Do It Yourself Privacy Guard (DIYPG)
Sujet V0.9

Vincent Dugat

Janvier 2020

NOTES IMPORTANTES :

Modalités du projet :

- La partie pilotage de projet et ses documents sera traitée dans le cours "Initiation à la gestion de projet" et gérée par les intervenants de ce cours.
- Vous aurez un tuteur pour la partie développement et organisation du code.
- Le code est à programmer en langages C et Java selon les modalités décrites dans les documents correspondants.
- La page Moodle est commune à la partie Pilotage de projet et codage.
- Normalement à plusieurs on va plus loin. L'organisation de votre équipe et son efficacité en tant que telle, est un point important du projet.

— **Modalités de correction** :

- Vous aurez une note de pilotage de projet et une de codage. La note de codage inclus la recette du code.
- La présence à la recette est obligatoire (sinon ABI).
- Les deux notes comptent pour 50% chacune dans la note finale.
- Le total de tous les points est de 40. Ce score est à partager entre tous les membres d'une équipe (la note finale est sur 10 points qui seront ajoutés à la note de pilotage de projet elle-même sur 10 points). Le total sera sur 20 points.
- Le partage du score est par défaut équitable. Ce principe peut être modifié par le tuteur de développement ou à la demande de l'équipe (il est alors souhaitable qu'il y ait consensus).
- Le projet est structuré en phases. Il est conseillé d'étudier ces phases pour déterminer celles qui peuvent être faites en parallèle (vous êtres plusieurs) et celles qui ont un ordre naturel.
- Toutes les phases sont à faire en C sauf la blockchain qui sera en Java.

Table des matières

1 Introduction

Le projet concerne la cryptographie asymétrique selon le système RSA. Dans ce système il y a deux clés : une clé publique servant à envoyer un message chiffré à quelqu'un, et une clé privé servant à ce quelqu'un à déchiffrer le message. Le même système peut servir aussi à signer électroniquement un message avec une clé privée et à vérifier une signature avec la clé publique.

Le sujet sera découpé en plusieurs parties. Chaque partie sera constituée d'une explication et d'un cahier des charges de réalisation. Il est conseillé de faire une première lecture globale de tout le sujet pour comprendre où le sujet cherche à vous amener.

!! Spoiler !! : il s'agit de programmer un système du type GPG.

2 Le système RSA.

Le sujet est basé sur le document suivant.

`https://zestedesavoir.com/tutoriels/723/la-cryptographie-asymetrique-avec-rsa/`

C'est là que vous trouverez les détails des explications.

Le système cryptographique RSA a été inventé en 1977, et publié en 1978, par trois mathématiciens qui travaillaient alors au MIT : Ronald Rivest, Adi Shamir et Leonard Adleman. Comme vous pouvez le constater, le nom de l'algorithme est simplement tiré des initiales de leurs noms de famille. La grande réussite de leur travail en commun est sans doute due au fait que leurs compétences se complètent.

Rivest est un grand spécialiste de la cryptographie, la discipline qui cherche à protéger les messages d'une lecture inopportune : il est notamment à l'origine de l'algorithme MD5.

Shamir est au contraire un spécialiste de la cryptanalyse, dont le but est de casser les protections mises en place par la cryptographie : il est notamment connu pour avoir cassé en 1982 le cryptosystème de Merkle-Hellman, un algorithme proche de RSA mais reposant sur une base mathématique un peu différente.

Adleman, quant à lui, travaillait à l'époque sur la bio-informatique, en gros, des ordinateurs utilisant de l'ADN pour faire leurs calculs.

Le système RSA est différent des systèmes cryptographiques dits symétriques où toute la solidité du chiffrement repose sur une « clé » secrète, qui doit être connue à la fois de l'envoyeur et du récipiendaire.

Par exemple, le chiffre dit « de César » est un système symétrique. Il s'agit de décaler chaque lettre du message d'un certain nombre de rangs dans l'alphabet : avec un décalage de trois rangs, A devient D, B devient E, Z devient C, etc. Ainsi, le message « trololo » devient « wururor ». Il suffit au récipiendaire de décaler les lettres du message chiffré de trois rangs dans l'autre sens pour retrouver le message d'origine. Et dans l'affaire, la clé n'est autre que le nombre de rangs dont il faut décaler les lettres : cette information doit être connue des seuls interlocuteurs, mais les deux interlocuteurs doivent la connaître.

Un algorithme de chiffrement est un ensemble d'opérations, généralement mathématiques, à effectuer sur le message que l'on veut protéger, afin de le rendre en théorie incompréhensible par ceux auxquels il n'est pas destiné. Il vient toujours avec un algorithme de déchiffrement, qui permet au récipiendaire du message de retrouver le message d'origine. Parfois, ces deux algorithmes sont exactement identiques, comme dans le cas du ROT-13, un cas particulier du chiffre de César.

Tant qu'on y est, le message d'origine, avant toute forme de chiffrement, est appelé message en clair. Une fois qu'un algorithme de chiffrement lui a été appliqué, on l'appelle message chiffré.

Un algorithme de chiffrement/déchiffrement décrit une suite d'opérations qui est la même pour tout le monde. Or, si tout le monde utilise exactement la même méthode pour protéger ses données, tout le monde peut lire les messages des autres, puisque l'algorithme de déchiffrement est le même que pour ses propres messages. C'est pourquoi les algorithmes laissent toujours un tout petit espace pour que l'utilisateur puisse les personnaliser : en gardant secret cet élément de personnalisation, que l'on appelle une (ou des) clé, la protection est assurée. Il s'agit généralement d'un mot ou d'une phrase de passe.

Pour désigner les personnes impliquées dans une opération de chiffrement, les spécialistes ont pris l'habitude de nommer Alice et Bob, les agents qui tentent de s'écrire sans être dérangés par l'espion Eve, qui leur veut du mal sans qu'on sache trop pourquoi.

2.1 Principe général

La cryptographie asymétrique, ou cryptographie à clef publique est fondée sur l'existence des fonctions à sens unique et à brèche secrète.

Les fonctions à sens unique sont des fonctions mathématiques telles qu'une fois appliquées à un message, il est extrêmement difficile de retrouver le message original.

L'existence d'une brèche secrète permet cependant à la personne qui a conçu la fonction à sens unique de décoder facilement le message grâce à un élément d'information qu'elle possède, appelé clef privée.

Supposons qu'Alice souhaite recevoir un message secret de Bob sur un canal susceptible d'être écouté par un attaquant passif Eve :

Alice transmet à Bob une fonction à sens unique pour laquelle elle seule connait la brèche secrète ; Bob utilise la fonction transmise par Alice pour chiffrer son message secret ; Alice réceptionne le message chiffré puis le décode grâce à la brèche secrète ; Si Eve réceptionne également le message alors qu'il circule sur le canal public, elle ne peut le décoder, même si elle a également intercepté l'envoi de la fonction à sens unique, car elle n'a pas connaissance de la brèche secrète. La terminologie classiquement retenue est :

— pour la fonction à sens unique : «clef publique» ;
— pour la brèche secrète : «clef privée».

En pratique, sont utilisées des fonctions de chiffrement classiques, les termes «clef publique» et «clef privée» correspondant alors à des paramètres employés pour ces fonctions.

Pour des explications plus détaillées il est conseillé de lire le document de référence :

`https://zestedesavoir.com/tutoriels/723/la-cryptographie-asymetrique-avec-rsa/`

Vous pouvez aussi lire :

`https://fr.wikipedia.org/wiki/Cryptographie_asym%C3%A9trique`
`https://fr.wikipedia.org/wiki/Chiffrement_RSA`

2.2 PHASE 1 : Fonctionnement pratique

Je ne vais pas vous mentir : c'est des maths.

2.2.1 Création des clés

L'étape de création des clés est à la charge d'Alice. Elle n'intervient pas à chaque chiffrement car les clés peuvent être réutilisées. La difficulté première, que ne règle pas le chiffrement, est que Bob soit bien certain que la clé publique qu'il détient est celle d'Alice. Le renouvellement des clés n'intervient que si la clé privée est compromise, ou par précaution au bout d'un certain temps (qui peut se compter en années).

1. Choisir p et q, deux nombres premiers distincts ;
2. calculer leur produit $n = p.q$, appelé module de chiffrement ;
3. calculer $\phi(n) = (p-1)(q-1)$ (c'est la valeur de l'indicatrice d'Euler en n) ;
4. choisir un entier naturel e premier avec $\phi(n)$ et strictement inférieur à $\phi(n)$, appelé exposant de chiffrement ;
5. calculer l'entier naturel d, inverse de e modulo $\phi(n)$, et strictement inférieur à $\phi(n)$, appelé exposant de déchiffrement ; d peut se calculer efficacement par l'algorithme d'Euclide étendu.

Comme e est premier avec $\phi(n)$, d'après le théorème de Bachet-Bézout il existe deux entiers d et k tels que $e.d = 1 + k.\phi(n)$, c'est-à-dire que $ed1(mod\phi(n))$: e est bien inversible modulo $\phi(n)$.

Le couple (n, e), ou (e, n), est la clé publique du chiffrement, alors que sa clé privée est le nombre d, sachant que l'opération de déchiffrement ne demande que la clef privée d et l'entier n, connu par la clé publique

2.2.2 Chiffrement du message

Si M est un entier naturel strictement inférieur à n représentant un message, alors le message chiffré sera représenté par :

$C \equiv M^e \pmod{n}$, l'entier naturel C étant choisi strictement inférieur à n.

2.2.3 Déchiffrement du message

Pour déchiffrer C, on utilise d, l'inverse de e modulo $(p-1)(q-1)$, et l'on retrouve le message clair M par :

$M \equiv C^d \pmod{n}$.

2.2.4 Exemple

Un exemple avec de petits nombres premiers (en pratique il faut de très grands nombres premiers) :

1. on choisit deux nombres premiers $p = 3, q = 11$;
2. leur produit $n = 311 = 33$ est le module de chiffrement ;
3. $\phi(n) = (3-1)(11-1) = 210 = 20$;
4. on choisit $e = 3$ (premier avec 20) comme exposant de chiffrement ;
5. l'exposant de déchiffrement est $d = 7$, l'inverse de 3 modulo 20 (en effet $e.d = 3 \times 7 = 1 mod 20$).

La clé publique d'Alice est $(n, e) = (33, 3)$, et sa clé privée est $(n, d) = (33, 7)$.

Bob transmet alors un message à Alice.

— Chiffrement de M = 4 par Bob avec la clé publique d'Alice : 43 31 mod 33, le chiffré est C = 31 que Bob transmet à Alice ;

— Déchiffrement de C = 31 par Alice avec sa clé privée : 317 4 mod 33, Alice retrouve le message initial M = 4.

2.2.5 Attaque pour casser le chiffre

Il faut distinguer les attaques par la force brute, qui consistent à retrouver p et q sur base de la connaissance de n uniquement, et les attaques sur base de la connaissance de n mais aussi de la manière dont p et q ont été générés, du logiciel de cryptographie utilisé, d'un ou plusieurs messages éventuellement interceptés etc.

La sécurité de l'algorithme RSA contre les attaques par la force brute repose sur deux conjectures :

1. « casser » RSA de cette manière nécessite la factorisation du nombre n en le produit initial des nombres p et q,
2. avec les algorithmes classiques, le temps que prend cette factorisation croît exponentiellement avec la longueur de la clé.

2.3 Code fourni

— rsa_header.h
— rsa_print_tools.c
— rsa_tools.c
— other_base64.c
— Sha-256

2.3.1 Implémentation de la phase 1 : code Moodle à lire et comprendre et un peu de code à écrire.

La phase 1 l'implémente les outils de nombres premiers, factorisation, algorithme d'Euclide étendu (Bézout) ; et la création des clefs. On se limite dans cette phase à des uint64, c'est à dire des entiers non signés codés sur 64 bits. Les nombres premiers générés et le calculs sont limités en taille, attention au dépassement de capacité, ils ne sont pas gérés en C.

Le fichier header.h contient les prototypes des fonctions citées.

Fichier : rsa_tools.c

C'est la base su système, c'est pourquoi ce fichier est (presque) donné. Vous le trouverez sur Moodle
Il utilise la fonction bezoutRSA du fichier bézout.c, lui aussi disponible sur Moodle mais avec une subtilité : **il est à votre charge de programmer la fonction bézout à partir des spécifications données**.

1. int premier (uint n) : test de primarité avec le crible d'Erathostène, renvoie 1si le nombre est premier, 0 sinon
2. uint random_uint(uint min,uint max) : génère un uint aléatoire entre min et max. min et max sont des uint, renvoie n : $min \leq n \leq max$
3. int decompose (uint facteur[], uint64 n) : décompose un entier en facteurs premiers facteur.(brute force), param[in] : n l'entier, pram[out] : facteur = tableau de facteurs, returns la dimension du tableau
4. uint puissance(uint a, uint e) : puissance russe, param[in] : a l'entier et e l'exposant, returns : a^e
5. uint64 puissance_mod_n (uint64 a, uint64 e, uint64 n) : puissance modulaire, calcule a^e mod n, ATTENTION : $a \times a$ peut dépasser la capacité d'un uint64
   Se référer à :
   ```
   // https://www.labri.fr/perso/betrema/deug/poly/exp-rapide.html
   ```
6. uint genereUint(uint max,int *cpt) : génère un nombre premier inférieur à max. param[in] max : la borne sup, param[out] cpt : nombre d'essais, returns : le nombre premier. cpt compte le nombre d'essai pour trouver le nombre premier.
7. int rabin (uint a, uint n) : test de Rabin sur la pimarité d'un entier, c'est un test statistique
8. uint genereUintRabin(uint max,int *cpt) : fournit un nombre premier vérifié avec le test de rabin
9. uint pgcdFast(uint a,uint b) : pgcd rapide de a et b, param[in] a et b les deux entiers, pgcd(a,b).
10. void genKeysRabin(rsaKey_t *pubKey,rsaKey_t *privKey) : génère une paire de clefs, param[out] pubKey : clef publique, param[out] privKey : clef privée
11. void inputKey(uint64 E,uint64 N,rsaKey_t *key) : pour debug : permet de saisir une clef

12. long bezoutRSA(uint a,uint b,long *u,long *v) : récupère (r,u,v) de Bézout. Si u est négatif on le remplace par le premier qui est supérieur à 2.
13. long bezout(uint a,uint b,long *u,long *v) : implémente l'algorithme d'Euclide étendu : **Code à écrire**

    `https://fr.wikipedia.org/wiki/Algorithme_d%27Euclide_%C3%A9tendu`

    Fonction utilisée par la précédente. param[in] : a, b entiers (naturels), param[out] : u, v entiers relatifs tels que r = pgcd(a, b) et r = a*u+b*v, returns r entier

A faire : Ecrire la fonction bézout en suivant les spécifications, vérifier avec phase1.c et Makefile-phase1, comprendre le code et savoir utiliser les fonctions.

Remarque 1 : les prototypes de certaines de ces fonctions ne figurent pas dans le header. Il n'est pas nécessaire de mettre dans le header général les prototypes des fonctions qui ne sont pas appelées par des fonctions d'un autre fichier.

Remarque 2 : Ecrivez dès maintenant ces fonctions d'impression à l'écran : fichier rsa_print_tools.c
void printKey(rsaKey_t key) ; écrit une clef
void printKeyPair(keyPair_t keyPair) ; écrit une paire de clefs (privée et publique)
void printUint64Array(tabUint64_t tab) ; écrit un tableau d'uint64 (voir header)
uint64 * giveMeMem(int dim) ; allocation dynamique pour un tableau à une dimension
void printBlock(block_t blk) ; écrit un bloc

2.4 PHASE 2 : Chiffrer, déchiffrer, base64

On commence les choses sérieuses, c'est à dire la production du code de cryptographie que vous devez écrire.

2.4.1 Phase 2.1 : sur des messages utf8 dans des buffers

Pour se faire la main, on va lire et chiffrer un message caractère par caractère pour tester. Cependant **c'est ce qu'il ne faut pas faire car cette méthode est sensible à l'attaque statistique**. On prend chaque octet du code utf8 que l'on considère comme un entier pour faire les calculs.

Fichier crypt_decrypt.c

1. void RSAcrypt(unsigned char *msg, uint64 *cryptedMsg, rsaKey_t pubKey) ;
2. void RSAdecrypt(unsigned char *msg, uint64 *cryptedMsg, rsaKey_t privKey) ;

Utilisez le fichier phase2-1.c pour tester.

2.4.2 Phase 2.2 : conversion base 64, messages dans des fichiers

Le processus de codage base64 consiste à coder chaque groupe de 24 bits successifs de données d'un fichier binaire par une chaîne de 4 caractères. On procède de gauche à droite, en concaténant 3 octets pour créer un seul groupement de 24 bits (8 bits par octet). Ils sont alors séparés en 4 nombres de seulement 6 bits (qui en binaire ne permettent que 64 combinaisons). Chacune des 4 valeurs est enfin représentée (codée) par un caractère de l'alphabet retenu. Ainsi 3 octets quelconques sont remplacés par 4 caractères, choisis pour être compatibles avec tous les systèmes existants.

Cette conversion est utile pour écrire des données binaires dans des fichiers et, par exemple, les transférer par mail.

Fichier : rsa_file_char.c

Nous allons utiliser les fonctions suivantes données sur Moodle :
— char *base64_encode(const unsigned char *data, size_t input_length, size_t *output_length)
— unsigned char *base64_decode(const char *data, size_t input_length,size_t *output_length)

Commande Unix pour vérifier la transcription si besoin :

```
$ echo -n 'Hi!' | base64
SGkh
```

Le fichier tests-base64.c donne quelques exemples d'utilisation des fonctions ci-dessus.

Cette conversion nous permet d'écrire les clefs et les messages chiffrés dans des fichiers, de les envoyer par mail en les rendant manipulables par tous les systèmes.

A implémenter

1. void RSAfile_crypt(char *inFilename, char *outFilename, rsaKey_t pubKey) : lit caractère par caractère dans le fichier inFilename, chiffre avec la clef publique, traduit en base64, écrit le résultat dans le fichier outFilename.
   Note : on peut tout chiffrer d'un coup et convertir après en base64, écrire ensuite dans le fichier.
2. void RSAfile_decrypt(char *inFilename,char *outFilename,rsaKey_t privKey) : fait l'opération inverse.

Utilisez le fichier phase2-2.c pour tester vos fonctions.

2.4.3 Phase 2.3 : rendre le chiffrage plus résistant : utilisation de blocs de quatre octets, GMP

Pour brouiller les cartes et gêner l'analyse statistique on peut prendre un bloc de plusieurs octets dans le message, en faire un entier et l'utiliser pour les calculs. On prend des blocs de quatre octets ce qui correspond à la taille d'un uint (unsigned int). C'est une simplification. Les systèmes cryptographiques effectivement employés utilisent des blocs bien plus grands.

Le problème est qu'avec ce système les calculs n'ont plus aucune chance de tenir sur un uint64. Donc on va utiliser une bibliothèque de calcul en précision illimité : GMP.

GMP La bibliothèque GMP (GNU Multiple Precision) est disponible ici :

`https://gmplib.org/`

C'est une bibliothèque logicielle de calcul multiprécision sur des nombres entiers, rationnels et en virgule flottante. Voici un tuto d'installation :

`https://openclassrooms.com/fr/courses/1362801-la-cryptographie-asymetrique-rsa/1363119-gmp-arithmetic-without-limitation`

Nous allons devoir ré-écrire certains des outils de théorie des nombre avec les fonctions de cette bibliothèque. GMP contient déjà en standard des fonctions qui vont nous servir comme *mpz_powm* qui fait la puissance modulaire.

void puissance_mod_n_gmp(mpz_t res,uint64 a, uint64 e, uint64 n) : puissance modulaire, calcule a^e mod n avec GMP **Ce code est donné sur Moodle**.

A FAIRE : Implémentation du calcul par blocs : On peut maintenant envisager de reprendre le calcul de chiffrage par blocs.

1. uint convert_4byte2int(uchar *b) : conversion de 4 octets en un int selon l'ordre little endian **Ce code est donné sur Moodle**.
2. void convertInt2uchar(uint nb,uchar *tab4bytes) : convert from an unsigned int to a 4-byte array **Ce code est donné sur Moodle**.
3. uint64 RSAcrypt1BlockGmp(uint64 blockInt, rsaKey_t pubKey) ;
4. uint64 RSAdecrypt1BlockGmp(uint64 blockInt, rsaKey_t privKey) ;
5. void RSAfile_crypt(char *inFilename, char *outFilename, rsaKey_t pubKey) : chiffre un message par blocs à la volée depuis un fichier, codage en base64 et écriture dans un fichier.

6. void RSAfile_decrypt(char *inFilename,char *outFilename,rsaKey_t privKey) : opération inverse.

Le fichier *phase-test-blocks.c* permet de tester la conversion des messages en blocs numériques et l'inverse.

Le fichier *test-crypt.c* permet de tester la fonction puissance_mod_n_gmp

Vous pouvez à ce stade envoyer des pièces jointes chiffrées dans des courriels en utilisant la clef publique de vos correspondants. C'est une bonne validation du code. Le mieux est d'échanger des messages avec une autre équipe qui utilisera son code.

2.5 PHASE 3 : interprète de commande et annuaire de clefs

Pour gérer les clefs, celles de l'utilisateur et de ses contacts, on va créer un annuaire sous forme d'un interprète en ligne de commande permettant de :

1. générer des paires de clés avec identificateur (table des symboles)
2. chiffrer, déchiffrer avec les clés d'un identificateur
3. supprimer des clés
4. sauver une clé publique en base64
5. retrouver l'identificateur correspondant à une clé publique en b64
6. sauvegarde des clés dans un fichier chiffré avec mot de passe

A FAIRE : Fichier rsa_cmd.c : Implémenter l'interprète et les fonctionnalités demandées. Vous devez aussi à ce stade faire un *main* qui appellera l'interprète. Ce sera le programme principal de l'application.

2.6 PHASE 4 : signatures et hash

La signature d'un document utilise aussi les clefs cryptographiques. Si le chiffrage est destiné à assurer la confidentialité des documents (personne d'autre que la personne ayant la clef privée ne peut les lire), la signature concerne l'intégrité (le document n'a pas été modifié) et la paternité (le document a bien été envoyé par la personne qui le prétent).

Le document peut être transmis en clair, s'il est publique, ou lui-même chiffré. Pour signer un document on va calculer son hash à l'aide d'une fonction de hashage (ici sha 265). Le résultat s'appelle le "digest". Le digest va alors être chiffré par la clef privée de l'expéditeur. Il transmet alors le document et la signature.

Pour vérifier la signature, on utilise la clef publique de l'expéditeur pour retrouver le hash. On hash le document et on vérifie que les deux hash sont bien identiques.

Avec un document chiffré on agit de la même manière.

Pour des raisons de sécurité, Il est fortement déconseiller d'utiliser la même paire de clefs pour chiffrer et pour signer.

Implémentation Fichier rsa_sign.c

1. générer des clés de signature
2. générer le digest d'un fichier : void text2sha(char *inFilename,char *shaStr) /// lit un message dans un fichier et renvoie le hash associé /// [in] inFilename : le fichier contenant le Message /// [out] shaStr le hash du message /// shaStr alloué à l'extérieur en statique
3. signer le digest en le chiffrant avec la clef privée
4. faire un fichier b64 avec tout ça
5. ajouter un mdp dans l'interprète (vérification par hash)

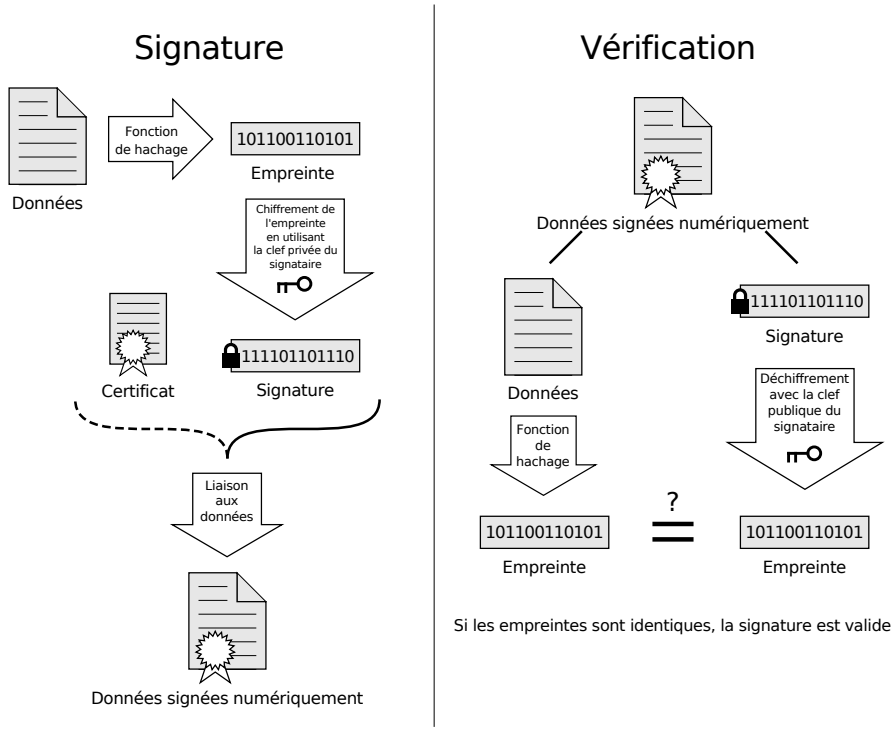

FIGURE 1 – Signature d'un documents par Guilieb Travail personnel,
CC BY-SA 3.0, https ://commons.wikimedia.org/w/index.php ?curid=45669258

6. vérifier une signature
7. signer un message chiffré
8. vérifier et déchiffrer le message signé et chiffré
9. ajouter la gestion des clefs de signatures à l'annuaire précédent.
10. ajouter à l'interprète de commande une commande certification suivie d'un identificateur de clef qui génèrera une requête de certification pour cette clef (publique). Cette requête sera écrite dans un fichier dont le nom est normalisé "TxEnAttente.txt". Structure de la commande : certify create ou revoke identificateur

Structure du fichier "TxEnAttente.txt" Le fichier sera un simple fichier texte au format CSV (séparateur : point-virgules, c'est plus visible)

- date de création
- type d'évènement :
  - création d'une clef cryptographique, code : CCK (create cryptographic key),
  - révocation d'une clef cryptographique, code : RCK (revoke cryptographic key),
  - création d'une clef de signature, code : CSK (create sign key),
  - révocation d'une clef de signature, code : RSK (revoke sign key)
- identité de l'émetteur (email)
- signature de l'émetteur s'il y en a déjà une d'enregistrée

2.7 PHASE 5 : Programme Java

Il existe des registres de clefs publiques qui sont garanties (les clefs et l'identité du propriétaire) par des autorités de certifications. Celles-ci peuvent elles-mêmes être certifiées par d'autres autorités. La certification est un problème délicat dans la mesure où certaines autorités sont connues pour vendre des certificats de complaisance.

Les autorités de certifications sont des organismes publiques ou privés selon un modèle centralisé. Nous allons simuler une telle autorité en utilisant un système "décentralisé" garanti par une communauté d'utilisateurs, les mineurs. Le registre lui même est une blockchain comparable à celle qui gère le Bitcoin (entre autre).

Donc en résumé la blockchain est un registre de documents publiques (registre de clés publiques et d'évènements). La blockchain fait figure d'autorité de certification.

Le principe de la blockchain et le type des documents sont définis dans un document à part.