficile, car on ne peut pas fabriquer la bonne valeur du « tag » t sans connaître la clef K. Par contre, on peut intercepter des valeurs de t « légitimes », et les messages qui vont avec. En faisant ça, on obtient donc un message M, dont on peut calculer l'empreinte $H(M)$, et on connaît la valeur de son MAC, $t = \mathcal{E}(K, H(M))$. Si on arrive à fabriquer un message M' dont l'empreinte est $H(M)$, alors t sera un MAC valide du message en question, et la fusée l'acceptera comme étant légitime.

Pour que ce protocole de communication soit sûr, il faut que la fonction de hachage soit « *résistante aux secondes préimages* », c'est-à-dire qu'aucun algorithme ne doit être capable, étant donné un message M, de produire un message M' différent qui collisionne avec M. Trouver une seconde préimage est plus dur que de trouver une collision, car on a pas la liberté de choisir M ; il nous est imposé. L'exercice 3.3 montre comment cela peut être utilisé sur le web.

3.1.6 Audit de stockage

Imaginons que nous voulions mettre en place un webservice de sauvegarde en ligne (style *dropbox*, *google drive*, *amazon S3*, etc.). Pour se démarquer de la concurrence, nous voudrions que les utilisateurs puissent *auditer* le stockage, c'est-à-dire obtenir une preuve que le webservice possède bel et bien leurs fichiers (qu'ils n'ont pas été perdu accidentellement, ni volontairement, pour « faire de la place »).

Evidemment, les utilisateurs pourraient télécharger périodiquement leurs fichiers pour en vérifier le contenu, mais cela consommerait beaucoup de bande passante. On pourrait imaginer que pour économiser la bande passante, le webservice renvoie uniquement l'empreinte du fichier demandé (en partant de l'idée qu'il faut connaître M pour calculer l'empreinte de M). Le problème, c'est que le serveur de stockage pourrait calculer les empreintes à la réception des fichiers et les stocker à part. Quand le besoin s'en fait sentir, il efface une partie des fichiers stockés — c'est moins cher que d'acheter de nouveaux disques durs. En cas de demande d'audit, il utilise les empreintes stockées pour répondre.

Pour éviter ça, on met en place le protocole suivant :

1. L'utilisateur choisit une courte chaine de bits C et l'envoie au webservice.
2. Le webservice calcule $h \leftarrow H(File \parallel C)$ et renvoie h.
3. L'utilisateur vérifie que le haché est correct.

L'avantage est que là aussi, seules de courtes chaines de bits sont échangées (disons 256 bits dans chaque sens). Si le serveur de stockage a « perdu » le fichier, il ne doit pas lui être possible de répondre. L'intérêt de C, dans cette optique, c'est de rendre inutile le stockage des empreintes des fichiers, car C modifie la réponse. Il faut posséder le fichier pour être capable de calculer h.

On a laissé cet exemple pour la fin car il est beaucoup plus subtil que tous les autres. En gros, il faudrait que la fonction de hachage possède la propriété suivante : « quelqu'un capable de calculer $H(x)$ connaît forcément x ». Cette propriété est plutôt difficile à formaliser (qu'est-ce que ça veut dire qu'un programme « connaît » une chaine de bits ?).

Conclusion partielle

On voit que les fonctions de hachage cryptographiques peuvent servir dans une grande variété de situations (et encore, il y en a bien d'autres). C'est un peu le « couteau suisse » de la cryptographie. Mais on a aussi vu que chaque situation nécessite que les fonctions de hachage satisfassent une propriété différente. C'est (une partie de) la raison pour laquelle il est délicat de dire précisément ce qu'est une fonction de hachage cryptographique.

3.2 Modèle de l'oracle aléatoire

S'il fallait résumer, on pourrait dire qu'une fonction de hachage est une fonction $\{0,1\}^* \to \{0,1\}^n$ dont le *code source est public* et qui ne possède *aucune structure exploitable* permettant de relier ses entrées et ses sorties.

On pourrait imaginer construire un tel dispositif de la façon suivante : un serveur mondial exécute un *web-service* qui permet à tous les utilisateurs d'internet de calculer des empreintes. Pour obtenir l'empreinte de M, les utilisateurs chargent une URL qui invoque la méthode `query` ci-dessous (c'est du code Python) :

```
class RandomOracle:
    log = {}

    def query(self, M):
        if M in self.log:
            return self.log[M]
        h = RandomGenerator.getrandbits(n)
        self.log[M] = h
        return h
```

Ce *web-service* fait deux choses :

— si ce n'est pas la première fois que M est demandé, il renvoie la valeur précédement renvoyée. Du coup, le résultat est une « vraie fonction » qui renvoie toujours la même valeur quand on lui soumet le même argument.

— Si M est « nouveau », alors il choisit une empreinte *complètement au hasard*, la stocke et la renvoie. Du coup, les sorties n'ont *aucun rapport* avec les entrées. En particulier, la sortie ne fait pas « fuir » d'information sur l'entrée.

Le gros problème c'est qu'il est impossible à réaliser dans la pratique : l'espace de stockage nécessaire n'est pas du tout borné ! Cependant, ce petit bout de code garantit, par construction, l'absence de correlation entre les entrées et les sorties. Il simule effectivement une *fonction aléatoire*, c'est-à-dire une fonction choisie uniformément au hasard parmi l'ensemble des fonctions $\{0,1\}^* \to \{0,1\}^n$. On l'appelle donc « un oracle aléatoire ».

C'est ce qu'on pourrait espérer de mieux comme fonction de hachage. Bien souvent, des raisonnements cryptographiques sont basés sur l'hypothèse que les fonctions de hachage habituelles sont des oracles aléatoires. Il est difficile de déterminer dans quelle mesure ceci est irréaliste. En 2009, il a été mis au jour que le protocole de backup auditable de § 3.1.6, qui est sûr dans le modèle de l'oracle aléatoire, n'est pas sûr du tout lorsque H est une des fonctions de hachages habituelles (MD5, SHA1, SHA256, ...).

Exercices

Exercice 3.1 : Une entreprise de spectacle gère une billeterie dématérialisée. Les billets sont expédiés par email, et les acheteurs présentent une version imprimée (ou leur smartphone) en venant au spectacle. Proposer une manière de réaliser le système. Quelles propriétés doit-il avoir ?

Exercice 3.2 : On peut construire un système de *mots de passes à usage unique* (« One-Time Passwords »). L'idée c'est qu'un mot de passe utilisé avec succès pour s'identifier devient automatiquement invalide, et ne peut pas servir à s'authentifier une nouvelle fois. Bien sûr, l'utilisateur pourrait envoyer sur un canal fiable un nouveau mot de passe après chaque authentification réussie, mais on peut faire un peu plus économique.

Un système possible fonctionne de la façon suivante. L'utilisateur fixe une graine de départ x_0. Il calcule $x_{i+1} = H(x_i)$, et stocke x_{1000} sur le serveur.

Expliquez ce qui doit se passer lorsque l'utilisateur tente de se connecter, et lorsqu'il y arrive. Quelle propriété doit avoir la fonction de hachage ?

▷ **Exercice 3.3 :** La société Amazon est un opérateur de cloud connu : on peut y louer de l'espace de stockage, de la puissance de calcul, de la bande passante, etc. Les utilisateurs peuvent administrer

leurs ressources grâce à une interface web, ou bien grâce à une API REST. Cela signifie en gros qu'on exécute des requêtes HTTP pour effectuer les actions d'administration. Voici un exemple d'appel à l'API :

```
DELETE /PAC/exam-2018.tex HTTP/1.1
User-Agent: dotnet
Host: s3.amazonaws.com
Date: Tue, 13 Mar 2018 21:20:27 +0000
Authorization: XXXXXXXXXXXXXXXXXXX
```

La requête (suppression de `/PAC/exam-2018.tex`) contient, dans l'en-tête HTTP `Authorization` une *preuve* que l'émetteur de la requête est bien le titulaire du compte. Ce dernier partage avec Amazon une clef symétrique K.

Expliquer comment pourrait fonctionner le mécanisme d'authentification. Que permet-il et qu'empêche-t-il à un adversaire actif ?

Exercice 3.4 : On s'intéresse à un algorithme de signature particulièrement simple proposé par Leslie Lamport (né en 1941, prix Turing en 2013). On veut signer des messages de n bits (par exemple, $n = 256$). Le schéma nécessite une fonction de hachage cryptographique H, comme SHA256. On génère une paire de clefs de la façon suivante :

1. Pour tout $0 \leq i < n$, générer deux chaines de n bits aléatoires $X_0[i]$ et $X_1[i]$.
2. Pour chaque i, calculer $Y_b[i] = H(X_b[i])$.

Les Y_i forment la clef de vérification, tandis que les $X_b[i]$ forment la clef de signature. On considère un message M de n bits, et on note $M[i]$ la valeur du i-ème bit. La signature de M est :

$$S = X_{M[0]} \parallel X_{M[1]} \parallel \ \ldots \ \parallel X_{M[n-1]}$$

Comment fait-on pour vérifier une signature ? Quelle est la complexité du processus ?

Exercice 3.5 : Pour prolonger l'exercice 3.4, pourquoi parle-t-on de signature « jetables » (*one-time signature* en anglais) ? Que se passerait-il si on produisait plusieurs signatures avec la même clef ?

Exercice 3.6 : Pour prolonger encore l'exercice 3.4, quelle(s) propriété(s) la fonction de hachage doit-elle posséder ?

Exercice 3.7 : Expliquez pourquoi si peut mettre en gage un seul bit, alors on peut aussi mettre en gage des messages arbitraires.

Exercice 3.8 : Montrer qu'un schéma de mise en gage est *information-theoretically binding* si et seulement s'il n'existe pas de paire r, r' telle que $\textsc{Commit}(r, 0) = \textsc{Commit}(r', 1)$.

Exercice 3.9 : Montrer qu'un schéma de mise en gage est *statistically hiding* si et seulement pour tout $y \in \{0,1\}^k$, il existe une de paire r, r' telle que $y = \textsc{Commit}(r, 0) = \textsc{Commit}(r', 1)$.

Exercice 3.10 : Expliquez comment réaliser un schéma de mise en gage avec une fonction de hachage. Quelles propriétés possède le schéma ? Et quelle(s) caracteristique(s) doit avoir la fonction de hachage ?

Deuxième partie

Cryptographie à clef secrète

Chapitre 4

Modes opératoires pour le chiffrement symétrique

4.1 Notion(s) de sécurité

On a vu qu'un mécanisme de chiffrement symétrique est contistué de deux algorithmes :

$$\mathcal{E} : \{0,1\}^k \times \{0,1\}^* \to \{0,1\}^*$$
$$\mathcal{D} : \{0,1\}^k \times \{0,1\}^* \to \{0,1\}^*$$

qui effectuent respectivement le chiffrement et le déchiffrement. Ici, k désigne la taille de la clef. Mais qu'est-ce qu'un mécanisme de chiffrement *de bonne qualité* ? Il doit être rapide, mais surtout il doit être *sûr*.

Nous sommes donc confrontés à la question de définir ce qu'est, précisément, la sécurité d'un mécanisme de chiffrement. Clairement, il ne faut pas qu'un adversaire puisse déchiffrer un message chiffré, ni qu'il puisse récupérer la clef secrète. On a vu qu'il y a des situations raisonnables où un adversaire peut apprendre des paires (clair, chiffré), et parfois même obtenir le (dé)chiffrement de messages de son choix.

Définir précisément ce qu'est un mécanisme sûr n'est pas complètement évident. En effet, un adversaire peut *toujours* récupérer la clef secrète... par la recherche exhaustive. Il faut donc limiter ses ressources, en particulier le temps qui lui est alloué. En effet, si on lui laisse la possibilité d'effectuer 2^k opérations, alors il peut faire la recherche exhaustive. On pourrait par exemple exiger qu'il soit retreint à une complexité polynomiale en k.

Et de toute façon, même si un adversaire ne peut pas entièrement déchiffrer un message, il peut éventuellement apprendre des choses sur son contenu : si, par exemple, il est capable de faire la différence entre le chiffrement d'un fichier `.mp3` et celui d'un fichier `.txt`, alors le mécanisme de chiffrement « laisse filtrer » des informations.

On en est donc venu, par approximation successives, à la définition suivante, la "*Left-Or-Right Security*" (LOR). En fait, il y a deux définitions : une où l'adversaire peut faire chiffrer ce qu'il veut ("*Chosen Plaintext Attack*" — CPA), et une ou il peut en plus faire déchiffrer ce qu'il veut ("*Chosen Ciphertext Attack*" — CCA).

On imagine une sorte de "jeu", dont le participant est un adversaire (c'est ne l'oublions jamais, une machine déterministe qui a à sa disposition une source de bits aléatoire). L'adversaire joue et gagne... ou perd. Le schéma est sûr si l'adversaire perd (presque) tout le temps. On s'intéresse à sa *probabilité de succès*.

4.1.1 Oracles

Pour jouer, l'adversaire a la possibilité d'interagir avec le mécanisme de chiffrement. Pour cela, il a accès à un « oracle », qui effectue pour lui, par exemple, le chiffrement.

D'après le dictionnaire de l'académie française, un oracle est la « réponse d'une divinité que l'on venait consulter en un lieu sacré, et dont un interprète inspiré [1] devait dévoiler le sens, sans parvenir toujours à l'éclairer ». Par extension, cela désigne aussi la divinité elle-même ou bien la personne s'exprimant en son nom.

Pour l'adversaire, un oracle est un mécanisme opaque et au fonctionnement inconnu, mais qui fournit les bonnes réponses. En particulier, l'adversaire ne peut pas examiner le fonctionnement interne de l'oracle. En gros, il accède à l'oracle via une API, ou bien via un web-service. L'oracle a une interface bien spécifiée.

4.1.2 LOR-CPA

L'adversaire a accès à un *oracle* "gauche-ou-droite", qui contient une clef symétrique (secrète) et un bit secret b. L'adversaire envoie deux messages M_0 et M_1 à l'oracle, qui lui renvoie le chiffrement de M_b.

En gros, l'oracle exécute le code suivant :

```
class LOR_CPA_Oracle:
    def __init__(self, E, K, b):
         self.K = K
         self.b = b
         self.E = E

    def left_or_right(self, M_0, M_1):
        if self.b == 0:
            return self.E.encrypt(self.K, M_0)
        else:
            return self.E.encrypt(self.K, M_1)
```

Le « jeu », pour l'adversaire consiste à deviner la valeur du bit secret b. Concrètement, cela signifie qu'à chaque fois qu'il donne deux messages, on lui renvoie le chiffrement de l'un des deux. Il doit réussir à réussir à « extraire » suffisament d'information du chiffré pour réussir à identifier le clair de départ.

Le jeu se passe donc de la façon suivante :

```
def experiment(E, k, b, adversary):
    K = RandomGenerator.getrandbits(k)
    oracle = LeftRightOracle(E, K, b)
    return adversary(oracle)
```

En gros, une fois qu'on a un adversaire sous la main, on initialise le jeu avec une certaine valeur du bit secret b, puis on exécute l'« expérience » : l'adversaire reçoit un accès à l'oracle [2] et renvoie son opinion sur la valeur de b. Il « gagne » s'il renvoie la bonne valeur.

On note que si jamais l'adversaire est capable de déchiffrer un message chiffré, complètement ou même partiellement, alors il peut facilement « gagner » à ce jeu.

4.1.3 Avantage

Il nous reste maintenant à « mesurer » l'efficacité de l'adversaire. Tout d'abord, on observe que s'il répond *au hasard*, l'adversaire est assuré de fournir la bonne réponse 50% du temps. Toute la question est de savoir dans quelle mesure il peut faire mieux que ça. On définit donc son *avantage* de la façon suivante :

$$\mathbf{Adv}^{LOR-CPA}(\mathcal{E}, \mathcal{A}, k) = \mathbb{P}\left[\texttt{experiment}(\mathcal{E}, k, 1, \mathcal{A}) = 1\right] - \mathbb{P}\left[\texttt{experiment}(\mathcal{E}, k, 0, \mathcal{A}) = 1\right]$$

La probabilité tient compte du choix aléatoire de la clef, ainsi que des choix aléatoires effectués par l'adversaire le cas échéant.

1. comprendre : en transe
2. En python, les objets ne peuvent pas avoir de variables « privées ». Par conséquent, dans le code donné en exemple, l'adversaire pourrait inspecter le contenu de l'oracle. On admet que ce n'est pas le cas ! Si l'oracle était implémenté via un web-service ce serait de toute façon impossible.

Si l'adversaire répond tout le temps au hasard, alors les deux probabilités sont égales à $1/2$, et leur différence vaut zéro. Au contraire, si l'adversaire donne toujours la bonne réponse, il a un avantage de 1. S'il arrive à calculer la bonne réponse dans 75% des cas, alors il aura un avantage de 0.5.

En fait, un adversaire qui a un avantage x donne la bonne réponse avec probabilité $(1 + x)/2$.

Au passage, si l'adversaire a tort plus souvent qu'il a raison, alors il a un avantage négatif. Mais dans ce cas-là, on peut imaginer fabriquer un nouvel adversaire qui répond systématiquement l'inverse du premier, et qui aurait, lui, un avantage positif...

Si l'adversaire s'amuse à essayer de faire une recherche exhaustive pendant un temps polynomial (disons qu'il teste $P(k)$ clefs), alors il a $P(k)$ chances sur 2^k de trouver la bonne clef. S'il trouve la bonne clef, il détermine la valeur de b, sinon il renvoie « b=0 ». Quel est son avantage ? Si $b = 1$, il va répondre $b = 1$ s'il parvient à trouver la clef, c'est-à-dire avec probabilité $P(k)/2^k$. Si $b = 0$, il ne répond jamais « 1 ». Son avantage est donc $P(k)/2^k$.... C'est-à-dire une fonction qui tend très vite vers zéro quand k augmente.

Un tel avantage est *négligeable* : il est dominé par l'inverse de n'importe quel polynôme.

4.1.4 Sécurité

La sécurité offerte par le schéma est fonction inverse de l'avantage du meilleur adversaire possible. On considère qu'un mécanisme de chiffrement est *sûr* si aucun adversaire qui fonctionne en temps polynomial peut atteindre un avantage non-négligeable. A contrario, on considère que n'importe quel adversaire dont l'avantage n'est pas négligeable parvient à « casser » le mécanisme de chiffrement.

On définit donc la fonction :

$$\mathbf{Adv}^{LOR-CPA}(\mathcal{E}, k, t, q) = \max_{\mathcal{A}} \mathbf{Adv}^{LOR-CPA}(\mathcal{E}, \mathcal{A}, k),$$

où le maximum est pris sur l'ensemble des adversaires qui s'exécutent en temps au plus t, et qui effectuent au plus q requêtes à l'oracle.

Le schéma $\mathcal{E}$ est « sûr » si la fonction $\mathbf{Adv}^{LOR-CPA}(\mathcal{E}, k, t, q)$ est négligeable lorsque t est une fonction polynomiale de k.

4.1.5 LOR-CCA

Intéressons-nous maintenant à la situation où l'adversaire peut obtenir le déchiffrement de messages de son choix. Il a donc toujours accès au même oracle « gauche-droite », mais en plus il a accès à un oracle de déchiffrement. Par contre... il n'a pas le droit de faire déchiffrer ce que lui renvoie l'oracle gauche-droite (ce serait trop facile sinon). On utilise donc l'oracle suivant :

```
class LOR_CCA_Oracle:
    def __init__(self, EncryptionScheme, K, b):
         self.K = K
         self.b = b
         self.log = []

    def left_or_right(self, M_0, M_1):
        if self.b == 0:
            result = E(self.K, M_0)
        else:
            result = E(self.K, M_1)
        self.log.append(result)
        return result

    def decrypt(self, C):
        if C in self.log:
             raise ValueError("This is forbidden")
        return D(self.K, C)
```

Les autres définitions sont exactement les mêmes.

4.2 Le masque jetable

On va voir qu'il existe un mécanisme de chiffrement *inconditionnellement sûr* : tous les adversaires ont un avantage de zéro, et ce quel que soit leur temps de calcul... mais à condition qu'ils ne fassent qu'une seule requête. Appelons ce mécanisme OTP (pour « One-time Pad », masque jetable en français).

L'idée est très simple. Le mécanisme chiffre des messages de n bits en utilisant des clefs qui font, elles aussi, n bits :

$$\mathcal{E}(K, M) = K \oplus M$$
$$\mathcal{D}(K, M) = K \oplus M$$

En gros, pour chiffrer, on XORe la clef sur le message (c'est pour cela qu'il faut qu'elle ait la même taille que ce dernier). Chaque clef ne peut être utilisée qu'une seule fois, et doit donc être jetée après usage (d'où le nom du système).

Pour comprendre pourquoi ce schéma est incassable (littéralement), imaginons qu'on chiffre des messages d'un seul bit. Supposons que j'examine le chiffré $C = 1$. Cela peut être le chiffré du message $M = 0$ (avec $K = 1$), mais cela pourrait tout autant être le chiffré de $M = 1$ (avec $K = 0$). On a aucun moyen de faire la différence entre ces deux situations !

Avec plus de 1 bit, si j'obtiens le chiffré "FOO", cela pourrait être le chiffrement de "PAC" avec $K =$ `0x160e0c`, ou bien le chiffrement de "CAR" avec $K =$ `0x050e1d`. Comme la clef est (en principe) choisie uniformément au hasard, ces deux situations sont tout aussi probables l'une que l'autre, et je n'ai aucun moyen de décider laquelle est la bonne.

Plus généralement, n'importe quelle chaine de n bits peut être le chiffré de n'importe quelle autre chaine de n bits. Ceci rend impossible toute forme de recherche exhaustive : comment savoir à quel moment on a le « bon » clair ? C'est pour ça que le système est *inconditionnellement sûr* : même avec une puissance de calcul illimitée, on ne peut pas en venir à bout.

Mettons-nous à la place d'un adversaire qui joue au jeu de la « Left-or-Right security ». On produit deux messages M_0 et M_1, et le système nous renvoie $Y = M_b \oplus K$. On peut s'amuser à déteminer les deux clefs potentiellement utilisées : ou bien $K = Y \oplus M_0$, ou bien $K = Y \oplus M_1$. Mais comme la clef K a été choisie au hasard, on ne sait pas laquelle de ces deux situation est la bonne. On ne sait même pas laquelle est la « plus probable » : elles ont exactement la même probabilité de se produire.

On a donc : $\mathbf{Adv}^{LOR-CPA}(\mathrm{OTP}, k, t, 1) = 0$ pour $k = n$ et pour toute valeur de t.

L'avantage du système est très clair : il est incassable. Les inconvénients, eux, sont aussi assez clairs : les clefs sont énormes et ne peuvent pas resservir. Le problème de leur distribution est donc particulièrement aigu. Le masque jetable était utilisé pour chiffrer le « téléphone rouge » entre la maison blanche et le Kremlin (c'était un telex, en fait, pas un téléphone). Les clefs étaient transportées en avion, dans des valises diplomatiques.

En dehors de ça, le système est très peu utilisé directement.

4.2.1 Optimalité

En fait, il n'est dans l'ensemble pas possible de faire mieux que le masque jetable tout en conservant la sécurité inconditionnelle.

En effet, on va voir que la sécurité inconditionnelle requiert des clefs au moins aussi longues que les messages. Si ce n'était pas le cas, on pourrait acquérir de l'information sur le message clair.

Pour cela, imaginons qu'on ait sous la main un chiffré de n bits, chiffré par un mécanisme de chiffrement symétrique (n'importe lequel) qui utilise des clefs de k bits, avec $k < n$. On effectue une recherche exhaustive sur la clef : pour chacune des 2^k clefs possibles, on déchiffre, et on obtient un clair « potentiel ». On peut atteindre de la sorte un sous-ensemble des clairs, de taille inférieure ou égale à 2^k. Il reste donc au moins $2^n - 2^k$ clairs « impossibles ».

On peut donc de la sorte exclure une partie des clairs possibles : on obtient donc un peu d'information sur le contenu du message de départ (on sait ce qu'il n'est pas).

4.2.2 Systèmes de chiffrement par flot

Dans la méthode du masque jetable, le « masque », c'est-à-dire la clef, doit être choisi uniformément au hasard parmi les chaines de bits de la taille du message.

On pourrait rendre le système plus pratique d'utilisation en générant le masque « à la volée » avec un *générateur pseudo-aléatoire* de bonne qualité.

Un « *Pseudo-Random Number Generator* » (PRNG) est un dispositif qui, à partir d'une *graine* (seed) de taille fixée (par exemple 128 bits) est capable de produire une séquence de bits « pseudo-aléatoires » arbitrairement longue.

Ces bits sont dits « pseudo-aléatoires » car ils sont produits par un mécanisme complètement déterministe (il n'y a pas de hasard !), et tout ce qu'on peut espérer de mieux c'est « qu'ils aient l'air aléatoire ».

Un PRNG est sûr s'il est impossible de faire la différence entre sa sortie (avec une graine aléatoire secrète) et une séquence de bits vraiment aléatoires. On peut définir un « jeu de distinguabilité » comme ci-dessus, une notion d'avantage, etc.

Ceci donne donc un moyen de chiffrer une quantité arbitraire de donnée avec une clef de taille fixe (de 128 bits par exemple) : pour chaque nouveau bit du clair, obtenir le prochain bit du flux pseudo-aléatoire et le XORer au bit du clair. Pour que la même clef puisse servir plusieurs fois, les PRNGs prennent généralement en argument un « vecteur d'initialisation » qui modifie le flux pseudo-aléatorie. Cet IV est inclu dans le chiffré et n'est pas secret.

L'un de ces systèmes, RC4, est assez largement utilisé, notamment dans le WEP. Il souffre cependant de quelques défauts qui rendent sa sortie distinguable de « vrais » bits aléatoires.

Bluetooth : protocole d'appariement

Les communications des objets connectés bluetooth sont chiffrées. Une fois que deux objets bluetooth sont appariés (« paired »), ils possèdent une clef secrète commune et peuvent communiquer de manière chiffrée. La partie intéressante est le protocole d'appariement, qui sert à établir cette clef partagée.

Ce protocole s'effectue par radio, mais un opérateur humain est censé saisir un code PIN secret de quelques chiffres sur les deux appareil, de manière sûre. Le protocole cherche donc à « amplifier » ce code PIN (secret très faible) en un secret fort. Les deux appareils négocient entre eux qui sera le Master (M) et le Slave (S).

Les périphériques possèdent une « addresse » ($\texttt{ADDR}_i$ pour le périphérique i) – c'est le numéro de série du périphérique bluetooth, un peu comme les adresses MAC des cartes ethernet. Chaque périphérique connaît l'adresse de l'autre.

Protocole B (*Appariement*). Le maître et l'esclaves choisissent des nonce aléatoire N_m et N_s (respectivement). Ils génèrent un masque jetable en chiffrant de l'aléa avec le code PIN. Ils s'en servent pour échanger leur nonce en les masquant.

B1. [Aléa.] $M \to S : N_r$.
M et S calculent $\text{MASK} \leftarrow \mathcal{E}_1(\text{PIN}, N_r)$.

B2. [N_s masqué.] $S \to M : N_s \oplus \text{MASK}$

B3. [N_s masqué.] $M \to S : N_m \oplus \text{MASK}$

Une fois les nonces echangés, ils peuvent calculer chacun de leur côté la clef partagée :

$$K \leftarrow \mathcal{E}_2\left(N_m, \texttt{ADDR}_m\right) \oplus \mathcal{E}_2\left(N_s, \texttt{ADDR}_s\right)$$

4.3 Chiffrement par bloc

Les chiffrement par flots sont utilisés dans certains contextes (WEP, Bluetooth), car ils peuvent être implémentés efficacement sur des circuits matériels. Mais en logiciel, c'est rarement la méthode

la plus répandue en pratique pour chiffrer des données de taille arbitraire avec une clef secrète de taille fixée (disons 128 bits).

Un *système de chiffrement par bloc* (« Block Cipher ») est un mécanisme qui chiffre des « blocs » de n bits avec une clef de k bits. Par exemple, le plus utilisé, l'AES, chiffre des blocs de $n = 128$ bits avec des clefs de $k = 128$ ou $k = 256$ bits.

Il s'agit donc d'opérer sur une entrée de taille fixe. On verra dans la section suivante comment utiliser un Block Cipher pour chiffrer des données de taille arbitraire.

Un Block Cipher, une fois la clef fixée, est une fonction qui transforme une chaine de n bits en une autre chaine de n bits. Il s'agit donc d'une *permutation* de l'ensemble $\{0,1\}^n$. En effet, si on prend tous les blocs de n bits dans l'ordre, et qu'on les chiffre tous, on obtient de nouveau l'ensemble des blocs de n bits, mais dans un autre ordre.

Chaque fois qu'on change la clef, on change la permutation. Un Block Cipher, mathématiquement, est donc une famille de permutations de $\{0,1\}^n$ (la famille est indexée par la clef).

4.3.1 Permutation aléatoire

Pour chiffrer des blocs de n bits, rien de tel qu'une permutation complètement aléatoire. Une telle permutation est tirée au hasard, uniformément, parmi l'ensemble des $2^n!$ permutations possibles (pour $n = 128$, cela fait à peu près $2^{2^{134}}$). Le bout de code suivant simule une permutation aléatoire. A chaque nouvelle question, une réponse uniformément aléatoire est choisie.

```
class RandomPermutation:
     def __init__(self, n):
        self.n = n
        self.forward = {}
        self.backward = {}

    def _f(self, direct, reverse, x):
        if x in direct:
            return direct[x]
        y = RandomGenerator.getrandbits(self.n)
        while y in reverse:
            y = RandomGenerator.getrandbits(self.n)
        direct[x] = y
        reverse[y] = x
        return y

    def E(self, x):
        return _f(self.forward, self.backward, x)

    def D(self, y):
        return _f(self.backward, self.forward, y)
```

L'avantage d'une permutation aléatoire, c'est qu'il n'y a pas de lien entre les entrées et les sorties, puisque les sorties sont aléatoires. En particulier, la sortie ne révèle pas d'information sur l'entrée.

En effet, si on vous dit $E(x) =$ `0x6af31de3986787e57134ad92e120f53d`, alors vous savez que cette dernière valeur a été générée aléatoirement, donc vous n'avez littéralement aucune information sur x.

4.3.2 Notion de sécurité

En gros, un Block Cipher est sûr s'il ressemble à une permutation aléatoire. En effet, on ne peut pas « attaquer » une permutation aléatoire, même avec une puissance de calcul illimitée. Si un adversaire même puissant ne peut pas distinguer notre Block Cipher d'une telle permutation aléatoire, c'est que ce dernier est sûr.

Étant donné un Block Cipher, l'ensemble des permutations qu'il peut produire représente donc une minuscule fraction de l'ensemble de celles qui seraient possible (2^k contre $2^n!$, soit 2^{128} contre $2^{2^{134}}$

avec $k = n = 128$). Mais on souhaite qu'il soit impossible de distinguer ce petit sous-ensemble du reste. Pour cela, on définit comme précédemment un jeu avec un bit secret b, où un adversaire a accès à un oracle. Si $b = 0$, cet oracle est le Block Cipher avec une clef aléatoire secrète. Si $b = 1$, l'oracle est une permutation aléatoire. Le but de l'adversaire est de deviner b.

Formellement, le jeu est défini comme ça :

```
class BC_Oracle:
    def __init__(self, BC, K):
        self.BC = BC
        self.K = K

    def E(self, x):
        return self.BC.encrypt(self.K, x)

    def D(self, y):
        return self.BC.decrypt(self.K, y)

def experiment(BC, b, adversary):
    if b = 0:
        K = RandomGenerator.getrandbits(k)
        oracle = BC_Oracle(BC, K)
    else:
        oracle = RandomPermutation(BC.blocksize)
    return adversary(oracle)
```

L'avantage de l'adversaire est :

$$\mathbf{Adv}^{IND-CCA}(\mathcal{BC}, \mathcal{A}) = \mathbb{P}\left[\texttt{experiment}(\mathcal{BC}, 1, \mathcal{A}) = 1\right] - \mathbb{P}\left[\texttt{experiment}(\mathcal{BC}, 0, \mathcal{A}) = 1\right]$$

Une différence sensible avec les « mécanismes de chiffrement » considérés au-dessus est que ces derniers peuvent avoir des clefs de taille variable, et que la sécurité était définie *asymptotiquement* en fonction de la taille de la clef. Un Block Cipher a une clef de taille fixe, donc la même définition n'est pas possible.

Un adversaire qui effectue une recherche exhaustive pendant un temps t peut trouver la bonne clef avec probabilité $t/2^k$. On considère qu'il « casse » le Block Cipher s'il obtient un avantage significativement plus grand. Seulement il est difficile de quantifier cette dernière notion. C'est tout le problème avec les mécanismes cryptographiques *concrets* : on ne peut pas parler, par exemple, d'adversaire fonctionnant en temps polynomial (polynomial *en quoi* ?).

4.3.3 En pratique

Sur le plan pratique, les Block Cipher comptent parmi les mécanismes cryptographiques les mieux connus et les plus étudiés. Il en existe depuis longtemps, qui ont résisté pendant longtemps à la cryptanalyse, et on comprend relativement bien les principes à appliquer pour en produire des sûrs.

Le DES d'abord, puis l'AES maintenant (en 2019) sont des exemples de Block Cipher très largement utilisés, et dont la sécurité est reconnue. Enfin, le DES avait des clefs trop petites, et il a été un peu mis à mal par l'invention des cryptanalyses différentielles et linéaires. Enfin, la meilleure attaque connue contre le DES nécessite la connaissance de 70 tera-octets de données chiffrées et des clairs correspondants, ainsi qu'environ 2^{40} opérations de calcul. Elle est donc assez peu réaliste, contrairement à la recherche exhaustive qui nécessite, elle 2^{56} opérations de calcul, mais très peu de données.

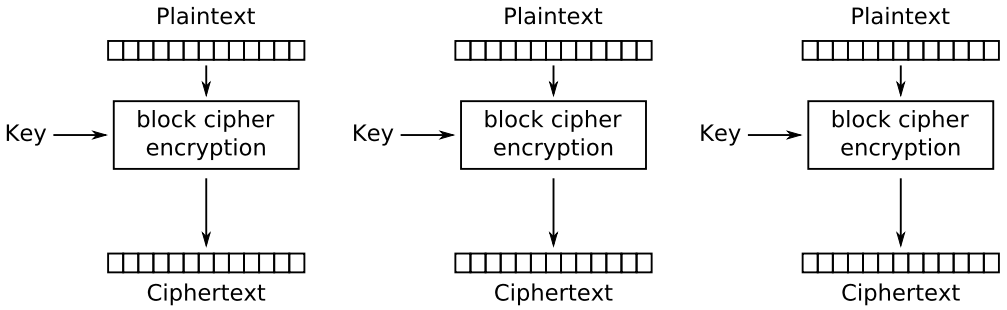
Aucune attaque sérieusement plus rapide que la recherche exhaustive n'est actuellement connue sur l'AES (en 2019).

4.4 Modes opératoires pour le chiffrement par bloc

Armés d'un Block Cipher, il nous reste maintenant à être capable de chiffrer des chaines de bits arbitrairement longues. Pour cela, on utilise le Block Cipher de manière répétée selon un *mode*

opératoire spécifié. Ces modes opératoires ont reçu des noms plus ou moins barbares.

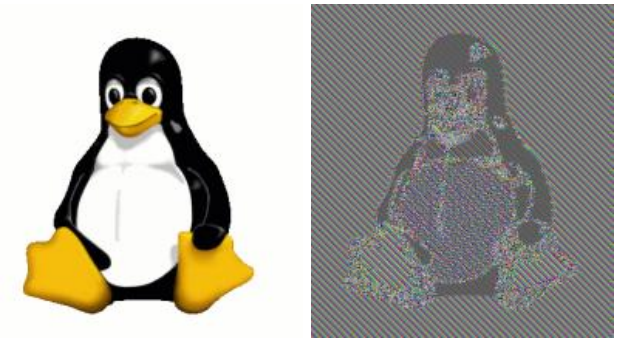
ECB. La solution la plus simple consiste à découper le message clair en blocs de n bits, et à chiffrer tous les blocs séparément, en parallèle.



Electronic Codebook (ECB) mode encryption

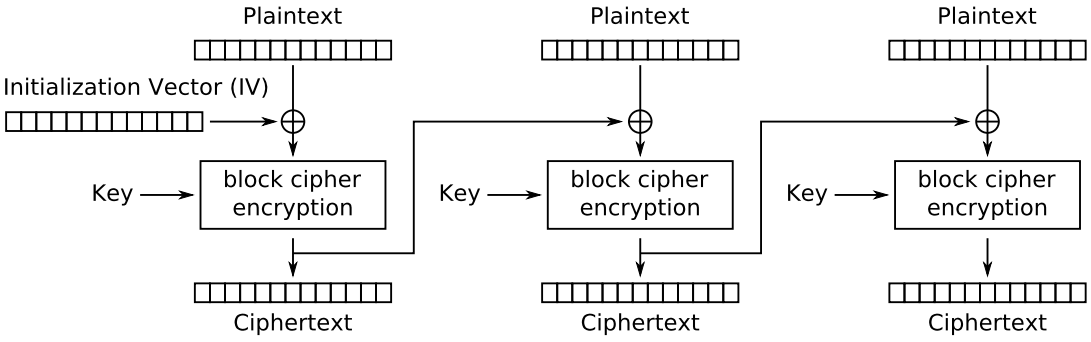
$$Y_i = E(K, X_i)$$

L'avantage est que c'est très simple, et que c'est parallelisable. L'inconvénient, c'est que ce n'est pas très sûr. En effet, la même séquence d'octets va être chiffrée de la même façon, quelle que soit sa position dans le message. Ceci permet à des informations de « filtrer » à travers le chiffrement, comme illustré ci-dessous. L'image de droite a été obtenue en chiffrant l'image de gauche avec le mode ECB.



Le chiffrement ne dissimule même pas l'identité du personnage représenté ! ECB signifie « Electronic Code Book ».

CBC. Pour éviter ce problème, il faut que le chiffrement d'un bloc du clair dépende de sa position dans le message. On peut en faire obtenir encore mieux que cela avec le « Cipher Block Chaining » :



Cipher Block Chaining (CBC) mode encryption

$$Y_{-1} = IV$$
$$Y_i = E(K, X_i \oplus Y_{i-1})$$

L'idée, c'est que le chiffrement du i-ème bloc dépend de tous les blocs précédents (et de la clef bien sûr). Afin que le chiffrement du premier bloc dépende lui aussi de quelque chose, on ajoute un « Vecteur d'Initialisation » (IV).

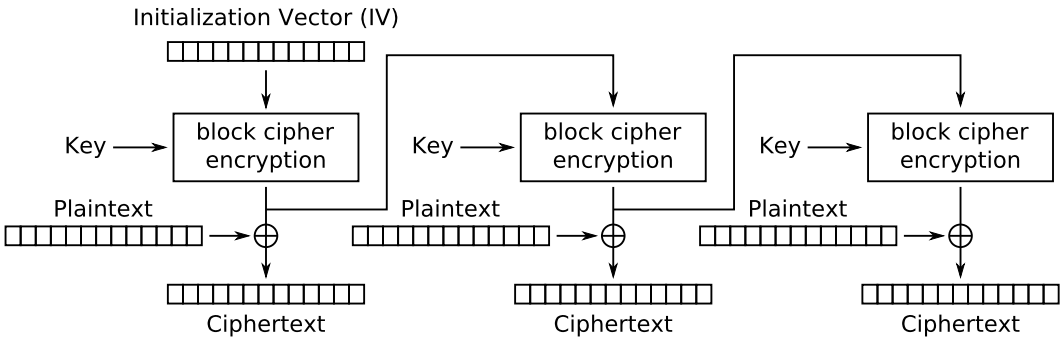
Pour déchiffrer, on calcule :

$$X_i = D(K, Y_i) \oplus Y_{i-1}.$$

Pour déchiffrer le premier bloc, on a besoin de connaître IV. Il fait donc l'inclure dans le chiffré, qui a donc nécessairement un bloc de plus que le clair. L'IV peut être choisi un peu n'importe comment. On peut par exemple le choisir au hasard, ou utiliser l'heure qu'il est, etc. Mais il est important de ne pas utiliser deux fois le même. La possibilité de choisir l'IV a l'avantage que ça permet de rendre le chiffrement non-déterministe : on peut chiffrer le même message de plein de manières différentes. Par exemple, si on fait du chiffrement de disque dur, on peut chiffrer chaque secteur en utilisant son numéro comme IV. De la sorte, si des données sont présentes plusieurs fois sur le disque, ceci n'est pas visible dans les données chiffrées.

Au passage, autant le chiffrement ne peut pas se faire en parallèle, autant le déchiffrement, lui, peut. De plus, le mécanisme est « auto-synchronisant » : si une partie du chiffré est perdue, on peut reconstituer les parties du clair autour du « trou » dans le chiffré.

OFB. Une autre idée naturelle consiste à transformer un Block Cipher en PRNG. En effet, si notre Block Cipher est indistinguable d'une permutation aléatoire, alors $E(K, x)$ est essentiellement indistinguable d'une chaine de bits aléatoires. C'est l'idée du mode « Output Feed-Back » :



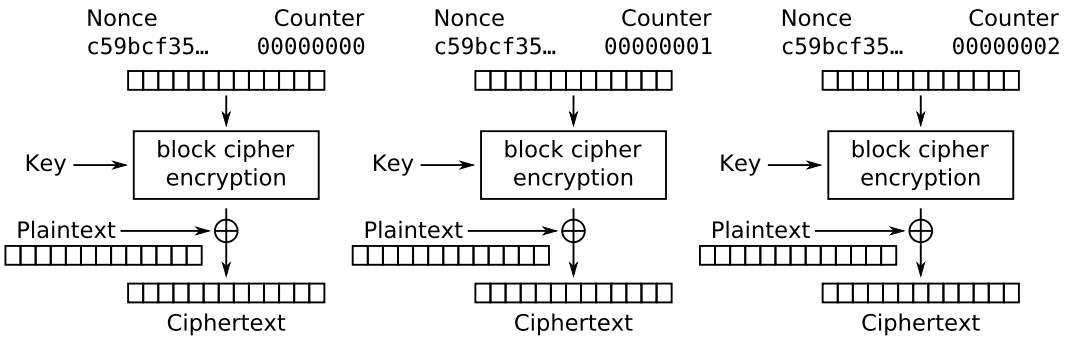
Output Feedback (OFB) mode encryption

$$Z_{-1} = IV$$
$$Z_i = E(K, Z_{i-1})$$
$$Y_i = X_i \oplus Z_i$$

On se sert du Block Cipher pour générer un masque pseudo-aléatoire, on le XORe sur le message à chiffrer, et voilà. La raison (informelle) pour laquelle ceci est sûr est simple : si le Block Cipher est indistinguable d'une permutation aléatoire, alors la suite $(Z_i)_{i \geq 0}$ est indistinguable d'une séquence de bits vraiment aléatoires. Donc, on est dans la situation du masque jetable.

L'inconvénient, c'est que ce n'est pas parallelisable.

CTR. On peut un peu améliorer l'idée précédente, notamment pour la rendre parallelisable. C'est le mode « CounTeR » :



Counter (CTR) mode encryption

$$Z_i = E(K, IV \| i)$$
$$Y_i = X_i \oplus Z_i$$

Ici, le masque pseudo-aléatoire est obtenu en chiffrant la séquence $(IV\|0), (IV\|1), (IV\|2), \ldots$ Cette séquence est sans répétition tant que le « compteur » ne déborde pas. Par exemple, on peut imaginer pour des blocs de 128 bits, un IV sur 96 bits et un compteur sur 32 bits : on peut chiffrer 64 giga-octets avec le même IV.

4.4.1 Schéma de bourrage

Le mode CBC, qui est largement utilisé en pratique, nécessite que la taille du message soit un multiple de la taille du bloc. Que faire si le message est trop court ? Il suffit de rajouter des bits ou des octets à la fin, de telle sorte qu'il atteigne la bonne taille.

Pour cela, on emploie un « mécanisme de bourrage » (« *padding scheme* »). Par exemple, on peut rajouter des bits à zéro. L'inconvénient, c'est que M et $M\|0$ peut alors avoir le même chiffrement, et on ne peut plus les déchiffrer de manière non-ambigüe.

Pour éviter ce problème, on peut ajouter un bit 1, puis autant de bits 0 que nécessaire. Si la taille du message de départ est exactement un multiple de la taille du bloc, on rajoute un bloc entier de bourrage (sinon on pourrait confondre la fin du message avec du bourrage).

D'autres solutions ont été proposées. L'une d'entre elles est simple : s'il faut rajouter k octets au message, alors on ajoute des octets représentant le nombre k (on rajoute aussi un bloc complet si nécessaire).

Quand on utilise un schéma de bourrage, on peut éventuellement se rendre compte que le déchiffrement a échoué : en effet, si on ne trouve pas, dans le dernier bloc, le schéma de bourrage bien en place, alors c'est que le déchiffrement s'est mal passé. Signaler « le déchiffrement a échoué car le bourrage n'était pas en place » à l'utilisateur n'est pas une bonne idée. Cela conduit à des attaques parfois subtiles.

4.4.2 Preuve de sécurité du mode compteur

Il est particulièrement simple de justifier que ce mode opératoire donne un mécanisme de chiffrement à clef secrète qui possède la « Left-or-Right security » tant que le Block Cipher utilisé est sûr.

Supposons dans un premier temps que le Block Cipher soit une permutation aléatoire. Dans ce cas, la séquence des Z_i est une séquence de bits vraiment aléatoire, tant que le compteur ne déborde pas (après, elle devient périodique). Par conséquent, ceci réalise *exactement* la méthode du masque jetable. Chaque IV différent aboutit à la production d'un masque différent. Par conséquent, tant qu'un IV n'est utilisé qu'une seule fois, l'avantage de n'importe quel adversaire est nul.

Si, à la place d'une « vraie » permutation aléatoire on utilise un Block Cipher, alors les choses sont différentes. En effet, comme on l'a vu au-dessus, l'adversaire choisit deux messages M_0, M_1 et reçoit

$Y = M_b \oplus Z$, où Z est le masque utilisé. Si les messages font ℓ blocs, on a :

$$Z = \mathcal{E}(K, IV\|0) \parallel \mathcal{E}(K, IV\|1) \parallel \ \ldots \ \parallel \mathcal{E}(K, IV\|\ell)$$

L'adversaire pourrait, par exemple, tenter une recherche exhaustive sur la clef K (il peut car il connait IV). On voit donc qu'il n'y a plus de sécurité inconditionnelle : au bout de suffisament de temps, un adversaire finirait par gagner.

L'idée générale consiste à dire que : « si un adversaire arrive à identifier le message à travers le masque, alors c'est que le masque n'est pas vraiment une suite de bits aléatoires ». En gros, on va faire une *réduction*.

Par conséquent, étant donné un adversaire $\mathcal{A}$ qui a un avantage non-négligeable au jeu de la « Left-or-Rigth security », on va construire un adversaire $\mathcal{B}$ qui distingue le Block Cipher d'une permutation aléatoire avec un avantage élevé. Si le Block Cipher est sûr, un tel adversaire $\mathcal{B}$ ne devrait pas exister. Comme on a construit $\mathcal{B}$ à partir de $\mathcal{A}$, c'est qu'on a eu tort de supposer que $\mathcal{A}$ existe. Par conséquent, il n'y a pas d'algorithme efficace qui casse la « Left-or-Right security » du mode CTR instancié avec le Block Cipher.

Voici le détail de la construction de $\mathcal{B}$ à partir de $\mathcal{A}$:

1. Choisir un bit aléatoire b'
2. Construire un oracle « gauche-droite » $\mathcal{O}$ qui renvoie toujours « gauche » si $b' = 0$ et « droite » si $b' = 1$. Il effectue le chiffrement en exécutant le mode compteur, et calcule le chiffrement de chaque bloc avec son propre oracle (« Block Cipher-ou-permutation-aléatoire »).
3. Appelle l'adversaire fourni, $\mathcal{A}$, en lui fournissant l'oracle $\mathcal{O}$.
4. Si $\mathcal{A}$ devine correctement la valeur de b', on répond « Block Cipher » ($b = 0$), sinon on répond « permutation aléatoire » ($b = 1$).

Pour calculer l'avantage de $\mathcal{B}$, examinons d'abord ce qui se passe si la permutation vraiment aléatoire lui est fournie.

On a vu au-dessus que dans ce cas-là, $\mathcal{A}$ reçoit des messages chiffrés avec le One-Time Pad, et donc l'avantage de $\mathcal{A}$ est zéro. Il va donc identifier b' avec probabilité $1/2$. Dans ce cas-là, $\mathcal{B}$ va donner la réponse « permutation aléatoire » ($b = 1$) avec probabilité $p_1 = 1/2$.

Lorsque le Block Cipher est fourni, $\mathcal{A}$ va identifier b' avec probabilité :

$$q = \frac{1}{2} \cdot \left(1 + \mathbf{Adv}^{LOR-CPA}(\mathrm{CTR}, \mathcal{A}, k)\right).$$

On va donc donner la réponse $b = 1$ lorsque b' n'a *pas* été identifié, donc avec probabilité $p_0 = 1 - q$.

L'avantage de $\mathcal{B}$ est donc :

$$\begin{aligned}
\mathbf{Adv}^{IND-CCA}(\text{Block Cipher}, \mathcal{B}) &= p_1 - p_0 \\
&= \frac{1}{2} - \left[1 - \frac{1}{2} \cdot \left(1 + \mathbf{Adv}^{LOR-CPA}(\mathrm{CTR}, \mathcal{A}, k)\right)\right] \\
&= \frac{1}{2} \cdot \mathbf{Adv}^{LOR-CPA}(\mathrm{CTR}, \mathcal{A}, k)
\end{aligned}$$

Par conséquent, si $\mathcal{A}$ a un avantage non-négligeable, alors on a une probabilité non-négligeable de distinguer le Block Cipher d'une permutation aléatoire en temps raisonnable.

Si le Block Cipher est sûr, alors c'est qu'un tel adversaire contre le mécanisme de chiffrement n'existe pas.

Chapitre 5

Modes opératoires pour les fonctions de hachage

On a vu au chapitre 3 qu'une « bonne « fonction de hachage cryptographique doit avoir tout un paquet de propriétés... et que dans le fond on aimerait que ce soit un oracle alétoire, c'est-à-dire quelque chose qu'une fonction déterministe ne peut pas être, par définition.

Du coup, on se donne de manière plus concrète des objectifs « raisonnables ».

5.1 Propriétés de base

5.1.1 Résistance aux préimages

Définition : One-Way Function

> Une fonction $f : A \to B$ est une une « One-Way Function » (OWF) (à **sens unique**, ou **résistante aux préimages**) si :
>
> *i)* Pour tout $x \in A$, calculer $f(x)$ est facile.
>
> *ii)* Pour tous les $y \in B$ (sauf éventuellement une fraction négligeable), trouver un x tel que $f(x) = y$ est calculatoirement impossible.
>
> Le y ci-dessus une une **préimage** de x (ou un antécédent).

Il existe un algorithme simple pour essayer de trouver une préimage : la recherche exhaustive. Un adversaire pourrait essayer de calculer les empreintes de chaines de bits aléatoires jusqu'à ce qu'il obtienne une préimage de l'empreinte voulue. A chaque essai, il va avoir grosso-modo une chance sur 2^n de réussir. Il lui faudra donc, en moyenne, 2^n essais (ceci utilise implicitement l'espérance d'une loi de probabilité géométrique). Ceci est analogue à l'exercice 1.8.

Par conséquent, pour qu'une fonction de hachage cryptographique soit à sens unique en 2019, il faut qu'elle produise des empreintes d'au moins $n \geq 128$ bits. Par comparaison, les fonctions de hachage utilisées pour les tables de hachage ou les bases de données ont une sortie de 32 ou 64 bits.

L'humanité ne connaît pas à ce jour de fonctions qui sont *prouvablement* à sens unique. Il y a beaucoup de fonctions qu'on ne sait pas inverser efficacement (par exemple, en temps polynomial en n), et il y en a même où l'on sait que réussir à les inverser est au moins ausssi dur qu'un problème mathématique difficile (tel que la factorisation des grands entiers). Mais on a pas de *preuve* que résoudre le problème en question est intrinsèquement dur.

5.1.2 Résistance aux secondes préimages

Définition : One-Way Hash Function

> Une fonction de hachage $f : \{0,1\}^* \to \{0,1\}^n$ est une une « One-Way Hash Function » (OWHF) si :

> *i*) Elle est à sens unique.
>
> *ii*) Étant donné $x \in A$ trouver un $x' \neq x$ tel que $f(x) = f(x')$ est calculatoirement impossible.
>
> Le x' ci-dessus une une **seconde préimage** de x.

La résistance aux secondes préimages implique la résistance aux préimages. Par contre, la résistance aux préimages n'implique *pas* la résistance aux secondes préimages. Ces deux affirmations sont l'objet des exercices 5.2–5.3.

Résistance aux collisions Définition : Collision-Resistant Hash Function

> Une fonction H est une **Collision-Resistant Hash Function** (CRHF) s'il est calculatoirement impossible de produire $x \neq x'$ tels que $H(x) = H(x')$.

La résistance aux collisions implique la résistance aux secondes préimages (c.a.d. qu'une CRHF est en particulier une OWHF). Cf. exercice 5.4. La résistance aux collisions est l'objectif le plus difficile à atteindre en pratique pour les concepteurs de fonctions de hachage. Il n'a, à ce jour, jamais été possible de calculer des (secondes-)préimages sur une fonction de hachage cryptographique largement déployée. Par contre, des collisions ont été trouvées dans MD5, SHA0, SHA1, ... Il existe des fonctions de hachage qui sont *prouvablement* résistantes aux collisions, sous l'hypothèse qu'un problème calculatoire bien identifié est difficile, mais elles ne sont pas utilisées en pratique car elles sont trop lentes.

Les empreintes produites par une CRHF identifient de manière non-ambigue leur entrée (en effet, si deux entrées produisaient la même empreinte ce serait... une collision). Pour cette raison, des systèmes tels que `git` utilisent l'empreinte d'un fichier pour l'identifier.

5.2 Le paradoxe des anniversaires

Comme on a fait pour la résistance aux préimages, considérons un adversaire qui essaye de trouver des collisions par « force brute ». Par exemple, il pourrait calculer les empreintes de messages aléatoires tous différents jusqu'à ce qu'il obtienne une collision pareils. Combien de temps cela va-t-il lui prendre ?

Raisonnement intuitif mais faux. Considérons deux messages aléatoires x et y. La probabilité que $H(x) = H(y)$ est de $1/2^n$ (si les sorties de H sont uniformément distribuées, mais si ce n'est pas le cas il faut mettre H à la poubelle tout de suite). Maintenant, si on a sous la main non pas deux, mais trois messages x, y et z, on peut obtenir une collision si $H(x) = H(y)$ ou bien si $H(x) = H(z)$, ou bien si $H(y) = H(z)$. On a trois fois plus de chance !

Si on possède les empreintes de k messages, alors on peut former $k(k-1)/2$ *couples* de messages. Chaque couple nous donne une chance sur 2^n de gagner, donc avec nos k messages, on peut espérer trouver une collision avec probabilité $\approx k^2/2^n$.

Cela signifie que si on veut trouver une collision avec probabilité 1, il suffit de prendre $k \approx \sqrt{2^n}$ messages, c'est-à-dire $2^{n/2}$. Pour un adversaire, trouver une collision par force brute est donc bien plus facile que de trouver une préimage par force brute. Si $n = 128$, alors trouver une collision demandera 2^{64} essais, ce qui est beaucoup mais n'est pas loin de faire partie du domaine du possible.

Versions rigoureuses. En fait, le raisonnement ci-dessus, même s'il donne à peu près le bon résultat, n'est pas correct mathématiquement. Le problème c'est que si on sait que $H(x) \neq H(y)$ et que $H(z) \neq H(y)$, alors la probabilité que $H(x) = H(z)$ n'est pas $1/2^n$ mais $1/(2^n - 1)$ — en effet il y a une valeur « exclue », qui est précisément $H(y)$.

Voici donc une version rigoureuse :

Théorème 1. *Supposons que n balles soient lancées uniformément et indépendamment au hasard dans m paniers. Notons $\mathcal{C}$ l'événement « au moins un panier contient au moins deux balles ». Alors :*

$$1 - e^{-\frac{n(n-1)}{2m}} \leq \mathbb{P}[\mathcal{C}] \leq \frac{n(n-1)}{2m}$$

Ce résultat a pris le nom de « paradoxe des anniversaires », bien que ce ne soit pas du tout un paradoxe. Ce qui est surprenant, c'est que la probabilité que deux étudiants parmis les 35 qui suivent PAC aient leur anniversaire le même jour de l'année est supérieure à 80% (on s'attendrait à moins).

5.2.1 Applications

Fonctions de hachage. Comment utiliser ceci ? Si on imagine que chaque nouvelle empreinte est une chaine de n bits aléatoire, alors c'est comme si on lançait une « balle » au hasard parmi 2^n « paniers ». Si, par l'effet du hasard, on génère deux fois la même chaine de bits, c'est qu'on a « lancé » deux balles dans le même panier.

Avec $2^{n/2}$ messages aléatoires, la probabilité d'obtenir une collision est comprise entre 36% et 50% d'après le théorème ci-dessus. Avec $4 \times 2^{n/2}$ messages aléatoires, elle est supérieure à 99.99%.

Cela signifie que trouver des collisions, même sur une fonction de hachage *idéale* (un oracle aléatoire), est bien plus rapide que de trouver des préimages, puisqu'on s'en tire en $2^{n/2}$ essais au lieu de 2^n. Si on veut que trouver une collision soit aussi dur que trouver une clef de l'AES par recherche exhaustive, alors il faut choisir des fonctions de hachage avec $n = 256$ bits.

PRNG à partir de Block Cipher. Le chiffrement en mode compteur (cf. § 4.4) revient à fabriquer un PRNG à partir d'un Block Cipher, puis à se servir du flux pseudo-aléatoire comme d'un masque jetable (c'est la technique des *stream ciphers*, cf. § 4.2.2). Quelle est la qualité du PRNG obtenu de la sorte ? Pour être « de qualité cryptographique », il faudrait qu'il soit impossible de distinguer sa sortie de bits « vraiment » aléatoires.

Ici, le paradoxe des anniversaires peut être utilisé comme *distingueur*. En effet, tant que le compteur i ne « déborde » pas, alors les blocs x_i sont tous différents. Si ces blocs font n bits de long, alors le paradoxe des anniversaires garantit qu'après en avoir observé $4 \cdot 2^{n/2}$, alors on devrait avoir vu une collision. Si $n = 64$, par exemple (avec le DES), il suffit d'environ 2^{34} blocs de 64 bits, soit 128Go de bits pseudo-aléatoires, pour être quasiment certain de distinguer la sortie du DES en mode compteur d'un flot de bits « vraiment » aléatoires.

5.3 Mode opératoire de Merkle-Damgård

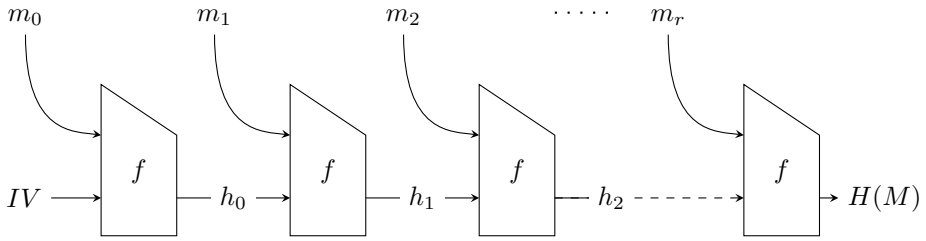
Pour hacher des chaines de bits arbitrairement longues, on peut utiliser une technique comparable à celle employée pour le chiffrement de longs messages vue au chapitre 4. On va traiter le message en entrée progressivement, morceau par morceau. La technique décrite ici est le mode opératoire de *Merkle-Damgård* du nom de ceux qui l'ont décrit précisément en 1990, et qui ont démontré certaines de ses propriétés. Il a été largement utilisé ensuite, dans MD5, SHA-1/2... Mais à cause d'un certain nombre de ses problèmes, SHA-3 est construit différement.

5.3.1 Description

L'idée consiste à utiliser de manière itérative une *fonction de compression*

$$f : \{0,1\}^{n+m} \longrightarrow \{0,1\}^n.$$

La fonction de compression est une « petite » fonction de hachage, qui « hache » $n+m$ bits en n bits. Elle possède généralement deux entrées qui ont des rôles distincts : le *bloc de message* et la *valeur de chainage*. Un petit dessin est plus explicatif :



Le message M donc on veur une empreinte est découpé en blocs de m bits, et ces blocs sont hachés successivements. La première valeur de chainage, l'IV, est une constante fixé par la spécification des fonctions de hachages concrètes. Un *padding* est appliqué sur le dernier bloc (ceci permet de hacher des messages dont la taille n'est pas un multiple de m bits). Ce *padding* DOIT contenir la *taille du message* (avant ajout du *padding*, bien sûr).

Pour donner une idée, dans SHA-256 (resp. SHA-512), les blocs font $m = 512$ bits (resp. $m = 1024$), et taille des valeurs de chainage est de 256 bits (resp. 512). Dans le dernier bloc, la taille du message est donnée en bits, et est codée sur 128 bits. Ceci impose une limite (absurdement élévée) sur la taille des messages qui peuvent être hachés.

5.3.2 Résistance aux préimages et aux collisions

L'intérêt principal de ce mode opératoire c'est qu'il possède des formes de « sécurité prouvée ». Plus exactement, on sait que si la fonction de compression n'a pas trop de problèmes, alors l'ensemble sera raisonnablement sûr.

Tout d'abord, il est facile de voir que si on est capable de trouver des préimages sur H, alors on est capable d'en trouver sur f.

Plus précisément, on sait que :

Théorème 2 (Merkle-Damgård, 1990)**.** *Si la fonction de compression f est résistante aux collisions, alors la fonction de hachage H^f construite à partir de f via le mode opératoire de Merkle-Damgård est résistante aux collisions.*

Démonstration

La preuve est une réduction qui transforme une collision sur H^f en collision sur f. Autrement dit, à partir d'une paire de message $M \neq M'$ telle que $H^f(M) = H^f(M')$, on produit deux messages $x \neq x'$ de $n+m$ bits tels que $f(x) = f(x')$. Si f est résistante aux collisions, trouver ces deux messages est calculatoireement impossible, donc par conséquent trouver M et M' est calculatoirement impossible aussi.

Supposons donc que nous ayons deux messages $M \neq M'$ tels que $H^f(M) = H^f(M')$, et tentons de produire une collision sur f. De deux choses l'une :

- Ou bien $|M| \neq |M'|$ (cette notation désigne la taille des messages). Dans ce cas, vu la définition du mécanisme de bourrage dans le mode opératoire de Merkle-Damgård, les blocs « bourrés » qui vont entrer dans la fonction de compression lors de sa dernière invocation seront différents lors du traitement de M et de M'. De plus, comme $H^f(M) = H^f(M')$, on sait que les sorties de la fonction de compression sont identiques dans les deux cas : c'est donc qu'il y a une collision sur f, qui a lieu lors du traitement du dernier bloc des deux messages.
- Ou bien $|M| = |M'|$. Supposons que les deux messages fassent r blocs, après bourrage. Il faut encore distinguer deux situations : soit $(h_{r-1}, m_r) \neq (h'_{r-1}, m'_r)$, et il y a alors une collisions sur f puisque $h_r = h'_r$. Ou bien $(h_{r-1}, m_r) = (h'_{r-1}, m'_r)$. Dans ce deuxième cas, l'argument se repète « un bloc avant ». Ou bien on trouve une collision sur f à un moment donné, or bien nous devons conclure que $m_i = m'_i$ pour tout i, ce qui est impossible vu que les messages sont différents.

L'avantage principal de ce mode opératoire est que pour produire une bonne fonction de hachage, il suffit de produire une bonne fonction de compression. En plus, il est rapide, et permet de produire les empreintes « à la volée », sans avoir à stocker l'ensemble du message.

On a longtemps cru que si la fonction de compression résistait aux secondes préimages, alors la fonction itérée y résistait aussi. C'est en 2006 qu'on s'est rendu compte que ce n'était pas vrai. Kelsey et Schneier ont montré qu'on pouvait calculer une seconde préimage d'un message M en temps $2^n/|M|$, donc plus rapidement que les 2^n auxquels on devrait avoir droit, et ce *quelle que soit la fonction de compression* (c'est un défaut du mode opératoire lui-même).

5.3.3 Problèmes

Le mode opératoire de Merkle-Damgård souffre cependant d'un certain nombre de défauts. Certains sont réparables, mais pas tous.

Recyclage des collisions. Une fois qu'on a trouvé une collision dans la fonction de compression, on peut s'en re-servir sans limite. En effet, si on connaît deux séquences de blocs différentes M_0 et M_1 qui produisent la même valeur de chainage, alors on peut rajouter n'importe quoi derrière et la collision continue.

Tant qu'il n'y a pas de collisions sur la fonction de compression, tout va bien, mais par exemple dans le cas de MD5, cela signifie que de telles collisions peuvent être utilisée de manière très dangereuse. Ceci permet par exemple de produire, à partir d'une collision sans signification et difficilement exploitable par elle-même, des fichiers `.pdf` aux contenus un peu arbitraires, qui collisionnent.

La « length-extension attack ». L'un des plus gros problème avec le mode opératoire de Merkle-Damgård est la *length-extension attack*. Si je connais $H(M)$, alors je peux calculer $H(M \,\|\, S)$ pour un suffixe S de mon choix. Et ceci, même sans connaître M (enfin, en connaissant sa taille, et éventuellement un petit bout de la fin).

L'idée, c'est que le début de P doit être le *padding* qui aurait été ajouté à la fin de M. En effet, $H(M)$ est précisément la valeur de chainage qu'on obtient après avoir traité M et son *padding*. Sa connaissance permet donc de « reprendre » le processus de hachage.

Ce problème est assez ennuyeux, par exemple parce qu'il interdit de construire un code d'authentification de message de la façon suivante :

$$\mathsf{MAC}(K, M) = H(K \parallel M).$$

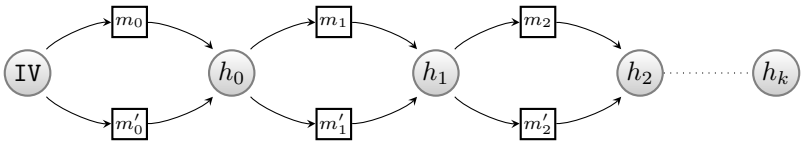
En effet, de la sorte, quelqu'un qui obtiendrait un *tag* d'un message M pourrait produire des tags pour une infinité de messages qui commencent par M. Une possibilité pour éviter ça consiste à transformer de manière irreversible la dernière valeur de chainage avant de la renvoyer. C'est comme ça qu'est définit dans la RFC 2104 le MAC standard :

$$\mathsf{HMAC}(sk, M) = \mathtt{H}\big(sk \oplus \mathtt{opad} \,\|\, \mathtt{H}\,(sk \oplus \mathtt{ipad} \,\|\, M)\big).$$

Ici, `ipad` et `opad` sont des constantes fixées et connues de tous.

Multi-collisions. Un autre problème, découvert en 2004 par A. Joux est lié au caractère itéré de la fonction (et il est *très* difficile à éviter). Une fois qu'on peut trouver des collisions sur la fonction de compression, on peut fabriquer un nombre exponentiellement grand de messages tous différents, mais qui ont la même empreinte.

L'idée est qu'il y a 2^k chemins entre IV et la valeur de chainage h_k dans le graphe ci-dessous, alors que k collisions sont nécessaires à sa réalisation.



5.3.4 Alternatives au mode de Merkle-Damgård

Deux alternatives attractives ont été proposées au mode opératoire de Merkle-Damgård. La première, la « *wide-pipe hash* due à Lucks en 2005, consiste à avoir des valeurs de chainages de taille $2n$, puis à tronquer la dernière à n bits pour produire l'empreinte. Avantage : trouver une collision sur l'état interne coûte maintenant 2^n au lieu de $2^{n/2}$. Trouver une collision sur la sortie coûte toujours $2^{n/2}$, mais n'offre plus du tout les mêmes possibilités d'exploitation.

La deuxième alternative, la « *sponge construction* » gère un état interne sensiblement plus gros que l'empreinte. Le prochain bloc de l'entrée est XORé sur (une partie de) l'état interne, puis une permutation fixée est appliquée à ce dernier. Une fois que toute l'entrée à été traitée, un certain nombre de « tours à blanc » de la permutation sont appliqués. Enfin, pour produire une empreinte de taille n, une partie de l'état interne est renvoyée, puis ce dernier est de nouveau permuté, et le processus recommence jusqu'à ce que n bits aient été produits. SHA-3 est construit de cette manière.

5.4 Fonctions de compression à base de Block Cipher

Pour faire marcher la *sponge construction*, il suffit d'avoir une bonne permutation (un Block Cipher avec une clef fixée peut éventuellement faire l'affaire). Pour obtenir une fonction de hachage avec le mode opératoire de Merkle-Damgård ou le *wide-pipe hash*, il faut pouvoir fabriquer de bonnes fonctions de compression. On peut en faire à base de Block Cipher. Une méthode largement utilisée est la *construction de Davies-Meyer* :

$$F(h, m) = \mathcal{E}(m, h) \oplus h.$$

En gros, on chiffre la valeur de chainage en uilitsant le bloc de message comme clef. On re-XORe la valeur de chainage par dessus pour que la fonction ne soit pas facilement inversible.

Réussir à trouver une collision n'est pas facile. En effet, si $F(h, m) = F(h, m')$, alors on a $E(m, h) = E(m', h)$. Pour un Block Cipher raisonnable, il n'est pas possible de trouver deux clefs différentes qui envoient h sur h'.

Les Block Cipher habituels ne sont cependant pas directement utilisables. D'abord, leur bloc n'est pas assez gros (il en faudrait avec des blocs de 256 bits). De plus, le problème est que les clefs sont connues ! Elles sont même sous le contrôle des adversaires (ce qui ouvre des possibilités concrètes d'attaques *à clefs liées*). La plupart des Block Cipher ne sont pas conçus pour un usage aussi détourné. C'est pour cela que dans les fonctions de hachage, on trouve des Block Cipher *ad hoc* (gros bloc, grosse clef, *key-schedule* particulièrement renforcé, etc.).

Notons une caractéristique particulière : la valeur de chainage $h_0 = \mathcal{D}(m, 0)$ est un *point fixe* pour le bloc de message m. On a en effet $F(h_0, m) = h_0$. En pratique ceci est très difficile à utiliser pour faire des attaques, car on ne sait pas trouver une séquence de blocs qui produit la valeur magique h_0 (sinon on saurait trouver des préimages).

On ne sait pas non plus faire l'inverse : calculer un bloc de message m qui ferait point fixe pour une valeur de h donnée.

Exercices

Exercice 5.1 : Démontrer que l'existence d'une fonction à sens unique f (qu'on ne saurait pas inverser en temps polynomial) impliquerait $P \neq NP$.

Indice : on peut considérer le problème suivant :

INPUT Deux chaines de bits x^* et y.

QUESTION Existe-t-il une chaine de bits x dont x^* est un préfixe, et telle que $f(x) = y$?

Exercice 5.2 : Démontrer que la résistance aux secondes préimages implique la résistance aux préimages.

Exercice 5.3 : Soit $h : \{0,1\}^* \to \{0,1\}^n$ une fonction de hachage résistante aux secondes préimages. On fabrique à partir de h une nouvelle fonction de hachage g de la façon suivante :

$$g(x) = \begin{cases} 1 \parallel x & \text{si } x \text{ est de longueur } n \\ 0 \parallel h(x) & \text{sinon} \end{cases}$$

Montrez que g résiste aux secondes préimages, mais pas aux préimages.

Exercice 5.4 : Soit $h : \{0,1\}^* \to \{0,1\}^n$ une fonction de hachage résistante collisions. Montrez que h résiste aussi aux secondes préimages.

Exercice 5.5 : Cet exercice démontre que retirer la taille du message dans le mode opératoire de Merkle-Damgård affaiblit la construction, car cela permet notamment la *long-message attack*. Expliquer comment on pourrait alors trouver (facilement) une seconde préimage d'un message M, en temps $\approx 2^n/|M|$, où $|M|$ désigne le nombre de blocs de message qui composent M. Pour SHA256, quel serait le coût total de l'attaque dans le meilleur des cas ?

Chapitre 6

L'Advanced Encryption Standard

6.1 Préliminaires

Définition. Un Block Cipher (où "système de chiffrement par blocs") est formé par deux programmes :

— $E(k, x)$, qui réalise le chiffrement du bloc x avec la clef k
— $D(k, y)$, qui réalise le déchiffrement du bloc y avec la clef k

L'idée est que si les blocs occupent n bits et les clefs k bits, alors l'ensemble :

$$\left\{ x \mapsto E(k, x) \;\middle|\; k \in \{0,1\}^k \right\}$$

est une famille de *permutations* de l'espace des blocs, c'est-à-dire de $\{0,1\}^n$.

Sécurité. Il y a plusieurs moyens de définir la sécurité d'un Block Cipher. Dans toutes les définitions, il y a un *adversaire* qui essaye de « casser » le dispositif. L'objectif de l'adversaire est de réussir à faire quelque chose *d'intéressant* :

1. En disposant d'un temps de calcul sensiblement inférieur à 2^k (sinon il pourrait faire la recherche exhaustive et retrouver la clef), et
2. En utilisant sensiblement moins que 2^n paires clair-chiffré (car sinon il connaît le [dé]chiffrement de tous les blocs, et récupérer la clef n'a plus vraiment de sens).

Pour l'adversaire, faire quelque chose *d'intéressant* peut avoir plusieurs significations :

— Réussir à mettre la main sur la clef secrète est en principe l'objectif n°1
— Réussir à déchiffrer un bloc donné, sans avoir le droit de faire des déchiffrements arbitraires, est aussi très fort.

La notion de sécurité la plus forte, et qui capture la notion intuitive de « faire quelque chose *d'intéressant* », consiste à introduire un jeu de *distinguabilité*. On donne à l'adversaire une *boite noire*[1] qui contient l'une de ces deux alternatives :

1. Le Block Cipher, où la clef k est fixée, et a été choisie au hasard parmi les 2^k possibilités
2. Une permutation complètement arbitraire, choisie au hasard parmi les $2^n!$ possibilités

Le but du jeu pour l'adversaire est de déterminer le contenu de la boite noire qui lui est donnée.

On considère qu'un Block Cipher est *sûr* si aucun adversaire ne peut gagner le jeu de distinguabilité avec une probabilité supérieure à $1/2$ (ceci car il peut toujours répondre au hasard). On définit son *avantage* comme $2p - 1$ où p est sa probabilité de succès.

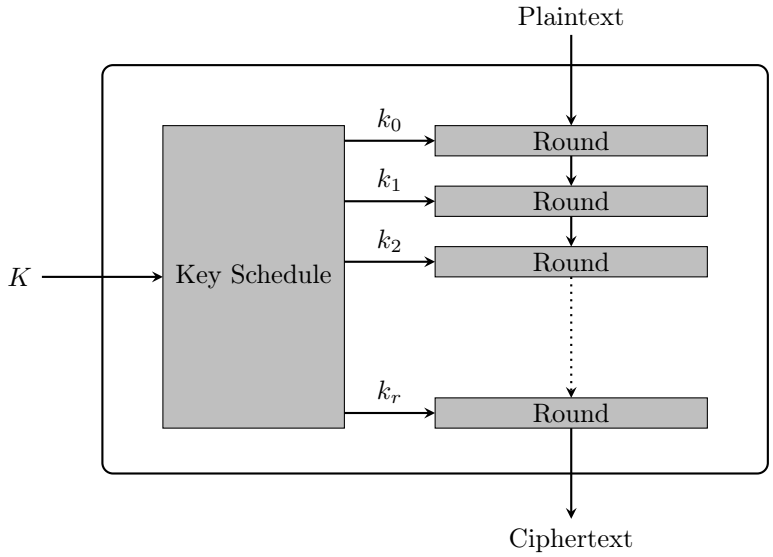
1. La notion de "boite noire" est courante dans l'informatique "théorique". Vous pouvez imaginer qu'il s'agit d'un webservice : il est possible d'envoyer au webservice des requêtes `http://webservice.crypto.fict.if/chiffre/x`, auquel le webservice répond par le chiffrement de x (idem avec le déchiffrement). L'idée c'est qu'on ne sait pas ce que fait vraiment le webservice, et qu'on cherche à deviner.

6.2 Structures communes à la plupart des Block Cipher

Itération. La plupart des Block Cipher sont des constructions *itérées* : le bloc est chiffré en appliquant plusieurs fois de suite une transformation simple, qu'on appelle généralement la « fonction de *tour* » (*round function*). Cette fonction de tour prend donc en entrée un bloc de n bits, et produit en sortie un bloc de n bits. Pour que l'ensemble de la construction soit une permutation, il faut que la fonction de tour soit elle aussi permutation (sinon, on ne pourrait pas inverser le chiffrement).

La fonction de tour prend de plus une clef en entrée, car cela permet que tous les tours ne soient pas identiques. Du coup, la fonction de tour est en fait elle aussi un Block Cipher ! L'idée est la suivante : même si la fonction de tour est un Block Cipher de *mauvaise qualité* (pas sûre), son *itération* peut être, elle, de bonne qualité (sûre). On peut même espérer que plus il y a de *tours*, alors plus la sécurité sera grande.

Pour que les tours puissent être différents, un mécanisme de « *diversification de clef* » (*key-schedule*), produit une clef par tour à partir de la « clef-maître » du Block Cipher.

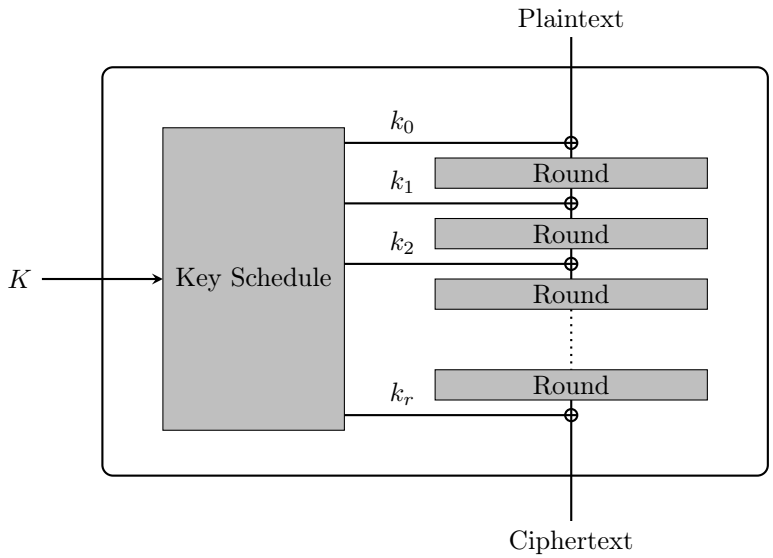


On retiendra que pour construire un Block Cipher, il faut en fait construire un Key-Schedule et une fonction de tour.

Structure des tours. Pour construire une fonction de tour, qui soit une permutation dépendant d'une clef, une méthode simple est utilisée presque tout le temps :

1. On applique d'abord une permutation fixée,
2. On XOR la clef sur le bloc en cours de chiffrement, comme dans le masque jetable.

Du coup, pour déchiffrer, il suffit de faire les opérations dans l'ordre inverse : d'abord XORer à nouveau la clef sur le bloc, puis appliquer l'inverse de la permutation fixée.

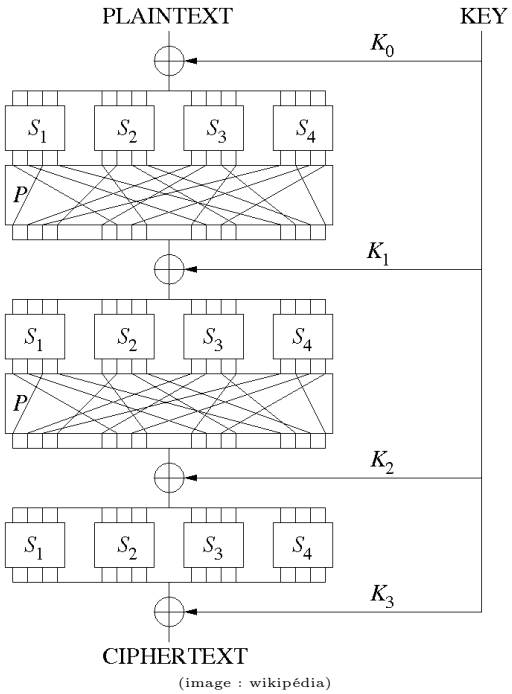


Du coup, il reste à voir comment on peut fabriquer UNE permutation, fixée une fois pour toute, qui « chiffre » un tant soit peu. Si on regarde l'ensemble des Block Cipher qui existent, on se rend compte qu'il y a essentiellement deux manières de faire : les *réseaux de Feistel* (comme le DES), qu'on verra en TD, et les *réseaux de substitution/permutation* (SPN, *Substitution-Permutation Network*). Comme l'AES est de type SPN, on va voir ça plus en détail.

Confusion et diffusion. Une fonction de tour de type SPN est composée de deux « couches » : une couche de *substitution*, et une couche de *permutation*. **Attention** : cette terminologie, qui est usuelle, est trompeuse : les deux couches sont des permutations, au sens mathématique du terme (si ce n'était pas le cas on ne pourrait pas inverser la fonction de tour).

La couche dite de « substitution » applique... une substitution fixée en parallèle à de petits paquets de bits du bloc (typiquement 8 bits dans le cas de l'AES). La substitution appliquée est appelée la *boite-S* (*S-box*). Il peut y en avoir plusieurs, comme sur le dessin ci-dessous, mais dans l'AES il n'y en a qu'une.

Dans les premiers Block Cipher qui utilisaient cette méthodologie, la couche dite de « permutation »... permutait les bits du bloc (cf. dessin). Depuis, les choses ont changé, et ce qui est généralement admis, c'est que la couche de permutation applique une *fonction linéaire* sur le bloc (c'est-à-dire qu'elle multiplie essentiellement le bloc par une matrice).



(image : wikipédia)

Les deux couches jouent des rôles différent.

— Les substitutions introduisent de la *confusion* : une petite partie du bloc est "mélangée" de manière très « profonde » (ce qui est bien), mais locale (ce qui n'est pas bien).
— La couche linéaire introduit de la *diffusion* : elle sert à ce qu'un changement sur une petite partie du bloc se répercute partout ailleurs. Comme elle est généralement linéaire, cette couche a une structure très forte (ce qui n'est pas bien), cependant elle affecte globalement le bloc (ce qui est bien).

La logique des choses est la suivante : on sait assez bien construire des substitutions « compliquées » qui s'appliquent à un petit nombre de bits (il suffirait presque de les choisir au hasard, et de les stocker dans une table). Cependant, si on savait en construire qui s'appliquent à autant de bits qu'on veut, alors on ne chercherait pas à construire des Block Cipher.

De plus, on sait facilement construire des applications linéaires inversibles, faciles à calculer, qui s'appliquent à autant de bits qu'on veut.

La combinaison de ces deux techniques donne la construction SPN. Notez que la S-boite est le seul composant *non-linéaire* de tout le Block Cipher. La présence de composants non-linéaires est indispensable : s'il n'y en avait pas, alors le Block Cipher serait une fonction linéaire, qu'on pourrait représenter par une matrice. On pourrait reconstituer tous les coefficients de cette matrice avec des paires clair-chiffré en résolvant un gros système d'équations linéaires, ce qui permettrait de faire le déchiffrement.

6.3 L'AES

L'AES (dont le nom original est rijndael) est un Block Cipher de type SPN. Il traite des blocs de $n = 128$ bits, et des clefs de 128, 192 ou 256 bits. Le nombre de tour est de 10, 12 ou 14 selon la taille de la clef. On sa se concentrer ici sur la version avec des clefs de $k = 128$ bits. Le bloc et la clef occupent donc 128 bits chacuns, soit 16 octets. Dans l'AES, ils sont organisé dans une matrice 4×4.

Pour commencer, la première sous-clef est XORée sur le bloc, puis 10 tours sont effectués. Chaque tour est composé de 4 opérations, dont les noms de code sont SubByte (substitution), ShiftRows et MixColumn (partie linéaire), puis AddRoundKey (XOR de la clef). Petite arnaque : dans le dernier tour, l'opération MixColumn est omise.

Pour décrire complètement l'AES, il nous reste donc à décrire son Key-Schedule, sa boite-S, et sa couche de « permutation ».

Structure agébrique dans l'AES. Il nous faut pour ça faire un petit détour par les mathématiques. L'AES repose en effet lourdement sur l'idée suivante, que nous admettrons :

> L'ensemble des séquences de 8 bits peut être muni d'une *addition* et d'une *multiplication*, de sorte que toutes les règles habituelles de l'arithmétiques soient encore vraies. De plus, toutes les chaines de bits différentes de 00000000 possèdent un inverse pour la multiplication.

La notion mathématique sous-jascente, c'est que l'ensemble des chaines de 8 bits $\{0,1\}^8$ peut-être vu comme un *corps fini à 256 éléments*. Là-dedans, « L'addition » en question est en fait le XOR. Pour la « multiplication », c'est plus compliqué... Mais on a au moins un bout de code C qui la calcule :

```
unsigned char mul(unsigned char a, unsigned char b) {
    unsigned char p = 0;
    unsigned char counter;
    unsigned char hi_bit_set;
    for(counter = 0; counter < 8; counter++) {
       if((b & 1) == 1)
           p ^= a;
       hi_bit_set = (a & 0x80);
       a <<= 1;
       if(hi_bit_set == 0x80)
```

```
            a ^= 0x1b;                  // constante magique
        b >>= 1;
    }
    return p;
}
```

Pour les petits malins qui veulent tout savoir : une chaine de 8 bits $a_0 a_1 \dots a_7$ est vue comme me polynôme de degré 8 à coefficients dans $\{0,1\}$ ($X^8 + a_7 X^7 + \dots + a_1 X + a_0$). Pour multiplier deux octets, on multiplie les polynômes, on divise le produit par le polynôme $X^8 + X^4 + X^3 + X + 1$, on jette le quotient et on garde le reste de la division. Le On obtient un polynôme de degré au plus 7, qui est donc représenté par 8 bits, et voilà ! Notez que le polynôme par lequel on divise est représenté par la chaîne de bits `10001101`, ce qui donne `0x11b` en hexadécimal (presque la constante magique).

La S-boite. La S-boite est en gros... la fonction qui à une chaine de bits x associe son inverse pour la multiplication. Mais, craignant que cela n'introduise une structure algébrique exploitable dans leur Block Cipher, les inventeurs de l'AES ont décidé que leur S-boite serait la composition de la fonction inverse, et d'une fonction affine.

Au final, on sait que le polynôme suivant représente la S-boite :

$$S(x) = \mathtt{05}X^{254} + \mathtt{09}X^{253} + \mathtt{F9}X^{251} + \mathtt{25}X^{247} + \mathtt{F4}X^{239} + \mathtt{B5}X^{223} + \mathtt{B9}X^{191} + \mathtt{8F}X^{127} + \mathtt{63}$$

Il se trouve que c'est un polynôme de degré *maximal*, tant que les chaines de 8 bits sont concernées (n'importe quel polynôme de degré plus grand calcule en fait la même fonction qu'un polynôme de degré inférieur ou égal à 254). Ceci n'est pas un hasard : la S-boite a été choisie pour être « la plus non-linéaire possible ».

La couche linéaire. La couche linéaire de l'AES traite le bloc comme un paquet de 16 élements, où chacun des éléments est une chaine de 8 bits comme on a vu ci-dessus.

La couche linéaire pourrait donc être décrite par une grosse matrice 16×16, par lequel il s'agit de multiplier le bloc. Cependant, pour des raisons d'efficacité, elle se compose de deux phase, qui opèrent sur une matrice 4×4 d'octets.

1. La phase ShiftRows décale la i-ème ligne de la matrice du bloc de i cases vers la gauche (les octets qui sortent à gauche reviennent à droite). Les valeurs des octets ne sont pas modifiées. La première ligne, qui porte logiquement le numéro zéro, n'est pas modifiée du tout.
2. La phase MixColumn consiste à multiplier chacune des 4 lignes du bloc par une matrice spéciale, fixée, qui est :
$$\begin{pmatrix} \mathtt{02} & \mathtt{03} & \mathtt{01} & \mathtt{01} \\ \mathtt{03} & \mathtt{01} & \mathtt{01} & \mathtt{02} \\ \mathtt{01} & \mathtt{01} & \mathtt{02} & \mathtt{03} \\ \mathtt{03} & \mathtt{01} & \mathtt{01} & \mathtt{02} \end{pmatrix}$$

La matrice en question a été choisie pour ses propriétés :
- ses coefficients sont petits (seuls les 2 bits de poids faibles sont non-nuls)
- elle est inversible, et les coefficients de son inverse sont petits
- elle possède la propriété MDS (*Maximum Distance Separable*). Cela signifie que si (y_0, y_1, y_2, y_3) est l'image de (x_0, x_1, x_2, x_3) par cette matrice, alors au moins 5 élements parmis les x_i et les y_i sont différents de zéro.

Le fait que les coefficients soient petits sert à accélerer les opérations de multiplications. La dernière propriété sert à obtenir des preuves de la résistance de l'AES contre certaines attaques, notamment la *cryptanalyse différentielle*.

Cryptanalyse différentielle. Cette technique de cryptanalyse, utilisée pour la première fois publiquement en 1990 par Biham et Shamir, a permis de casser plusieurs Block Cipher qui n'étaient pas conçus pour y résister. Curieusement, le DES, conçu par la NSA au milieu des années 1970, semblait y résister de manière optimale. En fait, la NSA connaissait apparemment déjà la technique 20 ans plus tôt.

C'est une technique qui peut servir à distinguer un Block Cipher d'une permutation aléatoire. L'idée générale consiste à effectuer en parallèle le chiffrement de deux messages, en suivant à la

trace la différence entre les deux. On fixe par exemple une différence en entrée Δ_{in}, puis on chiffre $Y_1 \leftarrow E(K, M)$ et $Y_2 \leftarrow E(K, M \oplus \Delta_{in})$. On examine la différence en sortie Δ_{out} qui vaut $Y_1 \oplus Y_2$.

L'intérêt de cette manoeuvre, c'est que les opérations linéaires (XOR des clefs, couches de permutations) affectent la différence de manière déterministe. Seuls les composants non-linéaires peuvent modifier la différence de façon imprévisible.

Cette présentation est un peu abstraite. C'est en essayant sur des exemples concrets qu'on se rend compte de ce qu'il se passe.

Troisième partie

Cryptographie à clef publique

Chapitre 7

Arithmétique pour la cryptographie

Toute une partie de la cryptographie à clef publique nécessite d'effectuer des calculs sur de « grands » entiers, d'une part. D'autre part, le fonctionnement de bon nombre d'algorithmes reposent sur des résultats mathématiques concernant les nombres.

7.1 Arithmétiques des entiers

7.1.1 Algorithmes de base

En 2019, un ordinateur moderne est capable d'effectuer des opérations arithmétiques sur des entiers qui tiennent dans des registres du processeur. Cela signifie que le processeur est capable d'effectuer des additions/soustractions/multiplications/divisions d'entiers sur 32 ou 64 bits.

Pour faire des calculs avec des entiers plus grands (des « entiers multi-précision »), il faut utiliser des routines logicielles. Il en existe heureusement déjà, notamment la bibliothèque Gnu Multi Precision (GMP). Quand on s'initie à l'analyse des algorithmes, on nous explique que les opérations arithmétques sont des opérations « élementaires » de coût constant. C'est vrai en première approximation. Cependant, on ne peut clairement pas additionner des nombres de n chiffres en temps constant quand n devient arbitrairement grand.

À l'école primaire, nous apprenons les algorithmes « classiques » pour effectuer les quatre opérations arithmétiques. Ils ne sont pas très compliqués à programmer, à l'exception de la division qui est, elle, très difficile. Comme vous les connaissez déjà, ils ne sont pas décrits ici, et seule leur complexité est donnée. Dans son ouvrage classique *The Art of Computer Programming*, D. Knuth propose des implémentations de ces algorithmes dans un langage assembleur fictif, et analyse le nombre de *cycles* nécessaires à leur exécution :

Opération	Taille de x	Taille de y	nombre de cycles
$x + y$	n	n	$10n + 6$
$x - y$	n	n	$12n + 3$
$x \times y$	n	m	$28mn + 4m + 7n + 3$
x/y	$m + n$	n	$30mn + 30n + 89m + 111$

On se contente d'additionner et de soustraire des nombres de la même taille, car on peut toujours prendre le plus petit des deux et le « bourrer » de zéros à gauche jusqu'à ce qu'il ait la taille souhaitée.

Il existe de nombreux algorithmes de multiplication de grands entiers dont la complexité asymptotique est meilleure que celle de l'algorithme classique. Un algorithme simple dû à A. Karatsuba en 1962 a une complexité en $\mathcal{O}\left(n^{1.585}\right)$. Un algorithme compliqué dû à Schönage en 1971 fonctionne en $\mathcal{O}\left(n \log n \log\log n\right)$, mais la constante cachée dans le $\mathcal{O}$ est si grande qu'il n'est plus rapide que les autres que pour des nombres qui ont environ un million de bits.

7.1.2 Division euclidienne

L'algorithme de division calcule à la fois le quotient et le reste, comme lorsqu'on fait l'opération à la main. Du coup, il s'ensuit que l'opération `mod` a la même complexité que la division.

Définition : Division euclidienne

> Soient a et b deux entiers. Effectuer la **division euclidienne** de a par b, c'est calculer deux entiers q et r tels que $a = bq + r$ et $0 \leq r < b$.
>
> On appelle q le **quotient** et r le **reste** de la division. Le reste est souvent noté $a\%b$ ou $a \bmod b$.

Le résultat $a = bq+r$ de la division euclidienne est *unique* : si on a à la fois $a = bq+r$ et $a = bq'+r'$ avec $0 \leq r, r' < b$, alors $q = q'$ et $r = r'$.

On sait tous que parfois des divisions ne « tombent pas juste », car il y a un reste non nul. Le cas où elles « tombent juste » est suffisamment intéressant pour mériter un nom :

Définition : diviseur

> On dit que $a \neq 0$ est un **multiple** de b, et que b est un **diviseur** de a, lorsqu'on peut écrire $a = bx$, où x est un entier. Ceci signifie que la division de a par b produit un reste nul.

Par définition, un nombre est toujours multiple de 1 et de lui-même. On dit que ce sont des diviseurs *triviaux*. Les autres sont donc, logiquement, *non-triviaux*. Quand on parle des diviseurs d'un nombre, on écarte souvent les diviseurs triviaux parce qu'ils ne sont pas intéressants.

7.1.3 Nombres premiers et factorisation

Imaginons qu'on nous fournisse un nombre. On pourrait se demander quels sont ses diviseurs ? Il est facile de déterminer que le nombre 42 est divisible par 2, 3, 7, 14 et 21. Par contre, le nombre 43 n'a aucun diviseur (non-trivial).

Définition : nombre premier

> Un nombre qui n'a pas de diviseurs (autres que 1 et lui-même) est un nombre **premier**. Un nombre qui n'est pas premier est **composite**.

Les nombres premiers jouissent d'un status particulier, car ce sont des « briques de base » à partir desquelles on peut fabriquer tous les autres nombres.

Théorème 3 (Théorème fondamental de l'arithmétique)**.** *Tout nombre entier n peut s'écrire de manière unique comme un produit de nombres premiers :*

$$n = p_1^{\alpha_1} \times p_2^{\alpha_2} \times ... \times p_k^{\alpha_k},$$

où les p_i sont des nombres premiers et les exposants α_i sont des entiers.

Démonstration

> L'existence de cette décomposition est facile à démontrer par récurrence sur n. Si $n = 1$, alors n n'est le produit de rien du tout et c'est réglé (la décomposition est vide, $k = 0$). Sinon, si $n > 2$, alors de deux choses l'une :
>
> — Soit n est lui-même premier, et donc la décomposition qui va bien est déjà fournie ($p_1 = 1, \alpha_1 = 1, k = 1$).
>
> — Soit n est composite, c'est-à-dire qu'on peut écrire $n = k\ell$ pour deux entiers $1 < k$ et $1 < \ell$. Par hypothèse de récurrence, ces deux entiers étant forcéments plus petits que n, on peut les écrire comme un produit de nombres premiers. Du coup, le produit de ces deux décomposition donne une écriture de n comme un produit de nombres premiers uniquement.
>
> On admet l'unicité de la décomposition. □

Par exemple, on trouve :

$$270006104 = 2^3 \times 17 \times 1051 \times 1889$$

Calculer cette décomposition, c'est *factoriser* n (l'écrire comme un produit de *facteurs*). En 2019, on ne connaît pas d'algorithme efficace pour factoriser de grands entiers. Le record actuel de factorisation (par des universités) est de 768 bits.

Considérons l'algorithme suivant, qui est généralement désigné sous le nom de « *factoring by trial division* » :

```
def FindDivisor(n):
    for i in range(2, math.sqrt(n)):
        if (n % i) == 0:
            return i
    return "prime"
```

Tout d'abord, cet algorithme est correct : s'il renvoie un nombre alors ce nombre est clairement un diviseur de n. De plus, s'il renvoie « prime », alors n est vraiment premier. En effet, dans ce cas-là, aucun nombre plus petit que $\sqrt{n}$ ne divise n. Si n était composite, alors il s'écrirait $n = ab$, et l'un des deux nombres a, b doit être plus petit que $\sqrt{n}$.

Essayons d'analyser son temps d'exécution. Si n est premier, ce qui est le pire des cas, il va y avoir $\sqrt{n}$ divisions « n/i », avec $i \leq \sqrt{n}$. Chacune de ces divisions prend un temps $\mathcal{O}\left(\log^2 n\right)$. La complexité totale est donc $\mathcal{O}\left(\sqrt{n}\log^2 n\right)$. Cet algorithme est donc *exponentiel* en la taille de n : le nombre n occupe $\log_2 n$ bits, et donc pour des nombres de k bits cet algorithme s'exécute en $\mathcal{O}\left(k^2 2^{k/2}\right)$.

Si n est composite, l'algorithme s'arrête dès qu'il a trouvé le plus petit diviseur non-trivial de n. Le nombre d'itérations est donc proportionnel à la taille de ce diviseur. Cet algorithme peut donc efficacement trouver de *petits* diviseurs, mais il est très inefficace pour trouver les grands diviseurs.

Générons un nombre plus petit que 10^{30} au hasard et calculons sa factorisation :

$$721230868692949606394508763765 = 9166752431102476669847 \times 15735799 \times 5$$

C'est la situation typique : il y a quelques petits diviseurs, et quelques énormes. Ceci fait que l'algorithme de *trial division* ne peut pas venir à bout de la plupart des entiers : les petits facteurs sont faciles à trouver, mais pas les gros.

Une autre conséquence de tout ceci est que les nombres de la forme $n = pq$ où p et q sont deux nombres premiers très grands sont « les plus difficiles » à factoriser, car il n'y a pas de petits facteurs. C'est pour cela qu'ils sont abondamment utilisés en cryptographie.

D'autres méthodes bien plus sophistiquées existent pour trouver des diviseurs de grands entiers. Aucune n'est polynomiale en le nombre de bits de l'entier. En 2019, il semble raisonnable de penser que des nombres (bien choisis) de 2048 bits sont hors de portée des algorithmes de factorisation existants.

7.1.4 Répartition des nombres premiers

Les nombres premiers fascinent les mathématiciens depuis l'antiquité. En effet, autant leur définition est très simple, autant nombres de leurs propriétés semblent insaisissables.

De la même manière que la matière est formée d'une combinaison d'atomes, tous les nombres sont formés d'une combinaison de nombres premiers. Cependant, la comparaison s'arrête là : contrairement aux différentes sortes d'atomes, qui sont en nombre limités, il y a une infinité de nombres premiers.

Théorème 4. *Il y a une infinité de nombres premiers. Par conséquent il y a des nombres premiers arbitrairements grands.*

Démonstration

On raisonne par l'absurde. Supposons qu'il n'y ait qu'un nombre fini de nombres premiers, et on les note $p_1, \ldots, p_k$. On pose :

$$N = p_1 \times \cdots \times p_k + 1.$$

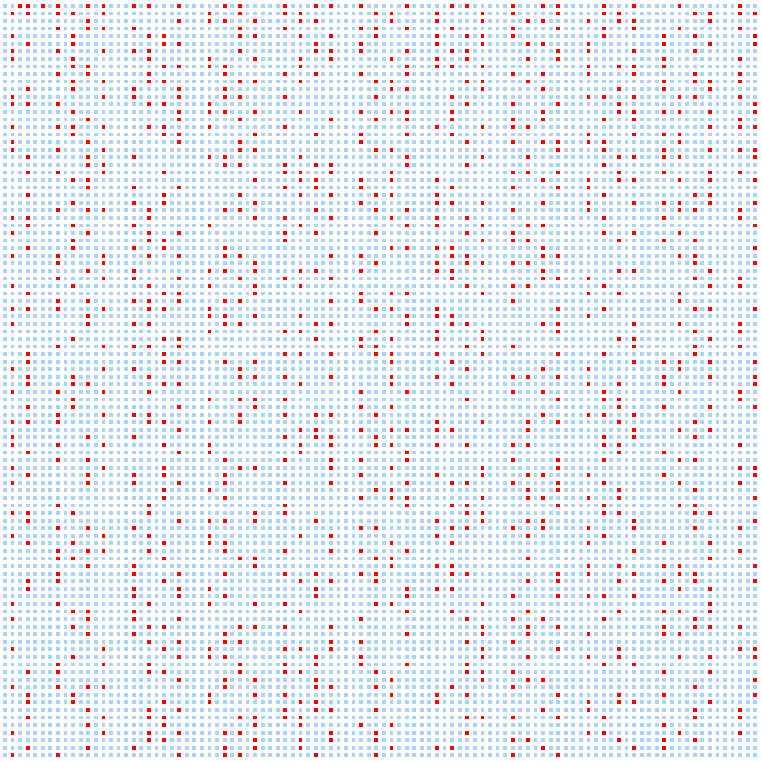
Ce nombre est plus grand que chaque nombre premier séparément, donc il est composite. Par conséquent, d'après le theorème fondamental de l'arithmétique, on peut l'écrire comme un produit d'au moins deux nombres premiers.

Prenons un nombre premier quelconque p_i. On prétend que $N \% p_i = 1$. On a en effet :

$$N = p_i \times \Big(p_1 \times \cdots \times p_{i-1} \times p_{i+1} \times \cdots \times p_k\Big) + 1$$

> On voit donc qu'on a écrit $N = p_i \times x + 1$, avec $1 < p_i$, donc par unicité du résultat de la division euclidienne on trouve que le reste est 1.
>
> Ceci prouve que p_i ne divise *pas* N (en effet il y a un reste non-nul). On a une contradiction : N est censé être le produit de plusieurs nombres premiers, et aucun nombre premier ne peut diviser N.
>
> C'est donc qu'il y a une infinité de nombre premiers différents. □

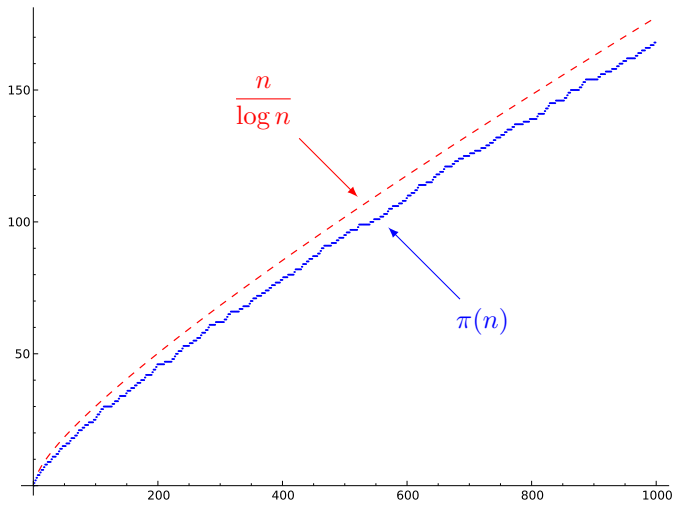
À première vue, la répartition des nombres premiers a l'air assez irrégulière. Voici les nombres de 1 à 10 000 (les nombres premiers dans une autre couleur) :



On ne voit pas émerger très clairement de régularité. Par contre, au fil du XIXème siècle, les mathématiciens ont réussi à cerner plus précisément la façon dont ils sont échelonnés. On note souvent $\pi(n)$ le nombre de nombres premiers plus petits que n. On connaît une approximation simple de $\pi(n)$:

Théorème 5 (theorème des nombres premiers, Hadamard et La Vallée Poussin, 1896)**.**

$$\pi(n) \approx \frac{n}{\log n}.$$



Ceci est intéressant pour la raison suivante. Imaginons qu'on veuille choisir uniformément au hasard un nombre premier plus petit que n. Si on disposait d'un algorithme `IsPrime` qui teste si un nombre est premier, alors l'algorithme suivant ferait l'affaire :

```
def RandomPrime(n):
    while True:
        k = random.randrange(n)
        if IsPrime(k):
            return k
```

Combien d'itérations va-t-il y avoir ? Il y a $\approx n/\log n$ nombres premiers plus petits que n, donc la probabilité que chacun des nombres k choisis au hasard soit premier est $\approx 1/\log n$. Cela signifie encore qu'il va y avoir environ $\log n$ itérations, ce qui est peu. Il existe des algorithmes efficaces de test de primalité, ce qui fait qu'on dispose d'un moyen efficace de générer de grands nombres premiers aléatoires.

7.2 Arithmétique modulaire

En cryptographie à clef publique on est souvent amené à faire des calculs *modulo* n. En première approximation, cela revient à faire des calculs, à diviser le résultat par n, puis à renvoyer le reste de la division.

Une des raisons pour laquelle cette opération est abondamment utilisée est qu'elle permet d'obtenir des résultats de taille contrôlée. En effet, dès qu'on fait des calculs avec des nombres, ils peuvent avoir tendance à grossir. En « coupant » le tout modulo n, on sait qu'on renvoie un résultat plus petit que n. Une autre raison, plus profonde, est qu'une bonne partie des raisonnements mathématiques habituels « se passe bien » modulo n. Enfin, une troisième raison est qu'un certain nombre de problèmes calculatoirement difficiles apparaissent quand on raisonne « modulo n ».

7.2.1 Congruence modulo n

Définition : congruence modulo n

On dit que deux entiers a et b sont **congrus modulo** n, ou bien **égaux modulo** n si $b - a$ est un multiple de n. Quand c'est le cas on écrit :

$$a \equiv b \mod n.$$

La relation $\equiv$ ressemble à s'y méprendre à l'égalité habituelle sur les entiers. En fait, écrire « $a \equiv b \bmod n$ » c'est écrire « a et b ont le même reste dans la division par n ». Cette affirmation est justifiée ci-dessous et quelques propriétés élémentaires sont démontrées.

Propriété 1. *La relation* $\equiv$ *est une* relation d'équivalence*, c'est-à-dire :*

i) $a \equiv a \bmod n$.

ii) *Si* $a \equiv b \bmod n$*, alors* $b \equiv a \bmod n$.

iii) *Si* $a \equiv b \bmod n$ *et* $b \equiv c \bmod n$*, alors* $a \equiv c \bmod n$.

Démonstration

i) On a bien $a \equiv a \bmod n$, car $a - a = 0 \times n$.

ii) L'hypothèse signifie que $b - a = kn$. Par conséquent, on a $a - b = (-k)n$, c'est-à-dire la conclusion voulue.

iii) Les hypothèses signifient que $b - a = kn$ et $c - b = \ell n$. En sommant les deux équations on trouve : $c - a = (k + \ell)n$.

□

Propriété 2. *La relation $\equiv$ est compatible avec l'addition et la multiplication. Si $a \equiv b \bmod n$ et $c \equiv d \bmod n$, alors :*

$$a + c \equiv b + d \mod n$$
$$a \times c \equiv b \times d \mod n$$

Démonstration

On sait qu'il y a deux entiers k et ℓ tels que $b - a = kn$ et $d - c = \ell n$. Du coup, il en découle :

$$(b + d) - (a + c) = (b - a) + (d - c) = (k + \ell)n.$$

Par conséquent, n divise $(b + d) - (a + c)$, et donc on a bien la première congruence annoncée.

Le cas de la multiplication n'est pas plus compliqué :

$$bd - ac = (a + kn)(c + \ell n) - ac = (a\ell + kc + k\ell n)n.$$

On voit que $bd - ac$ est un multiple de n, donc on a la deuxième congruence annoncée. □

Une dernière propriété utile est la possibilité de « réduire modulo n » des résultats intermédiaires à n'importe quel moment sans modifier le résultat final modulo n.

Propriété 3. $a \equiv (a\%n) \bmod n$.

Démonstration

On pose la division de a par n : $a = qn + r$ avec $0 \leq r < n$. Du coup, on trouve $a - r = qn$, et on voit bien que c'est un multiple de n. □

Pour voir l'intérêt de cette remarque, imaginons qu'on veuille calculer $(42X^2 + 3XY - 4Y)\%11$. On peut commencer par poser $x = X\%11$ et $y = Y\%11$. La combinaison des trois propriétés ci-dessus assure alors :

$$42X^2 + 3XY - 4Y \equiv \Big(9\big(x^2\%11\big) + (3xy\%11)\%11\Big) - (4y\%11).$$

L'intérêt c'est qu'une fois qu'on a obtenu x et y, on ne manipule plus que des nombres inférieurs à 11.

On peut donc ajouter et mutiplier des congruences comme on a l'habitude de le faire avec des équations habituelles qu'on manipule depuis le collège. Pour voir le phénomène en action, essayons de résoudre la congruence :

$$3x + 4 \equiv 5 \mod 7.$$

On « ajoute » 3 des deux côtés et on obtient :

$$3x \equiv 1 \mod 7.$$

On remarque qu'on a obtenu l'effet « soustraire 4 » en effectuant l'opération « ajouter 3 ». Ceci a lieu parce que $3 \equiv -4 \bmod 7$.

Ensuite, on multiplie les deux côtés de notre congruence par 5, et on trouve :

$$x \equiv 5 \mod 7.$$

De manière plus surpenante, on a obtenu l'effet « diviser par 3 » en effectuant l'opération « multiplier par 5 » ! Ceci s'explique parce que $3 \times 5 \equiv 1 \bmod 7$.

7.2.2 Application / digression : la preuve par 9

L'arithmétique modulaire permet de vérifier des multiplications de tailles arbitraire sans jamais manipuler des nombres plus grands que 100. En effet, si on a correctement calculé $c = ab$, alors en articulier on doit avoir $c \equiv ab \bmod 9$.

D'après ce qu'on a vu ci-dessus, ceci revient à vérifier si :

$$(c\%9) \equiv (a\%9)(b\%9) \mod 9$$

Cette vérification est facile à effectuer de tête (à condition de connaître ses tables de multiplication pour effectuer le produit de droite)... à condition d'être capable de calculer facilement « quelque chose modulo 9 ».

Pour cela, on remarque un phénomène bizarre :

$$1337 \equiv 1 + 3 + 3 + 7 \mod 9.$$

Un nombre en apparence anodin est égal à la somme de ses chiffres modulo 9. Bizarre ! En fait, il se trouve c'est vrai tout le temps. Le nombre n dont l'écriture en base 10 est $a_k a_{k-1} \dots a_1 a_0$ a pour valeur :

$$n = \sum_{i=0}^{k} a_k 10^k$$

Or il se trouve que $10^k \equiv 1 \bmod 9$: en effet $10^k - 1 = 999 \dots 999$, qui est clairement un multiple de 9. Il en découle que :

$$n \equiv \sum_{i=0}^{k} a_k \mod 9.$$

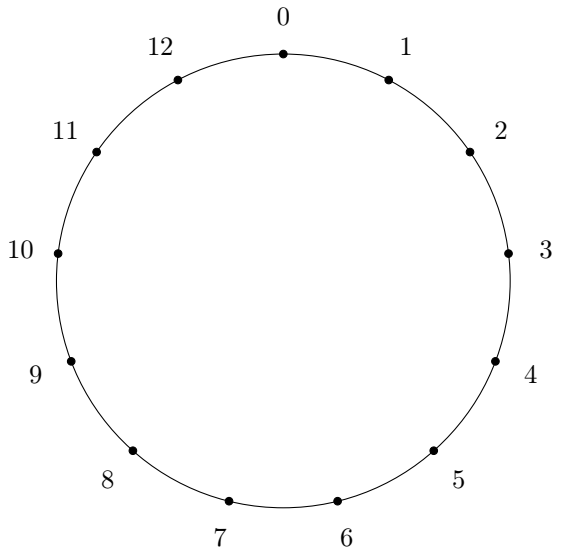
Du coup, pour calculer $a \bmod 9$ on calcule $\sum_{i=0}^{k} a_k \bmod 9$. Et la même ruse peut s'utiliser récursivement ! Par exemple, calculons :

$$\begin{aligned}
91799814 &\equiv (9+1) + (7+9) + (9+8) + (1+4) \mod 9 \\
&\equiv 10 + 16 + 17 + 5 \\
&\equiv (1+0) + (1+6) + (1+7) + 5 \\
&\equiv (1+7) + (8+5) \\
&\equiv 8 + 13 \\
&\equiv 8 + (1+3) \\
&\equiv 12 \\
&\equiv (1+2) \\
&\equiv 3.
\end{aligned}$$

7.2.3 Ensemble $\mathbb{Z}_n$ des entiers modulo n

Dans la section précédente, on vient de voir que l'essentiel des calculs et des raisonnements qu'on peut faire sur les entiers, on peut aussi les faire « modulo n ».

On note $\mathbb{Z}_n$ l'ensemble des entiers modulo n, c'est-à-dire l'ensemble $0, 1, \dots, n-1$. Cet ensemble est muni d'une addition et d'une multiplication qui calculent « modulo ». On peut représenter l'ensemble $\mathbb{Z}_n$ comme un cercle, car lorsqu'on atteint n on revient à zéro :



Quand il est clair qu'on est en train de parler d'éléments de $\mathbb{Z}_n$, on s'autorise à surcharger le prédicat d'égalité et les opérateurs arithmétiques. Ainsi, dans $\mathbb{Z}_{13}$, on a $6+9=2$ et $6\times 7=3$! Et en fait, comme $5+8=0$, il est logique de dire que 8 est l'opposé de 5 et d'écrire $8=-5$. De la même manière, on a $2\times 7=1$, donc il est tentant de dire que 2 est l'inverse de 7, et il est logique de le noter $2=7^{-1}$ ou encore $2=\frac{1}{7}$.

À défaut de disposer d'une soustraction et d'une division dans $\mathbb{Z}_n$, on peut obtenir les effets escomptés en ajoutant les opposés (pour soustraire) et en multipliant par les inverses (pour diviser).

Il faut cependant se méfier de quelque chose de nouveau. Dans $\mathbb{Z}_{12}$, on a $3\times 4=0$. Ceci est contraire à ce qui se passe d'habitude : on nous a bien appris que si x et y sont deux nombres non-nuls, alors le produit xy n'est jamais nul. Ceci est vrai quand on parle de l'ensemble des entiers relatifs $\mathbb{Z}$, mais ce n'est pas forcément vrai modulo n.

De plus, certains élements peuvent avoir des inverses, mais d'autres peuvent ne pas en avoir. Par exemple, dans $\mathbb{Z}_{12}$ toujours, on a $5\times 5=1$, donc $\frac{1}{5}=5$ (eh oui !), mais 4 n'a pas d'inverse. En effet, s'il existait x tel que $4x=1$, alors en multipliant des deux côtés par 3, on trouverait $0=3$, ce qui est absurde.

Comment déterminer quels élements sont inversibles, et comment calculer leurs inverses ?

7.2.4 Algorithme d'Euclide étendu et ses applications

Pour accomplir ces deux tâches, on a besoin d'un des plus anciens algorithmes connus de l'humanité : l'algorithme d'Euclide.

Définition : PGCD

> Étant donné deux entiers a et b, leur « **plus grand commun diviseur** » (PGCD) est le plus grand entier c qui divise à la fois a et b.
>
> Si deux nombres n'ont aucun diviseur en commun (autre que 1), on dit qu'ils sont **premiers entre eux** (dans ce cas leur PGCD vaut 1). Il est commode de fixer $PGCD(0,u)=u$.

Version de base

L'algorithme d'Euclide permet de calculer très efficacement le PGCD de deux entiers :

```
def PGCD(a, b):
    while b != 0:
        r = a % b
        a = b
        b = r
    return a
```

Pourquoi ceci renvoie-t-il bien le PGCD de a et b ? Tout repose dans le fond sur l'observations suivante :

Lemme 1.

Si b est un multiple de a, alors $PGCD(a,b)=a$, et sinon $PGCD(a,b)=PGCD(a,b\%a)$.

Démonstration

> Le cas où $b=ka$ est très facile : a est en effet un diviseur commun, et il ne peut y en avoir de plus grand (rien de plus grand que a ne peut diviser a).
>
> Supposons maintenant que b ne soit pas un multiple de a. On montre le résultat en deux étapes.
>
> 1. Soit d un diviseur commun à a et b. On pose donc $a=da'$ et $b=db'$. On pose la division euclidienne de b par a :
>
> $$\begin{aligned} b &= qa+r & 0<r<a \\ db' &= qda'+r & \\ r &= d(b'-qa'). & \end{aligned}$$

On voit bien que r, qui ne peut pas être nul, est un multiple de d.
Par conséquent, tout diviseur commun à a et b est aussi un diviseur commun à a et $b\%a$, donc il est inférieur ou égal à $PGCD(a, b\%a)$. Comme c'est vrai en particulier pour $PGCD(a, b)$, on trouve :

$$PGCD(a, b) \leq PGCD(a, b\%a).$$

2. Soit d un diviseur commun à a et $b\%a$. On écrit donc $a = da'$ et $b\%a = dr'$. On pose la division euclidienne de b par a :

$$\begin{aligned} b &= qa + (b\%a)b && = da' + dr' \\ b &= d(a' + r'). \end{aligned}$$

On voit donc que d divise b. Le même raisonnement qu'au-dessus montre donc que :

$$PGCD(a, b\%a) \leq PGCD(a, b).$$

En combinant les deux inégalités on obtient le résultat annoncé. □

La correction repose sur l'application répétée de cet argument : au début de chaque itération de la boucle, le PGCD de a et b est le PGCD des deux arguments de départ.

Complexité

L'algorithme est très efficace. On va montrer ici le résultat suivant :

Théorème 6. *Calculer le PGCD de deux nombres a et b de n bits nécessite $\mathcal{O}\left(n^2\right)$ opérations.*

Ceci est remarquable : calculer le PGCD de deux nombres a la même complexité asymptotique que le calcul de leur quotient par l'algorithme classique. Or il n'y a pas qu'une seule division !

Le reste de cette section (dont la lecture est facultative) est consacré à justifier ce résultat.

Tout d'abord, on considère la suite des nombre de Fibonacci que vous connaissez bien :

$$F_0 = 1, \quad F_1 = 1, \quad F_{i+2} = F_{i+1} + F_i$$

On remarque que l'algorithme effectue n itérations lorsqu'on calcule $PGCD(F_{n+2}, F_{n+1})$. En effet, comme $F_i < F_{i+1}$, il découle de la définition des nombres que le reste de la division de F_{i+2} par F_{i+1} est F_i. Par conséquent, si $a = F_{n+1}$ et $b = F_{n+2}$ au début d'une itération, on aura $a = F_n$ et $b = F_{n+1}$ à la fin.

En fait, il se trouve qu'il s'agit d'un *pire cas*, ce que nous admettrons sans démonstration.

Lemme 2. *Soient $u > v > 0$ deux nombres tels que le calcul de $PGCD(u, v)$ par l'algorithme d'Euclide nécessite exactement n itérations. Si u est le plus petit entier satisfaisant cette condition, alors $u = F_{n+2}$ et $v = F_{n+1}$.*

Grâce à cette remarque, on obtient un résultat intéressant :

Lemme 3. *Avec des nombres de n bits, le nombre total d'itérations est $\mathcal{O}(n)$.*

Démonstration

Remarquons d'abord qu'après la première itération, on a forcément $a > b$.

On justifie maintenant par récurrence que $F_{k+1} \geq (3/2)^k$. Pour $k = 0$ et $k = 1$, c'est vrai. Ensuite, pour $k > 1$, on a :

$$F_{k+2} = F_{k+1} + F_k \geq \left(\frac{3}{2}\right)^k + \left(\frac{3}{2}\right)^{k-1} = \left(\frac{3}{2}\right)^k \left(1 + \frac{2}{3}\right).$$

Comme $1 + 2/3 = 1.666$, on trouve bien que $F_{k+2} > (3/2)^{k+1}$.

Soit k le plus petit entier tel que $a < (3/2)^k$. En fait, $k = \lceil \log_{3/2} a \rceil \leq 1.71n$. On a donc $0 < b < a < (3/2)^k < F_{k+1}$. Vu le lemme précédent, le calcul du PGCD va nécessiter $k - 1$ itérations, auxquelles il faut ajouter la première qu'on a déjà faite. Le nombre d'itérations est donc inférieur à $1.71n + 1$. □

Le nombre d'itérations est donc faible. Chaque itération effectue une division dont le coût est majoré par $\mathcal{O}(n^2)$, ce qui donnerait une complexité totale en $\mathcal{O}(n^3)$.

On peut cependant améliorer ceci, car les divisions deviennent de plus en plus rapides.

Lemme 4. *On considère l'exécution de l'algorithme $PGCD(a,b)$ avec $a > b$. On note $q_1, q_2, \ldots, q_k$ la séquence des* quotients *obtenus dans les divisions. Alors :*

$$q_1 \times \cdots \times q_k \leq a.$$

Démonstration

On démontre le résultat par recurrence sur a. Si $a = 1$, alors forcément $b = 0$, il a pas de division, et comme le produit vide est égal à 1 tout va bien.

Si $a > 1$, alors on pose la division de a par b :

$$a = bq_1 + r, \quad 0 \leq r < b$$

Les divisions qui ont lieu ensuite on lieu lors du calcul de $PGCD(b,r)$. Comme $b < a$, par hypothèse de récurrence on trouve que $q_2 \ldots q_k \leq b$.

Par conséquent, $q_1 \ldots q_k \leq bq_1 \leq a$. □

En combinant tous ces ingrédients, on est maintenant capable de prouver le théorème 6.

Démonstration

Diviser a par b prend un temps $\mathcal{O}(\log a \times \log b)$ comme on l'a vu au début de ces notes. Si la division s'écrit :

$$a = bq + r$$

Alors $\log a \leq \log b(q+1) = \log b + \log(q+1)$. Par conséquent, la complexité de la division de a par b est aussi $\mathcal{O}(\log b \times (1 + \log q))$.

On note $q_1, \ldots, q_k$ les quotients des divisions qui surviennent dans l'algorithme. La somme de leurs complexité est donc de l'ordre de :

$$\log b \times (1 + \log q_1 + 1 + \log q_2 + \cdots + 1 + \log q_k) = \log b \times (k + \log q_1 \ldots q_k)$$

On sait que $k = \mathcal{O}(n)$ et $\log q_1 \ldots q_k = \mathcal{O}(n)$. □

Version étendue

Une extension de l'algorithme d'euclide, donnée ci-dessous renvoie un triplet (x, y, g) tel que $ax + by = g = PGCD(a,b)$.

```
def XGCD(a, b):
    u = (1, 0)   # représentation de a
    v = (0, 1)   # représentation de b
    while b != 0:
       # comme la version de base pour l'instant
        q, r = divmod(a, b)
        a = b
        b = r
        # met à jour les représentations
        tmp = (u[0] - q*v[0], u[1] - q*v[1])
        u = v
        v = tmp
    return a, u[0], u[1]
```

Si on note a_0 et b_0 les valeurs des arguments lors de l'appel, alors on a toujours :

$$a = a_0 \times u[0] + b_0 \times u[1]$$
$$b = a_0 \times v[0] + b_0 \times v[1]$$

En effet, c'est bien le cas lors de l'initialisation de u et v. Ensuite, c'est bien le cas après la mise à jour des représentations. Comme b est copié dans a, on copie la représentation de b dans celle de

a. Et comme $r = a - qb$ est copié dans b, on fabrique la nouvelle représentation de b pour prendre le calcul de r en compte.

Éléments inversibles de $\mathbb{Z}_n$

On est enfin armé pour se pencher sur la question du calcul des inverses dans $\mathbb{Z}_n$. Prenons un élément $\alpha \in \mathbb{Z}_n$. S'il est premier avec n, alors leur PGCD vaut 1. L'algorithme d'Euclide étendu renvoie donc x et y tels que $\alpha x + ny = 1$. Il s'ensuit que

$$\alpha x \equiv 1 \mod n$$

Et donc x est l'inverse de α modulo n.

A contrario, si $PGCD(\alpha, n) = d > 1$, alors α n'a pas d'inverse dans $\mathbb{Z}_n$. Dans ce cas-là on écrit $\alpha = d\alpha'$ et $n = dn'$. S'il existait x tel que $\alpha x \equiv 1 \bmod n$, alors on aurait :

$$\begin{aligned} d\alpha' x &\equiv 1 \mod n \\ (dn')\alpha' x &\equiv 1 \\ 0 &\equiv 1. \end{aligned}$$

Ceci est manifestement impossible, donc α n'a pas d'inverse. Pour résumer :

Théorème 7. *Un élément $\alpha \in \mathbb{Z}_n$ est inversible si et seulement si il est premier avec n. Dans ce cas-là, on peut calculer son inverse avec l'algorithme d'euclide étendu.*

Cas particulier : $\mathbb{Z}_p$ avec p premier

Le cas où n est un nombre premier est particulièrement intéressant. En effet, si $0 < \alpha < p$, alors α et p ne peuvent pas avoir de diviseurs en commun puisque p, étant premier, n'a pas de diviseurs non-triviaux.

Ceci implique, grâce au théorème qui précède, que tous les éléments non-nuls de $\mathbb{Z}_p$ sont inversibles. Autrement dit, dans $\mathbb{Z}_p$ on dispose non seulement d'une multiplication, mais aussi d'une division.

Autant sur $\mathbb{Z}_n$ on peut faire grosso-modo les mêmes opérations que sur les entiers, autant sur $\mathbb{Z}_p$ on peut faire grosso-modo les mêmes opérations que sur les *rationnels*.

7.3 Exponentiation et logarithme modulo n

Quand a et k sont deux grands entiers, il est difficile de calculer a^k car le résultat prend trop de place (sa taille est k fois la taille de a). Cependant, $a^k \bmod n$, lui, est plus petit que n, même si k est arbitrairement grand. Mais comment faire pour le calculer ?

7.3.1 Exponentiation modulaire rapide

L'algorithme suivant a été exposé par le mathématicien persan غیات الدین جمشید کاشانی (Ghiyās-ud-dīn Jamshīd Kāshānī) (1380–1429), plus connu en France sous le nom d'al-Kashi (« le natif de Kashan »).

```python
def pow_mod(a, k, n):
    y = 1
    z = a
    l = k
    while l > 0:
        # ici, a^k est congru à y*z^l modulo n
        q, r = divmod(l, 2)
        if l % 2 != 0:
            y = (y * z) % n
        z = (z * z) % n
        l = q
    return y
```

L'invariant de la boucle est clairement vrai lors de l'entrée dans la boucle. Vérifions maintenant qu'il est maintenu par une itération. On effectue la division euclidienne de ℓ par 2 :

$$\ell = 2q + r$$

En fait, les valeurs évoluent de la façon suivante : $z' \leftarrow z^2, y' \leftarrow y \cdot z^r$ et $\ell' \leftarrow q$. Par conséquent :

$$\begin{aligned} y' \cdot z'^{\ell'} &\equiv y \cdot z^r \cdot \left(z^2\right)^q \mod n \\ &\equiv y \cdot z^{2q+r} \\ &\equiv y \cdot z^{\ell}. \end{aligned}$$

Par hypothèse, ceci vaut $a^k \bmod n$. CQFD.

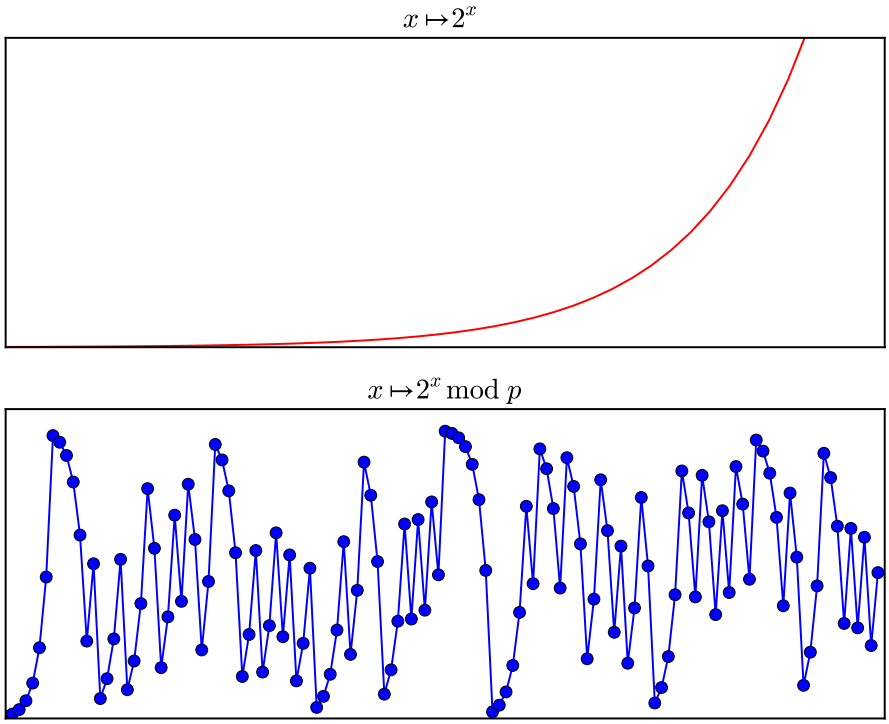
Enfin, sachez qu'en Python, ceci est calculé par la fonction prédéfinie `pow(a, k, n)`. Pour finir, cette procédure effectue au maximum $2 \log_2 k$ multiplications modulo n, et donc sa complexité est $\mathcal{O}\left(\log k \times \log^2 n\right)$. Elle est donc polynomiale en les tailles de k et n.

7.3.2 Logarithme « discret » modulo n

On a vu que la fonction « exponentielle modulo n » est facile à calculer. Qu'en est-il de sa réciproque, la fonction « logarithme modulo n » ? Le problème qui consiste à calculer x étant donné g et $g^x \% n$ est appelé le *problème du logarithme discret modulo n en base g*. Ici, « discret » ne signifie pas « qui n'attire pas trop l'attention », mais s'oppose à « continu ».

Ce problème est difficile. En effet, sur les entiers naturels, la fonction exponentielle est régulière. Étant donné deux entiers g et x, calculer x étant donné g et g^x est assez facile : x est environ le nombre de bits de g^x divisé par celui de g.

Mais la fonction « exponentielle modulo n » n'a pas du tout la même régularité.



En 2019, on ne connait pas d'algorithme efficace pour calculer le logarithme discret lorsque n est un grand nombre premier et lorsque g est choisi correctement. L'analogie avec la factorisation des grands entiers est frappante : la multiplication est facile, la factorisation est dure. L'exponentiation dans $\mathbb{Z}_p$ est facile, le logarithme est dur.

Evidemment, il existe des algorithmes dont la complexité augmente avec la taille de p. Mais ils ont en gros la même complexité que les algorithmes de factorisation. Il faut cependant se méfier : si n n'est pas premier et qu'il est très facile à factoriser, alors le logarithme discret modulo n n'est pas difficile.

Pour résumer, la fonction qui a x associe $g^x \bmod p$ est essentiellement à sens unique.

7.4 Comment génerer des nombres premiers ?

La production de clefs publiques nécessite des moyens de générer rapidement de grands nombres premiers connus de nous seuls. C'est le cas dans les systèmes RSA et ElGamal, entre autre. On a vu que si on choisit un nombre au hasard entre 1 et N, la probabilité qu'il soit premier est environ de $1/\log N$. Il « suffit » d'avoir un moyen efficace de tester si un grand nombre est premier, et au bout de $\log N$ tests on en obtiendra un, en moyenne.

Dans le fond, la solution repose sur la propriété suivante des nombres premiers :

Théorème 8 (Petit théorème de Fermat, 1601–1665)**.** *Si p est un nombre premier, alors $a^p \equiv a \bmod p$ pour tout entier a.*

Démonstration

D'abord, si $a = 0$ ou $a = 1$, alors c'est vrai. On se concentre donc sur les cas $a > 2$. Comme p est premier, le seul diviseur commun que a et p peuvent avoir est... p.

Si a est un multiple de p, alors $a \equiv 0 \bmod p$ et $a^p \equiv 0 \bmod p$, donc on a le résultat annoncé.

Si a n'est pas un multiple de p, alors $PGCD(a, p) = 1$. Considérons les nombres :

$$0 \ \% \ p, \quad a \ \% \ p, \quad 2a \ \% \ p, \quad \ldots, \quad (p-1)a \ \% \ p.$$

On prétend que ces nombres sont tous distincts. En effet, si on avait $ax\%p = ay\%p$, alors on trouverait $ax \equiv ay \bmod p$. Comme a est inversible, on aurait, en multipliant par l'inverse de a, $x \equiv y \bmod p$ (or ceci est impossible).

Puisque la liste ci-dessus est composée de p nombres distincts, et que le premier est zéro, alors les autres sont forcément $1, 2, \ldots, p-1$ mais dans un ordre qu'on ne connait pas. Du coup, en éliminant zéro et en multipliant les autres entre eux on trouve :

$$a \times 2a \times \cdots \times (p-1)a \equiv 1 \times 2 \times \cdot \times (p-1) \mod p$$

On multiplie des deux côtés par a et on rassemble les facteurs a côté gauche :

$$a^p \times 2 \times \cdots \times (p-1) \equiv a\Big(1 \times 2 \times \cdot \times (p-1)\Big) \mod p$$

Comme chacun des nombres $2, 3, \ldots, p-1$ est premier avec p, on peut tous les éliminer en multipliant par leurs inverses respectifs. On obtient donc :

$$a^p \equiv a \bmod p$$

□

Application L'intérêt de ce théorème est notamment le suivant. On veut tester si n est premier. Pour cela, on calcule $3^n \bmod n$. Si on ne trouve pas 3, c'est que n n'est *pas* premier. Si on trouve 3, alors n est *probablement* premier. Ce test est surprenament efficace. Il n'y a que 262 nombres inférieurs à 10^6 qui sont considérés à tort comme premier (c'est 91 le plus petit).

Comment faire pour améliorer la détection des composites ? Si un nombre n passe le test, alors on peut essayer de vérifier si $5^n \equiv 5 \bmod n$, par exemple. Cela élimine encore des candidats. Il n'y a plus que 59 nombres inférieurs à 10^6 qui passent les deux tests à tort.

On serait donc tenté d'écrire l'algorithme suivant :

```
def maybePrime(n, tries=32):
    for i in range(2, tries):
        if pow(i, n, n) != i:
            return False
    return True
```

Malheureusement ceci ne suffit pas. Il y a en effet des nombres composites qui passent le test, quel que soit le nombre d'essais choisis. Il s'agit des infâmes nombres de *Carmichael*. Le plus petit est 561. Il a été démontré en 1994 qu'il y en a une infinité, et même plus précisément, il y en a environ $n^{0.285}$ qui sont plus petits que n.

Il existe un algorithme plus sophistiqué, le test de Miller-Rabin, qui ne se trompe jamais quand il répond « composite », et qui se trompe avec probabilité $1/4$ quand il répond « premier ». Si on le répète k fois, il se trompe donc avec probabilité $1/4^k$.

Chapitre 8

Cryptographie basée sur le problème du logarithme discret

Ce chapitre décrit des techniques de cryptographie à clef publique dont la sécurité repose sur l'hypothèse que le problème du logarithme discret est difficile, ou autrement dit que la fonction exponentielle modulo p ($x \mapsto g^x \bmod p$) est à sens unique.

Pour fonctionner, ceci nécessite que les deux participants possèdent en commun un grand nombre premier p ainsi qu'un élément $g \in \mathbb{Z}_p$ (le « générateur »). Comme g est un élément de $\mathbb{Z}_p$, on n'écrira pas « modulo p » à chaque fois. Il est sous-entendu que les calculs effectués à partir de g restent dans $\mathbb{Z}_p$.

Tout le monde peut utiliser les mêmes valeurs de p et g. Il y a des valeurs recommandées dans certains standards. La RFC 3526 spécifie des valeurs possibles, comme par exemple :

$$p = 2^{2048} - 2^{1984} - 1 + 2^{64} \cdot \left(\left\lfloor 2^{1918} \cdot \pi \right\rfloor + 124476 \right)$$
$$g = 2$$

D'anciens standards contenaient des paramètres trop petits (768 bits, 1024 bits, ...) dont l'usage est déprécié. Si on est véritablement paranoïaque et qu'on pense que les standards sont moisis, on peut générer ses propres paramètres soi-même, à condition de faire un peu attention (cf. § 8.3).

Les paramètres décrivent une structure mathématique appelée un *groupe*, qu'on ne décrira pas précisément. Quand on parle « du groupe », on parle de g et p.

8.1 Échange de clef de Diffie-Hellman

Voyons comment faire de la cryptographie avec le problème du logarithme discret. Le protocole d'échange de clef de Diffie-Hellman est historiquement le premier mécanisme cryptographique publié qui ne nécessite pas de clef secrète. Il a été inventé en 1976 par Diffie (1944–) et Hellman (1945–). Il semble qu'il ait été également inventé un an plus tôt par les services secrets britanniques, mais cela a été gardé secret jusqu'en 1997. En 2019, l'échange de clef Diffie-Hellman est très largement utilisé (par exemple, c'est le cas quand je me connecte à ma banque).

Voici la description du protocole :

1. Alice choisit un nombre aléatoire $0 \leq x < p$. Elle calcule g^x et l'envoie à Bob.
2. Bob choisit un nombre aléatoire $0 \leq y < p$. Il calcule g^y et l'envoie à Alice.
3. Alice calcule $K \leftarrow (g^y)^x$ (modulo p, bien sûr).
4. Bob calcule $K \leftarrow (g^x)^y$ (idem).

À la fin du protocole, Alice et Bob possèdent donc en commun la valeur $K = g^{xy} \bmod p$. C'est essentiellement un élément aléatoire de $\mathbb{Z}_p$, si x et y sont choisis au hasard. Pour en tirer une clef secrète, on peut par exemple le hacher, ce qui permet l'utilisation ultérieure d'un procédé de chiffrement symétrique.

L'idée, c'est que d'éventuels adversaires passifs ne peuvent pas apprendre K en observant l'échange. En effet, ils voient passer g^x et g^y, mais à moins de pouvoir résoudre le logarithme discret dans $\mathbb{Z}_p$ ils ne peuvent pas apprendre x ou y. De plus, on ne connait pas d'algorithme efficace pour calculer g^{xy} étant donné g^x et g^y.

Attention au milieu. Sous cette forme de base, le protocole peut être victime d'une attaque par le milieu (cf. exercice 8.1). Une des solutions pour l'éviter consiste à ce que chaque participant signe les messages qu'il envoie. On parle alors d'échange de clef *authentifié*. Cependant la vérification des signatures nécessite la connaissance de la clef publique du partenaire (et donc, éventuellement un système de certificats).

Variante éphémère. Imaginons un serveur web sécurisé. De nombreux clients se connectent, et établissent une connexion sécurisée grâce à un échange de clef Diffie-Hellman authentifié. Ceci peut être réalisé de deux manières :

- Soit le serveur utilise le même g^x pour tout le monde. Dans ce cas-là, son g^x doit être authentifié par un certificat signé par une autorité. On parle de Diffie-Hellman *statique*.
- Soit le serveur génère un nouveau g^x à chaque nouvelle connection. Il doit alors le signer, et sa clef publique de vérification doit être authentifiée par un certificat. On parle de Diffie-Hellman *éphémère*.

La différence entre les deux, outre le temps de calcul pour le serveur, se trouve dans ce qui se passe si quelqu'un parvient à « casser » l'échange de clef, et à acquérir le nombre secret x du serveur.

Dans le cas statique, si quelqu'un parvient à récupérer x, alors il peut non seulement déchiffrer tous les échanges futurs du serveur, mais aussi les échanges *passés*. Ceci n'arrive pas avec le mode éphémère (cette propriété d'appelle la *Foward Secrecy*).

8.2 Chiffrement Elgamal

Est-ce qu'on peut faire du chiffrement à clef publique en « imitant » ce qui se passe dans l'échange de clef Diffie-Hellman ? Oui, comme cela a été démontré en 1985 par طاهر الجمل (Taher Elgamal) (1955–). Ce système a été largement implémenté dans les logiciels libres de crypto (GPG par exemple), car RSA était couvert par des brevets jusqu'à un passé récent. Le chiffrement Elgamal est du coup la principale alternative à RSA de nos jours.

Génération des clefs. Pour produire sa paire de clefs, Alice :

- choisit au hasard un exposant secret $0 \leq x < p$.
- calcule $h \leftarrow g^x$ (modulo p, bien sûr).

La partie publique de la clef est la description du groupe (p et le générateur g) ainsi que h. La clef secrète est x.

Chiffrement. Pour envoyer un message chiffré à Alice, Bob doit d'abord le coder comme un élément non-nul $m \in \mathbb{Z}_p$. Ensuite Bob :

- Choisit au hasard un nombre $0 \leq y < p$.
- Fabrique son chiffré $C \leftarrow \left(g^y, mh^y\right)$ et l'envoie à Alice :

L'idée c'est que le message m est « masqué » (comme avec un masque jetable) par $h^y = g^{xy}$. C'est ici que l'analogie avec l'échange de clef DH apparaît. Le chiffrement est non-déterministe. Sauf

dans de rares situations, c'est un avantage. Par contre, le chiffrement est *malléable* : étant donné les chiffrement de u et v, on peut produire celui de uv :

$$C(u) = (g^y, uh^y)$$
$$C(v) = (g^z, vh^z)$$

Si on les multiplie entre eux, coordonnée par coordonnée, on trouve (g^{y+z}, uvh^{y+z}), c'est-à-dire ce qu'on obtiendrait en chiffrant uv avec l'aléa $y + z$. Cette malléabilité peut être un problème dans certaines circonstances, et c'est pourquoi des précautions sont prises pour empêcher qu'elle ne pose trop de problèmes (cf. § 10.3).

Déchiffrement. Après avoir reçu le message chiffré $C = (a, b)$ de Bob, Alice peut :
— Calculer $h \leftarrow a^x$, qui vaut donc g^{xy}.
— Calculer h^{-1}, c'est-à-dire l'inverse de h dans $\mathbb{Z}_p$.
— calculer bh^{-1}, qui vaut m en principe.

On verra plus tard que le chiffrement Elgamal est sûr sous des hypothèses raisonnables (cf. § 9.2).

8.3 Comment fabriquer ses propres paramètres ?

On a vu comment fabriquer de grands nombres premiers. Par contre, il faut faire attention en fabriquant le « générateur » pour Diffie-Hellman où Elgamal. Voyons le problème qui peut se poser à travers un exemple. On choisit $p = 181$ (c'est bien un nombre premier) et $g = 42$.

Alice choisit $x = 118$ et calcule $g^x = 59$.

Bob choisit $y = 33$ et calcule $g^y = 59$.

Coincidence ? Pour s'en rendre compte, on calcule g^i pour plusieurs valeurs de i :

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
g^i	42	135	59	125	1	42	135	59	125	1	42	135	59	125	1	42	135

Et là, c'est l'horreur : il semble que g^i ne puisse prendre que 5 valeurs distinctes. Du coup, il n'y aurait que 5 secrets partagés possibles dans l'échange de clefs DH et seulement 5 « masques » différents possibles dans le chiffrement Elgamal. Autant dire que ça ne chiffre pas beaucoup.

On note $\langle g \rangle = \{g^i \mid i \in \mathbb{N}\}$ — g est le « générateur » d'un tel ensemble [1]. Il semble ici que la taille de $\langle 42 \rangle$ soit égale à 5, alors que $\mathbb{Z}_{181}$ possède beaucoup plus d'éléments.

En fait, le problème vient de ce que $g^5 = 1$. En effet, pour voir ce que vaut g^i, posons la division de i par 5 : $i = 5q + r$. On a alors :

$$g^i = g^{5q+r} = \left(g^5\right)^q \cdot g^r = g^r.$$

Et on voit bien que g^i ne peut prendre que 5 valeurs, de façon cyclique.

8.3.1 Ordre d'un élément modulo p

Définition : Ordre d'un élément modulo p

> Si $0 < a < n$ est premier avec n (donc si $a \in \mathbb{Z}_n$ est inversible), alors **l'ordre de a modulo n** est le plus petit entier $\lambda > 0$ tel que $a^\lambda \equiv 1 \bmod n$.

L'intérêt principal de cette notion d'ordre, par rapport à nos préoccupations, est exprimé dans le résultat suivant :

Théorème 9 (« passage à l'exponentielle »)**.** *Si a est inversible dans $\mathbb{Z}_n$ et que son ordre est q, alors on a :*

$$u \equiv v \bmod q \qquad \Longleftrightarrow \qquad a^u \equiv a^v \bmod n.$$

Démonstration

1. $\langle g \rangle$ est techniquement le sous-groupe de $\left(\mathbb{Z}_p^*, *\right)$ engendré par g...

$\Longrightarrow$ Par définition, on a $u - v = kq$ pour un certain entier k. On a donc :

$$\begin{aligned} a^u &\equiv a^{kq+v} \mod n \\ &\equiv (a^q)^k \cdot a^v \\ &\equiv a^v. \end{aligned}$$

$\Longleftarrow$ Il nous faut montrer que $u-v$ est un multiple de q. On suppose sans perte de généralité que $u \geq v$ (on peur les échanger si ce n'est pas le cas). Comme on a $a^u \equiv a^v \mod n$ et que a est inversible, on multiplie le tout par l'inverse de a puissance v :

$$a^{u-v} \equiv 1 \mod n.$$

Posons la division euclidienne de $u - v$ par q :

$$u - v = kq + r, \qquad 0 \leq r < q.$$

On va montrer que le reste r est nul, ce qui prouvera le résultat annoncé. Pour cela on écrit :

$$\begin{aligned} a^{u-v} &\equiv 1 \mod n \\ a^{kq+r} &\equiv 1 \\ (a^q)^k \cdot a^r &\equiv 1 \\ a^r &\equiv 1 \end{aligned}$$

Comme $r < q$ par définition de la division, et que q est par définition le plus petit entier *non-nul* tel que $a^q \equiv 1 \mod p$, on est forcé de conclure que $r = 0$. □

Pour que le logarithme discret soit vraiment difficile, il faut trouver un « générateur » g dont l'ordre soit très élevé. En effet, le nombre de valeurs différentes que peut prendre g^x est précisément l'ordre de g.

Lemme 5. *Pour $g \in \mathbb{Z}_p$, la taille de $\langle g \rangle$ est précisément l'ordre de g modulo p.*

Démonstration

Soit q l'ordre de g modulo p. Le theorème 9 nous dit que $g^i = g^{(i \ \% \ q)}$, donc clairement g^i ne peut pas prendre plus de q valeurs différentes.

Maintenant on va voir que pour tout $0 \leq i < j < q$, $g^i \neq g^j$. En effet, si on avait $g^i = g^j$ dans ces conditions, alors d'après le théorème 9 on aurait $i \equiv j \mod q$, ce qui est impossible car ils sont différents et plus petits que q.

8.3.2 Racines primitives modulo p

Il nous faut donc des éléments de grand ordre. Mais y en a-t-il ? Et si oui, comment les trouver ? Tout d'abord, le petit théorème de Fermat implique que l'ordre d'un élément modulo p est inférieur ou égal à $p - 1$. Les élements qui auraient cet ordre sont donc particulièrement intéressants.

Définition : Racine primitive modulo p

Si p est un nombre premier, alors on dit que $a \in \mathbb{Z}_p$ est une **racine primitive** modulo p si son ordre est $p - 1$.

En gros, les racines primitives sont les éléments d'ordre le plus élevé possible. Si g est une racine primitive, cela signifie que g^i prend toutes les valeurs possibles dans $1, 2, \ldots, p - 1$. La bonne nouvelle, c'est qu'il y en a toujours.

Théorème 10. *Si p est un nombre premier, alors il y a des racines primitives modulo p.*

On admet ce résultat, dont la preuve nous emmenerait un peu trop loin. En fait, la réalité se présente plutôt favorablement, car l'écrasante majorité des élements de $\mathbb{Z}_p$ sont en fait des racines primitives (environ $p/\log\log p$ en sont).

Alors, on peut espérer trouver une racine primitive modulo p en choisissant un élément au hasard, puis en vérifiant que son ordre est bien $p-1$. On espère s'en tirer en $\mathcal{O}(\log\log p)$ essais. Si l'hypothèse de Riemann généralisée était vraie, alors on aurait la certitude qu'il existe toujours une racine primitive modulo p plus petite que $\mathcal{O}\left(\log^6 p\right)$.

Mais comment faire pour tester si un élément g est une racine primitive ? De manière générale, déterminer l'ordre d'un élément de $\mathbb{Z}_n$ est au moins aussi dur que factoriser n, donc ça ne se présente pas bien. On pourrait vérifier que $g^i \neq 1$ pour toutes les valeurs de i... Mais ça prendrait vraiment trop longtemps si on voulait s'assurer que g est d'ordre 2^{1000}, par exemple. Heureusement, on peut s'épargner une bonne partie des tests grâce à l'observation suivante.

Lemme 6. *L'ordre d'un élément non-nul $a \in \mathbb{Z}_p$ est un diviseur de $p-1$. Plus généralement, si $a^k \equiv 1 \bmod p$, alors l'ordre de a est un diviseur de k.*

Démonstration

Montrons d'abord que la deuxième partie du résultat annoncé entraine la première (avec $k = p-1$). En effet, En effet, le petit theorème de Fermat affirme que $a^p \equiv a \bmod p$. Comme a n'est pas nul, il est forcément inversible modulo p, et on multiplie l'équation précédente par l'inverse de a des deux côtés pour obtenir : $a^{p-1} \equiv 1 \bmod p$.

Démontrons maintenant la deuxième partie. Supposons que $a^k \equiv 1 \bmod p$, notons q l'ordre de a modulo p. D'après le théorème 9 (« passage à l'exponentielle »), on trouve donc $k \equiv 0 \bmod q$, autrement dit q divise k. □

Ceci permet de tester un peu plus efficacement si un nombre est une racine primitive ou pas :

Lemme 7. *Les deux conditions suivantes sont équivalentes :*

i) $a \neq 0$ *est d'ordre q modulo p*

ii) $a^q \equiv 1 \bmod p$ *et* $a^{q/d} \not\equiv 1 \bmod p$ *pour tout nombre premier d qui divise q.*

Démonstration

$i \Rightarrow ii$: c'est facile. Cela découle de la définition de l'ordre, et du fait que q/d est plus petit que q.

$\neg i \Rightarrow \neg ii$: c'est le sens intéressant (et moins facile). Supposons donc que a n'est pas d'ordre q, et montrons que $a^q \not\equiv 1 \bmod p$ ou qu'il existe un nombre premier d qui divise q et pour lequel $a^{q/d} \equiv 1 \bmod p$).

Notons λ l'ordre de a, supposons que $\lambda \neq q$, que $a^q \equiv 1 \bmod p$ et montrons qu'il existe bien un diviseur premier d de q tel que $a^{q/d} \equiv 1 \bmod p$. D'après le lemme 6, on sait que λ divise q, et comme ce n'est pas q, c'est un diviseur strict de q. Autrement dit, il existe q' tel que $q = \lambda q'$, avec $q' > 1$. Considérons un diviseur premier d de q' (il y en a, cf. théorème 3), et vérifions qu'il fait l'affaire : notons $q' = q'' \times d$, et alors on a :

$$a^{q/d} = a^{\lambda q''} = \left(a^{\lambda}\right)^{q''} = 1 \qquad \text{(car } a \text{ est d'ordre } \lambda\text{)}.$$

CQFD.

Comment générer des racines primitives ? Le lemme 7 donne des critères permettant de vérifier qu'un générateur donné a bien un ordre donné. Il peut donc servir à tester si un générateur est une racine primitive, en vérifiant qu'il est bien d'ordre $p-1$. La difficulté, c'est qu'il faut connaître la factorisation de $p-1$ en facteurs premiers. Et si p est un nombre aléatoire de 2000 bits, calculer la factorisation de $p-1$ semble infaisable.

Pour être sûr d'avoir une racine primitive, plusieurs solutions sont possibles. L'une d'entre elles consiste à choisir un nombre premier p de telle sorte qu'on connaisse la factorisation de $p-1$. Le problème, c'est que s'assurer du caractère aléatoire et imprévisible de p devient plus compliqué, même si c'est possible.

Une solution plus simple consiste à utiliser des *nombres premiers sûrs*. On choisit au hasard un nombre premier p tel que $q = (p-1)/2$ est aussi premier. Alors, il est facile de tester si des élements de $\mathbb{Z}_p$ sont des racines primitives : la factorisation de $p-1$ en facteurs premiers est $p-1 = 2 \times q$. Du coup, un élément $a \in \mathbb{Z}_q$ est une racine primitive modulo q si et seulement si $a^2 \not\equiv 1 \bmod p$ et $a^q \not\equiv 1 \bmod p$.

Le problème est que tout ceci n'est pas très efficace (la génération d'un gros nombre premier sûr est assez lente, disons plusieurs minutes un ordinateur normal en 2019).

8.3.3 Générations efficace de paramètres offrant une sécurité suffisante

En fait, dans bon nombre de situations il n'est pas indispensable d'être sûr que le générateur est une racine primitive. On peut se contenter de la certitude que son ordre est « suffisamment grand ». Pour cela, une possibilité consiste à fabriquer un nombre premier p de telle sorte que $p-1$ est un multiple de q, pour q assez grand.

1. Choisir l'ordre q désiré, qui doit être un nombre premier d'au moins 256 bits en 2019.
2. Choisir un nombre uniformément aléatoire k de la bonne taille (1792 bits, par exemple).
3. Si $p = 1 + kq$ n'est pas premier, revenir à l'étape 2.
4. Choisir un élément $g \in \mathbb{Z}_p$ de manière arbitraire.
5. Si $g \neq 1$ et que $g^q \equiv 1 \bmod p$, alors g est d'ordre au moins q. Sinon, retourner à l'étape 4.
6. Si on veut que g soit d'ordre exactement q, il suffit de vérifier en outre que $g^k \not\equiv 1 \bmod p$.

Rien n'interdit d'utiliser des ordres q composites, mais ceci a plusieurs défauts : d'abord l'arithmétique modulo q est plus compliquée car tous les exposants ne sont pas inversibles modulo q, et d'autre part cela affaiblit généralement la sécurité (cf. ci-dessous).

8.3.4 Sécurité des paramètres

Sans trop rentrer dans les détails, la sécurité des algorithmes de calcul du logarithme discret dépendent de deux paramètres : la taille de p et la valeur de l'ordre.

Une première famille d'algorithmes (le « calcul d'index ») a une complexité qui est la même que celle qui consiste à factoriser un entier de la même taille que p. Ceci nécessite donc que p soit assez grand (2048 bits minimum en 2019).

D'autres algorithmes, plus simples, peuvent marcher si l'ordre a un problème : si l'ordre est très faible, une recherche exhaustive peut en venir à bout. D'autre part, il existe des algorithmes génériques dont la complexité est en $\mathcal{O}\left(\sqrt{q}\right)$ (la méthode ρ ou bien l'algorithme *baby steps / giant steps*. Ceci implique donc de s'assurer que l'ordre est un nombre de 256 bits au moins en 2019.

Enfin, si l'ordre est composite et s'écrit $q = q_1 \cdot q_2$, il faut savoir que calculer un logarithme peut se ramener à un calcul avec un ordre q_1 et un calcul avec un ordre q_2 (il s'agit de l'algorithm de Pohlig-Hellman). Si l'ordre est *friable* (n'a que de petits diviseurs), alors le calcul peut être facile. Les nombres premiers sûrs évitent ce problème, de même que le choix d'un ordre premier.

8.4 Signature de Schnorr

L'idée de départ du chiffrement Elgamal est que la connaissance de g^x permet le chiffrement, tandis que x permet le déchiffrement. Ici, il nous faudrait l'inverse : un mécanisme où x permette de signer, tandis que g^x permette de faire la vérification.

La méthode de signature numérique la plus simple reposant sur le problème du logarithme discret est sûrement la signature de Schnorr, décrit par Claus-Peter Schnorr en 1989. Il a été peu utilisé concrètement car il était protégé par un brevet jusqu'à 2008.

Le procédé de signature est dérivé d'un protocole d'authentification interactif. Il fonctionne dans un groupe muni d'un générateur d'ordre premier suffisament grand.

8.4.1 Protocole d'authentification de Schnorr

Protocole S (*preuve de connaissance d'un exposant*). Un prouveur P (Peguy) connaît un exposant secret x et il a publié son exponentielle $h = g^x$. Le protocole lui permet convaincre un vérifieur

V (Victor) qu'elle connaît bien x, mais sans lui révéler (« *Zero-Knowledge Proof of Knowledge* »). Ceci permet notamment à Peguy de démontrer son identité à Victor.

S1. [Mise en gage.] Peguy choisit aléatoirement un exposant secret r (donc un nombre non-nul modulo q), calcule $R = g^r$ puis transmet R à Victor.

S2. [défi.] Victor choisit aléatoirement un exposant c puis l'envoie à Peguy.

S3. [réponse.] Peguy calcule l'exposant $a \equiv r - cx \bmod q$ et l'envoie à Victor.

Après l'étape S3, Victor vérifie si $R \equiv g^a \cdot h^c \bmod p$. Si c'est le cas, il est convaincu que Peguy connaît bien x. Pendant une exécution du protocole, seuls R, c et a transitent entre les participants. On note (R, c, a) une *transcription* du protocole. Elle est *correcte* si $R \equiv g^a \cdot h^c \bmod p$.

8.4.2 Sécurité du protocole

Ce protocole possède deux propriétés de sécurité intéressantes. On peut d'abord démontrer que réussir à tromper Victor le vérifieur (c.a.d. le convaincre qu'on connaît le logarithme de h alors qu'on ne le connaît pas) est aussi dur que le problème du logarithme discret. Dans un deuxième temps, on démontre que ni Victor ni d'éventuels adversaires passifs ne peuvent apprendre d'information sur le secret.

Tricher dans le protocole est dur. Supposons qu'on dispose d'un algorithme déterministe efficace $\mathcal{A}$, capable d'exécuter correctement le protocole (à la place du prouveur). Cet algorithme prend h en entrée, mais pas x, et il reçoit une source de bits aléatoires en provenance de l'extérieur pour effectuer ses choix aléatoires.

On va montrer qu'on peut utiliser cet algorithme pour calculer le logarithme discret de h, c'est-à-dire reconstituer x. Il s'ensuit que « tricher » dans le protocole est au moins aussi difficile de calculer le logarithme discret.

L'idée consiste à exécuter $\mathcal{A}$ deux fois en lui fournissant les mêmes bits aléatoire. Par conséquent, chaque fois qu'on va exécuter $\mathcal{A}$, il va produire le même nombre aléatoire r et donc la même mise en gage R. On lui envoie deux défis différents $c \neq c'$, et on suppose que $\mathcal{A}$ produit deux réponses correctes $a \neq a'$. On obtient donc :

$$\begin{aligned} R &\equiv g^a \cdot h^c \mod p, \\ R &\equiv g^{a'} \cdot h^{c'} \mod p. \end{aligned}$$

On divisant ces deux congruences, on trouve $1 \equiv g^{a-a'} \cdot h^{c-c'} \bmod p$. D'après le théorème 9 (« passage à l'exponentielle »), et comme $h \equiv g^x \bmod p$, ceci implique $0 \equiv (a-a') + x(c-c') \bmod q$. On en déduit que $x \equiv (a-a') \cdot (c-c')^{-1} \bmod q$ est le logarithme de h dans le groupe, et qu'il est facile à calculer à partir de la sortie de $\mathcal{A}$.

Aucune information sur le secret n'est révélée. Que peut apprendre le vérifieur ? Que voient passer des adversaires passifs qui espionnent le protocole ? L'un et les autres observent des transcriptions correctes du protocole. On va montrer ici qu'il est très facile de produire des transcriptions correctes *sans posséder le secret* x.

Les adversaires « apprennent » donc des informations qu'on peut facilement produire sans posséder le secret x... et donc par conséquent ils n'apprennent *aucune* information sur x.

Pour produire une transcription correcte, c'est très simple : choisir un défi c et une réponse a uniformément au hasard modulo q. Calculer R de telle sorte que la transcription soit correcte, en posant $R = g^a h^c$. Grace au choix aléatoire de a (et c), on sait que R est uniformément distribué, comme s'il avait été choisi pendant une exécution normale du protocole.

8.4.3 Signature de Schnorr

En gros, ce qui fait qu'on ne peut pas tricher dans le protocole de Schnorr, c'est qu'on doit choisir R sans connaître le défi c. C'est donc le côté « interactif » qui bloque les tricheurs potentiels.

Pour transformer ceci en schéma de signature (non-interactif) on utilise une technique standard, la *transformation de Fiat-Shamir*, qui consiste à générer le challenge de manière déterministe mais imprévisible avec une fonction de hachage. La clef (publique) de vérification est $h = g^x$, la clef (secrète) de signature est x.

Pour signer un message $M \in \{0,1\}^*$:

1. Choisir r uniformément au hasard modulo q, calculer $R \leftarrow g^r \bmod p$.
2. Calculer le « défi », $c \leftarrow H(R \parallel M) \bmod q$.
3. Calculer la « réponse », $a \leftarrow r - cx \bmod q$.
4. La signature est (c, a).

Signer un message M revient à « démontrer » qu'on connaît x, avec un défi généré à partir de M : moralement, on ne peut le faire que si on connaît x (sinon, qu'est-ce qui nous empêcherait de tricher au protocole de schnorr ?).

Pour vérifier si (c', a') est bien une signature de M, on effectue la vérification du protocole d'authentification, avec un détail : on ne connaît pas R, mais on peut le reconstituer à partir du « défi » et de la « réponse ».

1. Calculer $R' \leftarrow g^{a'} h^{c'} \bmod p$.
2. La signature est valide si $c' \equiv H(R' \parallel M) \bmod q$.

Une signature de Schnorr est constituée de deux nombres modulo q, donc si q occupe 256 bits, une signature de Schnorr occupe 64 octets.

8.5 Signature ElGamal

> Historiquement, la signature ElGamal est le premier schéma à clef publique qui n'est pas complètement trivial.
>
> Phong Q. Nguyen

Est-ce qu'on peut utiliser le chiffrement Elgamal pour faire des signatures ? On ne peut pas vraiment utiliser la technique « signer = déchiffrer, vérifier = re-chiffrer ». Si la signature de M est le déchiffrement de M, et qu'on re-chiffre la signature, on obtiendra un résultat qui contient de l'aléa, donc qui n'a aucune chance d'être M.

Pour faire des signatures avec les mêmes idées que le chiffrement Elgamal, il faut ruser. La signature Elgamal est largement utilisée elle aussi. Elle a été standardisée par le gouvernement américain sous le nom de DSA (« Digital Signature Algorithm »), et elle a été largement déployée par exemple dans le logiciel SSH pour l'authentification sans mot de passe (en effet, la signature RSA était elle aussi couverte par des brevets). Maintenant, ECDSA (« *Elliptic Curves Digital Signature Algorithm* ») est encore plus largement utilisée .

La génération des clefs est identique à celle du chiffrement. On choisit un grand nombre premier p, et un générateur g d'ordre q (on suppose que l'ordre est elevé — si g est une racine primitive, son ordre est $p-1$).

Pour signer un message $m \in \mathbb{Z}_p$:

1. Choisir au hasard un élément *inversible* $k \in \mathbb{Z}_q$
2. Calculer $r \leftarrow g^k$
3. Calculer $s \leftarrow (m - xr)k^{-1} \bmod q$
4. Si $s = 0$, retourner à l'étape 1.
5. La signature est (r, s)

Pour vérifier que (r, s) est une signature de $m \in \mathbb{Z}_p$:

1. Vérifier que $0 < r < p$ et $0 < s < q$.
2. Vérifier que $g^m \equiv h^r r^s \bmod p$.

Note importante : il faut *toujours* choisir un nouveau k à chaque signature. Tout échec est fatal. Par exemple, les ingénieurs de Sony avaient apparemment mal compris la spécification de ce schéma de signature, car sur la console de jeu Playstation 3, les jeux sont signés avec ECDSA, et la console de charge que les jeux signés, donc « approuvés » par Sony. Seulement, tous les jeux ont été signés avec le même r... Et ceci permet de retrouver facilement la clef secrète !

Exercices

manque : rho, baby steps / giant steps / pohlig hellmann ? manque protocole SRP manque forgery existentielle sur signature Elgamal manque Naor-Pinkas OT et/ou Vitse OT.

Exercice 8.1 : Quel problème affecte le protocole d'échange de clef Diffie-Hellman si les messages ne sont pas authentifiés ?

Exercice 8.2 : On considère le mécanisme de chiffrement *symétrique* $\mathcal{E}(k, m) = m^k \bmod p$, pour un nombre premier p fixé et commun à tous. Les messages appartiennent à $\mathbb{Z}_p$. Ce mécanisme de chiffrement possède la particularité que $\mathcal{E}(k_1, \mathcal{E}(k_2, m)) = \mathcal{E}(k_2, \mathcal{E}(k_1, m))$.

Comment peut-on effectuer le déchiffrement ?

Exercice 8.3 : Bob est un opposant dans un régime répressif qui reçoit des messages avec Alice, une journaliste étrangère. Ils partagent une clef symétrique et s'échangent des messages chiffrés, car Bob a été mis sur écoute. En plus, il craint d'être enlevé par la police politique puis torturé jusqu'à ce qu'il révèle la clef symétrique. Les bourreaux vérifieraient alors que la clef est valide en déchiffrant tous les messages précédemment échangés avec Alice et en vérifiant s'ils ont un sens.

Concevez un mécanisme de chiffrement symétrique qui permet d'exprimer sa véritable opinion mais où, en cas d'urgence, on peut justifier de manière plausible que les messages précédemment chiffrés étaient favorables au régime.

Indice : il faut prévoir deux mécanismes de déchiffrement : un normal et un autre pour les cas d'urgence.

Exercice 8.4 : On considère un (mauvais) schéma de mise en gage basé sur le logarithme discret :

$$\textsc{Commit}(b) := g^b \bmod p,$$

Justifier que ceci est *perfectly binding*. Est-ce que le bit mis en gage est vraiment dissimulé ?

Exercice 8.5 : La première manière d'améliorer le mauvais mécanisme de l'exercice 8.4 repose sur le chiffrement Elgamal :

$$\textsc{Commit}(r, m) := (g^r, mh^r)\,,$$

où $r \in \mathbb{Z}_q$ est aléatoire, q est l'ordre du générateur g, $m \neq 0$ est le message à mettre en gage ($m \in \mathbb{Z}_p^*$) et h est un autre générateur du groupe engendré par g modulo p ($h = g^x$ pour un certain x et h a le même ordre que g). On n'essaiera pas de mettre en gage $m = 0$, pour des raisons évidentes.

Justifier que ceci est *perfectly binding*. Est-ce *computationally hiding* ?

Exercice 8.6 : La deuxième manière d'améliorer le mauvais mécanisme de l'exercice 8.4 donne des propriétés symétriques. On considère le schéma de mise en gage inventé par Torben Pryds Pedersen en 1991 :

$$\textsc{Commit}(r, m) := g^m h^r \bmod p,$$

où r est aléatoire ; m est le message à mettre en gage et h est un autre générateur du groupe engendré par g modulo p (il faut ici que l'ordre de q soit un nombre premier). Justifier que ceci est *statistically hiding* et *computationally binding*.

Exercice 8.7 : Que se passe-t-il si on fait deux signatures Elgamal avec le même aléa ?

Exercice 8.8 : On considère le protocole d'échange de clef authentifié par mot de passe (PAKE) suivant :

1. L'étudiant génère une paire de clefs Elgamal (avec un générateur g qui est une racine primitive modulo p), et envoie son nom ainsi que $\{pk\}_{pwd}$ à la borne wifi.
2. La borne wifi déchiffre pk avec le bon mot de passe, génère une clef de session aléatoire K et renvoie le chiffré Elgamal de K avec la clef publique reçue (autrement dit la borne répond $\{K\}_{pk}$).
3. L'étudiant et la borne continuent les échanges en chiffrant tout avec K.

Ce protocole est-il résistant aux attaques hors-ligne par dictionnaire ?

Exercice 8.9 : SPEKE (Simple Password Exponential Key Exchange) est un protocole d'échange de clef authentifié par mot de passe (PAKE). Il était protégé par un brevet jusqu'à mars 2017. Alice et Bob possèdent en commun un mot de passe pwd et veulent s'en servir pour établir une clef de session symétrique. Pour cela, ils choisissent un nombre premier *sûr* p, et effectuent l'échange de clef Diffie-Hellman en utilisant $g \leftarrow H(pwd)^2 \bmod p$.

Justifier que g est d'ordre $(p-1)/2$ avec très forte probabilité. Le protocole résiste-t-il aux attaques hors-ligne par dictionnaire ?

Exercice 8.10 : Démontrer que la propriété « être un nombre premier » est dans NP. Pour cela, on peut démontrer puis utiliser le résultat suivant :

« *S'il existe un nombre a d'ordre $n-1$ modulo n, alors n est un nombre premier.* »

Chapitre 9

Résiduosité quadratique modulo p et sécurité du chiffrement Elgamal

Est-ce que l'échange de clef Diffie-Hellman, le chiffrement Elgamal, la signature Elgamal, sont sûrs ? Il est évident que si on sait résoudre le problème du logarithme discret, on peut casser tous ces schémas.

Résoudre le problème du logarithme discret est donc *au moins aussi dur* que casser l'échange de clef de Diffie-Hellman ou que casser le chiffrement Elgamal. La question importante est la réciproque : est-ce que casser le protocole d'échange de clef Diffie-Hellman est « *aussi dur* » que résoudre le problème du logarithme discret ?

La réponse est : « ça dépend » (si on remplace $\mathbb{Z}_p$ par l'ensemble des points sur une courbe elliptique bien choisie, ça peut). En l'état actuel (en 2019), on ne sait pas si c'est vraiment équivalent ou pas sur $\mathbb{Z}_p$.

9.1 Sécurité de l'échange de clef Diffie-Hellman

Du coup, la solution consiste à faire de la sécurité de Diffie-Hellman un « problème » calculatoire à part entière :

Définition : Problème Diffie-Hellman calculatoire

> Soit p un nombre premier et $g \in \mathbb{Z}_p$. Un ensemble de trois éléments x, y, z de $\mathbb{Z}_p$ est appelé un **triplet Diffie-Hellman** s'il existe a et b tels que $x \equiv g^a \bmod p$, $y \equiv g^b \bmod p$ et $z \equiv g^{ab} \bmod p$.
>
> Le problème **Computational Diffie-Hellman** (CDH) consiste, étant donné p, g et deux éléments x et y appartenant au groupe, à trouver z tel que (x, y, z) forme un triplet DH.
>
> L'« hypothèse CDH » consiste à supposer que tout algorithme polynomial résoud CDH avec une probabilité de succès négligeable.

Si on sait résoudre CDH, on sait casser l'échange de clef de Diffie-Hellman, car on sait reconstituer le secret commun. Dans tous les groupes où on fait de la cryptographie, le meilleur algorithme pour résoudre CDH consiste à calculer des logarithmes discrets. La version décisionnelle du problème CDH joue également un rôle important en cryptographie.

Définition : Problème Diffie-Hellman décisionnel

> Le problème **Decisional Diffie-Hellman** (DDH) consiste, étant donné trois éléments de $\mathbb{Z}_p$, à déterminer s'il s'agit d'un triplet Diffie-Hellman.
>
> L'« hypothèse DDH » consiste à supposer que tout algorithme polynomial résoud DDH avec un avantage négligeable.

L'intérêt de ce problème est le suivant. J'ai espionné un échange de clef DH entre Alice et Bob, donc j'ai appris g^x et g^y. J'essaye de calculer g^{xy}, c'est-à-dire de résoudre CDH. Le service « action » de l'agence que je dirige a volé l'ordinateur de Bob, et me présente une chaine de bits trouvée

dessus qui est peut-être g^{xy}. Est-ce que je peux me rendre compte que c'est bien la solution que je cherche ? Ce n'est pas toujours facile.

Plus précisément, on peut concevoir un jeu dans lesquels un adversaire reçoit ou bien un triplet Diffie-Hellman, ou bien un triplet aléatoire. Il doit répondre 1 s'il détermine qu'il s'agit d'un triplet DH.

```
def experiment(E, p, g, b, adversary):
    x = random.randrange(p-1)
    y = random.randrange(p-1)
    if b == 0:
        z = random.randrange(p-1)
    else:
        z = x*y
    u = pow(g, x, p)
    v = pow(g, y, p)
    w = pow(g, z, p)
    return adversary(p, g, (u, v, w))
```

L'avantage de l'adversaire est, comme d'habitude :

$$\mathbf{Adv}^{\mathsf{DDH}}(p, g, \mathcal{A}, n) = \mathbb{P}\left[\texttt{experiment}(p, g, 1, \mathcal{A}) = 1\right] - \mathbb{P}\left[\texttt{experiment}(p, g, 0, \mathcal{A}) = 1\right]$$

Très clairement, CDH est plus dur que DDH (si on sait résoudre CDH, on peut facilement résoudre DDH). Il y a des situations dans lesquelles DDH est facile mais CDH est dur.

9.2 Sécurité du chiffrement Elgamal

Dans cette section, on va examiner les conditions sous lesquelles le chiffrement Elgamal est « sûr », et on va préciser la signification de cette notion.

Résistance de la clef publique Tout d'abord, récupérer la clef secrète à partir de la clef publique revient à calculer x à partir de $g^x \bmod p$, donc à casser le logarithme discret dans $\mathbb{Z}_p$. Aucun algorithme connu ne peut le faire si p et g sont bien choisis (cf. § 8.3.4). C'est un bon début.

Impossibilité du déchiffrement complet Ensuite, il faut bien sûr qu'il soit (calculatoirement) impossible de déchiffrer un message sans posséder la clef secrète. Dire ceci, c'est dire que la fonction de chiffrement doit être à sens unique (tant qu'on ne possède pas la clef secrète).

Définition : Sens unique

> On considère le jeu suivant, qui se joue entre un adversaire $\mathcal{A}$ (qui cherche à casser le chiffrement) et un challenger $\mathcal{C}$ (qui « teste » l'adversaire) :
>
> 1. $\mathcal{C}$ fabrique une paire de clefs (pk, sk) de paramètre de sécurité n, et donne pk à $\mathcal{A}$.
> 2. $\mathcal{C}$ choisit un message aléatoire M et envoie M à $\mathcal{A}$.
> 3. $\mathcal{A}$ gagne s'il devine M correctement.
>
> Le système de chiffrement est **à sens unique** (il possède la « *One-Wayness under Chosen Plaintext Attacks* » — OW-CPA) si tout adversaire fonctionnant en temps polynomial ne peut gagner qu'avec une probabilité négligeable (en fonction de n).

On avait construit le chiffrement Elgamal à partir de l'échange de clef Diffie-Hellman, donc il semble logique que leur sécurité soit liée.

Théorème 11 (Elgamal est OW-CPA)**.** *Le chiffrement Elgamal possède la propriété OW-CPA si et seulement si le problème CDH est difficile.*

Démonstration

> Si CDH est facile, alors le chiffrement Elgamal est cassé. En effet, étant donné la clef publique et la première partie du chiffré, avec un algorithme qui résoud CDH ont calcule le « masque jetable » appliqué sur le message.

Dans l'autre sens, prenons un algorithme $\mathcal{A}$ qui effectue le déchiffrement Elgamal en temps polynomial avec probabilité de succès non-négligeable, et « convertissons-le » en un algorithme qui résoud CDH efficacement.

Partons d'une instance de CDH, c'est-à-dire un nombre premier p, un générateur $g \in \mathbb{Z}_p$, et deux éléments u, v appartenant au groupe. On cherche à fabriquer un élément $w \in \mathbb{Z}_p$ tel que (u, v, w) est un triplet Diffie-Hellman. Si on écrit $u = g^x$ et $v = g^y$, alors il nous faut calculer g^{xy} en temps polynomial.

Pour cela, on utilise notre algorithme de déchiffrement $\mathcal{A}$. On lui donne u comme clef publique (x est implicitement la clef secrète), on choisit un élément z aléatoire dans le groupe, et on lui demande de déchiffrer le message (v, z). En temps polynomial et avec probabilité non-négligeable, $\mathcal{A}(p, g, u, (v, z))$ renvoie un message m. Ce message satisfait la relation $z \equiv m \cdot v^x \bmod p$. Il nous suffit donc de calculer $z \cdot m^{-1}$, et on récupère v^x, c'est-à-dire g^{xy}.

On a donc un algorithme qui a la même complexité que $\mathcal{A}$ (plus une inversion modulaire...), et qui a la même probabilité de succès. On note que ce que $\mathcal{A}$ reçoit est essentiellement aléatoire, donc il ne peut pas « détecter » qu'il est utilisé. □

Sécurité sémantique Il y a des situations raisonnables où la résistance au déchiffrement ne suffit pas. En effet, le caractère à sens unique du chiffrement garantit qu'il n'est pas possible de récupérer efficacement l'intégralité du message clair. Mais cela n'exclut pas la possibilité de récupérer *une partie* du message.

Voyons les problèmes que cela peut poser. Imaginons qu'on puisse voter par internet lors des prochaines élections. Pour garantir la confidentialité du vote, chaque bureau de vote fabrique une paire de clef, et affiche sa clef publique pk sur sa porte. Pour voter, je chiffre le nom de mon candidat (« Tartemolle ») avec pk, puis j'envoie mon vote chiffré. Le bureau de vote déchiffre mon vote avec sa clef secrète, et tout va bien [1].

La question qui se pose a nous est : « est-ce que quelqu'un qui intercepte mon vote chiffré peut déterminer si j'ai voté OUI ou si j'ai voté NON ? Si le chiffrement est déterministe, il peut le faire très facilement.

Pour éviter ceci, il faut un mécanisme de chiffrement qui « masque » davantage le contenu, la « signification » des messages. On dit d'un tel mécanisme qu'il possède la *sécurité sémantique*. C'est une adaptation aux systèmes à clef publique de la « *Left-or-Right security* » (cf. § 4.1).

Définition : Sécurité sémantique

On considère le jeu :

1. $\mathcal{C}$ fabrique une paire de clefs (pk, sk) de paramètre de sécurité n, et donne pk à $\mathcal{A}$.
2. $\mathcal{A}$ peut faire des calculs, puis envoie deux messages $M_0 \neq M_1$ à $\mathcal{C}$.
3. $\mathcal{C}$ choisit un bit aléatoire b, puis envoie $E(pk, M_b)$ à $\mathcal{A}$.
4. $\mathcal{A}$ gagne s'il devine b correctement.

Le système de chiffrement est **sémantiquement sûr** (« *INDistinguishability under Chosen Plaintext Attacks* » — IND-CPA) si tout adversaire fonctionnant en temps polynomial n'a qu'un avantage négligeable.

Ceci sous-entend un « jeu » qui teste la capacité de l'adversaire, et qu'on pourrait décrire comme ceci :

```
def experiment(E, n, b, adversary):
    (pk, sk) = E.KeyGeneration(n)
    m = adversary.stage1(n, pk)
    challenge = E.encrypt(pk, m[b])
    return adversary.stage2(challenge)
```

L'avantage de l'adversaire est, comme d'habitude :

$$\mathbf{Adv}^{\text{IND-CPA}}(\mathcal{E}, \mathcal{A}, n) = \mathbb{P}\left[\texttt{experiment}(\mathcal{E}, n, 1, \mathcal{A}) = 1\right] - \mathbb{P}\left[\texttt{experiment}(\mathcal{E}, n, 0, \mathcal{A}) = 1\right]$$

1. le bureau apprend mon vote, ce qui est un défaut important... mais ce n'est pas le problème ici

En gros, pour résumer, avec un système sémantiquement sûr, on ne peut pas faire la différence entre $\mathcal{E}(pk, \texttt{"Tartempion"}; r)$ et $\mathcal{E}(pk, \texttt{"Tartemolle"}; r')$ sans posséder la clef secrète. Pour cela, il est absolument indispensable que le chiffrement soit *randomisé*, c'est-à-dire que le chiffré contienne de l'aléa en plus du message clair.

Comme le chiffrement Elgamal est randomisé, on peut espérer qu'il possède la sécurité sémantique (IND-CPA). On aboutit donc au résultat majeur de cette section :

Théorème 12 (Elgamal est IND-CPA)**.** *Le chiffrement Elgamal possède la sécurité sémantique si et seulement si le problème DDH est difficile.*

Du point de vue de la sécurité, c'est bien le sens « si » qui est le plus intéressant. Cela montre qu'il « suffit » que DDH soit difficile pour que le schéma de chiffrement soit sémantiquement sûr. Du coup, on sait ce qu'il nous faut pour faire du chiffrement à clef publique de bonne qualité : un environnement dans lequel DDH est dur.

Le reste de cette section contient une preuve de ce théorème.

La sécurité sémantique implique la difficulté du problème DDH. D'abord, on va voir que si on a sous la main un algorithme $\mathcal{A}_{\mathsf{DDH}}$ qui résoud le problème DDH, alors on peut s'en servir pour casser la propriété IND-CPA.

On fabrique donc un algorithme qui joue au jeu IND-CPA :

1. [Choix des messages]. Après réception de la clef publique (p, g, h), choisir M_0 un élément aléatoire de $\mathbb{Z}_p$ et poser $M_1 = 1$.
2. [Détection de b]. Après réception du chiffré Elgamal (a, b), calculer $b \leftarrow \mathcal{A}_{\mathsf{DDH}}(p, g, (h, a, b))$. Renvoyer b.

L'idée est la suivante : le triplet (h, a, b) est un triplet Diffie-Hellman lorsque $b = 1$. En effet, on a $h = g^x$, $a = g^y$ (pour un certain y choisi au hasard par le « challenger ») et $b = M_b g^{xy}$. Lorsque $b = 1$, cela forme un triplet Diffie-Hellman.

Si $b = 1$, alors $\mathcal{A}_{DDH}$ reçoit donc toujours un triplet Diffie-Hellman. L'algorithme renvoie donc 1 (sachant que $b = 1$) avec la même probabilité que $\mathcal{A}_{DDH}$ renvoie 1 (sachant qu'il a reçu un triplet Diffie-Hellman).

Si $b = 0$, alors on fournit à $\mathcal{A}_{DDH}$ un triplet aléatoire. La probabilité qu'on renvoie 1 sachant que $b = 0$ est donc précisément la probabilité que $\mathcal{A}_{DDH}$ renvoie 1 (sachant qu'il a reçu un triplet aléatoire).

L'avantage de notre algorithme pour IND-CPA est donc exactement le même que celui de $\mathcal{A}_{\mathsf{DDH}}$, et sa complexité est également identique.

La difficulté de DDH implique la sécurité sémantique. On montre que si on a un adversaire $\mathcal{A}_{\mathsf{IND\text{-}CPA}}$ qui casse la sécurité sémantique du chiffrement Elgamal, alors on peut en tirer un algorithme qui resoud DDH.

1. [Initialisation.] On reçoit p, g ainsi qu'un triplet (u, v, w).
2. [1ère phase IND-CPA.] Lancer la première phase de l'algorithme fourni en lui donnant u comme clef publique :
$$(M_0, M_1) \leftarrow \mathcal{A}_{\mathsf{IND\text{-}CPA}}(p, g, u).$$
3. [Choix de b']. Choisir un bit b' au hasard.
4. [2ème phase IND-CPA] Récupèrer $b'' \leftarrow \mathcal{A}_{\mathsf{IND\text{-}CPA}}(y, M_{b'} \cdot w)$
5. [Fin.] Si $b' = b''$, renvoyer 1, sinon renvoyer 0.

Si $b = 0$, alors w est une valeur aléatoire et n'est pas lié à u ou v. Dans ce cas-là, le chiffré qu'on envoie à l'adversaire ne contient *aucune* information sur b' (en effet, son message $M_{b'}$ est « masqué » par un élement *aléatoire* du groupe. Que l'adversaire renvoie 0 ou 1, il va deviner b' avec probabilité $1/2$, car il essaye en fait de deviner un bit aléatoire secret, et il ne peut pas faire ni mieux ni moins bien que ça. Par conséquent, si $b = 0$, on va répondre « 1 » avec probabilité $1/2$.

Si $b = 1$, alors le « challenger » nous donne un triplet Diffie-Hellman. Dans ce cas-là, on ne « trompe » pas l'algorithme $\mathcal{A}_{\mathsf{IND\text{-}CPA}}$: on lui fournit bien un chiffré valide de $M_{b'}$. La probabilité

qu'on réponde « 1 » est alors la probabilité que $\mathcal{A}_{\mathsf{IND\text{-}CPA}}$ trouve b'. Il reste donc à estimer la probabilité que $\mathcal{A}_{\mathsf{IND\text{-}CPA}}$ ait raison. Il faut examiner ce qui se passe selon la valeur de b'

$$\mathbb{P}\left[b'' = b'\right] = \mathbb{P}\left[b'' = 1 \;\middle|\; b' = 1\right]\mathbb{P}\left[b' = 1\right] + \mathbb{P}\left[b'' = 0 \;\middle|\; b' = 0\right]\mathbb{P}\left[b' = 0\right]$$

Comme b' est choisi au hasard, on peut simplifier un peu :

$$\mathbb{P}\left[b'' = b'\right] = \frac{1}{2}\left(\mathbb{P}\left[b'' = 1 \;\middle|\; b' = 1\right] + \mathbb{P}\left[b'' = 0 \;\middle|\; b' = 0\right]\right)$$

Après, on peut exploiter la loi des évenements complémentaires pour écrire :

$$\mathbb{P}\left[b'' = b'\right] = \frac{1}{2}\left(\mathbb{P}\left[b'' = 1 \;\middle|\; b' = 1\right] + 1 - \mathbb{P}\left[b'' = 1 \;\middle|\; b' = 0\right]\right)$$

On a donc réussi à faire apparaitre l'avantage de $\mathcal{A}_{\mathsf{IND\text{-}CPA}}$:

$$\mathbb{P}\left[b'' = b'\right] = \frac{1}{2}\left(1 + \mathbf{Adv}^{\mathsf{IND\text{-}CPA}}(\mathcal{A}_{\mathsf{IND\text{-}CPA}})\right)$$

On peut donc conclure en exprimant notre propre avantage pour le jeu DDH :

$$\mathbf{Adv}^{\mathsf{DDH}} = \frac{1}{2}\left(1 + \mathbf{Adv}^{\mathsf{IND\text{-}CPA}}(\mathcal{A}_{\mathsf{IND\text{-}CPA}})\right) - \frac{1}{2} = \frac{1}{2}\mathbf{Adv}^{\mathsf{IND\text{-}CPA}}\left(\mathcal{A}_{\mathsf{IND\text{-}CPA}}\right)$$

Notre algorithme a grosso-modo la même complexité que $\mathcal{A}_{\mathsf{IND\text{-}CPA}}$, et il a un avantage deux fois plus faible. Si l'avantage de $\mathcal{A}_{\mathsf{IND\text{-}CPA}}$ n'est pas négligeable, alors le notre ne le sera pas non plus.

Par conséquent, tant qu'il n'existe pas d'algorithme efficace pour résoudre DDH, il n'existe pas d'adversaire contre la sécurité sémantique du chiffrement Elgamal.

Maintenant qu'il est établi que le chiffrement Elgamal offre un bon niveau de sécurité lorsque le problème DDH est difficile, il reste à voir si le problème DDH est vraiment difficile. Ceci nécessite de faire un détour par le calcul des racines carrées modulo p.

9.3 Résiduosité quadratique et bits faibles de l'exponentielle

Puisqu'on peut élever au carré modulo p, c'est bien qu'il y a des racines carrées modulo p. Par exemple, $4^2 \equiv 2 \bmod 7$, donc on peut dire que 4 est une racine carrée de 2 modulo 7 (c'est dépaysant !). En fait, il y a aussi une deuxième racine carrée qui est -4, c'est-à-dire 3 : $3^2 \equiv 2 (\mathrm{mod} 7)$.

Si on calcule tous les carrés modulo 7 on trouve :

i	0	1	2	3	4	5	6
$i^2 \bmod 7$	0	1	4	2	2	4	1

On voit donc que les seuls carrés possibles sont $0, 1, 2$ et 4. En fait, plus généralement, comme la fonction « élévation au carré » envoie deux éléments de $\mathbb{Z}_p$ sur le même carré, le nombre de carrés ne peut pas être plus grand que $p/2$.

Définition : Résidu quadratique

> On dit que $a \in \mathbb{Z}_p$ est un **résidu quadratique** si c'est un carré modulo p, c'est-à-dire s'il existe $x \in \mathbb{Z}_p$ tel que $a \equiv x^2 \bmod p$.

On va voir qu'il y a un moyen efficace pour déterminer si un élément de $\mathbb{Z}_p$ est un résidu quadratique. L'intérêt de cette notion pour nous est le suivant :

Lemme 8. *Soit $p > 2$ un nombre premier, et g une racine primitive de $\mathbb{Z}_p$. Alors :*

$$g^x \text{ est un résidu quadratique modulo } p \iff x \text{ est pair.}$$

Avant de donner la preuve, remarquons que si on a un moyen de tester si un nombre est un résidu quadratique, alors on a un moyen d'apprendre la parité de x étant donné g^x. On apprendrait donc un bit d'information sur l'entrée de la fonction exponentielle.

Démonstration

$\Longleftarrow$ Si $x = 2k$, alors $g^x \equiv g^{2k} \equiv \left(g^k\right)^2 \bmod p$. On voit bien que g^x est un carré modulo p.

$\Longrightarrow$ Si g^x est un résidu quadratique, alors cela signifie qu'il existe $a \in \mathbb{Z}_p$ tel que $g^x \equiv a^2 \bmod p$. Comme g est une racine primitive de $\mathbb{Z}_p$, alors tout élément de $\mathbb{Z}_p$ peut s'écrire comme une puissance de g. En particulier, il existe y tel que $a \equiv g^y \bmod p$.
Pour résumer, on a : $g^x \equiv (g^y)^2 \bmod p$, et donc, d'après le théorème 9 (« passage à l'exponentielle »), on obtient $x \equiv 2y \bmod (p-1)$. Cela signifie qu'il existe un entier ℓ tel que $x = 2y + \ell(p-1)$. Comme p est impair, alors $p-1$ est pair, et donc x est pair.

□

Au passage, ceci démontre que la moitié exactement des éléments non-nuls de $\mathbb{Z}_p$ sont des carrés : ce sont ceux dont le logarithme discret en base g est pair.

Symbole de Legendre et critère d'Euler Une notation commode a été introduite par Legendre (1752–1833) pour discuter des questions de résiduosité quadratique.

Définition : Symbole de Legendre

Soit p un nombre premier. Le **symbole de Legendre** de a modulo p s'écrit $\left(\frac{a}{p}\right)$ et est défini de la façon suivante :

$$\left(\frac{a}{p}\right) = \begin{cases} 0 & \text{si } a \equiv 0 \bmod p \\ 1 & \text{si } a \text{ est un résidu quadratique modulo } p \\ -1 & \text{sinon} \end{cases}$$

Attention de ne pas le confondre avec la fraction a/p !

Si on disposait d'un moyen de calculer le symbole de Legendre, alors on pourrait tester la résiduosité quadratique modulo p. Ceci est possible grâce à Euler (1707—1783), un autre mathématicien du XVIIIème siècle.

Théorème 13 (Critère d'Euler)**.** *Si $p > 2$ est un nombre premier, alors*

$$\left(\frac{a}{p}\right) \equiv a^{\frac{p-1}{2}} \bmod p.$$

Note : cette preuve fait appel au résultat (classique) qui affirme qu'un polynôme non-nul ne peut pas avoir plus de racines que son degré. Une justification complète de cette affirmation est fournie en annexe à la fin de ces notes.

Démonstration

D'abord, le cas $a \equiv 0 \bmod p$ est très facile. On considère donc que $0 < a < p$.

L'outil essentiel de cette preuve est le polynôme $P(X) = X^{p-1} - 1$. Vu que p est premier et supérieur à 2, il est impair, donc $p-1$ est pair, et :

$$P(X) = \left(X^{\frac{p-1}{2}}\right)^2 - 1 = \underbrace{\left(X^{\frac{p-1}{2}} - 1\right)}_{P_1(X)} \underbrace{\left(X^{\frac{p-1}{2}} + 1\right)}_{P_{-1}(X)}$$

La suite de la preuve consiste à montrer que :
- $P_1(x) = 0$ si et seulement si x est un résidu quadratique.
- $P_{-1}(x) = 0$ si et seulement si x n'est pas un résidu quadratique.

Ceci implique en effet le résultat annoncé. Allons-y par étapes. Prenons un résidu quadratique $\alpha \in \mathbb{Z}_p$. Ceci suppose que $\alpha \equiv \beta^2 \bmod p$ pour un certain β, et on trouve :

$$P_1(\alpha) \equiv P_1\left(\beta^2\right) \equiv \left(\beta^2\right)^{\frac{p-1}{2}} - 1 \equiv \beta^{p-1} - 1 \equiv 0 \bmod p$$

La dernière congruence est dûe au petit théorème de Fermat. On voit donc que P_1 s'annule sur les $(p-1)/2$ résidus quadratiques. On va maintenant montrer qu'il ne s'annule nulle part ailleurs. L'argument est le suivant : P_1 n'est pas le polynôme nul —car $P_1(0) \equiv -1$— et il est

de degré $(p-1)/2$, donc il ne peut avoir que $(p-1)/2$ racines. Ces racines, ce sont les résidus quadratiques, donc il n'y en a pas d'autres.

Passons maintenant à $P_{-1}(X)$: on prend un non-résidu quadratique $\alpha \in \mathbb{Z}_p$, et on va voir que $P_{-1}(\alpha) \equiv 0$. D'après le petit théorème de Fermat, on a $P(\alpha) \equiv 0 \bmod p$. Comme on a vu que $P_1(\alpha) \not\equiv 0 \bmod p$, alors forcément $P_{-1}(\alpha) \equiv 0 \bmod p$. Le même argument de comptage permet de conclure : $P_{-1}(X)$ est non-nul, de degré $(p-1)/2$, s'annule sur les $(p-1)/2$ non-résidus quadratiques, donc il ne peut s'annuler nulle part ailleurs. □

Le symbole de Legendre peut donc se calculer en temps polynomial avec une exponentation modulaire rapide, et permet de décider la résiduosité quadratique modulo p. Cela permet donc d'obtenir le bit de poids faible de x connaissant g^x. Aussi, on voit que *lorsque g est une racine primitive*, la fonction $x \mapsto g^x \bmod p$ laisse « fuir » (au moins) un bit d'information sur son entrée, bien qu'elle soit à sens unique a priori.

Ceci a une conséquence directe sur la difficulté du problème DDH.

9.3.1 Problème DDH dans $\mathbb{Z}_p$

Le problème. Voyons d'abord la mauvaise nouvelle. Si g est une racine primive modulo p, alors il existe un algorithme polynomial qui obtient un avantage de $1/2$ pour résoudre le problème DDH avec p et g. Ce n'est pas *du tout* un avantage négligeable : cet algorithme résoud correctement le problème dans 75% des cas, ce qui est bien mieux que s'il répondait au hasard.

Cet algorithme repose sur l'idée suivante : si on a un triplet Diffie-Hellman (g^x, g^y, g^{xy}), alors on regarde si g^{xy} est un résidu quadratique modulo p. On alors l'équivalence suivante, avec des symboles de Legendre :

$$\begin{aligned}\left(\frac{g^{xy}}{p}\right) = 1 &\iff xy \text{ est pair} \\ &\iff (x \text{ est pair }) \vee (y \text{ est pair}) \\ &\iff \left(\frac{g^x}{p}\right) = 1 \vee \left(\frac{g^y}{p}\right) = 1.\end{aligned}$$

De ceci, on déduit directement un algorithme pour le « jeu » DDH : prendre le triplet (u, v, w), et tester si l'équivalence a lieu. Si oui, répondre « 1 », sinon répondre « 0 ».

Cet algorithme repond toujours « 1 » quand son entrée est bel et bien un triplet Diffie-Hellman. On va maintenant voir qu'il répond (à tort) « 1 » quand son entrée est un triplet aléatoire (g^x, g^y, g^z) avec probabilité $1/2$. En fait, il y a 4 combinaisons possibles (sur 8) dans lesquelles l'algorithme répond incorrectement « 1 » : si aucun n'est un résidu quadratique, ou bien si g^z en est un et g^x ou g^y aussi.

L'avantage de cet algorithme est donc $1 - \frac{1}{2} = \frac{1}{2}$, comme annoncé.

La solution. Pour éviter l'algorithme en question, il suffit de choisir un « générateur » qui est un résidu quadratique. Par exemple on peut prendre $g' = g^2$ lorsque g est une racine primitive. L'ordre de g' est $(p-1)/2$, ce qui est encore suffisamment grand. Sous ces conditions, tous les éléments g'^x sont des résidus quadratiques, et le test précédent est impossible.

De manière générale, la meilleure solution consiste à choisir un générateur dont l'ordre est un grand diviseur premier de $p-1$. On peut se placer dans cette situation de la façon suivante. On choisit un nombre premier q suffisament grand, puis on génère des nombres p de la forme $qk+1$ en essayant différentes valeurs de k jusqu'à ce qu'il soit premier (ceci implique que k est pair). Ensuite on pose $g' = g^{(p-1)/q}$ et on trouve que g' est (très probablement) d'ordre q. Le nouveau générateur g' est forcément un résidu quadratique car $k = (p-1)/q$ est pair.

9.3.2 * Calcul des racines carrées modulo p

On a donc un moyen de tester si un élément de $\mathbb{Z}_p$ est un résidu quadratique. Lorsque c'est le cas, peut-on calculer une de ses deux « racines carrées » modulo p ?

Le calcul des racines carrées peut être plus ou moins facile, mais il y a un cas au moins où c'est très simple. Le point clef est le petit théorème de Fermat. En effet, comme $\alpha^p \equiv \alpha \bmod p$, on serait tenté de dire que $\alpha^{p/2}$ est une racine carrée de α. Le problème, c'est que p est impair. Cependant, une variante de cette idée fonctionne bien.

Lemme 9 (Racines carrées avec $p \equiv 3 \bmod 4$)**.** *Soit p un nombre premier tel que $p \equiv 3 \bmod 4$, et soit α un résidu quadratique modulo p. Dans ces conditions, $(p+1)/4$ est un entier, et $\alpha^{(p+1)/4}$ est une racine carrée de α.*

Démonstration

Vérifions le premier point. Si $p \equiv 3 \bmod 4$, alors $p+1$ est un multiple de 4, donc $(p+1)/4$ est bel et bien entier. Ensuite, comme α est un résidu quadratique, alors il existe beta tel que $\alpha \equiv \beta^2 \bmod p$. Par conséquent :

$$\begin{aligned}\left(\alpha^{(p+1)/4}\right)^2 &\equiv \alpha^{(p+1)/2} \mod p \\ &\equiv \beta^{(p+1)} \\ &\equiv \beta^2 \qquad \text{(par le petit théorème de Fermat)} \\ &\equiv \alpha\end{aligned}$$

9.3.3 * Le bit de poids fort est *hard-core* pour le logarithme discret

Le bit de poids faible de x est facile à obtenir étant donné g^x, lorsque g est une racine primitive. Plus généralement, *certaines* propriétés de x peuvent « filtrer » à travers $F(x)$ lorsque F est une fonction à sens unique. Mais pas toutes.

Définition : prédicat *hardcore*

Soit $F : \{0,1\}^n \to \{0,1\}^m$ une fonction à sens unique. Un prédicat $P : \{0,1\}^n \to \{0,1\}$ est un **prédicat *hardcore* pour** F si aucun algorithme ne peut déterminer $P(x)$ en temps polynomial avec un avantage non-négligeable, connaissant $F(x)$.

Le jeu qui correspond à la définition, et l'avantage de l'adversaire sont définis comme d'habitude. En gros, un prédicat *hardcore* « concentre » la difficulté de l'inversion de la fonction à sens unique. Un résultat célèbre du début des années 1980 affirme que toute fonction à sens unique admet des prédicats *hardcore*, et qu'il suffit en gros de prendre le XOR d'un sous-ensemble aléatoire des bits de l'entrée.

Pour la fonction exponentielle, le bit de poids faible n'est pas *hard-core*. Mais le prédicat suivant, lui, l'est :

$$\begin{array}{rccl}\mathsf{MSB}_{p,g}(x): & \mathbb{Z}_{p-1} & \longrightarrow & \{0,1\} \\ & x & \longmapsto & \begin{cases}0 & \text{si } 0 \le x < (p-1)/2 \\ 1 & \text{si } (p-1)/2 \le x < p\end{cases}\end{array}$$

Ce prédicat indique si le logarithme discret de α en base g modulo p appartient à la « première moitié » des valeurs possibles. C'est donc une sorte d'analogue du bit de poids fort.

Théorème 14. *Soit p un nombre premier tel que $p \equiv 3 \bmod 4$, et soit g une racine primitive modulo p. Alors $\mathsf{MSB}_{p,g}$ est un prédicat hard-core pour la fonction à sens unique $x \mapsto g^x \bmod p$.*

Tout le reste de cette section est dévoué à démontrer ce résultat.

Supposons qu'on dispose d'un algorithme $\mathcal{A}$ capable de calculer $\mathsf{MSB}_{p,g}(x)$ à partir de $g^x \bmod p$, en temps polynomial. On va construire un autre algorithme $\mathcal{B}$, qui s'exécutera lui aussi en temps polynomial, mais qui calcule x à partir de $g^x \bmod p$, c'est-à-dire qui résoud le problème du logarithme discret.

Schéma général. L'idée consiste à calculer x bit par bit, par la méthode suivante. On a vu qu'on peut calculer les racines carrées efficacement. Or calculer la racine carrée de g^x, cela revient plus ou moins à calculer $g^{x/2}$. Grosso-modo, ça « décale » x d'un bit vers la droite... ce qui tombe

bien car on peut connaître le bit de droite. En itérant cette procédure, on apprendrait donc ainsi tous les bits de x.

Il y a cependant deux détails à régler. Le premier c'est que g^x n'est pas forcément un résidu quadratique, donc on ne peut pas forcément calculer sa racine carrée. Ceci n'est pas un problème, car dans ce cas-là $g^{-1} \cdot g^x = g^{x-1}$ en est une.

L'autre problème est beaucoup plus embêtant : lorsque x est pair, g^x a *deux* racines carrées, et seulement une seule est $g^{x/2}$. Le *hic*, c'est qu'on ne sait pas si on récupère la « bonne » ou pas. C'est précisément à *ça* que sert l'algorithme $\mathcal{A}$ qui évalue $\mathsf{MSB}_{p,g}$.

Lemme 10. *Soit x un entier pair avec $0 \leq x < p-1$. On sait que g^x est un résidu quadratique. Ses deux racines carrées sont $g^{\frac{x}{2}}$ et $g^{\frac{x}{2}+\frac{p-1}{2}}$.*

Démonstration

Le fait que $g^{\frac{x}{2}}$ est une racine carrée est une évidence (il suffit d'élever au carré). Pour la deuxième, on trouve :

$$\begin{aligned}\left(g^{\frac{x}{2}+\frac{p-1}{2}}\right)^2 &\equiv g^x g^{p-1} \mod p \\ &\equiv g^x \qquad \text{(d'après le petit théorème de Fermat).}\end{aligned}$$ □

Toute l'astuce, c'est que $\mathsf{MSB}_{p,g}$ vaut 0 sur l'exposant dans la première racine carrée ($g^{x/2}$, qui est dite « principale ») et vaut 1 sur l'autre.

On peut donc écrire l'algorithme suivant, qui calcule le logarithme discret de y :

1. [initialisation.] $e \leftarrow y$, $z \leftarrow 0$, $i \leftarrow 0$.
2. [Fini ?] Si $e = 1$, alors renvoyer z.
3. [Résidu quadratique ?] Si e est un résidu quadratique modulo p, aller à l'étape 5.
4. [Rectification.] $z \leftarrow z + 2^i$ et $e \leftarrow g^{-1}e \bmod p$.
5. [Racine carrée.] Calculer les deux racines carrées de e. Sélectionner la principale avec un appel à l'algorithme $\mathcal{A}$ qui évalue $\mathsf{MSB}_{p,g}$. Remplacer e par sa racine carrée principale.
6. [Boucle.] $i \leftarrow i + 1$. Retourner à l'étape 2.

Notons x la réponse attendue, c'est-à-dire que $y \equiv g^x \bmod p$. On note $x \gg i$ le nombre x décalé de i bits vers la droite, conformément à un usage établi depuis longtemps dans les principaux langages de programmation.

Tout d'abord, on montre qu'à l'étape 2, $e \equiv g^{x \gg i} \bmod p$. C'est vrai lors du premier passage, vu l'initialisation. Ensuite, supposons que c'est vrai, et montrons que c'est encore vrai à la fin de la boucle. Écrivons la division euclidienne de $x \gg i$ par 2 : $x \gg i = 2q + r$. Dans cette formule, r est le bit de poids faible de $x \gg i$, c'est-à-dire que c'est le i-ème bit de x, et q est en fait $x \gg (i+1)$.

Si $r = 1$, c'est-à-dire si e n'est pas un résidu quadratique, alors l'étape 4 est exécutée, qui transforme e en g^{2q}. L'effet de l'étape 5 est alors de transformer e en $g^q = g^{x \gg (i+1)}$. L'invariant est donc établi.

On voit donc que lorsque l'étape 3 est exécutée, c'est que le i-ème bit de x est égal à 1. Alors, le i-ème bit de z est défini à 1. Ceci garantit qu'on a bien $z = x$ à la fin.

Le nombre d'itération est le nombre de bits de p au maximum, et le nombre d'opérations de chaque itération est polynomial en cette quantité. Le théorème est donc démontré.

Chapitre 10

Systèmes reposants sur la difficulté de la factorisation

L'échange de clef Diffie-Hellman et le chiffrement/signature Elgamal reposent sur la difficulté (supposée) du calcul du logarithme discret. On va maintenant voir des schémas cryptgraphiques, les plus utilisés, reposant sur la difficulté de la factorisation des grands entiers.

Dans tous ces schémas, un nombre composite (généralement noté n) est rendu public, tandis que sa factorisation est gardée secrète. L'idée c'est que n sert à effectuer les opérations publiques (chiffrement, vérification de signature, etc.). La connaissance de la décomposition de n en produit de facteurs premiers permet, elle, d'effectuer les opérations secrètes (déchiffrement, signature).

Pour que ceci puisse fonctionner, il faut qu'il soit impossible de calculer la factorisation de n. Ceci exige d'une part que n soit suffisamment grand (2048 bits au moins en 2019) et d'autre part que n n'ait pas de petit diviseur, qui serait facile à trouver. Le plus simple consiste à générer aléatoirement deux grands nombres premiers (souvent notés p et q), et de calculer $n = pq$. A priori, les nombres de cette forme sont les plus difficiles à factoriser. Le record actuel de factorisation date de 2009, où un nombre de cette forme de 768 bits a été factorisé (c'est le record *connu*). Le calcul a nécessité environ 2^{67} opérations, soit environ 2000 ans de calcul sur un seul coeur d'un CPU moderne.

La plupart du temps, les message sont représentés comme des éléments de $\mathbb{Z}_n$, c'est-à-dire des entiers modulo n. Un certain nombre de différences sont à souligner par rapport à la situation qui prévaut dans les schémas à base de logarithme discret, où tout a lieu modulo un nombre premier. Lorsqu'on travaille modulo un nombre composite, il faut faire attention que :

— Tous les éléments de $\mathbb{Z}_n$ ne sont pas inversibles modulo n.
— Le petit théorème de Fermat n'est pas vrai.

Ces préliminaires étant posés, il y a deux principaux schémas reposant sur la difficulté de la factorisation : celui de Rabin et RSA. RSA est de très loin le plus utilisé, mais Rabin a l'avantage que les arguments qui justifient sa sécurité sont plus clairs.

10.1 Chiffrement RSA

Inventé en 1978, il est nommé d'après les initiales de ses inventeurs (Rivest, Shamir et Adleman). Il a été couvert par un brevet, qui a maintenant expiré.

Génération des clefs. Pour générer une paire de clefs RSA :

— Choisir deux grands nombres premiers p et q, puis calculer leur produit $n = pq$.
— Choisir un *exposant public* e, qui doit être premier avec $(p-1)(q-1)$. Ceci implique que e est impair (le cas spécial $e = 2$ est le chiffrement de rabin, cf. infra). La plupart du temps, on prend $e = 3$ ou $e = 2^{16} + 1$.
— Calculer d, l'inverse de e modulo $(p-1)(q-1)$ (on a donc $ed \equiv 1 \bmod (p-1)(q-1)$).

La clef publique est composée de e et n (elle occupe donc environ 2000 bits). La clef secrète est le nombre d (qui occupe lui aussi environ 2000 bits).

Chiffrement. Les messages clairs, tout comme les chiffrés, sont des éléments de $\mathbb{Z}_n$. Pour chiffrer m, on calcule :

$$\mathcal{E}((n,e),m) = m^e \bmod n.$$

La complexité du chiffrement est celle d'une exponentiation modulaire, donc c'est plutôt $\mathcal{O}\left(n^3\right)$. En pratique elle dépend de la valeur de e. On choisit parfois $e = 3$ car cela accélère le chiffrement (qui ne demande plus que 3 multiplications et 3 divisions par n). Pour la même raison, la valeur $e = 2^{16} + 1$ permet un chiffrement plus rapide que si e était un grand entier arbitraire.

Déchiffrement. L'un des gros avantages de RSA c'est que le déchiffrement est particulièrement simple. Pour déchiffer un chiffré c, on calcule :

$$\mathcal{D}((n,d),c) := c^d \bmod n.$$

Correction. On part d'un message m, on le chiffre puis on le déchiffre. Qu'est-ce qui garantit qu'on retombe bien sur m ?

Théorème 15 (Correction de RSA)**.** *Le déchiffrement RSA est la fonction réciproque du chiffrement.*

Pour démontrer ce résultat, il faut faire appel au petit théorème de Fermat d'une part, et au théorème des *restes chinois* d'autre part.

10.1.1 Le théorème des restes chinois

En fait, il y a un lien entre $\mathbb{Z}_{uv}$ (l'ensemble des entiers modulo uv) d'un côté, et $\mathbb{Z}_u \times \mathbb{Z}_v$ de l'autre.

Descente. Prenons un exemple. Choisissons $n = 15 \cdot 32 = 480$. Aucun de ces deux facteurs n'est premier, mais ils sont premier entre eux (ils n'ont pas de diviseur commun). Considérons un entier a quelconque, et supposons que $a \equiv 42 \bmod 480$. Dispose-t-on d'un moyen simple de déterminer $a \bmod 15$ et $a \bmod 32$?

On sait que $a = 42 + 15 \cdot 32 \cdot k$, où k est un entier qu'on ne connaît pas. Par conséquent, on sait que :

$$\begin{aligned} a &\equiv 42 \mod 15 & \qquad a &\equiv 42 \mod 32 \\ a &\equiv 12 \mod 15 & \qquad a &\equiv 10 \mod 32. \end{aligned}$$

L'astuce, c'est que k disparait quand on passe au modulo. Du coup, les restes modulo 15 et modulo 32 sont complètement determinés par le reste modulo $15 \cdot 32$. Conclusion : si on connaît $a \bmod uv$ alors on peut facilement calculer $a \bmod u$ et $a \bmod v$. Ceci nous pousse à définir la fonction :

$$\begin{array}{rccc} \Psi_{u,v} : & \mathbb{Z}_{uv} & \longrightarrow & \mathbb{Z}_u \times \mathbb{Z}_v \\ & a & \longmapsto & (a \,\%\, u, \quad a \,\%\, v) \end{array}$$

Remontée. Maintenant, on pourrait aussi se poser la question dans le sens inverse : si on connaît $a \bmod u$ et $a \bmod v$, est-ce qu'on peut déterminer $a \bmod uv$? Par exemple, si :

$$\begin{aligned} a &\equiv 4 \mod 15 \\ a &\equiv 2 \mod 32, \end{aligned}$$

que vaut a ? Autrement dit, est-ce qu'on peut « inverser » la fonction $\Psi_{u,v}$?

La réponse, positive, à cette question a été fournie par le mathématicien chinois Sun Tsŭ (孙子), quelque part entre les années 280 et 473. Pour cette raison, le résultat a pris le nom de théorème des « restes chinois ».

Théorème 16 (Restes chinois)**.** *Soient u et v deux entiers positifs et premiers entre eux. Soient α et β deux entiers. Il existe un et un seul entier k qui satisfait les conditions :*

$$0 \leq k < uv \qquad \textit{et} \qquad \left\{ \begin{array}{rcl} k & \equiv & \alpha \mod u \\ k & \equiv & \beta \mod v \end{array} \right.$$

En d'autres termes, il y a un et un seul k tel que $\Psi_{u,v}(k) = (\alpha, \beta)$, donc $\Psi_{u,v}$ est une bijection.

Démonstration

Dans un premier temps, on va voir qu'il existe toujours au moins une solution. Pour cela, on lance l'algorithme d'euclide étendu sur u et v et on obtient (r, s) tels que $ru + sv = 1$. On a donc :

$$\Psi_{u,v}(ru) = (0, 1)$$
$$\Psi_{u,v}(sv) = (1, 0).$$

Par conséquent, il est facile de vérifier que :

$$\Psi_{u,v}(\beta ru + \alpha sv) = (\alpha, \beta).$$

Au passage, on peut aussi se convaincre que r est l'inverse de u modulo v et s est l'inverse de v modulo u.

Maintenant, on va voir qu'il y a *exactement* une solution. Pour cela, on remarque qu'il y a uv sorties différentes possibles de $\Psi_{u,v}$. On vient de voir que pour chacune de ces sorties possible il existe (au moins) une entrée correspondante. Ces entrées sont toutes différentes, car elles aboutissent à des sorties différentes. Or, il n'y a que uv entrées possibles. Par conséquent, chaque sortie ne peut être produite que par une et une seule entrée. □

Correspondance. On vient de voir qu'à chaque entier modulo uv correspond une seule paire d'entiers modulo u et v. En fait, cela va même un peu plus loin. On va maintenant voir que faire des calculs dans $\mathbb{Z}_{uv}$ ou bien dans $\mathbb{Z}_u \times \mathbb{Z}_v$, cela revient au même. Pour cela, notons :

$$\begin{array}{lll} (a,b) \boxplus (c,d) & = (& (a+c) \,\%\, u, \qquad (b+d) \,\%\, v \quad) \\ (a,b) \boxtimes (c,d) & = (& ac \,\%\, u, \qquad\quad bd \,\%\, v \quad). \end{array}$$

En gros, $\boxplus$ et $\boxtimes$ représentent l'addition et la multiplication « parallèle » modulo u et v. On a alors le résultat logique :

Lemme 11. *Pour toute paire d'entier a et b on a :*

$$\Psi(a + b) = \Psi(a) \boxplus \Psi(b)$$
$$\Psi(a \times b) = \Psi(a) \boxtimes \Psi(b)$$

On laisse au lecteur le soin de se convaincre que c'est vrai, mais c'est très facile.

Application : correction du chiffrement RSA. Voici maintenant la preuve du théorème 15. L'idée générale est qu'on raisonne modulo p et q séparément, ce qui permet l'utilisation du petit théorème de Fermat, puis on « recombine » les résultats avec le théorème des restes chinois.

Démonstration

Il suffit de montrer que $m^{ed} \equiv m \bmod n$, où $n = pq$. Pour cela, on note que $ed \equiv 1 \bmod (p-1)(q-1)$. Par conséquent, il existe un entier k tel que $ed = 1 + k(p-1)(q-1)$.

On « projette » m^{ed} modulo p :

$$\begin{aligned} m^{ed} &\equiv m \cdot m^{k(p-1)(q-1)} \mod p \\ m^{ed} &\equiv m \cdot \left(m^{k(q-1)}\right)^{p-1} \\ m^{ed} &\equiv m \qquad \text{(d'après le petit théorème de Fermat).} \end{aligned}$$

On peut faire le même raisonnement modulo q, et aboutir à la conclusion que $m^{ed} \equiv m \bmod q$.

Pour finir, le théorème des restes chinois garantit que $m^{ed} \equiv m \bmod pq$: on a $\psi_{p,q}\left(m^{ed}\right) = (m, m)$ d'une part et $psi_{p,q}(m) = (m, m)$, donc par injectivité de ψ, on sait que $m^{ed} \equiv m \bmod pq$.

10.2 Sécurité

10.2.1 Sécurité du chiffrement

Absence de « preuve » de sécurité. Dans le cas du système de Rabin, on sait qu'effectuer le déchiffrement sans connaître la clef secrète est aussi difficile que factoriser n. Dans RSA, il n'existe pas d'argument qui permette d'affirmer la même chose. En gros, pour effectuer le déchiffrement RSA, il faut être capable de calculer des « racines e-ème » modulo n. On ne connaît pas de moyen efficace de le faire sans connaître la factorisation de n. Cependant, l'existence d'un algorithme efficace de calcul des racines e-ème n'impliquerait pas automatiquement l'existence d'un algorithme efficace de factorisation. Il semble cependant improbable que le premier puisse exister sans que le deuxième ne soit déjà là.

Par contre, l'avantage, c'est que même si un adversaire peut faire déchiffrer ce qu'il veut, il ne peut pas utiliser cette capacité pour obtenir la clef secrète, contrairement à ce qui se passe dans le système de Rabin.

Malléabilité. Comme le chiffrement ElGamal, le système RSA est *malleable* : le produit des chiffrés (modulo n) est égal au chiffré des produits (modulo n). Par conséquent, et sauf dans des situations très particulières, il est nécessaire d'utiliser des précautions particulières pour son utilisation.

Déterminisme. Le chiffrement RSA n'est pas randomisé, donc il ne peut pas avoir la sécurité sémantique (IND-CPA, cf. § 9.2). On l'utilise donc généralement en injectant un aléa dans le message à chiffrer.

Petits messages. De toute façon, il ne faut *jamais* utiliser le chiffrement RSA tel qu'il est décrit ci-dessus, ne serait-ce que pour une raison simple : si $m^e < n$, alors le chiffrement n'offre aucune sécurité. En effet, dans ce cas-là, l'opération « modulo n » n'a aucun effet, or calculer des racines e-èmes sur les entiers (pas modulo) est facile. En pratique, on utilisera toujours un mécanisme de bourrage spécial qui empêche ce problème, en faisant en sorte que les messages (bourrés) soient proches de n.

Attention au « *Broadcast*. Supposons qu'Alice envoie un même message m à plusieurs destinataires. Elle possède les clefs publiques de tous les destinataires : $(e_1, n_1), \ldots, (e_k, n_k)$. Elle envoie $m^{e_i} \bmod n_i$ au i-ème destinataire. S'ils utilisent tous la même valeur de e, alors un adversaire passif récupère $m^e \bmod n_1, \ldots, m^e \bmod n_k$.

Grâce au théorème des restes chinois, il peut calculer $m^e \bmod n_1 n_2 \ldots n_k$. Si $e \leq k$, alors le modulo n'a pas d'effet, et il connaît l'entier m^e. Il lui suffit donc de calculer une racine e-ème (normale, pas modulo) pour retrouver m. Ceci est une des raisons pour lesquelles utiliser $e = 3$ n'est pas forcément une bonne idée.

10.2.2 Difficulté de récupérer la clef secrète

Il est clair que quelqu'un qui peut factoriser n peut récupérer la clef secrète. On va voir que c'est également vrai dans l'autre sens : récupérer l'exposant secret d à partir des données publiques est aussi difficile que de factoriser n.

Difficulté de $\phi(n)$. Tout d'abord, pour commencer, on note que si on peut obtenir $(p-1)(q-1)$, alors on peut en déduire d et tout casser. Heureusement, on va voir que calculer $(p-1)(q-1)$ est aussi dur que calculer n. Au passage, ce nombre est souvent noté $\phi(n)$. C'est le nombre d'éléments non nuls de $\mathbb{Z}_n$ qui sont inversibles.

L'idée c'est que $\phi(n) = n - (p+q) + 1$. Si on connait $\phi(n)$, on peut donc en déduire $S = p + q$. Il nous reste donc à résoudre le système :

$$\begin{cases} p + q &= S \\ p \times q &= n \end{cases}$$

Pour cela, on substitue q et on trouve $p \cdot (S - p) = n$. Ceci est une équation du second degré en p, qu'on apprend à résoudre en classe de 1ère scientifique.

Conclusion : si on connaît $\phi(n)$, alors on peut factoriser n.

Difficulté de d. Supposons qu'un adversaire soit capable de calculer d. On va voir qu'on peut en déduire la factorisation de n. Pour cela, on calcule $k = ed - 1$. Alors, pour tout g inversible dans $\mathbb{Z}_n$, on a $g^k \equiv 1 \bmod n$ (car $g^{k+1} \equiv g \bmod n$).

On sait que k est pair, donc on regarde $g^{k/2}$. Ceci est une racine carrée de 1 modulo n. On verra plus bas (§ 10.5.1) qu'il y en a $4 : 1, -1, x$ et $-x$. Si on trouve que $g^{k/2} \not\equiv \pm 1 \bmod n$, alors on va pouvoir factoriser n en utilisant le même truc que pour le chiffrement de Rabin.

Moralité : si on connaît e, d et n, alors on peut factoriser n. Ceci signifie qu'il n'y a pas d'algorithme sensiblement plus rapide que ceux de factorisation, capable de retrouver d à partir de la clef publique.

Considérations sur le choix de e. L'exposant public e n'a pas besoin d'être aléatoire, et en plus il est public. Ceci permet que tout le monde choisisse la même valeur. $e = 3$ est la plus petite valeur admissible (puisque $e = 2$ donne le chiffrement de Rabin). Cependant, pour un certain nombre de raisons, $e = 3$ peut poser des problèmes, et $e = 2^{16} + 1$ est généralement préféré.

Attention à la génération des nombres premiers. On a vu que pour générer un nombre premiers de k bits, il faut générer des nombres aléatoires et tester s'ils sont premiers. Environ $\mathcal{O}(k)$ essais sont nécessaires en moyenne, ce qui nécessite donc en tout $\mathcal{O}(k^2)$ bits aléatoires. En 2019, cela fait quelques millions. On utilise donc souvent un PRNG pour générer les bits aléatoires nécessaires à la production d'une paire de clefs.

Ceci peut poser un problème si ces PRNG sont mal initialisés. En effet, si les clefs publiques d'Alice et Bob possèdent un facteur premier en commun, celui-ci est facile à trouver en calculant le PGCD de leurs valeurs respectives de n. Une étude menée en 2012 a montré que sur un total d'environ 6.4 millions de clefs publiques RSA trouvées sur le web (dans des certificats ou des clefs publiques PGP d'individus), 26'965 partagent un facteur premier avec un autre, et n'offrent donc aucune sécurité.

10.3 *Paddings* standards

Il y a en fait une technique qui résoud tous les problèmes évoqués (déterminisme + malléabilité + « petits » messages). Il suffit d'ajouter un bourrage au message.

10.3.1 PKCS#1 v1.5

Le plus ancien mécanisme de padding utilisé a été spécifié dans la RFC 2313. C'est le bourrage par défaut dans openssl. Notons k la taille de la clef publique n en octets (avec des clefs de 2048 bits, on devrait avoir $k = 256$). Voici la procédure pour appliquer le bourrage sur un message m. On note $|m|$ la taille de m en octets.

1. Si $|m| > k - 11$ octets, alors le message est trop long et on déclenche une erreur.
2. Génèrer une chaine PS de $k - |m| - 3$ octets pseudo-aléatoires, tous différents de zéro.
3. Former le bloc « bourré » :

0x00	0x02	PS	0x00	m

4. Chiffrer le bloc bourré (c'est un nombre plus petit que n).

Ce mécanisme de bourrage est relativement simple à mettre en oeuvre, et il « randomize » assez bien le chiffrement (la partie aléatoire fait au moins 8 octets). En plus, il casse la malléabilité, empêche le problème des « petits » messages (ils deviennent tous grands).

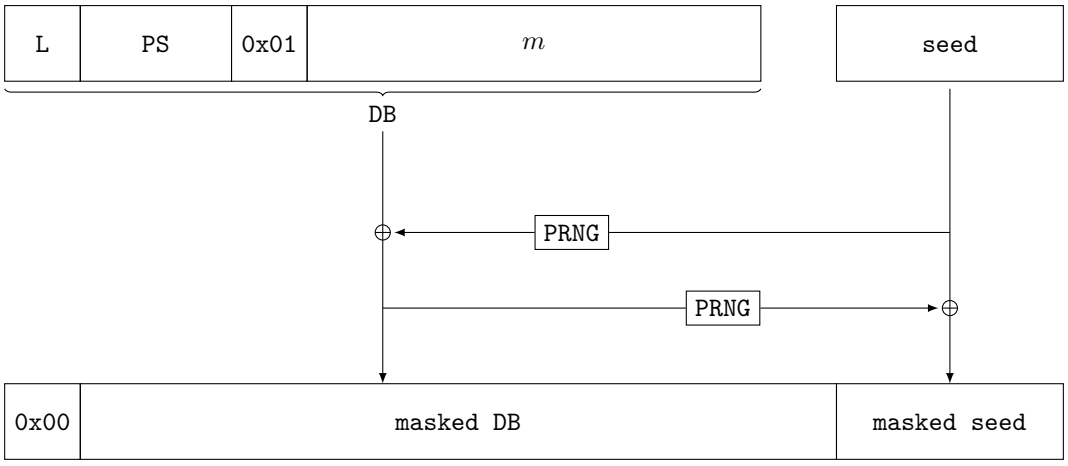
L'inconvénient c'est qu'il est assez facile de « forger » un chiffré qui présente le bourrage correctement. Et quand on y arrive, on sait que le message clair correspondant commence par `0x0002`. Une attaque sur RSA célèbre (mais pas très facile à implémenter) exploite cette propriété, et était faisable en pratique sur certains serveur SSL, dans le passé. Le problème avait lieu lorsque les serveurs répondaient différement selon le bourrage se présentait correctement ou pas. Ceci révèle alors une faible quantité d'information sur le message réellement chiffré. Ce problème, entre autre, a poussé à la conception de mécanismes de chiffrement qui possèdent une notion de sécurité plus forte que la sécurité sémantique (IND-CPA) : la sécurité face aux attaques à *chiffrés choisis*, c'est-à-dire où l'adversaire à le pouvoir d'obtenir le déchiffrement de certains messages (donc d'obtenir *toute* l'information sur le clair qui correspond à ses chiffrés).

10.3.2 OAEP

Pour faire face aux défauts du bourrage PKCS#1 v1.5, un autre mécanisme de bourrage plus sophistiqué a été développé : c'est l'« *Optimal Asymetric Encryption Padding* ». En gros, ce padding consiste à faire deux tours de réseau de Feistel, et utilise un PRNG (dans la pratique, c'est une fonction de hachage en mode compteur).

On note toujours k la taille de la clef publique n en octets. On note aussi h la taille de la sortie de la fonction de hachage, en octets. Étant donné un message m, on fabrique la version « bourrée » de la façon suivante

1. Si $|m| > k - 2h - 2$ octets, alors le message est trop long et on déclenche une erreur.
2. Génèrer une chaine `PS` de $k - 2h - |m| - 2$ octets tous égaux à zéro. Elle peut être vide.
3. Former la chaine `L` $= H($« nom de la fonction de hachage »$)$.
4. Former le « bloc de données » DB en concatenant `L`, `PS`, `0x01` et m.
5. Générer un nombre aléatoire `seed` de h octets.
6. Utiliser `seed` comme graine d'un PRNG. Générer un masque pseudo-aléatoire de $k - h - 1$ octets et le XORer sur `DB` pour former `masked DB`.
7. Utiliser `masked DB` comme graine d'un PRNG, et générer un masque pseudo-aléatoire de taille h, puis le XORer sur `seed` pour former `masked seed`.
8. Enfin, concaténer `0x00`, `masked DB` et `masked seed`. Le résultat est un nombre plus petit que n.



Bien qu'il soit beaucoup plus lourd à mettre en oeuvre, ce schéma de bourrage n'a pas les défauts de PKCS#1 v1.5. Notamment, la probabilité qu'un chiffré aléatoire soit valide est extrêmement faible (c'est la probabilité de deviner `seed`, en gros). De plus, il n'est pas possible de produire facilement des chiffrés valides, ce qui empêche un certain nombre de problèmes.

Pour effectuer le déchiffrement :

1. Reconstituer `seed` à partir de `masked DB` et de `masked seed`.

2. Reconstituer `DB` à partir de `masked DB` et de `seed`.
3. Vérifier que `DB` est correct (L, puis PS, puis `0x01`, etc.). Si ce n'est pas le cas, renvoyer « Erreur de déchiffrement ».
4. Renvoyer m.

C'est la vérification de l'étape 3 qui offre un vrai avantage en terme de sécurité par rapport au bourrage PKCS #1.5. En effet, ce test rend concrètement impossible de fabriquer des chaines de bits qui sont acceptées par la procédure de déchiffrement... autrement qu'en utilisant la procédure de chiffrement.

Belare et Rogaway, qui ont proposé le *padding* OAEP ont démontré que le chiffrement RSA combiné à OAEP possède la sécurité sémantique. Pointcheval et Stern ont démontré un peu plus tard qu'il possède même une notion de sécurité plus forte : la résistance aux attaques à chiffrés choisis de manière adaptative.

Ce n'est pas forcément très facile de voir pourquoi, mais les deux étapes intermédiaires décrites ci-dessous peuvent potentiellement donner des arguments.

10.3.3 (*) Transformation de Fujisaki et Okamoto

Dans la même famille que OAEP, il existe des moyens d'améliorer la sécurité de mécanismes déjà existants. Par exemple, on connaît un mécanisme de chiffrement sémantiquement sûr (le chiffrement Elgamal), mais qui n'est pas résistant aux attaques à chiffrés choisis, car il est malleable. On peut lui supprimer ce défaut grâce à une technique inventée en 1999.

Si E et D sont les procédures de chiffrement et de déchiffrement d'un système sémantiquement sûr (IND-CPA), alors on pose :

$$\mathcal{E}(pk, m, r) = E\big(pk, m \,\|\, r, H(m \,\|\, r)\big)$$

En gros, on chiffre $m \,\|\, r$ en utilisant $H(m \,\|\, r)$ comme aléa. Pour déchiffrer un message y :
— On calcule $x = D(sk, y)$. En principe on récupère $x = m \,\|\, r$.
— On vérifie que $y = E\big(pk, x, H(x)\big)$. Si oui, on renvoie m. Sinon, on renvoie une erreur.

Le truc, c'est que même s'il est possible de « casser » E grâce à des déchiffrement, ici ce n'est pas possible. D'une part l'entrée de E n'est pas entièrement sous le contrôle de l'adversaire, à cause de la fonction de hachage. L'adversaire peut essayer de choisir r comme ça l'arange (par exemple $r = 0$), mais alors il perd le contrôle de $H(r)$.

Si l'adversaire essaye d'utiliser des messages liés (par exemple, s'il essaye de déchifrer x puis y pour réussir à déchiffrer xy dans Elgamal), il se fait avoir d'abord par l'aléa puis par la fonction de hachage, qui « brise » la relations et rend le tout non-malléable.

Il est difficile de fabriquer un chiffré valide sans utiliser le mécanisme de chiffrement. En effet, il faudrait réussir à fabriquer une chaine de bits qui se déchiffre en quelque chose de la forme $x \,\|\, H(x)$. Si on essaye au hasard, on a une probabilité negligeable de réussite, car il faudrait fabriquer la bone sortie de H « par accident ». Du coup, même si on accès à un oracle de déchiffrement, on ne peut faire déchiffrer... que ce qu'on a soi-même chiffré, ce qui n'est pas très intéressant.

Tout ceci ne s'applique pas directement au chiffrement RSA, car il n'est pas sémantiquement sûr d'une part, et il ne prend pas d'aléa en entrée d'autre part.

10.3.4 (*) Transformation de Pointcheval

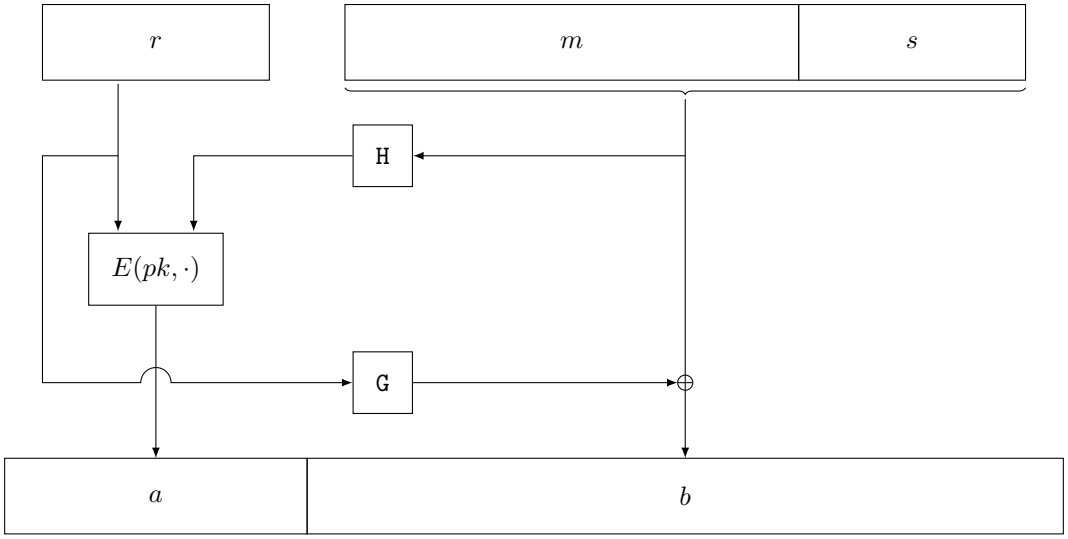
On présente ici une méthode, inventée en l'an 2000 par David Pointcheval, pour transformer un mécanisme de chiffrement à sens unique (comme le chiffrement RSA) en mécanisme de chiffrement sémantiquement sûr face à des adversaires qui peuvent faire déchiffrer ce qu'ils veulent. C'est une sorte de généralisation du bourrage OAEP.

Si E et D sont les procédures de chiffrement et de déchiffrement d'un système à sens unique, et G, H sont deux fonctions de hachage différentes, alors on pose :

$$\mathcal{E}(pk, m, r \,\|\, s) = \underbrace{E\big(pk, r \,\|\, H(m \,\|\, s)\big)}_{a} \,\Big\|\, \underbrace{(m \,\|\, s) \oplus G(r)}_{b},$$

où r et s sont des chaines de bits aléatoires.

L'idée générale est la suivante : $G(r)$ agit comme un masque jetable qui permet de chiffrer m. Comme il faut bien pouvoir déchiffrer, on effectue le chiffrement de r avec la clef publique. Le deuxième aléa, s, sert à « randomiser » le chiffrement du premier.



Pour déchiffrer un message, la procédure n'est pas plus compliquée :

1. Séparer les deux parties a et b.
2. Déchiffrer a, récupérer r.
3. Calculer $b \oplus G(r)$ qui vaut en principe $m \,\|\, s$
4. Vérifier que $a = E(pk, r \,\|\, H(m \,\|\, s))$. Si ce n'est pas le cas, renvoyer « erreur de déchiffrement ».
5. Renvoyer m.

On fait rentrer $H(m \,\|\, s)$ dans le chiffrement à sens unique E, pour la même raison que précédemment. Il est difficile de fabriquer des chiffrés valides sans effectuer le chiffrement, à cause de la « vérification » de l'étape 4.

10.4 Signature RSA.

Contrairement aux signatures de Schnorr ou Elgamal, qui sont assez différentes du chiffrement Elgamal, la signature RSA utilise le chiffrement RSA « à l'envers », et elle est déterministe.

Une signature de m est simplement $s = m^d \bmod n$ (c'est le « déchiffrement » de m). Pour vérifier que la signature est correcte, on vérifie que $s^e \equiv m \bmod n$ (on « re-chiffre »).

La malléabilité de RSA pose un problème particulier dans le schéma de signature. En effet, le produit des signatures est la signature du produit. Par conséquent, à partir de deux signatures légitimes, je pourrais en fabriquer une troisième (leur produit modulo n), alors que je ne possède pourtant pas la clef secrète.

Ce problème, comme beaucoup d'autres peut s'éviter avec l'utilisation de mécanismes de bourrage décrits ci-dessous.

10.4.1 PKCS#1 v1.5

Il y a un mécanisme simple, analogue à celui du § 10.3.1, mais pour des signatures. C'est aussi le bourrage par défaut dans openssl. Notons k la taille de la clef publique n en octets (avec des clefs de 2048 bits, on devrait avoir $k = 256$). Pour signer un message m, on calcule d'abord le « bloc bourré », puis on signe ce dernier. Ceci nécessite une fonction de hachage H, qui produit des empreintes de h octets.

La fonction de hachage n'est pas fixée a priori, donc il faut indiquer laquelle on a utilisé. Pour cela, une structure `ASN.1` contenant un « Universal Object Identifier » de la fonction de hachage, ainsi que l'empreinte, est sérialisée en `DER`. Concrètement, avec SHA-256, il faut former :

$T =$ | `0x3031300D060960864801650304020105000420` | SHA-256(m) |

1. Si $|T| > k - 11$ octets, alors la clef publique est trop courte et on déclenche une erreur.
2. Génèrer une chaine `PS` de $k - |T| - 3$ octets tous égaux à `0xff`.
3. Former le bloc « bourré » :

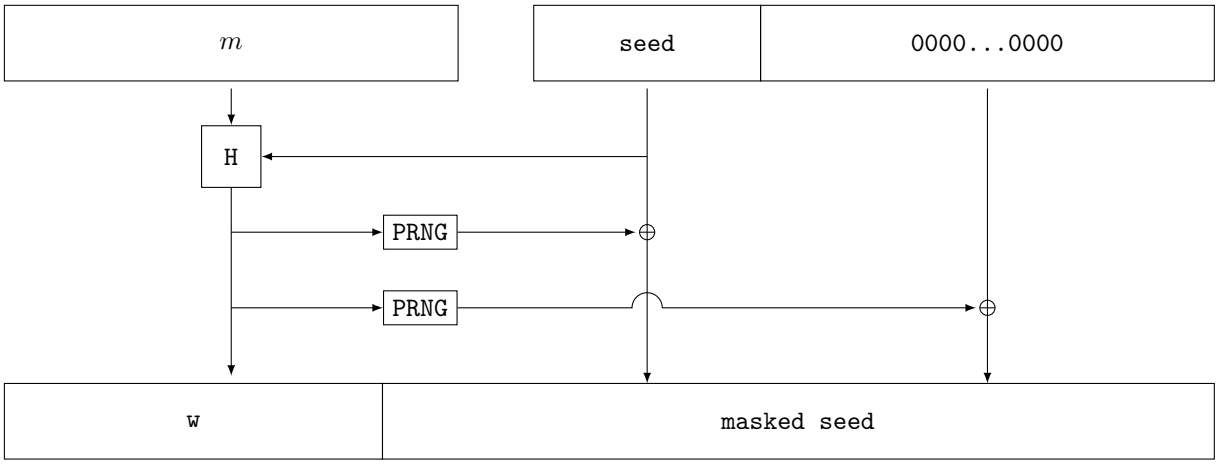
`0x00`	`0x01`	`PS = 0xffff...ffff`	`0x00`	T

4. Signer le bloc bourré (c'est un nombre plus petit que n).

A priori, forger une signature sans posséder la clef secrète est impossible. Si on choisit un message m et qu'on calcule le bloc bourré, il faudrait inverser la fonction RSA. A contrario, si on génère un signature un peu arbitrairement, il y a très peu de chance que si on applique la fonction RSA, ça produise PS correctement (et il reste la difficulté qui consiste à inverser la fonction de hachage).

10.4.2 RSA-PSS

Le bourrage précédent a un côté « bricolage ». Il peut y avoir des problèmes subtils s'il n'est pas implémenté correctement. Le standard actuel, RSA-PSS (« Probabilistic Signature Scheme ») a été conçu pour fournir des garanties. Il est démontré que forger une signature RSA-PSS est aussi difficile qu'inverser RSA.



Dans le fond, l'idée consiste à hacher le message avec des données aléatoires. Ces données aléatoires doivent être inclues dans la signature. Enfin, un « bourrage » pseudo-aléatoire est généré à partir du haché pour occuper tout l'espace du bloc qui va être « déchiffré » par RSA.

Pour vérifier qu'une signature est correcte, on l'élève puissance e modulo n. On récupère en principe le bloc bourré. On recalcule les masques à partir de w, on vérifie qu'on trouve bien les `0000....0000`. On recupère la graine aléatoire, puis on recalcule le haché et on vérifie qu'il est correct.

10.4.3 Signatures « en blanc »

La malléabilité de RSA rend cependant possible des choses surprenantes. Par exemple, on peut faire des « signatures en blanc ». C'est un système de signature où Alice envoie des données à Bob. Bob les signe, mais sans les connaitre !

Typiquement, ce système se passe avec une autorité, à qui on doit faire signer des données qu'on ne souhaite pourtant pas lui révéler. Pour obtenir une signature de m, on choisit un « masque »

aléatoire dans $\mathbb{Z}_n$ (il doit être inversible). On envoie $m' = m \cdot r^e \bmod n$ à l'autorité. Cette dernière calcule une signature RSA (sans bourrage) de m', et nous renvoie donc :

$$\begin{aligned} s' &\equiv (m \cdot r^e)^d \mod n \\ &\equiv m^d \cdot r. \end{aligned}$$

On calcule $s \leftarrow s' \cdot r^{-1} \bmod n$, et on obtient ainsi une signature de m par l'autorité.

L'autorité, elle, ne peut pas connaître m. En effet, comme le « masque » r a été choisi uniformément au hasard, il agit comme un masque jetable. Du coup, tous les messages m sont possibles, avec le bon choix de r. L'autorité ne peut pas les distinguer.

Application Ceci s'applique par exemple au vote électronique. Pour voter, chaque electeur démontre son identité auprès du bureau de vote (en prouvant la connaissance d'une clef secrète, par exemple), et fait signer « en blanc » une autorisation de voter.

Au moment du vote, chaque électeur envoie son vote, chiffré avec la clef publique du bureau de vote, ainsi que son autorisation de voter signée.

Le bureau de vote ne peut pas savoir qui vote pour quoi, car il ne peut pas relier les autorisation de vote à des identités. Et pourtant, personne ne peut voter deux fois !

Une fois le vote terminé, toutes les couples (vote + autorisation) sont publiés. Chacun peut vérifier que son vote a bien été pris en compte (et que la somme est correcte).

10.5 Chiffrement et signature de Rabin

Alors que le système RSA repose sur la difficulté de calculer des racines e-ièmes modulo n avec $e \geq 3$, le chiffrement de Rabin, lui, repose sur la difficulté de calculer des racines *carrées* modulo n.

L'idée de base du chiffrement de Rabin est la suivante : élever un nombre au carré modulo n est très facile, tandis que l'opération inverse (le calcul des racines carrées modulo n) nécessite la connaissance de p et q, comme on va le voir. Ceci pemet d'envisager une méthode de chiffrement à clef publique où le chiffré de x soit $y = x^2 \bmod n$.

Avantages par rapport à RSA : a) le (chiffrement/vérification des signatures) est plus simple et plus rapide, b) les arguments de sécurité sont plus forts. Inconvénients par rapport à RSA : le chiffrement n'est pas injectif (à chaque chiffré correspondent 4 clairs possibles), le (déchiffrement/signature) est plus compliqué.

Pour commencer, oour rendre cette idée concrète, il faut quand même que le propriétaire de la clef secrète puisse déchiffrer, donc qu'on puisse retrouver x à partir de $y = x^2 \bmod n$.

10.5.1 Racines carrées modulo n

On a vu comment calculer des racines carrées modulo un nombre premier p tel que $p \equiv 3 \bmod 4$ dans le § 9.3.2. Rappelons comment ça marche : si a est un résidu quadratique, alors $a^{\frac{p+1}{4}}$ est une racine carrée de a modulo p. En effet :

$$\left(a^{\frac{p+1}{4}}\right)^2 \equiv a^{\frac{p+1}{2}} \equiv a^{1+\frac{p-1}{2}} \equiv a \mod p$$

La dernière égalité découle du critère d'Euler (théorème 13) : puisque a est un résidu quadratique, alors $a^{\frac{p-1}{2}} \equiv 1 \bmod p$. Il nous reste à voir comment calculer une racine carrée modulo n. Pour cela, on va calculer une racine carrée modulo p, puis modulo q, et on va les *recombiner* ensemble pour obtenir une racine carrée modulo n avec la technique des restes chinois.

Mais avant cela, une première remarque s'impose. Posons $n = 11 \cdot 19 = 209$. On calcule :

$$36^2 \equiv 1296 \equiv 42 \bmod 209$$
$$74^2 \equiv 5476 \equiv 42 \bmod 209$$
$$135^2 \equiv 18225 \equiv 42 \bmod 209$$
$$173^2 \equiv 29929 \equiv 42 \bmod 209$$

Il s'ensuit que 42 possède non pas deux, mais *quatre* racines carrées modulo 209. En fait, il y en a deux modulo 11 et deux modulo 19. La « recombinaison » peut alors produire les quatre combinaisons possibles. On voit bien que $36 \equiv -173 \bmod 209$ et $74 \equiv -135 \bmod 209$. Mais en plus des deux racines carrées « normales » x et $-x$, il y en a deux autres.

Principe du calcul des racines carrées. On part d'un nombre a, puis on calcule sa racine carrée modulo p et modulo q (en supposant que $p \equiv q \equiv 3 \bmod 4$) :

$$\alpha^2 \equiv a \mod p$$
$$\beta^2 \equiv a \mod q.$$

Autrement dit : $(\alpha, \beta) \boxtimes (\alpha, \beta) = \Psi_{p,q}(a)$. D'après le théorème des restes chinois, il existe un entier k tel que $\Psi_{p,q}(k) = (\alpha, \beta)$. Ce qu'on a au-dessus se ré-ecrit donc

$$\Psi_{p,q}(a) = \Psi_{p,q}(k) \boxtimes \Psi_{p,q}(k) = \Psi_{p,q}(k^2)$$

L'unicité de la solution garantie par le théorème des restes chinois garantit que $a \equiv k^2 \bmod pq$. Par conséquent, k est une racine carrée de a modulo pq.

Relation entre les racines carrées. On peut aisément se rendre compte que $\Psi_{u,v}(1) = (1, 1)$ et que $\Psi_{u,v}(-1) = (-1, -1)$. D'après le théorème des restes chinois, il existe aussi f tel que $\Psi_{u,v}(f) = (-1, 1)$, et donc logiquement $\Psi_{u,v}(-f) = (1, -1)$.

On a donc trouvé les 4 racines carrées de 1 modulo uv : il s'agit de l'ensemble $\{1, -1, f, -f\}$. L'intérêt de ceci est qu'une fois qu'on a trouvé une racine carrée x d'un nombre quelconque a modulo pq, alors on sait que les 3 autre sont $-x$, fx et $-fx$. Le nombre f est facile à calculer avec l'algorithme d'Euclide. En effet, on a $f \equiv ru - sv \bmod pq$, où r et s sont des entiers tels que $ru + sv = 1$.

Cas particulier des facteurs premiers. Dans le cas où u et v sont des nombres premiers (qu'on va noter p et q), on peut même faire un peu mieux. On a déjà observé que $r \equiv p^{-1} \bmod q$ et $s \equiv q^{-1} \bmod p$. D'après le petit théorème de Fermat, on a alors $r \equiv p^{q-2} \bmod q$ et $s \equiv q^{p-2} \bmod p$. Il s'ensuit que :

$$f = p^{q-1} - q^{p-1} \bmod pq.$$

Du coup, il n'est même pas nécessaire de faire appel à l'algorithme d'Euclide.

Algorithme complet de calcul des racines carrées modulo n**/** On dispose maintenant de tous les ingrédients pour calculer des racines carrées modulo n. On suppose que $n = pq$, où p et q sont deux nombres premiers qui satisfont $p \equiv 3 \bmod 4, q \equiv 3 \bmod 4$. Voici comment calculer les 4 racines carrées de k modulo n

1. [Descente mod p et q] Poser $n_p \leftarrow k \ \% \ p$, $n_q \leftarrow k \ \% \ q$.
2. [Racine carrée mod p et q] Si n_p (resp. n_q) n'est pas un résidu quadratique modulo p (resp. q), alors k n'est pas un carré modulo n. Sinon, calculer une racine carrée α de n_p modulo p et une racine carrée β de n_q modulo q.
3. [Remontée mod n] Calculer les entiers r et s tels que $rp + sq = 1$ avec l'algorithme d'euclide étendu, et calculer $x \leftarrow (\beta rp + \alpha sq) \ \% \ n$.
4. [Autres solutions] Poser $f \leftarrow (rp - sq) \ \% \ pq$. Les racines carrées de k modulo n sont $\{x, -x, fx, -fx\}$.

10.5.2 Chiffrement de Rabin

Génération des clefs. Pour produire sa paire de clefs, Alice choisit deux nombres premiers p et q tels que $p \equiv 3 \bmod 4$ et $q \equiv 3 \bmod 4$. Elle calcule $n = pq$. La clef publique est n, la clef secrète est (p, q).

Chiffrement. Pour envoyer un message chiffré à Alice, Bob doit d'abord le coder comme un élément $m \in \mathbb{Z}_n$. La procédure de chiffrement est alors :

$$\mathcal{E}n, m := m^2 \bmod n$$

Comme le chiffrement RSA, celui de Rabin est déterministe. Tel quel, il ne peut donc pas non plus offrir la sécurité sémantique. En plus, si $m = 0$ ou $m = 1$, le chiffrement ne fait rien. Enfin, si $m < \sqrt{n}$, alors le modulo n'a aucun effet, et peut déchiffrer facilement même sans connaître p et q, juste en calculant une racine carrée habituelle sur les entiers (pas modulo). Pour finir, le chiffrement est malléable : le produit des chiffrés est le chiffré du produit, modulo n.

Déchiffrement. Après avoir reçu le message chiffré C de Bob, Alice calcule les 4 racines carrées modulo n, détermine laquelle correspond au message envoyé par Bob et ignore les 3 autres. Il y a là une nouvelle difficulté : comment retrouver le message envoyé par Bob parmi ces 4 possibilités ?

10.5.3 Sécurité

Le chiffrement de Rabin est-il sûr ? On a listé ci-dessus un certain nombre de défauts. Il reste à se demander s'il est vraiment à sens unique. On a vu comment l'utilisateur légitime, qui possède la clef secrète, peut effectuer le déchiffrement. Qu'en est-il pour un adversaire qui ne possède que la clef publique ? Le résultat ci-dessous est le point fort du mécanisme.

Théorème 17. *Le chiffrement de Rabin est à sens unique tant que la factorisation de n est difficile.*

Démonstration

Supposons qu'on dispose d'un adversaire qui effectue le déchiffrement de Rabin en connaissant juste n. On a donc un algorithme SQRT qui calcule une racine carrée modulo n. On va s'en servir pour trouver la factorisation de n.

Voici l'algorithme. Des remarques le suivent.

1. Choisir aléatoirement $y \in \mathbb{Z}_n$, et calculer son « chiffré » $y^2 \bmod n$.
2. Utiliser l'adversaire pour obtenir $x = \mathrm{SQRT}(y, n)$.
3. Si $x \equiv \pm y \mod n$, alors retourner à l'étape 1.
4. Renvoyer $\mathrm{PGCD}(x - y, n)$.

Voici la logique de son fonctionnement. Si $x \equiv \pm y \bmod n$, alors l'appel à SQRT ne nous a rien appris que nous ne connaissions déjà. De plus, dans ces cas-là, le PGCD calculé à la fin n'a pas une valeur intéressante.

Par contre, quand on nous renvoie une des deux autres racines carrées de y^2 modulo n, en dehors de x et $-x$, là on apprend quelque chose d'utile. En effet on a alors $y^2 \equiv x^2 \bmod n$, et donc $(x - y)(x + y) \equiv 0 \bmod n$. À ce stade, on sait que $(x - y)(x + y)$ est un multiple de pq, donc —par exemple— p divise soit $x - y$ soit $y + x$ (et q divise l'autre). Par conséquent, le PGCD calculé à la fin renvoie soit p soit q.

Quelle est la complexité de tout ceci ? En gros, chaque appel à SQRT nous donne ce qu'on veut avec probabilité $1/2$. En effet, il y a 4 racines carrées possibles, 2 sont inutiles et 2 sont valables. En plus, comme on a choisi y au hasard, l'algorithme SQRT n'a aucun moyen de savoir de laquelle il s'agit. La boucle va donc effectuer deux itérations, en moyenne.

La méthode de chiffrement de Rabin est donc *prouvablement* à sens unique, sous l'hypothèse que la factorisation est un problème difficile. Ce n'était pas le cas du système RSA (ou en tout cas, une preuve d'un tel résultat n'est pas encore connue).

Faiblesse face aux attaques à chiffré choisi. La « preuve de sécurité » peut malheureusement se retourner contre nous. En effet si un adversaire peut nous faire déchiffrer un message de son choix, et obtenir le clair correspondant, alors il a beaucoup de chance de pouvoir factoriser notre clef publique !

Résumons : le chiffrement de Rabin est à sens-unique sous l'hypothèse que la factorisation est difficile, mais il n'est pas *du tout* résistant aux attaques à *chiffré choisi*. C'est dans un tel contexte que la transformation de Pointcheval (cf. § 10.3.4) est particulièrement utile. Une fois qu'elle est appliquée, il n'est plus possible de « produire » des messages chiffrés valides et d'en obtenir le déchiffrement.

10.5.4 Signature de Rabin

Comme la signature RSA, on peut obtenir des signatures de Rabin en utilisant la procédure de chiffrement « à l'envers ». Historiquement, c'est l'un des premiers mécanismes de signature connus.

Contrairement au chiffrement de Rabin, qui n'est pas très largement utilisé, notamment à cause du problème de l'identification du bon clair, la signature de Rabin est largement déployée, au moins en France. Elle est utilisée dans les badges Vigik qui contrôlent l'accès à de nombreux halls d'immeubles.

Les badges Vigik sont justes des mémoires passives, qui contiennent la signature de Rabin d'une autorisation d'ouverture de la porte. Les lecteurs muraux contiennent la clef publique de vérification de signature (c'est-à-dire n). Du coup, voler un lecteur mural (!) ne permet pas de forger des badges valides. La signature de Rabin a été préférée à d'autres alternatives pour des raisons de coût : vérifier une signature de Rabin ne nécessite qu'une élévation au carré modulo n. C'est plus rapide que la plupart des autres solutions concurrentes, et donc ça permet de se contenter d'un processeur *low cost* dans le lecteur mural.

Pour produire une signature de m, on calcule une racine carrée de m modulo n. Pour vérifier qu'une signature est correcte, on l'élève au carré et on vérifie qu'on trouve bien le message original.

Il y a cependant un petit problème qui empêche cette belle idée de fonctionner directement. Comment faire pour signer un message m qui n'est pas un carré modulo n ? Autrement dit : que faire si le calcul de la racine carrée de m échoue ? On peut faire la chose suivante :

1. Choisir un « bourrage » U aléatoirement et calculer $y = H(m \,\|\, U)$.
2. Si y n'est pas un carré, retourner à l'étape 1.
3. Calculer une racine carrée x de y ($x^2 \equiv y \bmod n$). La signature est la paire (x, U).

La sécurité du chiffrement garantit qu'un adversaire ne peut pas produire de signature, même s'il en connaît déjà. En effet, s'il choisit un message m, alors il est confronté au problème de calculer une racine carrée modulo n, et on a vu que c'était aussi dur que la factorisation.

S'il cherche à contourner le problème, c'est-à-dire s'il choisit $z \in \mathbb{Z}_n$, il peut bien calculer $y = z^2 \bmod n$. Il connaît donc une racine carrée de y. Mais alors, il est confronté au problème d'inverser la fonction de hachage, car il doit trouver x tel que $H(x) = z$.

.1 Le nombre de racines d'un polynôme est borné par son degré

On rappelle que α est une **racine** du polynôme P si $P(\alpha) = 0$. Dans cette section on démontre le résultat suivant :

Théorème 18. *Un polynôme non-nul de degré d possède au plus d racines distinctes. Ceci est vrai pour les polynômes à coefficients et à valeurs dans l'ensemble $\mathbb{Z}_p$ des entiers modulo p ainsi que dans $\mathbb{C}$ des nombres complexes.*

On va faire la preuve dans $\mathbb{Z}_p$, mais il suffit de remplacer ceci par $\mathbb{C}$ et elle reste vraie.

Lemme 12. *Soit P un polynôme de degré $d \geq 1$ et $\alpha \in \mathbb{Z}_p$ une racine de P. Alors il existe un polynome Q à coefficients dans $\mathbb{Z}_p$ de degré $d-1$ tel que :*

$$P(X) = (X - \alpha) \cdot Q(X).$$

Démonstration

D'abord, on remarque que si Q existe, alors son degré est $\deg P - 1$. Ecrivons donc :

$$P(X) = \sum_{i=0}^{d} p_i X^i \qquad\qquad Q(X) = \sum_{i=0}^{d-1} q_i X^i$$

Tout d'abord, on fait le petit calcul :

$$\begin{aligned}
(X-\alpha)\cdot Q(X) &= (X-\alpha)\cdot \sum_{i=0}^{d-1} q_i X^i \\
&= X \cdot \left(\sum_{i=0}^{d-1} q_i X^i \right) - \alpha \cdot \left(\sum_{i=0}^{d-1} q_i X^i \right) \\
&= \sum_{i=0}^{d-1} q_i X^{i+1} - \sum_{i=0}^{d-1} \alpha q_i X^i \\
&= q_{d-1} X^d + \sum_{i=1}^{d-1} \left(q_{i-1} - \alpha q_i \right) X^i - \alpha q_0.
\end{aligned}$$

Pour que ceci soit égal à $P(X)$, il faut que ceci ait les mêmes coefficients que $P(X)$, à savoir $p_d, \ldots, p_0$. Pour cela, on effectue le choix magique suivant :

$$q_i = \sum_{k=i+1}^{d} p_k \alpha^{k-(i+1)}, \qquad 0 \leq i \leq d-1$$

Il ne reste plus qu'à vérifier que ceci donne les résultats attendus lorsqu'on le substitue dans les formules d'avant.

— Tout d'abord, on voit que $q_{d-1} = p_d$. Ceci découle de la définition de q_{d-1}, il n'y a rien à justifier.

— Ensuite, on calcule pour $1 \leq i \leq d-1$:

$$\begin{aligned}
q_{i-1} - \alpha q_i &= \sum_{k=i}^{d} p_k \alpha^{k-i} - \alpha \sum_{k=i+1}^{d} p_k \alpha^{k-(i+1)} \\
&= \sum_{k=i}^{d} p_k \alpha^{k-i} - \sum_{k=i+1}^{d} p_k \alpha^{k-i} \\
&= p_i \alpha^{k-k} = p_i \qquad \text{(seul le terme } i = k \text{ reste)}
\end{aligned}$$

— Enfin, on remarque que :

$$\begin{aligned}\alpha q_0 &= \alpha \sum_{k=1}^{d} p_k \alpha^{k-1} \\ &= \sum_{k=1}^{d} p_k \alpha^k \\ &= P(\alpha) - p_0 \\ &= -p_0 \qquad \text{(par hypothèse)}\end{aligned}$$

On en arrive donc à la conclusion que $(X-\alpha)Q(X) = \sum_{i=0}^{d} p_i X^i = P(X)$. □

Preuve du théorème 18. On démontre en fait sa contraposée : si P possède au moins $d+1$ racines, alors $P = 0$. On effectue la preuve par récurrence sur le degré du polynôme P.

Si P est de degré 0, alors le polynôme est constant, et on a $P(X) = v$. Comme on suppose que P possède au moins une racine, alors il est nécessaire que $v = 0$ (sinon P ne peut pas s'annuler). On a donc $P(X) = 0$, et le théorème est bien vrai dans ce cas-là.

Si P est de degré $d > 0$. On suppose que P possède au moins $d+1$ racines $\alpha_0, \ldots, \alpha_d$. D'après le lemme ci-dessus, il existe Q de degré $d-1$ tel que $P(X) = (X-\alpha_0)Q(X)$. En fait, $Q(X)$ doit nécessairement s'annuler sur les *autres* racines. Comme $P(\alpha_i) = 0$, avec $i \geq 1$, on a donc $(\alpha_i - \alpha_0)Q(\alpha_i) = 0$. Comme les racines sont toutes différentes, alors $\alpha_i - \alpha_0 \neq 0$. On peut donc diviser par $\alpha_i - \alpha_0$ et on trouve que $Q(\alpha_i) = 0$. Le polynôme Q, qui est de degré $d-1$, possède donc d racines distinctes. Par hypothèse de récurrence, $Q(X) = 0$. Mais ceci entraine immédiatement $P(X) = 0$. CQFD.

Annexe A

Solutions des exercices

A.1 Chapitre 1

1.1 On va montrer qu'une chaine de bits aléatoire de taille $|M| + k$ va être déchiffrée sans produire d'erreur avec probabilité supérieure à 2^{-k}. Fixons une clef K et choisissons un entier ℓ. On chiffre tous les messages de taille ℓ : il y en a exactement 2^ℓ. L'ensemble des chiffrés qu'on obtient (en essayant tous les aléas possibles, tant qu'on y est) est de taille supérieure ou égale à 2^ℓ, sinon le chiffrement ne serait pas injectif (et serait donc ambigu). Les chiffrés occupent $\ell + k$ bits : parmis les $2^{\ell+k}$ possibles, au moins 2^ℓ sont valides. Donc si on choisit une chaine de $\ell + k$ bits au hasard, la probabilité que ce soit un chiffré valide est supérieure à 2^{-k}.

1.8 Comme il y a 2^n clefs possibles et qu'on en choisi une au hasard, la probabilité de trouver la bonne est 2^{-n}.

En fait, la probabilité que l'algorithme retourne une paire (P, K) est plus grande que ça et dépend de la qualité du mécanisme « d'authentification » du chiffrement. L'exercice 1.1 montre en effet que la probabilité d'obtenir autre chose que $\perp$ est supérieure à 2^{-k} lorsque le mécanisme produit des chiffrés k bits plus gros que les clairs. Pour cette raison, on choisit généralement $k = n$. Du coup, la probabilité de succès de SingleTrial est bien 2^{-n}.

1.9 La probabilité de gagner 4 fois de suite à l'euro-millions est donc de 1 chance sur $(139\,838\,160)^4 \approx (1.4 \cdot 10^8)^4 \approx 3.85 \cdot 10^{32}$. La probabilité de « deviner » une clef de 128 bits en un seul essai est de 1 chance sur 2^{128}. Qu'est-ce qui est plus grand, 10^{32} ou 2^{128} ? Pour le savoir, le plus simple est de passer au logarithme :

$$\begin{array}{lll} \ln 10^{32} & = 32 \times \ln 10 & \approx 32 \times 2.3 \quad \approx 73.6, \\ \ln 2^{128} & = 128 \times \ln 2 & \approx 128 \times 0.7 \quad \approx 89.6. \end{array}$$

On voit donc que 2^{128} est environ $e^{89.6-73.6} \approx 8\,886\,110$ fois plus grand que 10^{32}. Par conséquent, il est plus probable de gagner 4 fois de suite à l'Euro-millions que de casser une clef de 128 bits par force brute.

1.12 Non : il peut envoyer des chaines de bits arbitraire à Bob, et Bob va essayer de les déchiffrer... Mais comme l'adversaire ne connaît pas K, il ne peut pas produire de nouveaux chiffrés valides que Bob réussirait à déchiffrer correctement.

1.13 Un adversaire pourrait tenter de se faire passer pour Bob avec une certaine chance de succès. Il ne peut pas déchiffrer le message de l'étape I1, mais il pourrait essayer de « deviner » la valeur de N choisie par Alice. Il génère donc N', au hasard, et le renvoie à Alice. Si N est codé sur n bits, il a une chance sur 2^n de réussir. Il faut donc que n soit assez grand ($n = 128$ est un bon choix).

1.14 Un adversaire (Charlie) qui capte une session du protocole peut la *rejouer* et ré-expédier à Alice le message de l'étape J1 : ceci devrait convaincre Alice que Charlie connaît K, or c'est faux !

1.15 Bob démontre qu'il est capable de chiffrer avec la clef K (et donc qu'il la connaît). Sans connaître K, il est impossible de répondre. Un adversaire actif peut (en se faisant passer pour Alice) faire chiffrer ce qu'il veut par Bob. Il faut donc que le chiffrement résite aux attaques par clair choisi.

1.16 Il faut stocker $(\{K\}_{pwd}, \{data\}_K)$ sur le disque, où K est une clef aléatoire choisie une bonne fois pour toute, et pwd est le mot de passe. Lors d'un changement de mot de passe, il suffit de rechiffrer K.

1.17 Il ne pourra pas lire la réponse chiffrée de l'étape P2. Il ne connaîtra donc pas K_{ab}, et ne pourra pas envoyer correctement le message de l'étape P4.

1.18 Il ne pourra ni fabriquer le message de l'étape P2 (car il ne possède pas K_a).

1.19 B ne vérifie pas que A connait bien K_{ab}. Ceci permet l'attaque (active) suivante : un adversaire enregistre une session du protocole. Plus tard, il ré-emet le message de l'étape NS3 : B ouvre une session croyant parler à A. L'adversaire, qui ne connaît pas K_{ab}, peut néanmoins rejouer les messages précédents (correctement chiffrés avec K_{ab}). C'est ennuyeux si un des messages dit « voici un virement de $xxxxx$ € » !

1.20 Dans les étapes d'interaction avec Bob, l'identité de l'initiateur du protocole ne figure dans aucun des messages. Bob ne peut identifier son partenaire que grace à ses propres déclarations ou au protocole réseau (qui n'est pas forcément fiable).

Par exemple, C exécute normalement le protocole pour entrer en contact avec B. Une clef de session K_{cb} est établie et lors de l'étape NS3, C envoie (légitimement) à B le message $\{K_{cb}\}_{K_b}$.

Une fois que tout ceci est terminé, C tente de se faire passer pour A auprès de C. Il suffit qu'il exécute les étapes NS3–NS5 en envoyant exactement les mêmes choses, mais en prétendant être Alice. Comment Bob pourrait-il faire la différence ?

1.21 Le problème c'est que dans l'étape NS2, rien ne lie la réponse de S au message précédent. Un adversaire actif pourrait donc se faire passer pour S auprès de A – ceci dit, s'il ne connaît pas K_a ni K_b, il ne pourra que rejouer des messages de l'étape NS2 précédemment captés. C'est toutefois gênant car cela force Alice et Bob à utiliser toujours la même clef de session K_{ab}.

A.2 Chapitre 2

2.1 En gros, la complexité est $\exp\left(\alpha \sqrt[3]{n \ln^2 n}\right)$ où $\alpha \approx 1.70$. Ce qu'on cherche est donc :

$$\begin{aligned} T(1024)/T(768) &= \exp\left(\alpha \left[\sqrt[3]{1024 \ln^2 1024} - \sqrt[3]{768 \ln^2 768} \right]\right) \\ &\approx e^{7.27} \\ &\approx 1440 \end{aligned}$$

2.8 On note $I(A)$ lorsqu'Irène prétend être Alice.

NS1-a.	A	$\longrightarrow$	I	:	$\{N_a\}_{pk_i}$
NS1-b.	I(A)	$\longrightarrow$	B	:	$\{N_a\}_{pk_b}$
NS2-b.	B	$\longrightarrow$	I(A)	:	$\{N_a, N_b\}_{pk_a}$
NS2-a.	I	$\longrightarrow$	A	:	$\{N_a, N_b\}_{pk_a}$
NS3-a.	A	$\longrightarrow$	I	:	$\{N_b\}_{pk_i}$
NS3-b.	I	$\longrightarrow$	B	:	$\{N_b\}_{pk_b}$

2.9 Dans le protocole modifié, Bob va répondre $\{B, N_a, N_b\}_{pk_a}$ dans l'étape NSL2. La feinte de l'attaque consistait en ceci Irène relayait ce message, sans le modifier (il est opaque pour elle, qui n'a pas la clef secrète d'Alice). Maintenant, si elle le relaye, Alice reçoit un message qui contient « Bob » dedans, alors qu'elle est en train d'exécuter le protocole avec Irène, donc la fraude est détectée.

2.17 Pour montrer que le problème est dans NP, il faut exhiber un certificat vérifiable en temps polynomial lorsque la réponse est « OUI ». En fait, (m, r) est un tel certificat, et la vérification est très simple : il suffit de calculer $\mathcal{E}(pk, m, r)$ et de vérifier si ça donne bien c.

Pour montrer que le problème est dans coNP, il faut exhiber un certificat vérifiable en temps polynomial lorsque la réponse est « NON ». La technique précédente ne marche plus. Mais on peut alors utiliser la clef secrète sk qui correspond à pk comme certificat : il suffit de lancer l'algorithme de déchiffrement et de s'assurer qu'il échoue (au passage, sk peut aussi faire office de certificat pour « OUI »).

Un problème est qu'il faudrait vérifier que sk et pk sont bien liées. Une manière de résoudre ce problème consiste à remarquer que pk et sk sont produit par un algorithme de génération de clef :

$$(sk, pk) \leftarrow \mathcal{KG}(n, r), \qquad r \text{ est aléatoire.}$$

Du coup, au lieu de fournir sk comme certificat, on fournit (n, r). Vérifier le certificat revient à vérifier que pk est correctement obtenu à partir de (n, r), puis effectuer le déchiffrement avec sk.

A.3 Chapitre 3

3.2 Si le serveur stocke x_n, l'utilisateur envoie $y \leftarrow x_{n-1}$. Le serveur vérifie que $H(y) = x_n$. Si c'est bien le cas, l'utilisateur est connecté, et le serveur stocke y. Au prochain coup, le client devra envoyer x_{n-2}. La fonction de hachage doit être à sens unique.

3.7 Pour effectuer une mise en gage de deux bits b_1 et b_2, il suffit de choisir deux chaines aléatoires r_1 et r_2, et de dire que :

$$\textsc{Commit}\left(r_1 \parallel r_2, b_1 b_2\right) = \textsc{Commit}\left(r_1, b_1\right) \parallel \textsc{Commit}\left(r_2, b_2\right).$$

Bref, il suffit de concatener des mises en gage (avec des aléas indépendants) des deux bits. La technique se généralise aisément à autant de bits qu'on veut.

3.8 $\Rightarrow$ Raisonnement par contraposée. Supposons qu'il existe r, r' tels que $\textsc{Commit}(r, 0) = \textsc{Commit}(r', 1)$. Il est donc possible d'effectuer une mise en gage de zéro (avec l'aléa r) puis de l'ouvrir comme une mise en gage de 1 (avec r'). Trouver r et r' peut être calculatoirement difficile, mais ils existent : avec une puissance de calcul illimitée on finirait bien par les trouver. Le schéma ne peut pas être *information-theoretically binding*.

$\Leftarrow$ Supposons qu'il n'existe pas de « mauvaise paire » (r, r'). Cela signifie qu'une mise en gage du bit b ne peut pas être ouverte autrement qu'en mise en gage du bit b, et ceci quelle que soit la puissance de calcul déployée.

3.9 $\Leftarrow$ Si une telle paire existe tout le temps, alors n'importe quelle chaine de k bits peut être une mise en gage de 0 tout comme une mise en gage de 1, et on ne dispose d'aucun moyen de savoir ce qui s'est passé.

$\Rightarrow$ Raisonnons par l'absurde, et supposons donc qu'il existe une mise en gage y pour laquelle il n'y a pas de telle paire r, r'. Démontrons alors qu'un adversaire non-limité en puissance de calcul peut « cracker » y et détecter si c'est une mise en gage de zéro ou de un. La stratégie consiste à tester tous les aléas r possibles avec $b = 0$ et $b = 1$ (en commençant par ceux de taille 1, puis 2, puis...). Au bout d'un moment on va tomber sur une valeur de r qui ouvrira correctement la mise en gage y avec une certaine valeur de b. Le point, c'est que c'est la bonne valeur, puisqu'il n'existe pas de paire r, r' qui permette d'ouvrir y des deux manières possibles. Le temps de calcul de la procédure est très élevé, mais on s'en fiche, on a un budget illimité.

3.10 On pose $\textsc{Commit}(r, b) = H(r \parallel b)$.

Le schéma est *computationally binding* si la fonction de hachage est résistante aux collisions. En effet, pour pouvoir « tricher » lors de l'ouverture, il faut être capable de produire une paire r, r' tels que $H(r \parallel 0) = H(r' \parallel 1)$, donc en particulier une collision sur H.

Pour que le schéma de mise en gage puisse être *information-theoretically binding*, il faudrait qu'il ne puisse pas y avoir de paire r, r' tels que $H(r \parallel 0) = H(r' \parallel 1)$. Or, ceci serait inhabituel pour une fonction de hachage cryptographique. En particulier, ce n'est pas le cas si H est un oracle aléatoire. Justification : fixons une empreinte y quelconque, et prenons une séquence de messages $x_1, \ldots, x_n$. La probabilité que « $H(x_i \parallel 0) \neq y$ pour tout $1 \leq i \leq n$ » tend vers zéro lorsque $n \to +\infty$. Même chose pour « $H(x_j \parallel 1) \neq y$ pour tour j ». Par conséquent, la probabilité qu'il n'existe pas de « mauvaise paire » (r, r') est nulle.

Le même raisonnement permet de se convaincre que le schéma de chiffrement est *statistically hiding* dans le modèle de l'oracle aléatoire.

A.4 Chapitre 5

5.1 On fait le raisonnement par contraposée : si $P = NP$, alors il existe un algorithme polynomial qui inverse f.

Tout d'abord, le problème de décision donné ci-dessus est bien dans NP, car x fait office de certificat compact et vérifiable en temps polynomial.

Si $P = NP$, alors ce problème peut être décidé en temps polynomial. Il ne reste plus qu'à construire un algorithme polynomial qui inverse f à partir de ça.

Tout d'abord, on vérifie si ε est bien un préfixe d'une preimage de y (ε désigne la chaine vide). Si ce n'est pas le cas, alors y n'a pas d'inverse pour f et on s'arrête là.

Sinon, on initialise $x^* \leftarrow \varepsilon$. Tant que x^* est trop petit, tester si x^*0 est un préfixe d'une préimage de y. Si oui, faire $x^* \leftarrow x^*0$, sinon faire $x^* \leftarrow x^*1$ — Chaque nouveau test révèle un bit de la préimage de y qu'on cherche.

5.2 On le montre par contraposée : on montre en fait que si on peut trouver facilement des préimages, alors on peut trouver facilement des secondes préimages. Et en effet, si on avait sous la main un algorithme qui calcule des préimages, alors on pourrait l'utiliser directement pour calculer une préimage de l'empreinte du message de départ.

5.3
- — *Résistance aux secondes préimages de g.* On nous donne un message M, et on cherche à produire $M' \neq M$ tel que $g(M) = g(M')$. Si M occupe n bits, on ne va pas y arriver : une minute de reflexion nous convaincra que M est la seule entrée possible qui peut produire cette sortie pour g.
  Si M ne fait pas n bits, alors une seconde préimage M' ne peut pas non plus faire n bits. Mais alors $g(M) = g(M')$ implique que $h(M) = h(M')$. Le M' qu'on aurait trouvé serait alors une seconde préimage de M pour la fonction h, qui est supposée être résistante aux secondes préimages.
- — *g n'est pas à sens unique.* On nous donne une empreinte y de $n+1$ bits et on nous met au défi de trouver une préimage x (c.a.d. x t.q. $g(x) = y$). Si $y[0] = 0$, on laisse tomber. Sinon, on renvoie $x \leftarrow y[1 : n+1]$. On arrive donc à inverser g sans effort dans 50% des cas.

5.4 Admettons qu'on nous donne un algorithme $\mathcal{A}$ qui trouve des secondes préimages. Voici comment produire des collisions : 1) choisir un message M aléatoirement, 2) calculer $M' \leftarrow \mathcal{A}(M)$, une seconde préimage de M. On a alors $M \neq M'$ et $h(M) = h(M')$. On a donc un mécanisme qui produit des collisions pour H. Comme ceci n'est pas possible efficacement, c'est que $\mathcal{A}$ ne peut pas être efficace.

5.5 L'attaque fonctionne de la façon suivante :

1. on choisit un bloc de message aléatoire m, on calcule $x = F(IV, m)$, puis on regarde si x est l'une des valeurs intermédiaires h_i obtenues pendant qu'on hache M.
2. Si $x = h_k$, alors on renvoie $M' = m\|M_{k+1}\|M_{k+2}\|M_{k+3}\| \ldots$
3. Sinon, on retourne à l'étape 1

Lorsque cette procédure s'arrête, elle renvoie bien une seconde préimage de M : La même valeur de chainage est obtenue après le traitement de m et $M_0, \ldots, M_k$, et ensuite les préfixes sont identiques. Comme la taille du message n'apparaît pas dans le bourrage, les deux blocs de « bourrés » sont identiques, et du coup les empreintes sont identiques.

Avec SHA256, la taille des messages en bits est codée sur 128 bits, et les blocs font 512 bits, donc le nombre maximal de blocs est $2^{128-9} = 2^{119}$. On peut donc espérer trouver une seconde préimage d'un tel message en testant $2^{256-119} = 2^{137}$ blocs m, au lieu de 2^{256} en théorie. Ce n'est aas du tout envisageable en réalité, mais c'est plus (sensiblement) rapide que prévu en théorie.

A.5 Chapitre 8

8.1 On n'a pas de garantie sur l'identité de son partenaire. On pourrait croire qu'on communique avec un serveur bancaire, alors que la communication a été interceptée et qu'on révèle ses identifiants bancaires à un serveur pirate.

8.2 On calcule $d \leftarrow k^{-1} \bmod p-1$ (ceci limite le choix de la clef k, qui doit être inversible modulo $p-1$) et on pose $\mathcal{D}(k,y) = y^d \bmod p$.

Vérifions que c'est correct. Par construction, $kd \equiv 1 \bmod p-1$ donc il existe un entier q tel que $kd = 1 + q(p-1)$. On trouve alors $\left(x^k\right)^d \equiv x^{1+q(p-1)} \equiv x\left(x^{p-1}\right)^q \equiv x \bmod p$ (en vertu du petit théorème de Fermat).

8.3 L'idée générale est la suivante : Bob et Alice partagent un nombre secret x ainsi qu'une clef de l'AES k. Lorsqu'Alice envoie un message sensible M, elle génère en passant un autre message M' favorable au régime. Elle effectue le chiffrement Elgamal du message M', mais en utilisant $\texttt{AES}(k,M)$ comme aléa. Autrement dit :

$$\mathcal{E}(M,M') = \left(\texttt{AES}(k,M), M' \times \texttt{AES}(k,M)^x \bmod p\right).$$

Dans la situation normale, Bob déchiffre M avec k (déchiffrement AES). Dans la situation d'urgence, Bob prétend que le chiffrement Elgamal a été utilisé et révèle sa clef secrète x. Le déchiffrement avec x révèle les messages M'.

8.4 Le mécanisme est *perfectly binding* si une mise en gage de 0 ne peut pas être « ouverte » comme une mise en gage de 1. Or ici, c'est bien le cas : $g^0 = 1 \neq g = g^1$...

Le bit mis en gage n'est pas du tout dissimulé : si la mise en gage est 1, c'est que le bit mis en gage est 0.

8.5 D'après l'exercice 3.8 (en généralisant un peu), le mécanisme est *perfectly binding* s'il n'existe pas de paire r, r' telle que $\textsc{Commit}(r,m) = \textsc{Commit}\,(r',m')$ avec $m \neq m'$.

Or, $\textsc{Commit}(r,m) = \textsc{Commit}\,(r',m')$ implique la double congruence

$$\begin{aligned} g^r &\equiv g^{r'} \mod p, \\ mh^r &\equiv m'h^{r'} \mod p. \end{aligned}$$

La première implique (en vertu du théorème 9) que $r \equiv r' \bmod q$. Comme h est lui aussi d'ordre q, il s'ensuit que $h^r \equiv h^{r'} \bmod p$ (toujours théorème 9, mais dans l'autre sens cette fois). Comme h doit être inversible, multiplier des deux côtés par l'inverse de h^r donne $m \equiv m' \bmod p$, et donc le schéma de mise en gage est bien *binding*.

Pour le côté *hiding* : tout d'abord, si on est capable de casser le logarithme discret, alors on peut récupérer r à partir de g^r, puis récupérer le message. Le mécanisme de mise en gage ne peut donc pas être *statistically hiding*. Mais dans le fond, la mise en gage est essentiellement le chiffrement élgamal du message (où h joue le rôle de la clef publique), donc extraire le message de la mise en gage ne doit pas être facile.


Il est possible d'argumenter que briser la propriété de *hiding* du schéma de mise en gage revient grosso-modo à briser la sécurité sémantique du chiffrement Elgamal, ce qui revient précisement à résoudre le problème Diffie-Hellman décisionnel. Tous les détails dans le chapitre 9.

8.6 D'après l'exercice 3.9, le mécanisme est *statistically hiding* si pour tout y (de la bonne taille) il existe une paire r, r' telle que $y = \textsc{Commit}(r,0) = \textsc{Commit}\,(r',1)$. Soit donc y un élément du groupe engendré par g modulo p. Il existe donc un exposant u tel que $y = g^u$. Notons aussi $h = g^x$.

Pour que $y = \textsc{Commit}(r,0)$, il faut que $y = h^r$, donc que (th. 9) $u \equiv rx \bmod q$. Il suffit donc de choisir $r \leftarrow ux^{-1} \bmod q$ (x est inversible modulo q car $x \neq 0$ et q est premier).

Pour que $y = \textsc{Commit}(r,1)$, il faut que $y = gh^r$, donc que $u \equiv 1 + rx \bmod q$. Il suffit donc de choisir $r' \leftarrow (u-1)x^{-1} \bmod q$.

Le mécanisme est donc *statistically hiding*. Calculer r et r' tels que définit ci-dessus nécessite de connaître les logarithmes de y et de h en base g, ce qui n'est pas forcément faisable calculatoirement, mais ce n'est pas le problème : il suffit de savoir qu'ils existent.

Passons au côté *binding*. Le raisonnement ci-dessus montre que quelqu'un qui pourrait calculer des logarithmes en base g pourrait briser la propriété de *binding*. Supposons qu'on dispose d'un algorithme $\mathcal{A}$ qui prend en argument g, h, y et qui produise r, r' tels que $y \equiv h^r \equiv gh^{r'} \bmod p$. Comme h est inversible, on déduit de ces deux congruences que $1 \equiv gh^{r'-r} \bmod p$. On en déduit (th. 9) que $0 \equiv 1 + x(r' - r)$, où x est tel que $h = g^x$. Alors, on trouve $x = 1(r' - r)^{-1} \bmod q$, et on a obtenu le logaritme discret de h en base g modulo p. Par conséquent, l'algorithme $\mathcal{A}$ ne peut pas être sensiblement plus efficace qu'un algorithme qui résoud le logarithme discret, et le schéma mise en gage est *computationally binding*.

8.7 On obtient deux signatures de deux messages m_1, m_2 avec le même $r = g^k$, notons-les (r, s_1) et (r, s_2). Si elles sont valides, on a

$$\begin{aligned} g^{m_1} &\equiv h^r r^{s_1} \mod p, \\ g^{m_2} &\equiv h^r r^{s_2} \mod p. \end{aligned}$$

Comme g est inversible, on multiplie la première par l'inverse de la second et on trouve : $g^{m_1 - m_2} \equiv r^{s_1 - s_2} \bmod p$. Par le théorème 9, il s'ensuit que $m_1 - m_2 \equiv k(s_1 - s_2) \bmod q$. Ceci permet de calculer $k \leftarrow (m_1 - m_2)(s_1 - s_2)^{-1} \bmod q$ (attention, si q n'est pas premier, l'inverse n'existe pas forcément. Dans ce cas-là, il faut éventuellement d'autres signatures...).

Un fois qu'on a k, on re-regarde la première congruence qui nous donne, via le th. 9 : $m_1 \equiv rx + ks_1 \equiv q$, et ceci permet de récupérer la clef secrète ! $x \leftarrow (m_1 - ks_1)r^{-1} \bmod q$.

8.8 Ce protocole est résistant aux attaques hors-ligne par dictionnaire. Un adversaire peut tenter le déchiffrement de $\{pk\}_{pwd}$ avec tous les mots de passes possibles. Mais il n'a aucun moyen de vérifier qu'il est en train d'essayer le bon : il ne peut pas obtenir K sans casser le chiffrement Elgamal, et donc pour « confirmer » *pwd* il lui faudrait un moyen de décider si pk est une clef publique Elgamal valide. Or vu que g est une racine primitive, $pk = g^x$ est uniformément distribué dans $\mathbb{Z}_p^*$, et donc il n'est pas possible de distinguer pk d'un nombre aléatoire inférieur à p. Par contre, si on trouve un nombre supérieur à p on sait que *pwd* n'est pas bon (ceci doit se produire en moyenne avec probabilité 50%).

Voilà une situation où il ne faut *PAS* utiliser un chiffrement authentifié !

8.10 Démontrons le résultat intermédiaire

Démonstration

> Si a est d'ordre $n - 1$ modulo n, alors les $n - 1$ nombres :
>
> $$a^1 \bmod n, \quad a^2 \bmod n, \quad \ldots, \quad a^{n-2} \bmod n, a^{n-1} \bmod n,$$
>
> sont tous différents (en vertu du lemme 5). Il y a donc là les entiers $1, 2, 3, \ldots, p - 1$, puisqu'il ne peut pas y avoir zéro. De plus chacun de ces nombres est inversible (comme a est inversible, on écrit b son inverse, et l'inverse de a^i est b^i).
>
> Cela signifie que n n'a aucun diviseur en commun avec chacun des entiers inférieurs à n. Autrement dit, n est premier.

Ceci nous dit que pour assembler un certificat compact et facilement vérifiable de la primalité de n, on peut se contenter d'exhiber un élément a de $\mathbb{Z}_n$ et prouver qu'il est d'ordre $n - 1$. Pour cela, on fournit la liste de tous les diviseurs premiers $d_1, \ldots, d_k$ de $n - 1$ et l'algorithme de vérification consiste vérifier que les conditions du lemme 7 sont bien réunies.

Pour que le certificat soit complet, il faut *aussi* qu'il permette de justifier que les d_i sont tous premiers ! Mais pour ça on peut utiliser la même technique récursivement (ça donne bien des certificats de taille polynomiale, qui se vérifient en temps polynomial).

On connaît depuis 2002 un algorithme polynomial déterministe pour tester la primalité, dû à trois chercheurs indiens : Manindra Agrawal (1966–), Neeraj Kayal (≈ 1984–), et Nitin Saxena (1981–). Mais il est très lent (bien plus que le test de Miller-Rabin qui est probabiliste) et... trop compliqué pour nous.

Annexe B

Solution des exercices de TD

Réservé à l'équipe pédagogique !

B.1 Chapitre 1

1.2 $16 \times 8 \times 4 \times 140 \cdot 10^6 = 71\ 680\ 000\ 000$ DES/s.

1.3 $t = 2^{56}/71.68 \cdot 10^9 \approx 1005268$ secondes, soit entre 11 et 12 jours.

1.4 $t = 2^{80}/71.68 \cdot 10^9 \approx 0.5$ millions d'années.

1.5 Si on autorise 64 caractères différents, on a 64^8 possibilités, soit $(2^6)^8 = 2^{48}$ combinaisons. Un mot de passe « compliqué » (`p@B3Lz+vS^'`) a donc la même « force » qu'une clef de 48 bits, ce qui est très peu !

1.6 Un seul coeur fait $7 \times 24 \times 3600 \times 8 \cdot 10^6 \approx 4.8 \cdot 10^{12}$ essais par semaine. Il faut faire 2^{48} essais en tout, donc il faut $2^{48}/4.8 \cdot 10^{12} \approx$ une soixantaine de coeurs.

1.7 La terre peut faire au maximum $5.972 \cdot 10^{24} \times 1.36 \cdot 10^{50} \approx 8 \cdot 10^{74}$ opérations par seconde. Donc, pour faire les $2^{512} \approx 1.34 \cdot 10^{154}$ opérations qui s'imposent, il faut au moins $\approx 5 \cdot 10^{71}$ années. Rappel : a priori, le soleil disparaît dans 10^9 années.

1.10 Admettons que chaque essai a une probabilité de succès de 2^{-n} (cf. correction de l'exercice 1.8). On fait k essais, donc la probabilité de succès totale est inférieure à $k2^{-n}$ (inégalité de Boole). Si k est faible devant 2^n, alors cette majoration est précise.

Pour obtenir un résultat plus précis, il faut regarder la probabilité que ManyTrials ne trouve *pas* la solution. Chaque essai rate avec probabilité $\left(1 - 2^{-n}\right)$. Chacun des k essai est indépendant des autres, donc la probabilité que les k essais ratent est $\left(1 - 2^{-n}\right)^k$. La probabilité que l'attaque réussisse après k essais est donc :

$$1 - \left(1 - 2^{-n}\right)^k.$$

Cette formule est valable quelles que soient les valeurs de k et n. Des approximations existent lorsque k est faible (on retrouve $k2^{-n}$) ou lorsque k est proche de 2^n.

1.11 On a vu dans la correction de l'exercice du dessus qu'il faudrait environ 10 millions d'essais pour dépasser la probabilité de 4 gains consécutifs à l'Euro-millions.

B.2 Chapitre 2

2.2 Il s'agit de calculer $\log_2 T(2048)$, ce qui donne :

$$\begin{aligned}\log_2 T(2048) &= (\log_2 e) \times \alpha \sqrt[3]{2048 \ln^2 2048} \\ &\approx 120.6.\end{aligned}$$

Ceci est assez approximatif, car cette complexité est asymptotique et ne prend pas en compte la constante dans le $\mathcal{O}(\ldots)$.

2.4 Commençons par calculer le ratio des complexités :

$$\begin{aligned} T(1024)/T(512) &= \exp\left(\alpha(\sqrt[3]{1024\ln^2 1024} - \sqrt[3]{512\ln^2 512})\right) \\ &\approx 10^7 \end{aligned}$$

Il faut donc 10^7 coeurs pour tenir la deadline. Cela fait $10^7/36 = 277\ 778$ serveurs, soit un budget d'acquisition d'environ un milliards d'euros.

2.5 $277\ 778 * 400 = 100$MW tout pile.

2.6 Oui, d'ailleurs ça tient dans un sous-marin. On peut empiler 48 serveurs de ce type dans un rack, qui occupe environ $1m^2$ au sol, et 2m de haut. Un sous-marin fait 100m de long, et sa section est une ellipse de 9m par 15m (au meilleur endroit). La surface de la tranche est donc $\pi \cdot 9 \cdot 15 \approx 423m^2$, et le volume est (environ) $423\ 000m^3$. Il faut 5800 racks, qui occupent comparativement un volume faible.

2.7 Commençons par calculer le ratio des complexités :

$$\begin{aligned} T(2048)/T(1024) &= \exp\left(\alpha(\sqrt[3]{2048\ln^2 2048} - \sqrt[3]{1024\ln^2 1024})\right) \\ &\approx 10^9. \end{aligned}$$

B.3 Chapitre 3

3.3 Ceci pourrait se faire avec une technique à clef publique

Cependant, pour des raisons d'efficacité, chez Amazon le mécanisme mis en place repose sur la crypographie symétrique. Amazon affecte à chaque compte une chaine de bit secrète, qu'on va noter K. On note `REQUEST` le contenu de la première ligne de la requête, qui contient donc le type de requête HTTP (« `DELETE` ») ainsi que l'URL demandé (`/PAC/exam-2017.tex`), le tout suivi de « `HTTP/1.1` ».

Le contenu de l'entête `Authorization` est :

$$\texttt{Authorization} = \mathsf{HMAC}_K(\texttt{REQUEST} \,\|\, \texttt{Host} \,\|\, \texttt{Date})$$

La définition de HMAC n'est pas importante ici ; il suffit de savoir que c'est un MAC sûr.

Quel est l'intérêt d'inclure `REQUEST` dans le MAC ? Et `Date` ?

mon voisin, qui n'aime pas la musique que j'écoute trop fort, cherche à me nuire. Il essaye d'envoyer à mon fournisseur de cloud des requêtes à ma place, pour me perturber et altérer mes fichiers. Cependant, il n'est pas très bon en crypto.

Qu'est-ce qui l'empêche d'envoyer une requête `DELETE PAC/* HTTP/1.1` ?

Supposons que mon voisin parvienne à intercepter une requête que j'ai moi-même envoyée (et dont l'`Authorization` est donc valide). Que peut-il en faire ?

B.4 Chapitre 5

Index