Déchiffre et Delettre Partie 1
Chiffrement symétrique avec clé unique.

sur github : https://github.com/flashline/Atelier-crypto

Le fameux code César avait une clé à un nombre unique qu'on additionnait à chaque caractère du message.
Exemple :
Si A=1; B=2; etc.
et que la clé est 6.
Le A est codé G. Le B est codé H.
Le message 'BABA' donne 'HGHG'.

Ce système est très mauvais car il suffit d'essayer toutes les clés de 1 à 25 ou même de 1 à 127 si on a chiffrer les majuscules, minuscules, caractères spéciaux, les accents, espace, les points, etc (soit la table des caractères étendue français par exemple).

Meilleur système, on peut avoir une clé différente pour chaque caractère.
Mais tous les systèmes substituant un code différent pour chaque lettre mais identique pour une même lettre (chiffrement mono-alphabétique) est relativement facile à craquer plus le texte est long, si on connaît la langue du message clair. Car les lettres ont une fréquence d'apparition connue pour chaque langue.
Exemple en français la fréquence du "e" dans un texte est plus ou moins 17,26%, le "a" 8,4%, etc.

A noter que le système RSA que nous verront dans le prochain atelier est sensible à ce genre d'attaque. C'est la raison pour laquelle il est augmenté de différentes astuces pour pallier à ce problème).

On va programmer dans cet épisode :

Un système à clé secrète qui sous certaines conditions est "théoriquement"* inviolable.

* C-à-d qu'un théorème démontre cette absolue inviolabilité.

Théorie

Avec une clé privée:

Identique pour le chiffrement et déchiffrement,
la plus longue possible,
les lettres identiques n'ont pas toutes le même chiffrement, contrairement au code césar de base ou par substitution.

En particulier un théorème de Claude Shannon (Ingénieur et mathématicien) prouve en 1949, qu'un message chiffré avec:

- une clé de même longueur que le message,
- réellement aléatoire, (ne pas prendre un algo. pseudo-aléatoire encore moins des lettres choisies au hasard)
- à usage unique (One Time Pad ou Masque jetable)

est **indéchiffrable par quiconque ne connaissant pas la clé**.

Voir : https://www.lavachequicode.fr/cryptographie/cle-a-usage-unique
http://www.acrypta.com/telechargements/fichecrypto_300.pdf

Dans la version moderne de ces systèmes :

1. on prend comme valeur du caractère "clair" non pas sa position dans l'alphabet mais son code informatique c-à-d son code ASCII (Si le message contient des caractères Unicode, JS renverra un nombre > à 255. Dans ce cas une astuce consiste à remplacer le caractère par sa notation utilisée en html. Ex un 'œ' qui a le code 339 doit être remplacé par œ
   La fonction toUnicode() dans index.js le fait automatiquement.).
2. L'opération de codage n'est pas une addition mais un "OU exclusif"; ce qui est très proche :

Parenthèses apprendre à compter en binaire et conversions binaire <=> décimal :

Un nombre décimal s'exprime par des colonnes de droite à gauche valant 1,10,100,1000,... 10^n
Un nombre binaire aussi sauf que ces entêtes de colonnes converties en base 10 donne
1,2,4,8,16,32,64,128,256,... 2^n

Ex : le A a le code ascii 65.
65 est la somme des entêtes 64 et 1 ; ce qui donne en binaire sur 1 octets :

256	128	64	32	16	8	4	2	1
	0	**1**	**0**	**0**	**0**	**0**	**0**	**1**

Le nombre binaire ci-dessous :

256	128	64	32	16	8	4	2	1
	0	**1**	**0**	**0**	**1**	**1**	**1**	**0**

Est égal à 64+8+4+2 soit **78**. En regardant une table Ascii comme
http://ourhouzz.site/ascii-table-decimal-binary-hexadecimal/) on voit que 78 est le code du 'N'.

Un message M est donc une suite de bits.
  Ex: "AB" est égal à en ascii à 0100.0001|0100.0010
Une clé K est aussi une suite de bits aléatoires.
  Ex: 0101.1001|1111.0110 ("Yö")

L'opérateur XOR (ou ⊕) s'applique respectivement sur chacun des bits de 2 nombres (en l'occurrence celui du message M et celui de la clé K.) avec cette simple règle :
1 ⊕ 1 ou 0 ⊕ 0 donnent 0 . 1 ⊕ 0 ou 0 ⊕ 1 donnent 1.

On chiffre ainsi

M= **0100.0001**|0100.0010
⊕
K= 0101.1001|1111.0110
donne
C= 0001.1000 ...etc

On déchiffre ainsi

C= 0001.1000
⊕
K= 0101.1001
donne:
M= **0100.0001**

Note :
Le transfert sécurisé de clés secrètes (une par message) n'est pas adapté pour l'instant, aux transmissions fréquentes et anonymes d'internet.

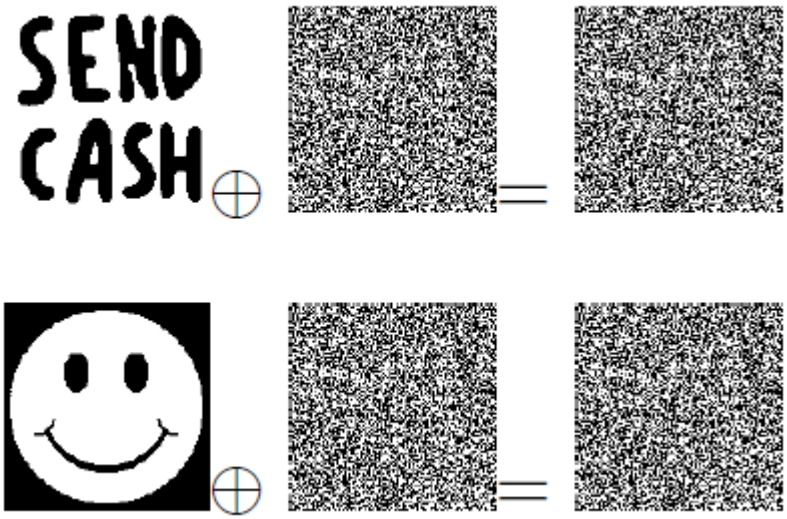
A la fin de ce support je donne une idée de transmissions de clés secrètes utilisant les propriétés quantiques de la lumière qui elles, sont rapides, sécurisées et adaptées aux transmissions anonymes sur Internet.

Les systèmes quantiques marchent déjà mais sont encore très chers.
En attendant leur démocratisation la clé secrète est elle même chiffrée avec un autre système comme RSA que nous verrons dans le prochain atelier.

Note :

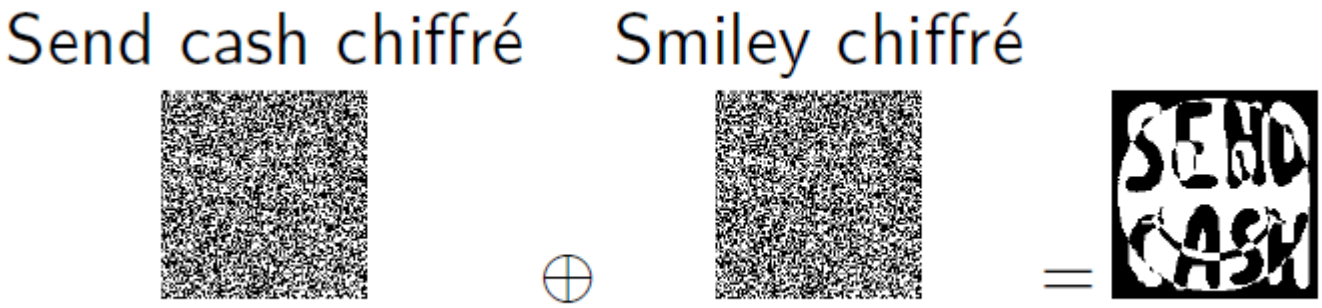
Risque dans le cas de ré-utilisation de la même clé.

Exemple avec 2 images et une même clé représentée elle aussi comme une image (0=blanc ; 1 = noir)



On voit que les chiffrements (à droite) sont efficaces.

Si par contre une personne quelconque récupère 2 messages chiffrés il obtient (en effectuant un XOR) une superposition des 2 images qui peut être très lisible.
Dans l'exemple ci-dessous le message se lit très bien et le smilley se devine aussi :



Encore pire !!!

En récupérant (peut-être par du 'phishing") un couple clair et chiffré, on retrouve la clef par :

$$\text{Chiffré} \oplus \text{clair} = \text{Clé}.$$

Lien:
risque si réutilisation de clé: https://cryptosmith.com/2008/05/31/stream-reuse/

Pratique: On ouvre le fichier ./Atelier-crypto-master/onePassSecretKey/js/index.js

Création de la clé aléatoire:

Note:
Dans le prog. principal, on a
```
  var o=createKey (plainTxt.length);
  var key=o.key; //var binKey=o.binKey;
```
On écrit donc la function:

```
/**
* Random key creation
* @param  len                Length of plain text
* @return object with
*                            key           Created key
*                            binKey        binary string of key (only to display it)
*/
function createKey (len) {
        var key=""; var k=0; var binKey="";
        //* var binKey8="";
        for (i=0;i<len*8;i++) {                    //  Pour chaque bit de la clé à construire
                var bit=random();                  //  on choisi au hasard 1 ou 0
                var k=k+bit;                       //  on range le bit dans k
                //* binKey8+= bit.toString() ;          // on range le bit pour former une string tempo.
                if ( i%8==7) {                     // quand on a traité 8 bits.
                        key+=String.fromCharCode(k);  //   on range l'octet k en tant que carac.
                        //* binKey+=binKey8+".";        //   on range la string tempo binKey8.
                        //* binKey8="";                 //   on remet à zéro la string tempo.
                        k=0;                          //   on remet à zéro l'octet traité.
                }
                k=k<<1; // or // k*=2; //          // on ajoute un zéro à droite de k pour accueillir
                                                   // le bit de la boucle suivante.
        }
        return {binKey:binKey,key:key};            // Retour de la clé key (sera utilisée)
                                                   // et binKey qui affichera la clé en binaire. (facultatif)
}
```

* Enlever les **//*** pour avoir dans binKey la string permettant l'affichage binaire, de la clé.
// …

```
function random () {
        return Math.floor(Math.random() * 2);      // Math.random renvoie un nombre 0>= N <1
                                                   // la partie entière de N * 2 renvoie aléatoirement
                                                   // 0 ou 1.
};
```
Notes:
Dans le prog. principal, après
```
  var o=createKey (plainTxt.length);
  var key=o.key;
  var binKey=o.binKey;
```

On a dans key une clé de même longueur que la message (var plainText)
et dans binKey la clé binaire.

ON TESTE
Atelier-crypto-master/onePassSecretKey/index.html

Théorie

Il vaut mieux grouper les octets de la clé et du message clair par **4** et stocker la valeur numérique de ces 4 octets dans un tableau.
Pourquoi? Parce que en JS l'opérateur ^ (xor) s'opère sur 32 bits max. 32=8x**4.**
La même fonction est utilisée (et cette fois fortement recommandée) pour l'atelier RSA.

Pour ça on fera pour chaque caractère C (de gauche à droite) en démarrant avec M = 0 :
Ajouter 8 zéros à droite M;
M = M+ caractère C suivant.
Quand M contient 4 caractères (ou 4 octets ; ou 32 bits) il est stocké comme élément d'un tableau mémoire (array)

Explications pour rajouter 8 zéros à droite et additionner le caractère suivant.

En base 10 pour ajouter 8 zéros à un nombre, on le multiplierai par 100000000 (équivalent à 10^8 soit 100 millions) .
Et bien en base 2 nous faisons la même chose à ceci près que le nombre binaire qui s'écrit 100000000 (2^8) est égal à 256.
On peut dire que multiplier un nombre binaire par 256 lui ajoute 8 zéros.

Alternative :
Avec le préfixe **ob** qui permet d'entrer directement un nombre binaire, on obtient le même résultat en JS soit
M x 0b100000000 est équivalent à M x 256

Pratique:

Regroupement de 4 octets en une valeur de 32bits et rangement dans un tableau (Array).

Note:
Dans le prog. principal, on a au début :

```
const by=4;
```

by donne le nombre de caractères qui vont être regroupés

```
// Convert plain text to array of 4 bytes elements
var arrBy= stringToArrayBy(plainTxt,by);
```

et

```
// Convert key to array of 4 bytes elements
var keyBy=stringToArrayBy(key,by);
```

La fonction est déjà écrite et expliquée dans **théorie** ci-dessus

```
/**
* Creation of an array of 4 bytes (32 bits) elements
* @param  str                A text string (message or key)
* @param  by                 Number of  bytes (by=4)
* @return An array of 32 bits elements
*/
function stringToArrayBy (str,by) {
        var arrOut=[]; var nc=0;var c;
        for (i=0;i<str.length;i++) {
                c=str.charCodeAt(i);
                if (c>255) error("If message  contains unicode 16 bits, you have to use toUnicode() ");
                nc=nc*256+c ;  // or nc=nc*0b100000000+c ; //
                if (i%by==(by-1) || i==str.length-1) {
                        arrOut.push(nc);      nc=0;
                }
        }
        return arrOut;
}
```

Dans le prog. principal, après

```
var arrBy= stringToArrayBy(plainTxt,by);
```

et

```
var keyBy=stringToArrayBy(key,by);
```

On doit avoir 2 tableaux de même longueur contenant chacun des valeurs de 32 bits.

ON enlève le breakPoint() précédent et on TESTE

Le chiffrement

Note:
Dans le prog. principal, on a

```
// Ciphering
var cipherArr=cipher (arrBy,keyBy);
```

On écrit donc le corps de la function:

```
/**
 * Ciphering
 * @param  msgArr                Message's 32 bits array
 * @param  keyArr                Key's 32 bits array
 * @return Ciphered message's 32 bits array
 */
function cipher (msgArr,keyArr) {
        var arrOut=[];  var j=0;
        for (var i in msgArr) { // Pour chaque élément de 32 bits
                //* if (j>keyArr.length-1) j=0 ;
                arrOut.push((msgArr[i] ^ keyArr[j])); // on chiffre. ^ est l'opérateur xor en JS
                //* j++;
        }
        return arrOut; // retour du tableau chiffré.
}
```

* remarquer que si on laisse comme ça, on se demande pourquoi utiliser j !
En fait j est requis pour que cipher() fonctionne même si la clé était plus petite que le message.
Il faut dans ce cas enlever les commentaires //* .

Déchiffrement

Note:
Dans le prog. principal, on a
```
  // Unciphering
  uncipherArr=uncipher(cipherArr,keyBy);
```

On écrit donc le corps de la function:
```
/**
* Unciphering
* @param  cipherArr                Message's 32 bits array
* @param  keyArr                   Key's 32 bits array
* @return Ciphered message's 32 bits array
*/
function uncipher (cipherArr,keyArr) {
                                                // le chiffrement et le déchiffrement sont symétriques.
        return cipher (cipherArr,keyArr) ;  // retour du tableau déchiffré (en clair mais non affichable).
}
```

Conversion de tableau déchiffré en texte

Note:
Dans le prog. principal, on a
// Uncipher array to string
var unciphText=arrayByToString (uncipherArr,by)

La fonction est déjà écrite et est l'inverse de stringToArrayBy() vue plus avant :

```
/**
* Reconstruction of plain text
* @param  arr                Unciphered array
* @param  by                 Number of  bytes (by=4)
* @return Original text reconstituted
*/
function arrayByToString (arr,by) {
        var str=""; var strBy=""; var c;
        for (var i=0;i<arr.length;i++) {
                var nc=arr[i];
                while (nc!=0) {
                  c=nc%256;                                  // ou c=nc%0b100000000;
                                                             //    extraction de la valeur de l'octet de droite.
                  strBy=String.fromCharCode(c)+strBy; // mettre le car. à gauche de la string tempo
                  nc=Math.floor(nc/256);                     //  ou nc=Math.floor(nc/0b100000000);
                                                             //    suppression de l'octet de droite.
                }
                str+=strBy;strBy="";                         // On range la string tempo de 4 car. Dans
                                                             // la string finale.
        }
        return str;                                          // retour du message en clair.
}
```

Manière dont les caractères sont rangés en tant que nombre, dans un élément du tableau :

P **l** **e** **a**
01010000.01101100.01100101.01100001

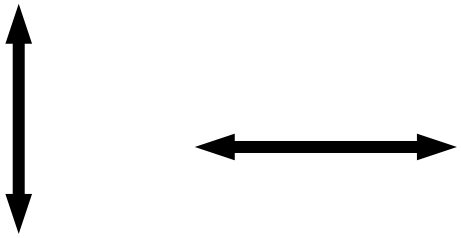
ON enlève le breakPoint() précédent
et ON TESTE
avec des messages + ou – longs
et éventuellement avec des caractères Unicode sur 16 bits
dans ce dernier cas utiliser la fonction toUnicode() en ligne 6 de index.js.

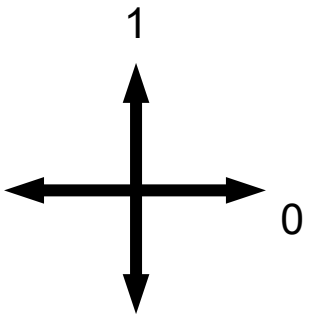
Démo de simulation de transmission quantique de clés.

Théorie

La cryptographie quantique n'existe pas. Le chiffrement se fait avec la méthode de clé secrète à usage unique vu précédemment.
Ce qui existe est la transmission de clés secrètes en étant sûr que la clé n'a pas été piratée sur le réseau.
Si au contraire il y a un doute, on jette la clé. Et c'est cette garantie qui est obtenu grâce à certaines principes de la physique quantique.
En particulier on utilise la polarisation de photons unique.
Un photon peut être représenté comme étant un grain de lumière (même si toute représentation d'un photon que ce soit un corpuscule ou une onde, est aussi fausse l'une que l'autre)

Un photon peut être polarisé (sens de sa vibration) :





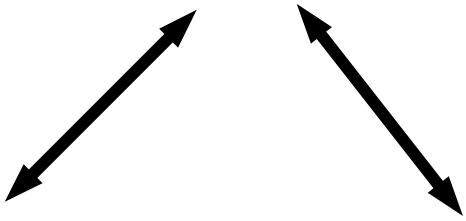
Verticalement ou Horizontalement;

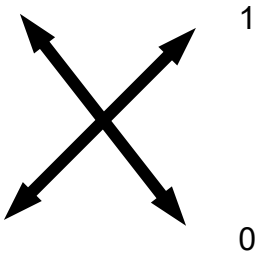
Bits: 1 0

On va parler de base Orthogonal

---

ou bien :





Diagonale ou Anti-diagonale

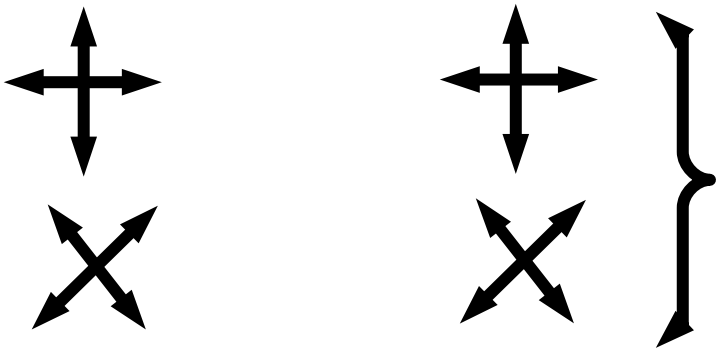
Bits: 1 0

On va parler de base Croisée

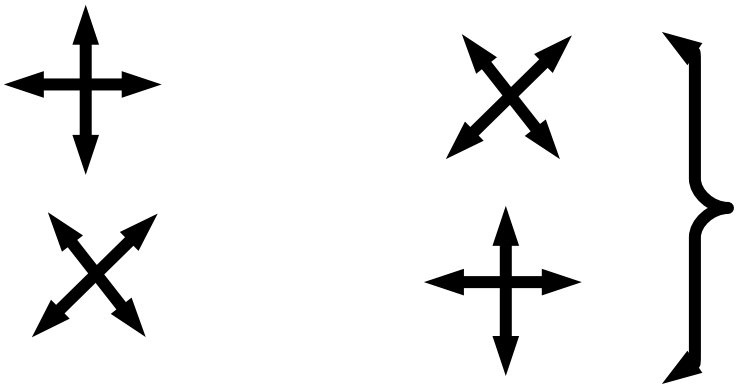
Les principes fondamentaux:
Si le photon est créé et lu dans la même base il sera lu correctement.
Si le photon est crée dans une base et lu dans l'autre il sera correctement lu que dans 50 % des cas.

Digression: Dans le monde du chiffrement on nomme **Alice** et **Bob** les émetteur/récepteur du message (Le pirate est Eve en anglais prononcer iv' . Nous l'appellerons **Yves**)

Alice créé en Bob lit en



==>> C'est déterministe c'est de l'info



==>> C'est probabiliste à 50% c'est **pas** de l'info !

Notes:

Le choix des bases X et + est aléatoire chez Alice et Bob et indépendamment l'un de l'autre.
Le choix 1 ou 0 est aléatoire chez Alice.

Simulation en ligne:

Fonctionnement
(avec le programme http://www.pixaline.net/intra/Atelier-2019/crypto/done/quantic/)

Alice envoie 1000 photons par exemple et sauve les valeurs des bits et des bases en interne.

Bob reçoit les photons et sauve les valeurs des bits et des bases en interne.

Alice envoie 10 % des bits choisis au hasard avec leur base de création sur canal classique pour tester.
(c'est 10% ne seront pas utilisés pour former la clé)

Bob ne va comparer que les valeurs des bits créés et lus par une même base.

Si il n'y a pas d'erreur alors la transmission n'a pas été piratée.

Les 10% sont jetés.

Les bases des autres photons sont échangées entre Alice et Bob. (**Pas les valeurs des bits!**)

Ainsi les 2 peuvent reconstituer la clé à partir des bits déterministes c-à-dire de base identique chez Alice et Bob.

Si il y a des erreurs (+ ou – 25 %):

Le processus est abandonné (Pas de clé , pas de transmission)

On ne connaît pas en détail ce qu'a fait Yves le pirate mais la seule chose qu'il pouvait faire est:

Choisir 1 base au hasard comme Bob. Sans connaître celle d'Alice. Et donc ajouter 50 % d'erreur sur les photons qui seront testés par Alice et Bob (rappel: de bases identiques pour ces 2 derniers).

Puis créer un nouveau photon dans la même base et de même valeur que celle qu'il a lu,
A noter qu'on ne peut pas cloner un photon. On ne peut que le détruire en le lisant puis en créer un autre. On ne peut pas anticiper la polarisation d'un photon.

Bob lira le photon de Yves avec une probabilité de 50% de rétablir la bonne valeur. Ce qui explique que la probabilité d'erreurs est de 25%, Dans les faits cela oscille entre 15 et 35 % dans ce programme de simulation.

Pour un rappel des explications voir cette vidéo:
https://www.youtube.com/watch?v=zx9XkeX2YaY