Cryptanalyse de la fonction de hachage MD4

* Etude de l'attaque de H. Dobbertin et le calcul de sa complexité *

Introduction

Sous le nom propre « Information » se cachent aujourd'hui de nombreux domaines. Un nombre qui ne cesse de croître avec le développement continu des moyens de communication. Il est connu de tous que l'Internet joue aujourd'hui un rôle fondamental dans ces communications. L'Internet est à la fois le vecteur de transport de l'information en tant que réseau mondial, et le lieu où se crée cette information. Cependant, dans le cadre de l'étude que nous mènerons aujourd'hui, nous ne considérerons que ce premier aspect de vecteur du transport pour développer notre problématique.

Les échanges sur Internet sont d'une immense diversité. Certaines sont des plus banales, d'autres concernent au contraire des données très sensibles. Sensibles au point où leur modification accidentelle ou délibéré au cours du transfert constituerait un important danger. Pensons notamment aux données bancaires. C'est dans un tel cadre qu'est née la science moderne de la cryptologie. Moderne puisqu'elle existait déjà sous d'autres formes depuis l'antiquité. Nous ne nous attarderons pas sur ce qu'est la cryptologie et supposerons que le lecteur est familier avec ses enjeux.

Une partie de la cryptologie concerne les fonctions de hachage. Le but d'une fonction de hachage est de créer, à partir de données de taille arbitraire, une clé de taille fixe et relativement petite. Cette clé servira d'empreinte digitale au fichier dont elle a été extraite et permettra de détecter d'éventuels changements dans le fichier en ceci qu'elle variera avec le moindre changement de ce dernier.

Or, comme toute autre code secret, les fonctions de hachage sont eux aussi susceptibles d'être pris pour cible par des attaques et des actes de piraterie informatique. Afin de garantir la sécurité des protocoles en cours d'utilisation, les chercheurs sont obligés de se mettre dans la peau des pirates informatiques afin d'évaluer par eux-mêmes la vulnérabilité d'un protocole donnée. C'est la tâche de la cryptanalyse et constituera le fil directeur de notre étude.

1. Le protocole MD4

a. Introduction au hachage

Comme exposé dans l'introduction, le hachage a pour but de créer une clé de taille fixe à partir d'un fichier initial de taille arbitraire. Par la suite nous préfèrerons le nom de « hash » au terme de « clé ». Nous pouvons donc voir les fonctions de hachage comme des applications de l'ensemble des fichiers, soit l'ensemble des mots sur l'alphabet {0,1}, dans l'ensemble des mots de taille k sur ce même alphabet. L'entier k est défini par la fonction de hachage utilisé et a pour valeurs habituelles 64, 128, 256, 512.

Une remarque élémentaire mais fondamentale est qu'à cause de la taille des ensembles de départ et d'arrivée, ces applications seront toujours largement surjectives et non bijectives. Il est donc tout à fait possible de trouver deux fichiers différents ayant le même hash. Toutefois nous ne devons pas considérer tous les mots de l'alphabet {0,1} puisque nous ne traitons que des fichiers ayant un sens informatique, mais ceci, tout en limitant le risque d'un tel évènement, ne saurait exclure l'existence de deux fichiers ayant le même hash.

b. Le traitement des données

En excluant éventuellement quelques cas particuliers, les fonctions de hachage prennent en argument des blocs de taille fixe et non des fichiers de taille quelconque. Ce fait induit la nécessité d'un certain prétraitement des données avant de pouvoir appliquer la fonction de hachage. Nous utiliserons l'exemple de la fonction MD4 pour exposer ce en quoi consiste un tel traitement.

MD4 travaille sur des blocs 512-bits. Ainsi le fichier, avant hachage doit être modifié pour avoir une longueur qui est un multiple de 512. Cela se fait au moyen du procédé décrit sur la figure (1) de l'annexe.

Il reste toutefois la multiplicité des blocs à traiter, sachant que MD4 ne traite qu'un unique bloc de taille 512. Ici nous découvrons encore une fois un procédé utilisé dans la quasi-totalité des protocoles de hachage. Il s'agit de la construction de Merkle-Damgård. Cette construction repose sur l'utilisation d'un vecteur d'initialisation et le fait que le résultat issue du premier bloc sert de vecteur d'initialisation au traitement du deuxième bloc et ainsi de suite. Une illustration du procédé peut être trouvé dans les annexes (2), le sigle MD4 pouvant être remplacé par n'importe quel autre fonction de compression ayant les bonnes caractéristiques d'entrée et de sortie.

Un dernier avantage de la construction de Merkle-Damgård est que la sécurité d'un protocole est équivalente à la sécurité de la fonction de compression qui est utilisé.

c. La fonction MD4

Pour le calcul d'un bloc la fonction MD4 a besoin des données suivantes :

- Un vecteur d'initialisation :
  IV = {67452301, efcdab89, 98badcfe, 10325476}
- Trois constantes : $K_1 = 0$
  $K_2 = 5a827999$
  $K_3 = 6ed9eba1$
- Trois fonctions binaires non-linéaires :

$$F : (A,B,C) \rightarrow (A \wedge B) \vee (\neg A \wedge C) \quad + \quad K_1$$
$$G : (A,B,C) \rightarrow (A \vee B) \wedge (A \vee C) \wedge (B \vee C) \quad + \quad K_2$$
$$H : (A,B,C) \rightarrow A * B * C \quad + \quad K_3$$

Enfin, le bloc 512-bit sera traité comme une succession de 16 sous-mots 32-bits. Ceux-ci sont notées comme les X_i.

A partir de ces données la fonction procède de façon itérative pour obtenir son résultat. Le schéma fonctionnel d'une itération est donné en (3) et il faut au totale 48 itérations pour aboutir.
Les divers paramètres dont la valeur dépend du numéro de l'itération sont rassemblés dans le tableau (4).

Le hash du bloc est alors la suite des résultats des quatre dernières itérations auxquels ont été additionnées le vecteur d'initialisation.

Cette fonction de hachage a été inventé par Ron Rivest et fut publié en 1990. Les premières failles apparurent en 1994. A l'origine MD4 avait été spécifiquement conçu pour un hachage extrêmement rapide conduisant ainsi à une sécurité diminué du fait de la simplification des calculs le constituant.

2. Analyse de l'attaque de Dobbertin

a. Création d'une collision

Le principal et unique danger pour une fonction de hachage est qu'il existe un algorithme pouvant calculer en temps raisonnable une collision.

Une collision survient lorsque deux messages M_1 et M_2 tels que $M_1 \neq M_2$ ont le même hash. En cryptologie nous distinguons ainsi deux types de collisions :

- Les collisions faibles : Connaissant un message M donné et son hash K, trouver un message M' ≠ M, ayant le même hash K.
- Les collisions fortes : Trouver deux messages quelconques M et M' différents ayant le même hash.

Dans le cadre de cette étude, l'attaque que nous étudierons fournira une collision forte. Cette attaque a été présenté pour la première fois dans son intégralité en 1998 par le cryptologue allemand Hans Dobbertin. Elle a la spécificité d'utiliser la structure même de MD4 et d'être entièrement adapté à la cryptanalyse de cette fonction.

b. Les trois phases de l'attaque

Nous présenterons cette attaque « à l'envers » puisque nous commencerons par décrire la dernière étape pour ensuite remonter vers le début. Le but final de l'algorithme de Dobbertin est d'avoir deux messages M et M' tels qu'à partir de l'itération 32, les résultats de MD4 sont identiques pour les deux fonctions. D'autre part M et M' ne différeront que par leur 12ème sous-mot X_{12} en ce que:

$$X'_{12} = X_{12} + 1$$

Les itérations 19 à 35 seront traités au moyen d'hypothèses sur les résultats des itérations 15 à 19 sous forme d'équations. Dans cette phase de l'attaque il s'agira de maitriser la différence entre les itérations des deux messages. C'est une application directe du principe de la cryptanalyse différentielle. Nous voudrons ainsi vérifier les données du tableau (5) où les Q_i représentent le résultat de la i-ème itération et les ΔQ_i la différence $Q'_i - Q_i$. La colonne « p » donne la probabilité que, si la ligne précédente est vérifiée, la ligne courante le sera aussi.

Les itérations 12 à 18 permettent alors d'obtenir un système de 7 équations non-linéaires à vérifier pour satisfaire l'hypothèse nécessaire aux itérations 19 à 35. La résolution de ce système constitue la deuxième phase de l'attaque.

Enfin les étapes 1 à 11 nous donnent la possibilité de déterminer les X_i qui ne peuvent pas être déterminés par les autres phases. Ceci se fait par de simples calculs.

c. Justification du mode opératoire : les faiblesses de MD4

Dans son algorithme, H. Dobbertin a fait le choix d'une différence minimaliste entre les messages M et M'. En principe une fonction de hachage est construite de telle façon que son résultat peut être considéré comme pseudo-aléatoire. Cela constitue en quelque sorte une forte discontinuité de l'application en question. En principe il ne devrait donc pas importer que la différence entre M et M' soit grande. Toutefois minimiser cette différence permet de faire appel à la cryptanalyse différentielle, phase essentielle de l'attaque présentée ici.

D'autre part le choix de l'endroit où appliquer cette différence n'est pas anodin non plus. En consultant (4) nous pouvons remarquer que le i = 12 minimise la différence entre la première et la dernière utilisation de X_i dans MD4. Ce choix minimise donc aussi le nombre d'itérations sur laquelle nous devons maîtriser la différence entre les deux hashs et optimise donc notre probabilité de réussite.

3. Évaluation de la complexité temporelle

a. La phase de cryptanalyse différentielle

Nous travaillons ici donc sur les itérations 19 à 35 et il s'agit de vérifier les probabilités présentées dans (5). Une étude plus fine nous amène à n'avoir à étudier que trois types de cas différents. Les trois itérations types sont 28 → 29, 30 → 31 et 33 → 34.

Pour 28 → 29, il faut tenir compte selon (5) des différences entre les Q_i et les Q'_i. Les effets de ces différences au niveau binaire sont représentées en (6). De même (7) représente les équations à vérifier et le calcul de la probabilité que celles-ci soient effectivement vérifiées.
Le type 30 → 31 (resp. 33 → 34) se calcule de façon assez semblable et ce calcul est présenté en (8) (resp. (9)).

La complexité temporelle de l'algorithme cette phase est exactement égal à la probabilité de réussite de la maîtrise de la différence entre les hashs de M et M'. C'est donc le produit des probabilités de réussite de toutes les itérations. Soit d'environ $p = 1/2^{30}$

b. La résolution des équations non-linéaires

La résolution des équations induites par les itérations 12 à 18 forme la partie la plus complexe de l'attaque. Pourtant ce ne sera pas forcément l'étape la plus gourmande en temps de calcul. Les équations et leurs formes simplifiées que nous utiliserons sont écrites en (10).

En abordant de façon « naïve » les équations nous aurions une probabilité de $1/2^{96}$ d'obtenir une seule solution à cette équation sachant qu'il en faudra 2^{22} pour espérer obtenir une solution globale, soit une collision.

Cependant Dobbertin fait ici usage d'une approximation « continue » sur ces équations. Ainsi si une solution approximative est trouvé il serait possible en procédant par de petites modifications aléatoires d'aboutir à une vraie solution.

Il est admis aujourd'hui que cette étape possède une complexité bien inférieure à celle de la phase de cryptanalyse différentielle. Et elle n'augmentera donc pas la complexité globale de l'attaque au-delà d'un facteur $1/2^6$

c. Résultats empiriques et améliorations

Enfin les itérations 1 à 11 ne donnent lieu qu'à une complexité négligeable puisqu'elle n'impliquent que de simples calculs en temps constant.
La complexité temporelle théorique de l'ensemble d'attaque de H. Dobbertin peut donc être évalué à environ 2^{36}. Ce résultat reste cependant plutôt élevé même si elle implique la faisabilité technique d'une création systématique de collisions pour le protocole MD4.

Cependant divers résultats empiriques démontrent que cette complexité est largement surestimée et qu'en réalité elle ne serait que de 2^{26}.

D'autre part des avancées récentes en cryptanalyse ont conduit une équipe de chercheurs en Chine à présenter une nouvelle méthode de création de collisions pour le protocole MD4 et de nombreuses autres fonctions de hachage, dont le successeur MD4 et les fonctions de la famille SHA qui sont aujourd'hui largement répandues.

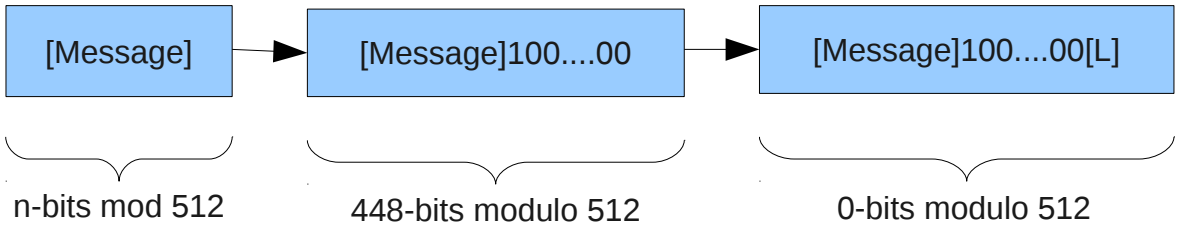
Conclusion et remerciements

Arrivés au bout de cette étude nous avons donc pu d'un côté montrer une méthode complète pour la création de collisions pour le protocole MD4. D'un autre côté nous avons aussi pu montrer son efficacité en temps. Cette dernière, lors de sa présentation a achevé de démontrer le caractère dépassé de la fonction MD4 et a encouragé la recherche et le développement de nouvelles fonctions de hachage plus sûres. Ce procédé se poursuit aujourd'hui avec l'arrivée à leur maturité de méthodes pouvant créer des collisions pour les protocoles de hachage utilisés de nos jours.

Pour ce travail je me dois de remercier Anne Canteaut, directrice du projet SECRET à l'Inria. Elle m'a notamment aidé au début de mes recherches dans le domaine de la cryptologie et plus spécifiquement de la cryptanalyse. De même je remercie mon professeur de Mathématiques Mr. Pommelet pour ses encouragements devant l'ampleur de mon travail.

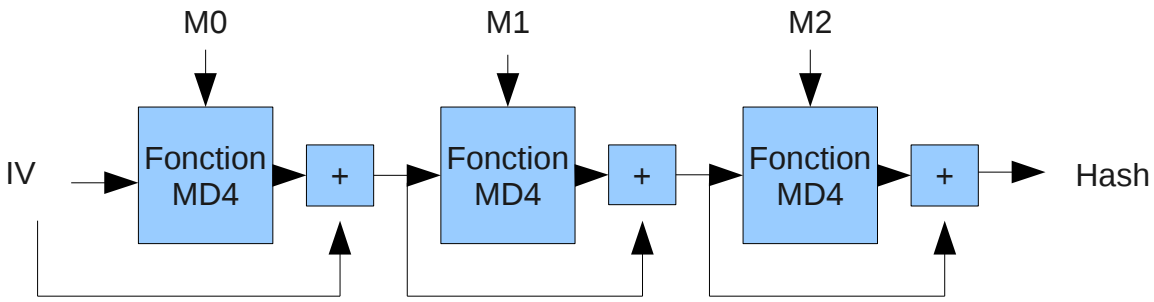
Annexes

(1)

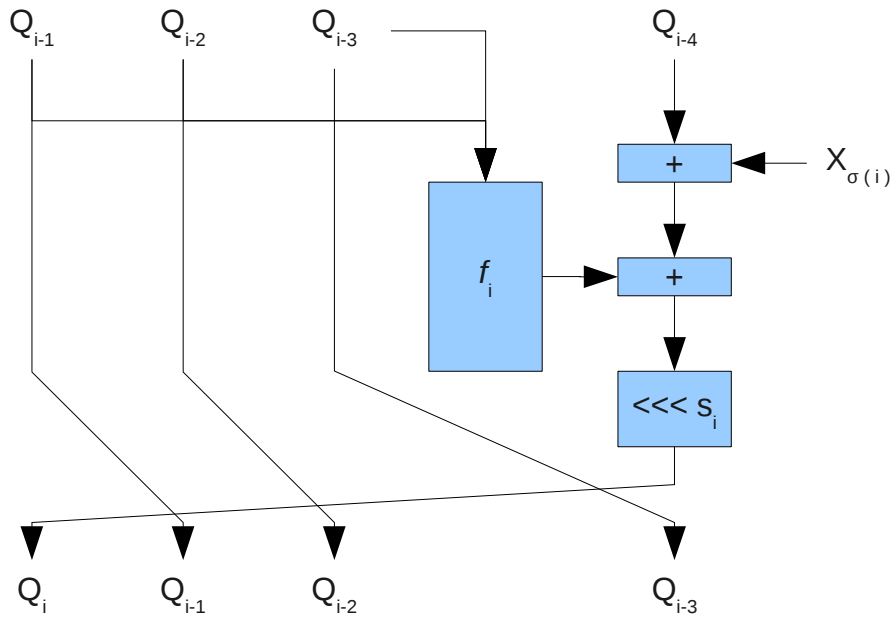


[L] = longueur du message avant le « padding » , soit une chaine de 64 bits.

(2)



(3)



(4)

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
s	3	7	11	19	3	7	11	19	3	7	11	19	3	7	11	19
f	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
σ	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

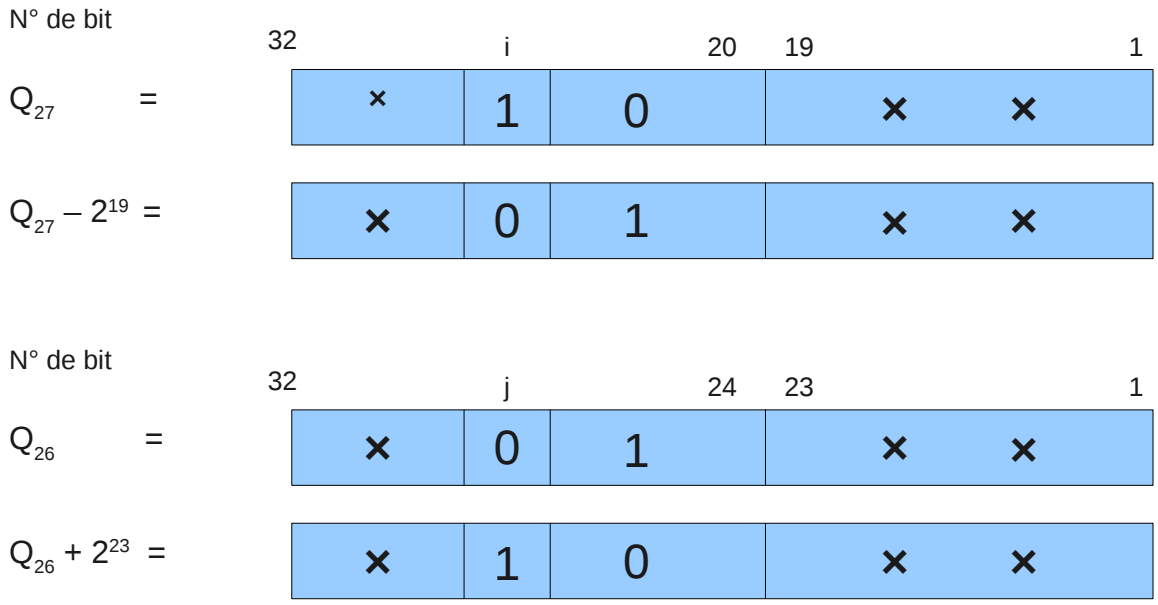
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
s	3	5	9	13	3	5	9	13	3	5	9	13	3	5	9	13
f	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
σ	0	4	8	12	1	5	9	13	2	6	10	14	3	7	11	15

i	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
s	3	9	11	15	3	9	11	15	3	9	11	15	3	9	11	15
f	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
σ	0	8	4	12	2	10	6	14	1	9	5	13	3	11	7	15

(5)

	Δ_j						
j	**ΔQ_j**	**ΔQ_{j-1}**	**ΔQ_{j-2}**	**ΔQ_{j-3}**	f_j	s_j	p
19	2^{25}	-2^{5}	0	0	*	*	*
20	0	2^{25}	-2^{5}	0	G	3	1
21	0	0	2^{25}	-2^{5}	G	5	1/9
22	-2^{14}	0	0	2^{25}	G	9	1/3
23	2^{6}	-2^{14}	0	0	G	13	1/3
24	0	2^{6}	-2^{14}	0	G	3	1/9
25	0	0	2^{6}	-2^{14}	G	5	1/9
26	-2^{23}	0	0	2^{6}	G	9	1/3
27	2^{19}	-2^{23}	0	0	G	13	1/3
28	0	2^{19}	-2^{23}	0	G	3	1/9
29	0	0	2^{19}	-2^{23}	G	5	1/9
30	-1	0	0	2^{19}	G	9	1/3
31	1	-1	0	0	G	13	1/3
32	0	1	-1	0	H	3	1/3
33	0	0	1	-1	H	9	1/3
34	0	0	0	1	H	11	1/3
35	0	0	0	0	H	15	1

(6)



(7)
$\Delta_{28} = (0 , 2^{19} , -2^{23} , 0)$
$Q_{29} = (Q_{25} + G(Q_{28} , Q_{27} , Q_{26}) + X_7) <<< 5$
$Q'_{29} = (Q'_{25} + G(Q'_{28} , Q'_{27} , Q'_{26}) + X'_7) <<< 5 =$
$(Q_{25} + G(Q_{28} , Q_{27} - 2^{19} , Q_{26} + 2^{23}) + X_7) <<< 5$

D'où nous voulons vérifier :
$G(Q_{28} , Q_{27} , Q_{26}) = G(Q_{28} , Q_{27} - 2^{19} , Q_{26} + 2^{23})$

Cela donne :
- p(i) est la probabilité que i ait une certaine valeur, de même pour p(j)
- $p_{i,j}$ est la probabilité à i et j fixé que l'égalité soit vérifié

$$p(i) = \left(\frac{1}{2}\right)^{i-19}$$

$$p(j) = \left(\frac{1}{2}\right)^{j-23}$$

$$p_{i,j} = \begin{cases} \left(\frac{1}{2}\right)^{(i-19)+(j-23)} & si \quad i < 24 \\ \left(\frac{1}{2}\right)^{4} & si \quad i = j \\ 0 & sinon \end{cases}$$

$$p = \sum_{i,j} p(i) * p(j) * p_{i,j}$$

$$p = \sum_{i=20}^{23} \sum_{j=24}^{33} \left(\frac{1}{2}\right)^{2(i-19)} \left(\frac{1}{2}\right)^{2(j-23)} + \sum_{i=24}^{33} \left(\frac{1}{2}\right)^{i-19} \left(\frac{1}{2}\right)^{i-23} \left(\frac{1}{2}\right)^{4}$$

$$p \simeq \frac{1}{9}$$

(8)
A présent l'égalité à vérifier est :
$G(Q_{30}, Q_{29}, Q_{28}) = G(Q_{30} + 1, Q_{29}, Q_{28})$

Soit :

$$p(i) = \left(\frac{1}{2}\right)^{i}$$

$$p_i = \left(\frac{1}{2}\right)^{i}$$

$$p = \sum_{i=1}^{33} p(i) * p_i$$

$$p = \sum_{i=1}^{33} \left(\frac{1}{2}\right)^{i} \left(\frac{1}{2}\right)^{i}$$

$$p \simeq \frac{1}{3}$$

(9)
Dernière égalité à vérifier :
$H(Q_{33}, Q_{32}, Q_{31}) + 1 = H(Q_{33}, Q_{32}, Q_{31} + 1)$

D'où le calcul de la probabilité donne :
- p(i) est la probabilité que les i-1 premiers bits de Q_{31} soient des 1 et le i-ème un 0
- p_i la probabilité à i fixé que les i premiers bits de Q_{32} et Q_{33} aient la même valeur

$$p(i) = \left(\frac{1}{2}\right)^{i}$$

$$p_i = \left(\frac{1}{2}\right)^{i}$$

$$p = \sum_{i=1}^{33} p(i) * p_i$$

$$p = \sum_{i=1}^{33} \left(\frac{1}{2}\right)^{i} \left(\frac{1}{2}\right)^{i}$$

$$p \simeq \frac{1}{3}$$

(10)

$$
\begin{aligned}
1 &= (Q'_{12}<<<29)-(Q_{12}<<<29)\\
F(Q'_{12,}Q_{11,}Q_{10})-F(Q_{12,}Q_{11,}Q_{10}) &= (Q'_{13}<<<25)-(Q_{13}<<<25)\\
F(Q'_{13,}Q'_{12,}Q_{11})-F(Q_{13,}Q_{12,}Q_{11}) &= (Q'_{14}<<<21)-(Q_{14}<<<21)\\
F(Q'_{14,}Q'_{13,}Q'_{12})-F(Q_{14,}Q_{13,}Q_{12}) &= (Q'_{15}<<<13)-(Q_{15}<<<13)\\
G(Q'_{15,}Q'_{14,}Q'_{13})-G(Q_{15,}G_{14,}G_{13}) &= Q_{12}-Q'_{12}\\
G(Q_{16,}Q'_{15,}Q'_{14})-G(Q_{16,}Q_{15,}Q_{14}) &= Q_{13}-Q'_{13}\\
G(Q_{17,}Q_{16,}Q'_{15})-G(Q_{17,}Q_{16,}Q_{15}) &= Q_{14}-Q'_{14}+(Q'_{18}<<<23)-(Q_{18}<<<23)\\
G(Q'_{18,}Q_{17,}Q_{16})-G(Q_{18,}Q_{17,}Q_{16}) &= Q_{15}-Q'_{15}+(Q'_{19}<<<19)-(Q_{19}<<<19)-1
\end{aligned}
$$

Ce qui pourra se simplifier en :

$$
\begin{aligned}
Q'_{15} &= Q_{15}-G(Q'_{18,}Q_{17,}Q_{16})+G(Q_{18,}Q_{17,}Q_{16})+(Q'_{19}<<<19)-(Q_{19}<<<19)-1\\
Q'_{14} &= Q_{14}-G(Q_{17,}Q_{16,}Q'_{15})+G(Q_{17,}Q_{16,}Q_{15})+(Q'_{18}<<<23)-(Q_{18}<<<23)\\
Q_{13} &= (Q_{14}<<<21)-(Q'_{14}<<<21)\\
Q'_{13} &= Q_{13}-G(Q_{16,}Q'_{15,}Q'_{14})+G(Q_{16,}Q_{15,}Q_{14})\\
Q_{10} &= (Q'_{13}<<<25)-(Q_{13}<<<25)
\end{aligned}
$$

et :

$$
\begin{aligned}
G(Q_{15,}Q_{14,}Q_{13})-G(Q'_{15,}Q'_{14,}Q'_{13}) &= 1\\
F(Q'_{14,}Q'_{13},0)-F(Q_{14,}Q_{13,}-1)-(Q'_{15}<<<13)+(Q_{15}<<<13) &= 0\\
G(Q_{19,}Q_{18,}Q_{17}) &= G(Q'_{19,}Q'_{18,}Q_{17})
\end{aligned}
$$

en supposant que $Q'_{12} = 0$, $Q_{12} = -1$ et $Q_{11} = 0$.

Implémentation partielle de l'attaque en Caml

```
#open "random";;
#open "big_int";;

let logic = function
      | 0 -> false;
      | 1 -> true;
      | _ -> failwith "func_error - logic";;

let rev_logic = function
      | false -> 0;
      | true -> 1;;

let conv_list l =
      let new_l = rev(l) in
      let rec aux = fun
            | [] k -> big_int_of_int 0;
            | (t::q) k ->
add_big_int (mult_int_big_int t (power_int_positive_int 2 k))(aux q (k+1));
      in
      aux new_l 0;;

let conv_bin n =
      let rec aux m = function
            | 0 -> [(int_of_big_int m)];
            | k -> let x =
      (int_of_big_int (div_big_int m (power_int_positive_int 2 k))) in
      x::(aux (mod_big_int m (power_int_positive_int 2 k)) (k-1));
      in
      aux n 31;;

let add_list l1 l2 =
      let rec aux mem = fun
            | [] [] -> [];
            | (t::q) (k::l) ->
            ((t + k + mem) mod 2)::(aux ((t + k + mem) / 2) q l);
            | _ _ ->
            failwith "Add_list error : input lists of different length";
      in
      rev(aux 0 (rev(l1)) (rev(l2)) );;

let substract_list l1 l2 =
      let rec aux mem = fun
            | [] [] -> [];
            | (t::q) (k::l) ->
      (abs((t - k - mem) mod 2))::(aux (rev_logic(t < (k + mem))) q l);
            | _ _ ->
      failwith "Substract_list error : input lists of different length";
      in
      rev(aux 0 (rev(l1)) (rev(l2)) );;

let lshift l1 n =
      let rec aux accu l1 p = match (l1,p) with
            | (t::q),0 -> accu,(t::q);
            | (t::q),k -> aux (t::accu) q (k-1);
            | _,_ -> failwith "lshift error : over 32-lshift";
      in
      let temp = aux [] l1 n in
      let right = fst(temp) and left = snd(temp) in
```

```
      left@(rev(right));;
let rec make_list n x = match n with
      | 0 -> [];
      | k -> x::(make_list (n-1) x);;

let rec make_list_rand n = match n with
      | 0 -> [];
      | k -> (random__int 2)::(make_list_rand (k-1));;

let rec func_f (x,y,z) = match (x,y,z) with
      | ([],[],[]) -> [];
      | (a::d,b::e,c::f) -> (rev_logic((logic(a) && logic(b)) ||
            (logic((a-1) mod 2) && logic(c))))::(func_f (d,e,f));
      | (_,_,_) ->
            failwith "func_f error : input lists of different length";;

let rec func_g (x,y,z) = match (x,y,z) with
      | ([],[],[]) -> [];
      | (a::d,b::e,c::f) -> (rev_logic((logic(a) && logic(b)) ||
      (logic(b) && logic(c)) || (logic(c) && logic(a))))::(func_g (d,e,f));
      | (_,_,_) ->
      failwith "func_g error : input lists of different length";;

let rec func_h (x,y,z) = match (x,y,z) with
      | ([],[],[]) -> [];
      | (a::d,b::e,c::f) -> ((a+b+c) mod 2)::(func_h (d,e,f));
      | (_,_,_) ->
            failwith "func_h error : input lists of different length";;

let gen() =
      let result = make_vect 15 [] in
      result.(1) <- make_list 32 0 ;
      result.(2) <- make_list 32 1 ;
      result.(11) <- make_list 32 0 ;
      for i=4 to 9 do
            result.(i) <- (make_list_rand 32);
      done;
      result;;

let compute_system basic =
      let temp_q_18 =
            add_list basic.(8) (conv_bin (power_int_positive_int 2 5)) in
      let temp_q_19 = lshift (substract_list basic.(9)
                        (conv_bin (power_int_positive_int 2 25)) ) 19 in
      let temp = ref (substract_list basic.(5)
                        (func_g (temp_q_18,basic.(7),basic.(6)) )) in
      temp := add_list !temp (func_g (basic.(8),basic.(7),basic.(6)) );
      temp := add_list !temp temp_q_19;
      temp := substract_list !temp (lshift basic.(9) 19);
      basic.(14) <- substract_list !temp ((make_list 31 0) @ [1]);
      temp := substract_list basic.(4)
                              (func_g (basic.(7),basic.(6),basic.(14)) );
      temp := add_list !temp (func_g (basic.(7),basic.(6),basic.(5)) );
      temp := add_list !temp (lshift temp_q_18 23);
      basic.(13) <- substract_list !temp (lshift basic.(8) 23);
      basic.(3) <- substract_list (lshift basic.(4) 21)
                                                (lshift basic.(13) 21);
      temp := substract_list basic.(3)
                              (func_g (basic.(6),basic.(14),basic.(13)) );
      basic.(12) <- add_list !temp
                              (func_g (basic.(6),basic.(5),basic.(14)) );
```

```
      basic.(0) <- substract_list (lshift basic.(12) 25)
                                                  (lshift basic.(3) 25);;

let check1 basic =
      (
      eq_big_int (sub_big_int (conv_list(func_g (basic.(5),basic.(4),basic.
(3)) )) (conv_list(func_g (basic.(14),basic.(13),basic.(12)) )))
(big_int_of_int(1))
       );;

let gen_initlist alea_init_vect =
      full_init alea_init_vect;
      let temp = gen() in
      compute_system temp;
      let test = ref (check1(temp)) in
      while not(!test) do
            let temp = gen() in
            compute_system(temp);
            test := check1(temp);
      done;
      temp;;
```

Bibliographie et remerciements

- **J. Buchmann**, Introduction à la cryptographie, Dunod, 2004

- **M. Stamp & R.M Low**, Applied cryptanalysis, Wiley-Interscience, 2007

- Wikipedia – Articles sur le hachage, la fonction MD4, MD5, SHA-1.
  http://fr.wikipedia.org/wiki/Fonction_de_hachage
  http://fr.wikipedia.org/wiki/MD4
  http://fr.wikipedia.org/wiki/MD5
  http://fr.wikipedia.org/wiki/SHA-1

- RFC de Ronald Rivest concernant le protocole MD4,
  http://www.faqs.org/rfcs/rfc1320.html