



# RAPPORT SUR EXPOSÉ:

## FONCTIONS HACHAGE CRYPTOGRAPHIQUE

Réalisé par:

**MAKHOUKHI AYMAN**

**EL WAHIDI EL ALAOUI NADA**

Encadré par:

**Mr.BENAZZI**

**Département: Génie informatique.**

**Filière: Développement des Applications  
Informatiques.**

**Année universitaire :2019-2020**

## Table des matières

<b>Introduction .....</b>	5
<b>Chapitre 1: FONCTIONNEMENT DES FONCTIONS HACHAGE CRYPTOGRAPHIQUES .....</b>	6
I.Définition du “Hachage” .....	7
II.Qualités des fonctions hachages .....	8
III.Différence entre hachage et cryptographie .....	9
IV. Fonctions de hachage .....	11
1. Mauvais exemple d'enregistrement des données .....	12
2. Bon exemple d'enregistrement des données.....	13
V. Historique des fonctions de hachage cryptographiques ..	14
VI. La collision et la différence entre les fonctions de hachage. ....	16
VII.Pourquoi SHA-1?.....	19
VIII.Définition de SHA-1 : .....	20
IX.Caractéristiques de SHA-1.....	21
X.Fonctionnement de SHA-1.....	22
XI.Construction de Keccak.....	24
<b>Chapitre 2 : EXEMPLES DES FONCTIONS DE HACHAGE CRYPTOGRAPHIQUES.....</b>	26
I. Code de la fonction SHA-1 sur JAVASCRIPT.....	27
II. Sécurité du mot de passe grâce à SHA-1 dans une application de gestion .....	35

III. Site montrant les différents algorithmes de hachages :...41

<b>Conclusion</b> .....	42
<b>Webographie</b> .....	43

## Table des figures

Figure 1 : Explication du cryptage et décryptage en image.....	9
Figure 2 : Explication de l'agorithme de hachage en image .....	9
Figure 3 : Exemple d'enregistrement du couple identifiant/mot de passe “Sans hachage” .....	12
Figure 4 : Exemple d'enregistrement du couple identifiant/mot de passe avec “Hachage” .....	13
Figure 5 : Sécurité et collision des fonctions de hachage.....	14
Figure 6 : Tableau montrant les différences entre les fonctions de hachage. ....	16
Figure 7 : Tableau montrant la taille des message en “bits”. ...	17
Figure 8 : Schéma du SHA-1.....	20
Figure 9 : Code d'authentification du message en hash. ....	22
Figure 10 :Schéma de la construction Keccak .....	24
Figure 11 : Screen du code de la function SHA1. ....	27
Figure 12:Screen du code pour l'obtention de 8bits.....	27
Figure 13: Screen du code pour concatener les éléments .....	27
Figure 14: Screen du code pour montrer la boucle while. ....	28
Figure 15: Screen du code montrant un code binaire de 8bits ASCII. ....	28
Figure 16 : Screen du code montrant le pad jusqu'à obtention de	

64bits.....	28
Figure 17 : Screen du code montrant la décomposition de 512 caractères.....	29
Figure 18 : Screen du code avec des opérations XOR sur chaque bit .....	31
Figure 19: Résultat du code en binaire .....	31
Figure 20: Screen du code avec initialisation des variables .....	32
Figure 21: Screen du code montrant les différentes boucles sur chaque morceau.....	33
Figure 22: Screen du code qui va retourner les variables en hexadécimal .....	34
Figure 23 : Fomulaire de connexion.....	35
Figure 24 : L'espace d'administration web de la base de données.	
.....	36
Figure 25 : Page de gestion des utilisateurs .....	37
Figure 26: Code php de la création des utilisateurs. ....	38
Figure 27: Le hash du mot de passe “Math2020” stocké dans la base de données.....	38
Figure 28: Code php de l'authentification d'utilisateur. ....	39
Figure 29: Cas d'erreur pendant l'authentification .....	40
Figure 30: Screen des fonctions de hachages . .....	41

# INTRODUCTION

Dans le cadre de notre formation à l'Ecole Supérieure de Technologie d'Oujda, nous sommes amenés à effectuer des exposés lors de nos séances en classe.

Le notre a été fait lors du troisième semestre au cours du module M11: **Mathématiques modélisation math Probabilité et Stat, Recherche opérationnelle.**

Ce rapport établit une synthèse de ce qu'on a appris sur les **“Fonctions Hachage cryptographiques”**.

Pour cela, on va en première partie expliquer le fonctionnement des fonctions hachages cryptographiques en détails.

Puis en deuxième partie, on va expliquer par des exemples concrets.

# **CHAPITRE 1:**

## **FONCTIONNEMENT DES FONCTIONS**

## **HACHAGES CRYPTOGRAPHIQUES**

## I. Définition du “Hachage” :

Hachage est un traitement informatique et mathématique qui transforme une donnée quelconque en une donnée de taille fixée.

Hachage est aussi appelée fonction de hachage à sens unique ou "**one-way hash function**" en anglais.

Ce type de fonction est très utilisé en cryptographie, principalement dans le but de réduire la taille des données à traiter par la fonction de cryptage.

En effet, la caractéristique principale d'une fonction de hachage est de produire un haché des données, c'est-à-dire un condensé de ces données. Ce condensé est de taille fixe, dont la valeur diffère suivant la fonction utilisée : nous verrons plus loin les tailles habituelles et leur importance au niveau de la sécurité.

Les fonctions de hachage cryptographiques sont employées pour **l'authentification, les signatures numériques et les codes d'authentification de messages**.

## II. Qualités des fonction hachages :

Une fonction de hachage doit disposer de ces qualités :

- **Rapide à calculer** : parce qu'elle est fréquemment sollicitée.
- **Non réversible** : chaque condensé peut provenir d'un très grand nombre de messages, et seule la force brute peut générer un message qui conduit à un condensé donné.
- **Résistant à la falsification** : la moindre modification du message aboutit à un condensé différent.
- **Résistant aux collisions** : il devrait être impossible de trouver deux messages différents qui produisent le même condensé.

### III. Différence entre hachage et cryptographie :

Le cryptage est une fonction bidirectionnelle qui inclut le cryptage et le décryptage tandis que le hachage est une fonction à sens unique qui transforme un texte brut en un résumé unique qui est irréversible.

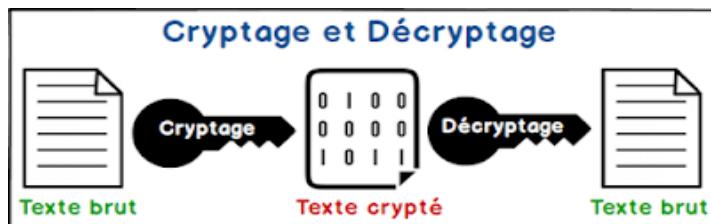


Figure 1 : Explication du cryptage et décryptage en image



Figure 2 : Explication de l'algorithme de hachage en image

Un hachage peut simplement être défini comme un nombre généré à partir d'une chaîne de texte. En substance, un hach est plus petit que le texte qui le produit. Il est généré de manière à ce qu'un hachage similaire de même valeur ne puisse pas être produit par un autre texte. De cette définition, on peut voir que le hachage est le processus de production de valeurs de hachage dans le but d'accéder aux données et pour des raisons de sécurité dans les systèmes de communication. En principe, le hachage prendra une entrée arbitraire et produira une chaîne de longueur fixe. En règle générale, le hachage aura les propriétés suivants:

Une entrée donnée qui est connue, doit toujours produire une sortie connue.

Une fois que le hachage a été fait, il devrait être impossible de passer de la sortie à l'entrée.

Différentes entrées multiples devraient donner une sortie différente.

Modifier une entrée devrait signifier un changement dans le hachage.

Il ya plusieurs différents types d'algorithmes de hachage. mais certains ont été rejetés au fil du temps :

**MD5, SHA, RIPEMD, TIGER.**

En bref, le cryptage est une fonction bidirectionnelle qui inclut le cryptage et le décryptage, a condition d'avoir une clé de sécurité utilisée pour le décryptage Il existe un certain nombre de systèmes de cryptage, où un cryptage asymétrique est également connu sous le nom de cryptage à clé publique., tandis que le hachage est une fonction à sens unique qui transforme un texte brut en un résumé unique qui est irréversible.

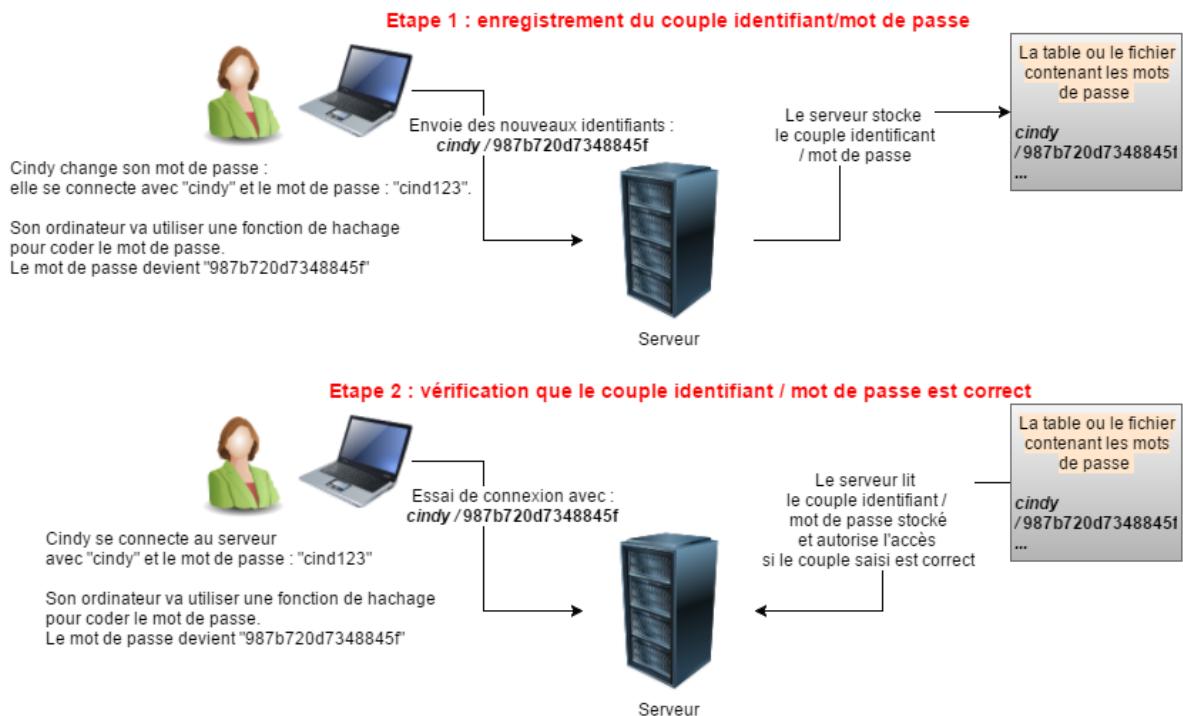
#### IV. Fonctions de hachage :

Les fonctions de hachage utilisés pour cracker les mots de passes tel que les mots de passes qui se trouvent dans votre disque dure, les mots de bases qui se trouvent sur les serveurs. Si quelqu'un accède à cette base de données, il peut récupérer ces mots de passes.

Ces fonctions d'hachages ont plusieurs propriétés, mais la plus importante c'est qu'il n'y a pas de réciproque. On prend un exemple le carré de **5** c'est **25** et si je vous demande **25** et qu'il est le nombre de départ c'est facile à trouver, car vous avez pris la racine de **25** c'est **5**, ça veut dire qu'il y a une fonction réciproque : c'est la racine carré qui permet de revenir en arrière, ces fonctions peuvent prendre en entrée d'importe quelle chaîne de caractères de n'importe quelle longueur, alors il est facile de donner un mot à un ordinateur qui peut calculer le hash de ce mot, mais s'il donne le hash en entrée il n'y a pas une fonction réciproque pour récupérer le mot de départ.

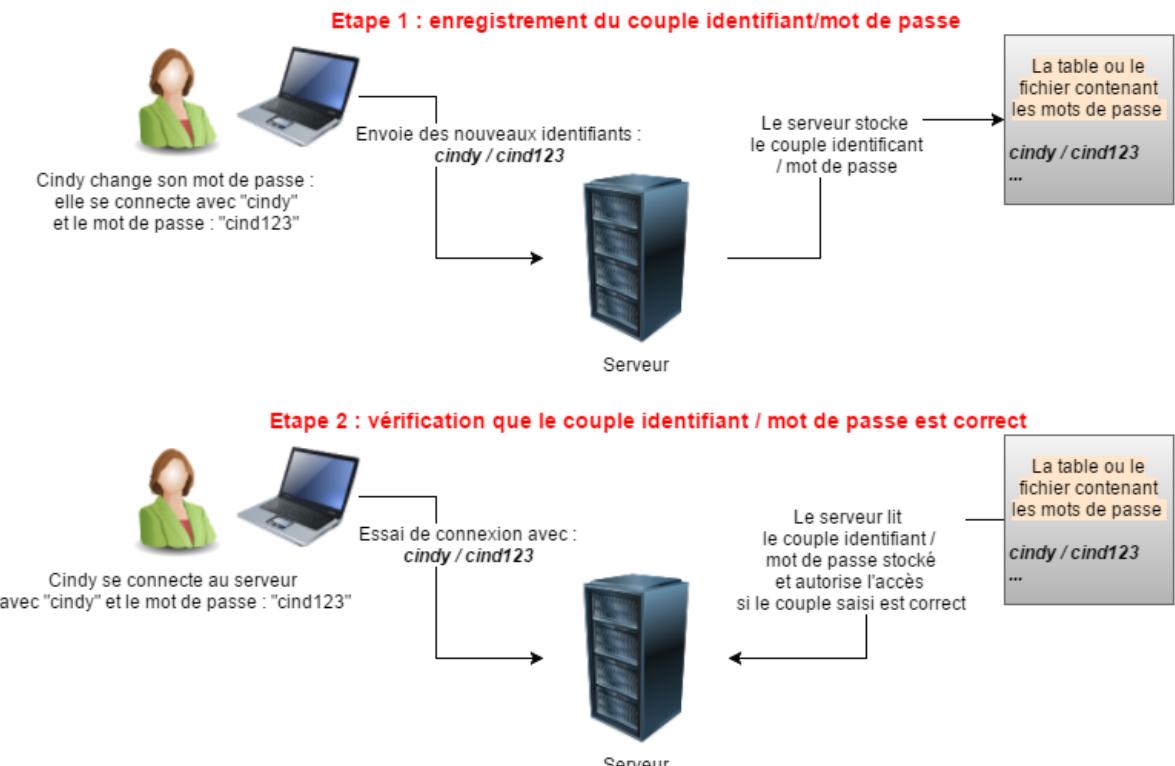
par exemple, du côté serveur, la base de données stocke les mots de passes des utilisateurs **user1/gZu9a1** lorsque vous connectez, le serveur calcule le hash de mot passe saisi et le compare avec le hash qui se trouve dans la base de données.

## 1. Mauvais exemple d'enregistrement des données (sans hachage):



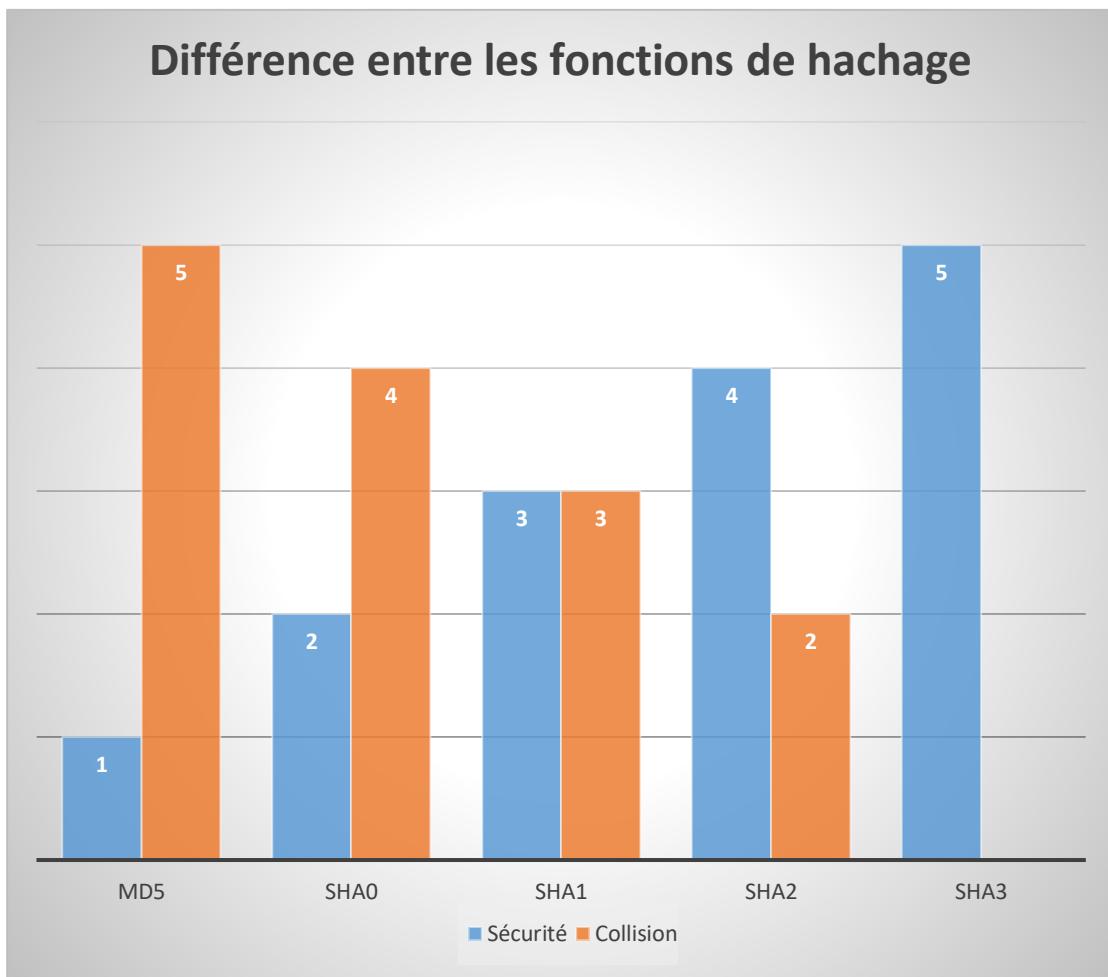
**Figure 3 : Exemple d'enregistrement du couple identifiant/mot de passe “ Sans hachage”**

## 2. Bon exemple d'enregistrement des données avec le hachage :



**Figure 4 : Exemple d'enregistrement du couple identifiant/mot de passe avec « Hachage »**

## V. Historique des fonctions de hachage cryptographiques :



**Figure 5 : Sécurité et collision des fonctions de hachage**

Sha-1 est une fonction cryptographique qui prend en entrée un message d'une longueur maximale de **2 ^ 64 bits** et génère un hachage de **160 bits**, soit **40 caractères**.

Sha-1 est une amélioration de **Sha-0**, il a été créé par la NSA et améliore la sécurité cryptographique en augmentant le nombre d'opérations avant une collision.

Cependant, **Sha-1** n'est pas considéré comme sécurisé car

$2^{63}$  pourrait être atteint assez facilement. Il a été remplacé par **Sha-2** et plus récemment par **Sha-3**.

En ce qui concerne la sécurité, **MD5** et **Sha-0** n'est plus considéré comme un type de hachage sécurisé. Si vous souhaitez toujours l'utiliser ,vous devriez envisager d'utiliser un salt pour rendre la vie des pirates plus difficile.

Un salt consiste en une chaîne que vous ajoutez au mot de passe de l'utilisateur avant de le hacher. Cette chaîne peut être une séquence aléatoire, un horodatage ou tout élément rendant le mot de passe plus difficile à trouver.

## VI.La collision et la différence entre les fonctions de hachage :

Algorithm and variant		Output size (bits)	Internal state size (bits)	Block size (bits)	Rounds	Operations
<b>MD5</b> (as reference)		128	128 (4 × 32)	512	64	And, Xor, Rot, Add (mod $2^{32}$ ), Or
<b>SHA-0</b>		160	160 (5 × 32)	512	80	And, Xor, Rot, Add (mod $2^{32}$ ), Or
<b>SHA-1</b>						
<b>SHA-2</b>	<i>SHA-224</i>	224	256 (8 × 32)	512	64	And, Xor, Rot, Add (mod $2^{32}$ ), Or, Shr
	<i>SHA-256</i>	256				
	<i>SHA-384</i>	384	512 (8 × 64)	1024	80	And, Xor, Rot, Add (mod $2^{64}$ ), Or, Shr
	<i>SHA-512</i>	512				
	<i>SHA-512/224</i>	224				
	<i>SHA-512/256</i>	256				
<b>SHA-3</b>	<i>SHA3-224</i>	224	1600 (5 × 5 × 64)	1152	24 <sup>[40]</sup>	And, Xor, Rot, Not
	<i>SHA3-256</i>	256		1088		
	<i>SHA3-384</i>	384		832		
	<i>SHA3-512</i>	512		576		
	<i>SHAKE128</i>	<i>d</i> (arbitrary)		1344		
	<i>SHAKE256</i>	<i>d</i> (arbitrary)		1088		

**Figure 6 : Tableau montrant les différences entre les fonctions de hachage**

L'attaque des anniversaires (les collisions):

une fonction à valeurs dans n bits aura  $2^n$  valeurs différentes possibles.

Si la fonction de hachage est définie sur n bits, on aura très probablement une collision après avoir calculé  $2^{n/2}$  hachés.

Algorithm	Message Size (bits)	Block Size (bits)	Word Size (bits)	Message Digest Size (bits)
SHA-1	$< 2^{64}$	512	32	160
SHA-224	$< 2^{64}$	512	32	224
SHA-256	$< 2^{64}$	512	32	256
SHA-384	$< 2^{128}$	1024	64	384
SHA-512	$< 2^{128}$	1024	64	512
SHA-512/224	$< 2^{128}$	1024	64	224
SHA-512/256	$< 2^{128}$	1024	64	256

Figure 7 : tableau montrant la taille des messages en “bits”

La collision par exemple c'est ou la lecture de deux différents mots de passes produisant la même empreinte.

C'est pourquoi, il est conseillé d'utiliser des fonctions de hachage avec un faible taux de collisions.

L'attaque générique des anniversaires permet de trouver une collision en  $2^{80}$  opérations pour les hash avec condenses de **160 bits**, ce qui est donc la sécurité attendue pour une telle fonction de hachage. Mais dans le cas de **SHA-1**, il existe une attaque théorique en  $2^{69}$  connue depuis 2005, qui a été améliorée en une attaque en  $2^{63}$  opérations.

On peut se demander pourquoi il existe plusieurs tailles de condensés ou encore pourquoi celle-ci est fixe. Il faut garder à l'esprit le but ultime d'un haché qui est d'être le plus court possible, tout en gardant ses propriétés. Or, cela nous amène tout naturellement au problème des collisions, également connu sous la dénomination de théorème ou paradoxe des anniversaires.

Prenons donc notre haché H, qui présente une longueur de n bits. Nous pouvons déjà déduire qu'il n'existe que  $2^n$  hachés de ce type possibles (puisque chaque bit n'a que 2 valeurs possibles, 0 ou 1. Nous risquons donc, un jour ou l'autre, de

produire un haché qui pourrait correspondre à un autre: c'est la perte de la propriété principale d'un condensé, qui est l'unicité : nous avons trouvé une collision.

Le théorème des anniversaires prouve qu'il faut  $2^{n/2}$  essais pour trouver une collision au hasard. C'est le chiffre qui sera utilisé pour évaluer la force d'une fonction de hachage.

Pourtant, il ne faut pas négliger le fait que la collision citée précédemment a été obtenue au hasard, ce qui n'est pas exploitable par une personne malveillante.

D'un point de vue pratique, et dans l'état de l'art actuel, il est généralement accepté que **256** calculs représentent un défaut réalisable. Comme exemple, les clés de **56 bits** sont réellement faibles et crackables. En conséquence, avec  **$n/2=56$**  et  **$n=112$** , le théorème des anniversaires nous indique que les hachés de **112** bits sont faibles et donc insuffisants à l'heure actuelle. De la même manière, les hachés de **128 bits ( $n/2=64$ )** ne représentent plus une sécurité à moyen terme. C'est pour cela que la norme actuelle est à **160 bits ( $n/2=80$ )** voire plus dans le cas de **SHA-1**.

## VII.Pourquoi SHA-1 ?

On a choisi la fonction **SHA-1**, par rapport a la fonction **MD5** car ces collisions étaient si facile a générer et à la portée de tout le monde, le **MD5** serait à bannir et non plus seulement déconseillé, en plus les algorithmes performants permettent de générer des collisions.

Pour les fonctions **SHA-2** et **SHA-3**, elles ont des algorithmes difficiles à expliquer car elles sont utilisées par les grandes organisations qui disent que l'algorithme de **SHA-1** pourrait ne plus être suffisant pour continuer à l'utiliser dans le future parce qu'il y a des cryptanalystes qui ont découvert des attaques sur **SHA-1**,alors que depuis 2010 Microsoft, Google et Mozilla ont recommandé son remplacement par SHA-2 ou **SHA-3**.

## VIII. Définition de SHA-1 :

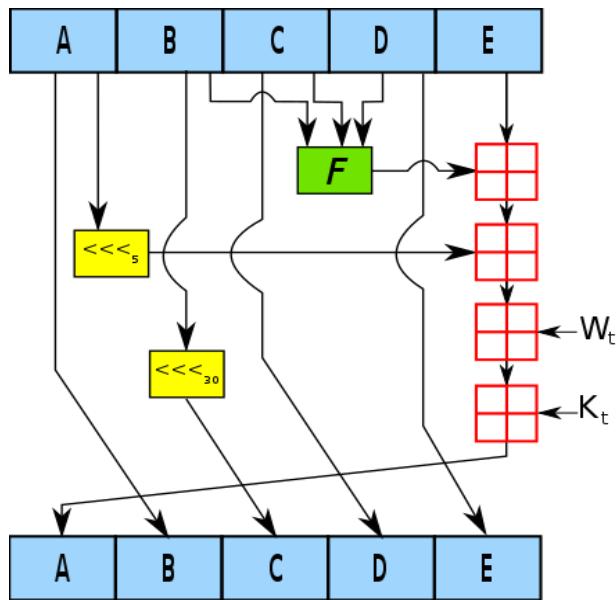


Figure 8 : Schéma du SHA-1

**SHA-1** (prononcé Secure Hash Algorithm) est une fonction de hachage cryptographique conçue par la National Security Agency des États-Unis (NSA), et publiée par le gouvernement des États-Unis comme un standard fédéral de traitement de l'information. Elle produit un résultat appelé « hash » ou condensat de **160 bits**.

L'attaque des anniversaires ne fonctionne évidemment pas, car  **$2^{80}$  ( $=2^{160}/2$ )**, ça fait tout de même cent mille milliards de milliards, et rien que pour stocker les résultats.

L'attaque utilisée sur le **SHA-1**, ils avaient trouvé en 2005 une attaque en  **$2^{69}$** . C'est beaucoup mieux que le  **$2^{80}$  ( $=2n/2$ )** de l'attaque des anniversaires), mais c'est compliqué à mettre en pratique (surtout avec la puissance de calcul de 2005), mais ça explique que le **SHA-1** était déjà à l'agonie même si on avait pas trouvé de collision. Mais avec SHAttered en  **$2^{63}$** , on a fini par trouver une collision.

## IX. Caractéristiques de SHA-1 :

- Taille du message :  $2^{64}$  bits maximum
- Taille des blocs : 512 bits
- Taille de hash : 40 digits hexadecimal
- Taille du condensé : 160 bits ( $5 \times 32$  bits )
- Niveau de sécurité : collision en  $2^{63}$  opérations.
- Nombre d'étapes : 80 ( 4 tours de 20)

## X. Fonctionnement de SHA-1 :

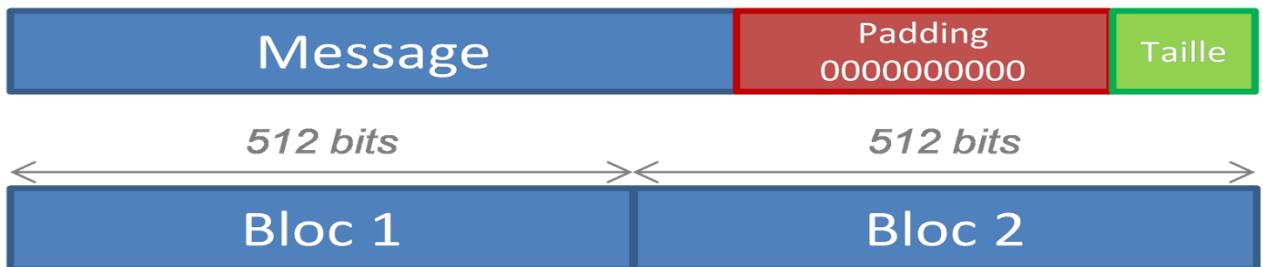


Figure 9 : Code d'authentification du message en hash

1. Le **SHA-1** prend un message d'un maximum de **2<sup>64</sup> bits** en entrée. Quatre fonctions booléennes sont définies, elles prennent 3 mots de **32 bits** en entrée et calculent un mot de **32 bits**. Une fonction spécifique de rotation est également disponible, elle permet de déplacer les bits vers la gauche ainsi le mouvement est circulaire et les bits reviennent à droite.
2. Le **SHA-1** commence par ajouter à la fin du message un bit à **1** suivi d'une série de bits à **0**, puis la longueur du message initial (en bits) codée sur **64 bits**. La série de **0** a une longueur telle que la séquence ainsi prolongée a une longueur multiple de **512 bits**. L'algorithme travaille ensuite successivement sur des blocs de **512 bits**.

Si un bloc de **512 bits** a déjà été calculé auparavant, les variables sont initialisées avec les valeurs obtenues à la fin du calcul sur le bloc précédent.

4. Il s'ensuit **80 tours** qui alternent des rotations, des additions entre les variables et les constantes. Selon le numéro du tour, le **SHA-1** utilise une des quatre fonctions booléennes. L'une de ces

fonctions est appliquée sur **3** des **5** variables disponibles. Les variables sont mises à jour pour le tour suivant grâce à des permutations et une rotation. En résumé, le **SHA-1** change sa méthode de calcul tous les **20 tours** et utilise les sorties des tours précédents.

5. À la fin des **80 tours**, on additionne le résultat avec le vecteur initial. Lorsque tous les blocs ont été traités, les cinq variables concaténées ( **$5 \times 32 = 160$  bits**) représentent la signature.

## XII. Construction Keccak :

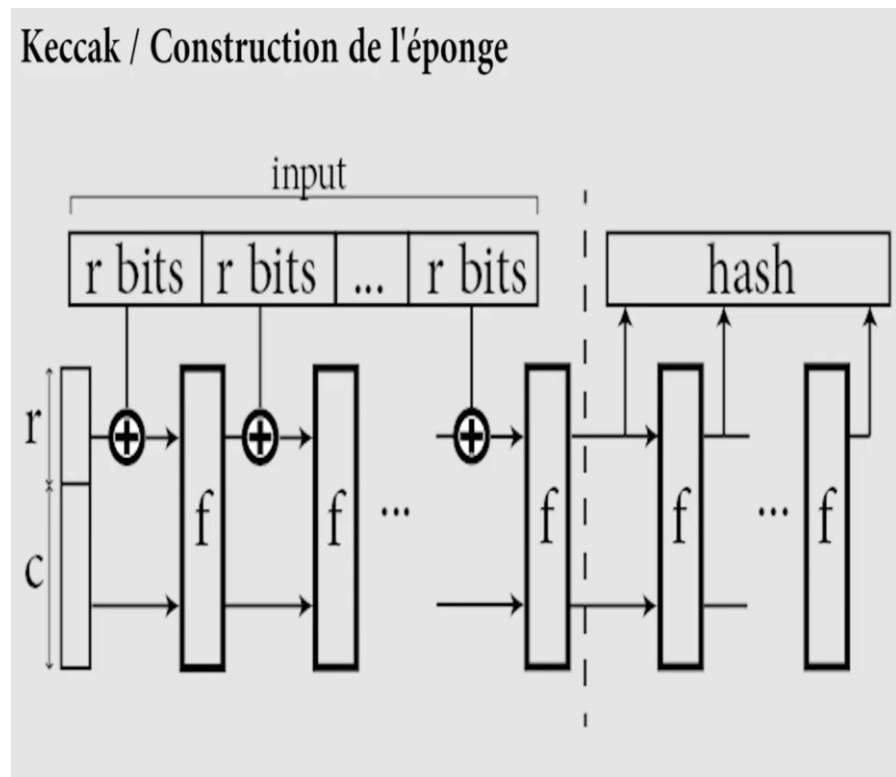


Figure 10 : Schéma de la construction Keccak

Construction de l'éponge fonctionne en deux phases :

- Absorption de l'information
- Essorage pour obtention du hash voulu.

Première phase :

1. On définit deux nombres de bits R et C.
2. On décompose l'entrée en bloc de R bits et on choisit un vecteur initial de longueur R+C.
3. On fait un XOR entre le premier bloc et les R bits initiaux, le résultat est donné en entrée à une fonction  $f$ .
4. La fonction en entrée prends les C bits de l'état initial et produit deux sorties une de R bit et une de C bit.
5. On fait un XOR entre le R bits résultant et le deuxième

bloc en entrée, le résultat donnée en entrée est de nouveau la fonction  $f$ .

On continue ce processus jusqu'à avoir traité tous les blocs en entrée.

### Deuxième phase :

1. On va appliquer de nouveau la fonction  $f$  mais cette fois le résultat de  $R$  bit va être ajouter à une chaîne binaire qui sera le résultat final, à chaque iteration de  $f$ .
2. On ajoute  $R$  bit au hash final.
3. On s'arrête qu'on le hash à au moins la taille désirée exemple 256 bits.

# **CHAPITRE 2 :**

## **EXEMPLES DES FONCTIONS HACHAGE CRYPTOGRAPHIQUES**

## I. Code de la fonction SHA-1 sur JAVASCRIPT :

1) On va transformer l'entrée de texte à un tableau des caractères du code ASCII :

A Test [A, ,T,e,s,t] = [65,32,84,101,115,116]

```
function sha1(text) {  
    const asciiText = text.split('')  
    .map((letter) => utils.charCodeAt(letter));
```

**Figure 11 : Screen du code de la fonction SHA1**

2) Ensuite on va convertir le code ASCII en binaire :

[1000001,100000,1010100,1100101,1110011,1110100]

3) Après on ajoute des zéros à chaque élément du tableau jusqu'à obtention de 8bits :

[01000001,00100000,01010100,01100101,  
01110011,01110100]

```
let binary8bit = asciiText  
    .map((num) => utils.asciiToBinary(num))  
    .map((num) => utils.padZero(num, 0));
```

**Figure 12 : Screen du code pour l'obtention de 8bits**

4) Concatenant les éléments du tableau et on ajoute 1 à droite  
[0100000100100000010101000110010101110011011101001]

```
let numString = binary8bit.join('') + '1';
```

**Figure 13 : Screen du code pour concatener les éléments**

5) Concatenant les éléments du tableau et on ajoute à cela des zeros jusqu'à obtention d'une longueur de 448 bits :

[0100000100100000010101000110010101110011011101001]

=>

```
[0100000100100000101010001100101011100110111010010  
0000000000000000000000000000000000000000000000000000000000  
000000000000000000000000000000000000000000000000000000000000  
000000000000000000000000000000000000000000000000000000000000  
000000000000000000000000000000000000000000000000000000000000  
000000000000000000000000000000000000000000000000000000000000  
000000000000000000000000000000000000000000000000000000000000  
000000000000000000000000000000000000000000000000000000000000  
000000000000000000000000000000000000000000000000000000000000  
000000000000000000000000000000000000000000000000000000000000  
000000000000000000000000000000000000000000000000000000000000]
```

```
while (numString.length % 512 != 448) {  
    numString += '0';  
}
```

Figure 14 : Screen du code pour montrer la boucle while

6) Prenant un code binaire de 8 bits ASCII de l'étape 3

48 : 110000

```
const length = binary8bit.join('').length;  
const binaryLength = utils.asciiToBinary(length);
```

Figure 15 : Screen du code montrant un code binaire de 8bits ASCII

7) On ajoute des zeros jusqu'à obtention d'une longueur de 64 bits :

```
000000000000000000000000000000000000000000000000000000000000  
000000110000
```

```
const paddedBinLength = utils.padZero(binaryLength, 64);  
numString += paddedBinLength;
```

Figure 16 : Screen du code montrant le pad jusqu'à obtention de 64bits

8) On ajoute le code binaire à la dernière étape au code qu'on a obtenu dans l'étape 5.

9) Décomposer le message en un tableau de «morceaux» de 512 caractères :

```
const chunks = utils.stringSplit(numString, 512);
```

**Figure 17 : Screen du code montrant la décomposition de 512 caractères**

10) Diviser chaque morceau en un sous-tableau de 16 ‘mots’ de 32 bits :

```
[0100000100100000101010001100101 ,  
01110011011101001000000000000000,  
00000000000000000000000000000000 ,  
00000000000000000000000000000000,  
00000000000000000000000000000000,0000000000000000  
0000000000000000,  
00000000000000000000000000000000,0000000000000000  
0000000000000000,  
00000000000000000000000000000000,0000000000000000  
0000000000000000,  
00000000000000000000000000000000,0000000000000000  
0000000000000000,  
00000000000000000000000000000000,0000000000000000  
000000001100000]
```

11) Faire une boucle à travers chaque «bloc» de seize «mots» de 32 bits et étendre chaque tableau à 80 «mots» en utilisant des opérations XOR au niveau du bit :

```
// chunkWords is an array of 'chunk' subarrays
const word80 = chunkWords.map((chunk) => {
    // we start with a 'chunk' array of 16 32-bit 'words'
    for (let i = 16; i <= 79; i++) {
        // take four words from that chunk using your current i in the loop
        const wordA = chunk[i - 3];
        const wordB = chunk[i - 8];
        const wordC = chunk[i - 14];
        const wordD = chunk[i - 16];

        // perform consecutive XOR bitwise
        // operations going through each word
        const xorA = utils.XOR(wordA, wordB);
        const xorB = utils.XOR(xorA, wordC);
        CONST xorC = utils.XOR(xorB, wordD);

        // left rotate by one
        const newWord = utils.leftRotate(xorC, 1);
        // append to the array and continue the loop
        chunk.push(newWord);
    }
    return array;
})
```

**Figure 18 : Screen du code avec des opérations XOR sur chaque bit**

F

**Figure 19 : Résultat du code en binaire**

## 12) Initialiser certaines variables :

```

let h0 = '01100111010001010010001100000001';
let h1 = '11101111100110110101110001001';
let h2 = '10011000101110101101110011111110';
let h3 = '00010000001100100101010001110110';
let h4 = '11000011110100101110000111110000';

let a = h0;
let b = h1;
let c = h2;
let d = h3;
let e = h4;

```

**Figure 20 : Screen du code avec initialisation des variables**

13) Boucle sur chaque morceau: opérations au niveau du bit et réaffectation des variables :

**h0 = 01100111010001010010001100000001**

**h1 = 1110111110011011010101110001001**

**h2 = 10011000101110101101110011111110**

**h3 = 0001000000110010010101010001110110**

**h4 = 11000011110100101110000111110000**

**h0 = 10001111000011000000100001010101**

**h1 = 10010001010101100011001111100100**

**h2 = 1010011110111100001100101000110**

**h3 = 10001011001110000111010011001000**

**h4 = 10010000000111011111000001000011**

```

    }
    for (let i = 0; i < words80.length; i++) {
        for (let j = 0; j < 80; j++) {
            let f;
            let k;
            if (j < 20) {
                const BandC = utils.and(b, c);
                const notB = utils.and(utils.not(b), d);
                f = utils.or(BandC, notB);
                k = '01011010100000100111100110011001';
            } else if (j < 40) {
                const BxorC = utils.xOR(b, c);
                f = utils.xOR(BxorC, d);
                k = '';
            } else if (j < 60) {
                const BandC = utils.and(b, c);
                const BandD = utils.and(b, d);
                const CandD = utils.and(c, d);
                const BandCorBandD = utils.or(BandC, BandD);
                f = utils.or(BandCorBandD, CandD);
                k = '';
            } else {
                const BxorC = utils.xOR(b, c);
                f = utils.or(BxorC, d);
                k = '';
            }
            const word = word80[i][j];
            const tempA = utils.binaryAddition(utils.leftRotate(utils.leftRotate(a, 5), f));
            const tempB = utils.binaryAddition(tempA, e);
            const tempC = utils.binaryAddition(tempB, k);
            let temp = utils.binaryAddition(tempC, word);

            temp = utils.truncate(temp, 32);
            e = d;
            d = c;
            c = utils.leftRotate(b, 30);
            b = a;
            a = temp;
        }
        h0 = utils.truncate(utils.binaryAddition(h0, a), 32);
        h1 = utils.truncate(utils.binaryAddition(h1, a), 32);
        h2 = utils.truncate(utils.binaryAddition(h2, a), 32);
        h3 = utils.truncate(utils.binaryAddition(h3, a), 32);
        h4 = utils.truncate(utils.binaryAddition(h4, a), 32);
    }
    return [h0, h1, h2, h3, h4].map((string) => utils.binarytoHex(string).join(''));
}

```

**Figure 21 : Screen du code montrant les différentes boucles sur chaque morceau**

14) Convertir chacune des cinq variables résultantes en hexadécimal :

**h0 = 10001111000011000000100001010101 -> h0= 8F0C0855**

**h1 = 10010001010101100011001111100100 -> h1=915633E4**

**h2 = 101001111011100001100101000110 -> h2=a7de1946**

**h3 = 10001011001110000111010011001000 -> h3=8B3874C8**

**h4 = 10010000000111011111000001000011 -> h4=901DF043**

```
return [h0, h1, h2, h3, h4].map((string) => utils.binarytoHex(string).join(''));
```

**Figure 22 : Screen du code qui va retourner les variables en hexadécimal**

15) Les rejoindre et le retourner :

**H = 8F0C0855915633E4A7DE19468B3874C8901DF043**

## II. Sécurité du mot de passe grâce à SHA-1 dans une application de gestion :

Voici un exemple d'une bonne façon d'enregistrement des données d'une manière sécurisée.

Ce site web a été créé pendant un stage d'initiation au sein de l'académie régionale de l'éducation et la formation professionnelles d'oriental.

pour s'authentifier l'utilisateur saisit son mail et son mot de passe, le système vérifie ces informations dans la base de données et lui renvoie la page d'accueil correspondante pour se connecter à l'application.

En cas d'erreur le système affiche un message d'erreur et demande de ressaisir le mail et le mot de passe.

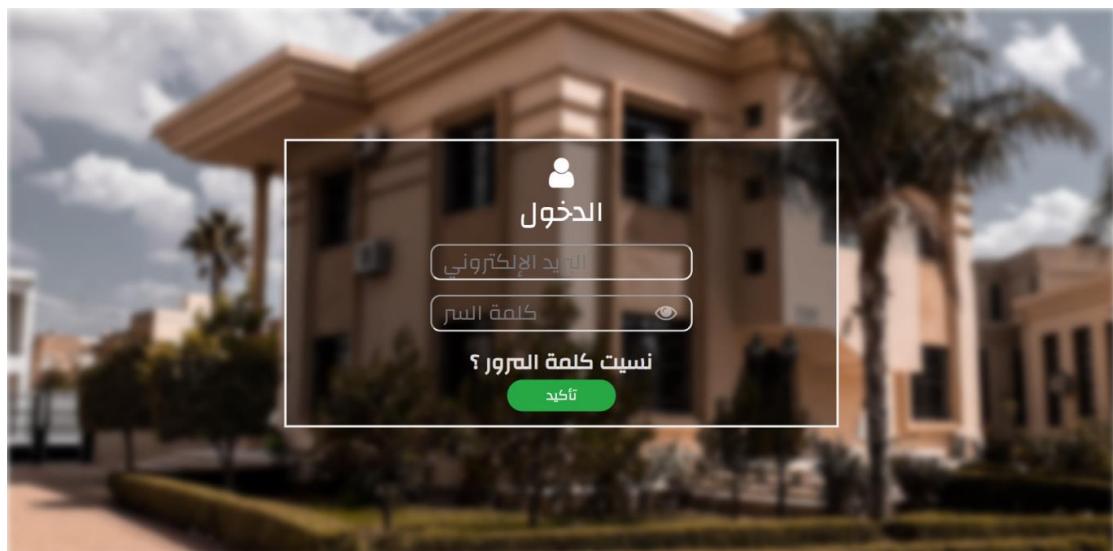


Figure 23 : Formulaire de connexion

Nous pouvons maintenant nous demander comment le système gère l'authentification dans cette application web d'une façon sécurisé ?

Alors dans notre exemple, on peut voir que dans notre espace d'administration web de bases de données qu'on a des mots de passe hachés.

La figure suivante représente l'espace d'administration web des bases de données :

The screenshot shows the phpMyAdmin interface for MySQL version 3.30.6. The left sidebar lists databases: academy, New, etablissement, users, ville, z\_elab, information\_schema, mysql, performance\_schema, sys, and tp11. The academy database is selected. The main area displays the 'users' table with the following data:

	id	username	email	password	role	ID_Ville
<a href="#">Edit</a>	87	Nada	nada@gmail.com	1e92f2c3063620494a76eb5a09e743879924f0d		
<a href="#">Edit</a>	85	Admin	ayman.makhoukhi@gmail.com	3a8e4ccded0cc9e93a496ce7393c4ef72c86410c		
<a href="#">Edit</a>	78	shawnCam	shawnv@gmail.com	e38f496a403c15ca97359a447aca815769384d1f		
<a href="#">Edit</a>	74	joyner	joyner@gmail.com	5e563300aaaaaa07bd40002a0d933785534ec0402		

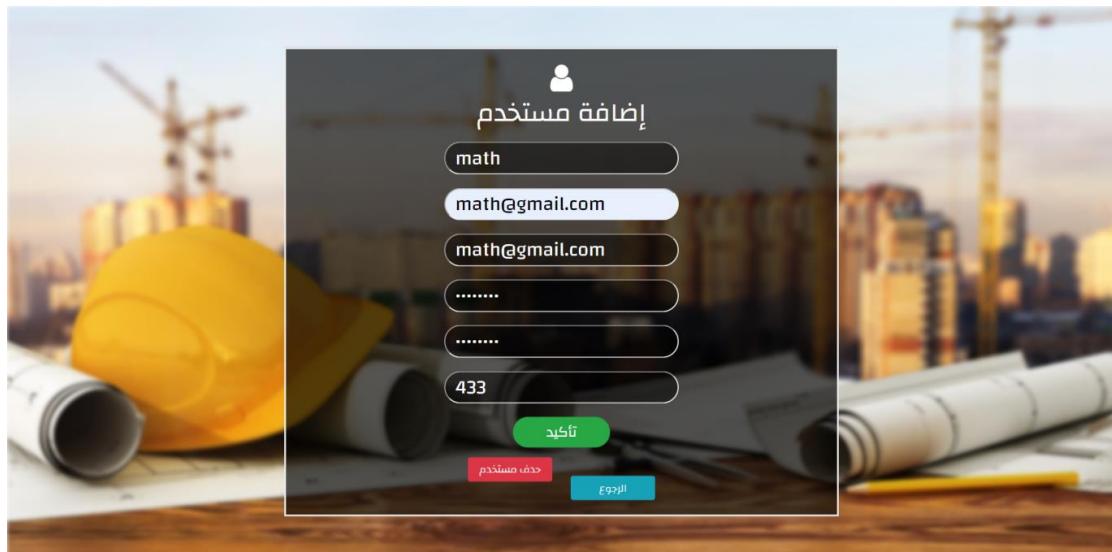
**Figure 24: L'espace d'administration web de la base de données**

Accédant à l'espace administrateur de l'application web, l'administrateur a le privilége de créer des nouveaux utilisateurs.

Alors on va créer un utilisateur avec un couple identifiant/mot de passe (math/math2020):

La figure suivante représente la création d'un nouveau utilisateur

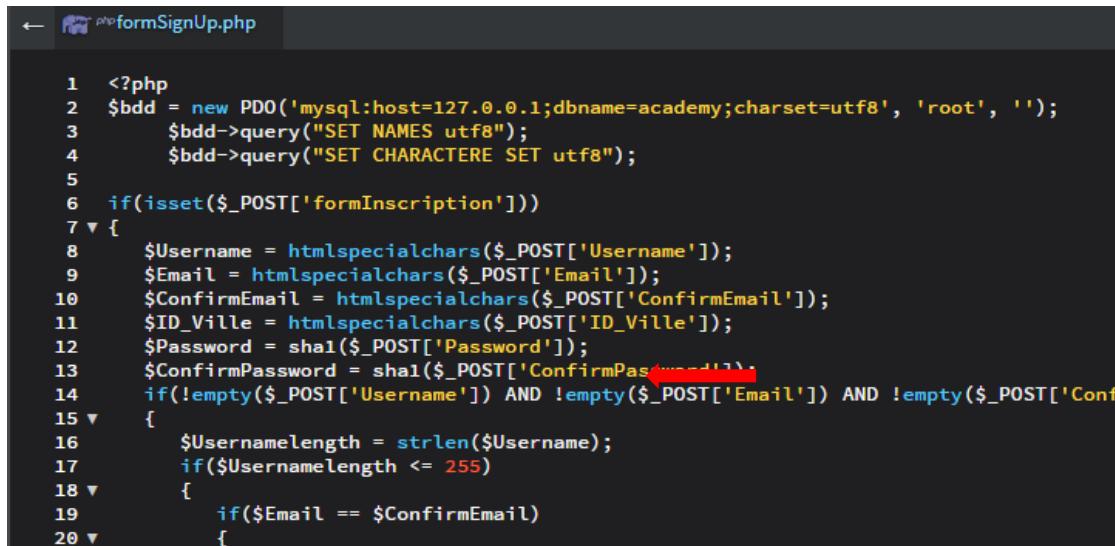
par l'administrateur:



**Figure 25 : Page de gestion des utilisateurs**

Alors après la saisie de mot de passe le système va hacher le mot de passe utilisant une fonction de hachage de n'importe quel algorithme donnée au système, dans notre cas on a utilisé la fonction sha1, et il va le stocker dans la base de données de l'application.

La figure suivante représente le code utilisé pour créer le nouveau utilisateur avec un mot de passe hachée utilisant la fonction sha1:



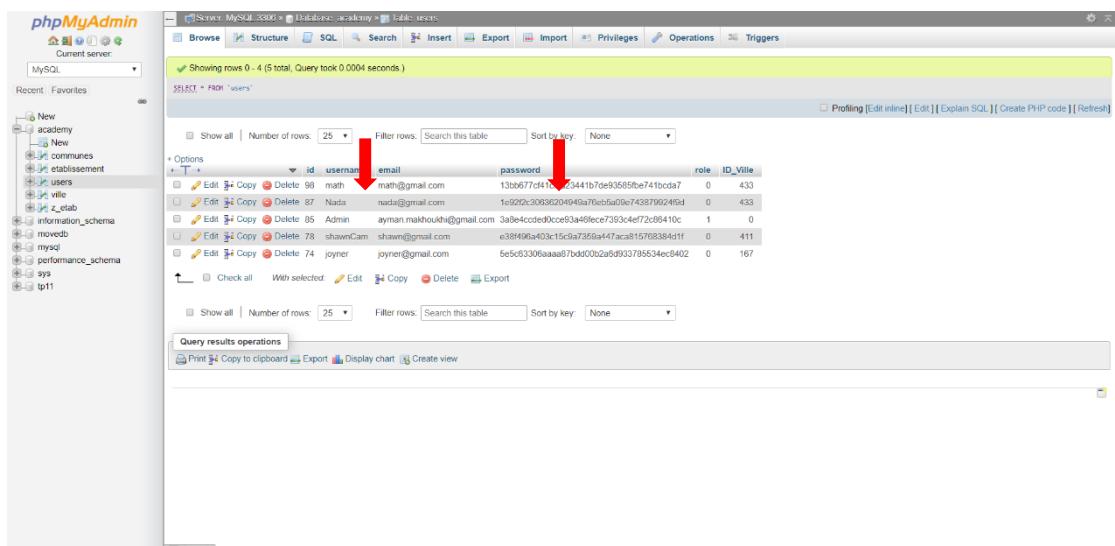
```

1 <?php
2 $bdd = new PDO('mysql:host=127.0.0.1;dbname=academy;charset=utf8', 'root', '');
3     $bdd->query("SET NAMES utf8");
4     $bdd->query("SET CHARACTER SET utf8");
5
6 if(isset($_POST['formInscription']))
7 {
8     $Username = htmlspecialchars($_POST['Username']);
9     $Email = htmlspecialchars($_POST['Email']);
10    $ConfirmEmail = htmlspecialchars($_POST['ConfirmEmail']);
11    $ID_Ville = htmlspecialchars($_POST['ID_Ville']);
12    $Password = sha1($_POST['Password']);
13    $ConfirmPassword = sha1($_POST['ConfirmPas...d1']);
14    if(!empty($_POST['Username']) AND !empty($_POST['Email']) AND !empty($_POST['Conf...d1'])
15 {
16        $Usernamelength = strlen($Username);
17        if($Usernamelength <= 255)
18 {
19            if($Email == $ConfirmEmail)
20 {

```

**Figure 26 : Code php de la création des utilisateurs**

La figure suivante représente que l'utilisateur a été bien créé avec un mot de passe haché irréversible:



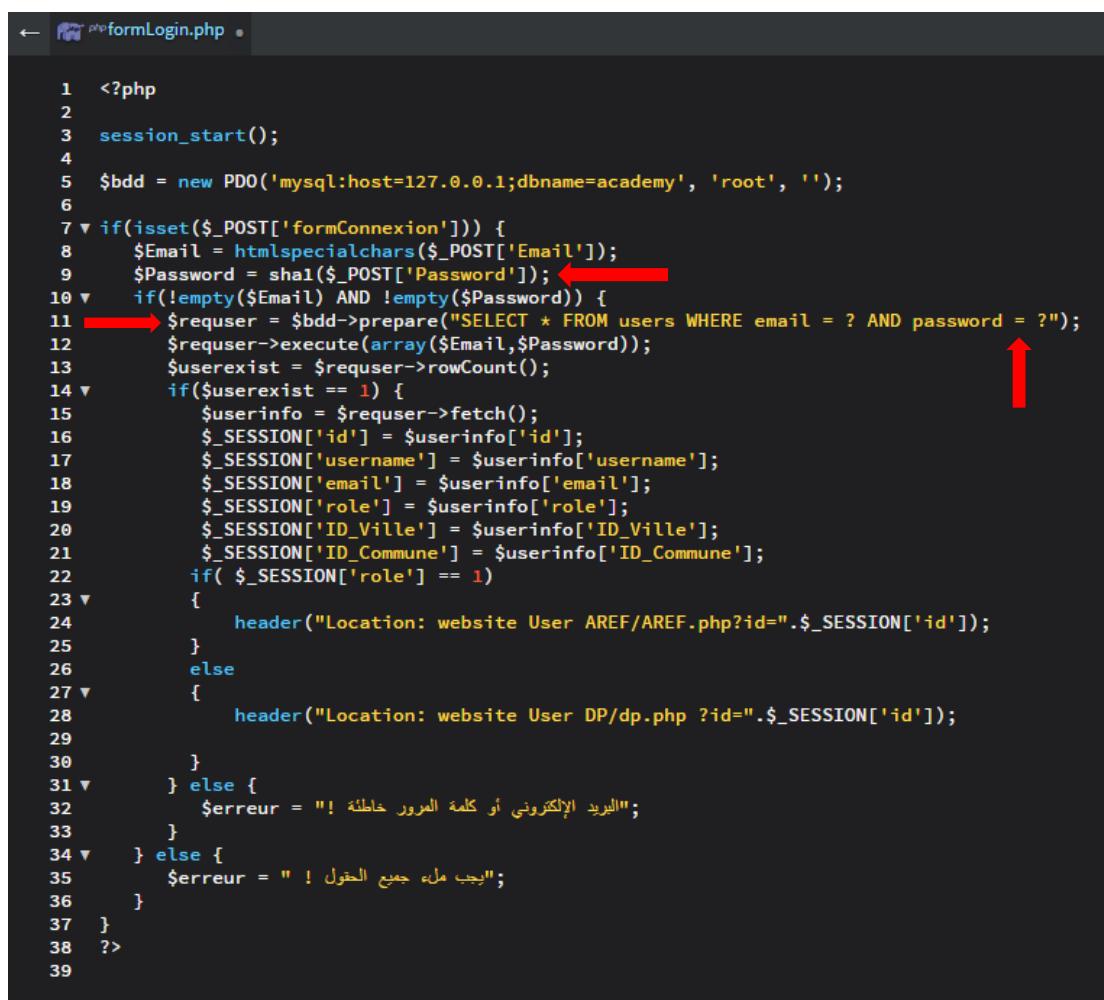
	id	username	email	password	role	ID_Ville
<input type="checkbox"/>	98	math	math@gmail.com	13bb677cf41cf0d23441b7de93585fbe741bcda7	0	433
<input type="checkbox"/>	87	Nada	nada@gmail.com	1e922c30620493a07e65fa00e743879b924fd	0	433
<input type="checkbox"/>	88	Admin	ayman.makhloufi@gmail.com	3a8e4cc0ed0cc693a49fece7393c4ef72b5410c	1	0
<input type="checkbox"/>	78	shawCam	shawm@gmail.com	e38496a403c15cda7359a4f7aca815768384d1f	0	411
<input type="checkbox"/>	74	joyne	joyne@gmail.com	6e5d3306aaaa07bdd002a5d933795534ec6402	0	167

**Figure 27 : Le hash de mot de passe “Math2020” stocké dans la base de données**

**Math2020 => 13bb677cf41cf0d23441b7de93585fbe741bcda7**

Si l'utilisateur veut accéder à l'application web, comme on a déjà mentioné, le système doit hasher le mot de passe saisi par l'utilisateur et lui comparer le mot de passe hachée qui se trouve dans la base de données.

Le code d'authentification sera comme suivant:



```
1 <?php
2
3 session_start();
4
5 $bdd = new PDO('mysql:host=127.0.0.1;dbname=academy', 'root', '');
6
7 if(isset($_POST['formConnexion'])) {
8     $Email = htmlspecialchars($_POST['Email']);
9     $Password = sha1($_POST['Password']); ↑
10    if(!empty($Email) AND !empty($Password)) {
11        $requser = $bdd->prepare("SELECT * FROM users WHERE email = ? AND password = ?"); ↑
12        $requser->execute(array($Email,$Password));
13        $userexist = $requser->rowCount();
14        if($userexist == 1) {
15            $userinfo = $requser->fetch();
16            $_SESSION['id'] = $userinfo['id'];
17            $_SESSION['username'] = $userinfo['username'];
18            $_SESSION['email'] = $userinfo['email'];
19            $_SESSION['role'] = $userinfo['role'];
20            $_SESSION['ID_Ville'] = $userinfo['ID_Ville'];
21            $_SESSION['ID_Commune'] = $userinfo['ID_Commune'];
22            if( $_SESSION['role'] == 1)
23            {
24                header("Location: website User AREF/AREF.php?id=".$_SESSION['id']);
25            }
26            else
27            {
28                header("Location: website User DP/dp.php ?id=".$_SESSION['id']);
29            }
30        }
31    } else {
32        $erreur = "البريد الإلكتروني أو كلمة المرور خاطئة !";
33    }
34 } else {
35     $erreur = "يجب ملء جميع الحقول !";
36 }
37 }
38 ?>
39
```

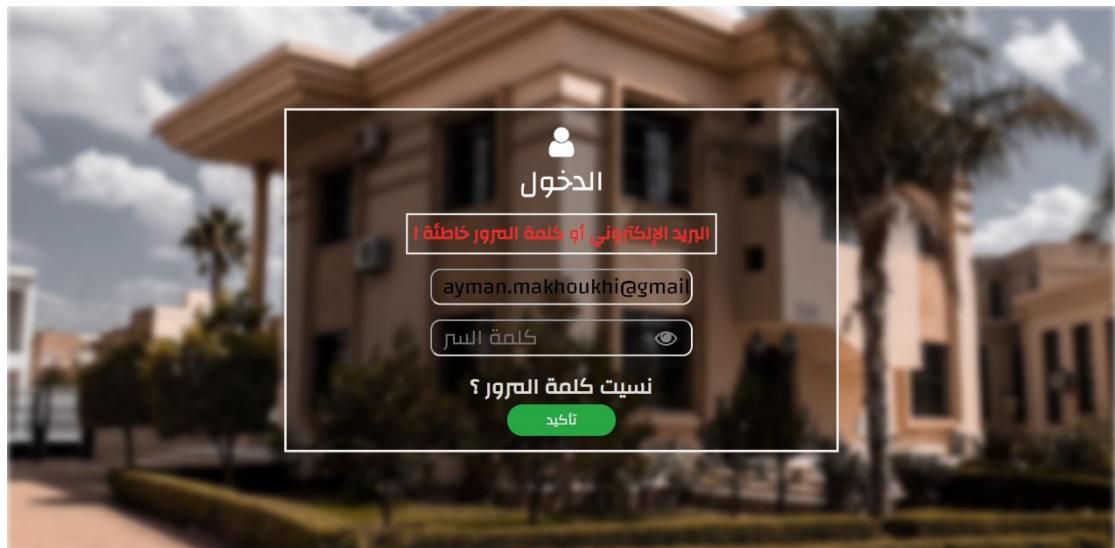
Figure 28 : Code php de l'authentification d'utilisateur

Si quelqu'un accède à la base de données de cette application, il peut récupérer les mots de passes des utilisateurs, mais dans notre serveur on n'a pas des vrais mot de passes, on a que les

haches de ces dernières, ces fonction d'hashages ont plusieurs

propriétés mais la plus importante c'est qu'il n'y a pas de réciproque, cela veut dire que le hacker peut ne pas avoir les mots de passe des utilisateurs pour accéder à ses profils.

Supposons que le hacker utilise les hash de mots de passes pour s'authentifier, le système va afficher un message d'erreur, il lui demandera de ressaisir le mail et le mot de passe car ils sont incorrectes, car le système a comparé le couple identifiant/"hach de mot de passe saisie" par le hacker avec le couple qui se trouve dans la base de données et il a trouvé qu'ils ne sont pas identiques.



**Figure 29 : Cas d'erreur pendant l'authentification**

### III. Site montrant les différents algorithmes de hachages :

Voila ce petit site qu'on a créé, ce site nous permettra de voir les résultats des différents algorithmes de hachage sur le même mot afin que nous puissions constater la différence entre les hashes.

```
Hashes Functions example

md4 hash:
7bdc3410354386a8a721daef20ff3323
md5 hash:
47aa0781e6dd48f4a0fe39e5bf4ba8f3
sha1 hash:
16e07b228eb601450502f458ac9e8e6f03f1fec
la longeur de hash sha1 du mot 'Nada' est :
40

sha256 hash:
b74e0813630ee3576bc3a15894a83633063a5e7cf3e7a3fb162a20d6a5be37be
sha384 hash:
87e8f6f76e47c21181006a2220a61236e1d773a52891a798d3bbad330d5eb3f5c791ceee3e02c3fc804a085b31e20259
sha512 hash:
b146bab704c6979323c5cbfb49cf5208f4a4b2cc8d30d360c65414bc823d67c3b400614cd6f64910fcb3743459a0dafc29faad4831150692330a25d777712ab
la longeur de hash sha512 du mot 'Nada' est :
128

whirlpool hash:
82370df06541805fbf449771a34baeb3fdbf2b13ec85bae7fceef1113104fd1dbd83bc63d8c33c242301ab0ff2386363624344f42f31dbd7a176a5165ec2cfb6b
tiger192,4 hash:
9505c134dd3e9ab8808c8a99c61f81733d2e5c6bd294c575
```

Figure 30 : Screen des fonctions de hachages

## **Conclusion :**

L'importance des fonctions hachage cryptographiques dans le monde de l'informatique :

- Comparaison des mots de passe.
- Vérification des données téléchargées.
- La signature électronique (ou signature numérique) des documents.
- Pour l'utilisation de tables de hachage, utilisées en développement des logiciels.
- Pour le stockage des données.

## Webographie :

<https://fr.wikipedia.org/>

<http://www.sha1-online.com/>

<https://www.youtube.com/>

<https://www.culture-informatique.net/>