TIPE

Résistance aux collisions des fonctions de hachage cryptographiques

Nathan Stchepinsky [1]

MP
Lycée Henri Poincaré

10 août 2021

1. nathanstchepinsky@protonmail.com

Table des matières

Chapitre 1

Présentation et objectifs

I. Avant propos

La numérisation exponentielle de nos communications et de nos données personnelles nécessitent de très hauts standards de sécurité. Les *fonctions de hachage cryptographiques* s'imposent comme l'un des piliers fondamentaux de la sécurité informatique garantissant l'authentification et la sécurité de toutes nos communications ainsi que l'accès à nos données personnelles.

Étant donné le caractère crucial du hachage et l'évolution rapide de la puissance de calcul des ordinateurs, l'objectif de mon travail est d'étudier les vulnérabilités théoriques et pratiques des fonctions de hachage et de proposer une nouvelle méthode de hachage satisfaisant des critères établis lors de l'étude des vulnérabilités.

II. Mise en cohérence des objectifs

II.1. Positionnement thématiques

INFORMATIQUE *(Informatique Pratique)*, **MATHÉMATIQUES** *(Mathématiques appliquées)*, **INFORMATIQUE** *(Informatique Théorique)*.

II.2. Mots-clés

Mots-clés	**Keywords**
Fonctions de hachage cryptographiques	*Cryptographic hash functions*
Résistance aux collisions	*Collision resistance*
Algorithme Rho de Pollard	*Pollard's rho algorithm*
Polynôme de Lagrange	*Lagrange polynomial*
Paradoxe des anniversaires	*Birthday paradox*

III. Bibliographie commentée

Les *fonctions de hachage cryptographiques* figurent parmi les têtes de proue de la sécurité informatique. Leur ubiquité leur ont value le surnom de "chevaux de traits de la cryptographie moderne" [1]. Elles sont notamment utilisées pour authentifier et vérifier l'intégrité d'un message ou protéger des mots de passe au sein d'une base de données. Ce sont des fonctions à sens unique, qui à un message de taille arbitraire associent un *haché* ou une *empreinte* de taille fixe dont il est à priori impossible - du moins en un temps raisonnable - de calculer un antécédent [4].

La sûreté et la robustesse des fonctions de hachage cryptographiques sont garanties des propriétés essentielles. Tout d'abord, ces dernières doivent être *resistantes à la pré-image* [2] c'est-à-dire qu'il doit être

impossible en pratique de trouver un message à l'origine d'une empreinte donnée. La seconde propriété fondamentale est la *resistance à la seconde pré-image* [2], qui définit l'impossibilité pratique de trouver, pour tout message m, un second message m', différent de m, tel que m et m' aient même empreinte. Enfin, une fonction de hachage cryptographique doit être *resistante aux collisions* [2]. Une collision étant l'existence de deux messages ayant la même empreinte. La résistance aux collisions traduit donc la difficulté pratique de trouver des collisions. C'est la propriété que je propose d'étudier au sein de ce travail.

Il faut noter que par définition, les fonctions de hachage sont fortement non injectives, ayant un ensemble de départ de cardinal infini et un ensemble d'arrivée, l'ensemble des empreintes, fini. Les collisions existent donc par construction même, de ces fonctions. Trouver des collisions sert deux objectifs principaux. Le premier est l'usurpation de signature [3]. Les fonctions de hachage cryptographique étant utilisées, pour *signer*, c'est-à-dire authentifier, générer des collisions permet à l'attaquant de créer un message altéré, ayant même signature que le message initial. Le deuxième objectif sert la compréhension du mécanisme fondamental de la fonction de hachage à attaquer dans le but d'attaquer la résistance à la seconde pré-image, voir même, la résistance à la pré-image et ainsi effectuer la cryptanalyse complète de la fonction de hachage.

À cet égard, une attaque possible de la resistance aux collisions se base sur l'algorithme Rho de Pollard qui est historiquement attribué à la factorisation d'entiers [5]. Cette attaque efficace et systématique permet de trouver des collisions avec une complexité temporelle probabiliste en $\mathcal{O}(2^{\ell/2})$ appels à la fonction de hachage à attaquer, avec ℓ la taille des empreintes considérées [6]. Comme la fonction de hachage est à valeur dans un ensemble fini, on utilise l'algorithme du lièvre et de la tortue, *(ou algorithme de Floyd)* qui permet de détecter des cycles au sein d'une suite récurrente [5, 6]. Ainsi, la fonction atteint une complexité spatiale en $\Theta(1)$, ce qui démarque cette attaque des autres attaques systématiques qui visent à enregistrer toutes les empreintes générées [6].

Enfin, l'étude de la résistance aux collisions des fonctions de hachage cryptographiques fera l'objet de la recherche d'optimisation de cette résistance aux collisions au travers de la création d'une toute nouvelle fonction de hachage. Pour cela, j'ai utilisé le polynôme interpolateur de Lagrange, dont j'ai extrait des propriétés intéressantes pour le hachage, telles que la majoration du degré du polynôme et l'intervention de la majorité des points à interpoler dans l'expression des coefficients du polynôme interpolateur [7]. D'autres propriétés comme le caractère bijectif de l'interpolation ont toutefois dû être corrigées pour créer une fonction de hachage viable et respectant les propriétés fondamentales requises.

IV. Problématique

Comment la résistance aux collision d'une fonction de hachage cryptographique peut-elle être systématiquement et efficacement attaquée ? Par quels procédés peut-on optimiser la résistance aux collisions d'une fonction de hachage cryptographique ?

V. Objectifs

1. Identifier, formaliser et évaluer la criticité des propriétés fondamentales des fonctions de hachage cryptographiques
2. Formaliser, implanter l'algorithme Rho de Pollard et attaquer la résistance aux collisions d'une fonction de hachage déjà existante avec cet algorithme.
3. Étudier les propriétés sous-jacentes de la résistance aux collisions afin de créer une nouvelle fonction de hachage dont la résistance aux collisions est optimisée. Formaliser, et implanter cette nouvelle fonction de hachage cryptographique.
4. Vérifier la bonne définition et construction de la fonction de hachage créée. L'attaquer avec l'algorithme Rho de Pollard. Comparer avec les attaques déjà effectuées.

Chapitre 2

Recherche de collisions basée sur l'algorithme Rho de Pollard

I. Préliminaires

Dans cette partie nous nous attelons à la démonstrations de plusieurs résultats qui fondent l'algorithme Rho de Pollard.

I.1. Recherche de périodicité

n appelle $\mathcal{M} = \{0,1\}^*$ l'ensemble des messages et $\mathcal{H} = \{0,1\}^\ell$ l'ensemble des empreintes possible de taille $\ell \in \mathbb{N}$.

Soit R une application **injective** de $\mathcal{H}$ dans $\mathcal{M}$.
Soit H l'application associant à chaque mot de $\mathcal{M}$ son empreinte dans $\mathcal{H}$.
On posera alors dans la suite l'application $f = H \circ R$ définie sur $\mathcal{H}$ à valeurs dans $\mathcal{H}$. On munit alors $\mathcal{H}$ de la loi de composition interne usuelle sur les applications, notée $\circ$. Ainsi, pour tout $k \in \mathbb{N}$, on notera f^k la composition k-ième de f avec elle-même.

On choisit arbitrairement un mot $m_0 \in \mathcal{M}$ et on définit la suite $(u_n)_{n\in\mathbb{N}}$ telle que

$$\begin{cases} u_0 = m_0 \\ \forall i \in \mathbb{N}, u_{i+1} = f(u_i) \end{cases}$$

Propriété 1

La suite $(u_n)_{n\in\mathbb{N}}$ est périodique.

□ ***Démonstration*** *(Périodicité de $(u_n)_{n\in\mathbb{N}}$)*

↪ Par construction,

$$\forall i \in \mathbb{N},\ (H \circ R)(u_i) = u_{i+1} = f^{i+1}(u_0)$$

Notons alors que $\mathcal{H}$ est fini. Il existe donc deux entiers i_0 et i_1 tels que $i_0 < i_1$ et $u_{i_0} = u_{i_1}$. Donc, pour tout $k \in \mathbb{N}$, $f^k(u_{i_0}) = f^k(u_{i_1})$, i.e. $u_{i_0+k} = u_{i_1+k}$. On pose alors $\lambda = i_1 - i_0$. Donc pour tout $k \in \mathbb{N}$, $u_{i_0+k} = u_{(i_0+k)+(i_1-i_0)} = u_{(i_0+k)+\lambda}$.
Autrement dit,

$$\exists \lambda \in \mathbb{N};\ \forall i \in \mathbb{N},\ i \geqslant i_0 \Longrightarrow u_i = u_{i+\lambda}$$

Donc la suite $(u_n)_{n\in\mathbb{N}}$ est périodique à partir d'un certain rang, noté i_0.

CQFD

On supposera de plus que $i_0 \neq 0$. Si tel est le cas, on choisira un nouveau u_0.

Enfin, on notera n le plus petit entier tel que pour tout $i \in \mathbb{R}$, $u_{i+n} = u_i$

I.2. Optimisation de la recherche de cycle : l'algorithme du Lièvre et de la tortue

On extrait alors une sous-suite de $(v_n)_{n\in\mathbb{N}}$ de $(u_n)_{n\in\mathbb{N}}$ telle que pour tout $k \in \mathbb{N}$, $u_{2k} = v_k$.
Cela revient à construire $(v_n)_{n\in\mathbb{N}}$ telle que,

$$\begin{cases} v_0 = u_0 \\ \forall i \in \mathbb{N},\ v_{i+1} = f^2(v_i) \end{cases}$$

Propriété 2

S'il existe $j \in \mathbb{N}^*$ tel que $u_j = u'_j$ alors j est un multiple de n et le plus petit indice vérifiant cette propriété, noté j_0, vérifie

$$0 \leqslant j_0 < i_0 + n$$

□ ***Démonstration*** *(Propriété 2)*

↪ Supposons qu'il existe $(i, i') \in \mathbb{N}^2$ supérieurs à i_0 tels que $i < i'$ et $u_i = u_{i'}$. Le théorème de division euclidienne assure qu'il existe $(q, r) \in \mathbb{N} \times [\![0, n-1]\!]$ tels que

$$i' - i = qn + r \tag{2.1}$$

On a donc $u_i = u'_i = u_{i+qn+r}$, soit par définition de n, $u_i = u_{i+r}$.
Or, n est le plus petit entier tel que $u_i = u_{i+n}$. Donc $r = n$ ou $r = 0$. Or $0 \leqslant r < n$. Donc $r = 0$, donc $i - i' = qn$.
Donc, s'il existe $(i, i') \in \mathbb{N}^2$ supérieurs à i_0 tels que $i < i'$ et $u_i = u_{i'}$ alors $i - i'$ est un multiple de n.

Supposons qu'il existe $j \in \mathbb{N}$ tel que $u_j = v_j$.
On a donc, $u_j = v_j = u_{2j}$
Si $j < i_0$, cela contredit la définition i_0. j est donc nécessairement plus grand que i_0.
La condition précédente énonce que $2j - j = j$ est un multiple de n.
Ainsi, on note j_0 le plus petit entier, multiple de n supérieur à i_0.
Il existe alors $\lambda \in \mathbb{N}$ tel que $j_0 = \lambda n$. On a alors,

$$v_{j_0} = u_{2j_0} = u_{j_0+\lambda n} = u_{j_0}$$

On a donc bien construit un indice j_0, le plus petit indice possible tel que $v_{j_0} = u_{j_0}$ et $i_0 \leqslant j_0 < i_0 + n$.

CQFD

Propriété 3

i_0 est alors le premier indice telle que,

$$f(u_{i_0-1}) = f(u_{i_0-j_0-1})$$

Les empreintes $R(u_{i_0-1})$ et $R(u_{i_0-j_0-1})$ constituent une collision pour H.

□ ***Démonstration*** *(Propriété 3)*

↪ On pose finalement,

$$\begin{cases} w_0 = v_{j_0} \\ \forall i \in \mathbb{N},\ w_{i+1} = u_{i+j_0} \end{cases}$$

Or, comme j_0 est un multiple de n, on a

$$u_{i_0} = u_{i_0+j_0} = w_{i_0}$$

Il suffit alors de parcourir les suites $(u_n)_{n\in\mathbb{N}}$ et $(v_n)_{n\in\mathbb{N}}$ jusqu'à trouver un rang k tel que $u_k = v_k$. Le premier rang trouvé correspond à j_0.
On définit alors la suite $(w_n)_{n\in\mathbb{N}}$, et on parcourt les $(u_n)_{n\in\mathbb{N}}$ et $(w_n)_{n\in\mathbb{N}}$ jusqu'à trouver un rang p tel que $u_p = w_p$. Le premier rang trouvé correspond à i_0.

On aura alors, $H(R(u_{i_0-1})) = H(R(u_{j_0+i_0-1}))$.

Donc $R(u_{i_0-1})$ $R(u_{j_0+i_0-1})$ constituent une collision pour H.

CQFD

II. Complexité

II.1. Complexité spatiale

Propriété 4 (Complexité spatiale de Rho de Pollard)

La **complexité spatiale de l'algorithme Rho de Pollard** est en

$$\Theta(1)$$

Remarque C'est l'intérêt premier de cet algorithme On ne garde en mémoire que deux empreintes de longueur ℓ.

II.2. Complexité temporelle

Propriété 5 (Complexité temporelle de Rho de Pollard)

La **complexité temporelle de l'algorithme Rho de Pollard** est en

$$\mathcal{O}(2^{\ell/2})$$

□ ***Démonstration*** *(Complexité temporelle de Rho de Pollard)*

↪ On pose, $h = \text{Card}(\mathcal{H}) = 2^\ell$. Notons alors P la probabilité d'obtenir une collision pour $\mathcal{H}$ avec n à partir de n valeurs aléatoires . Le paradoxe des anniversaires assure que

$$P = 1 - \prod_{k=1}^{n-1} \left(1 - \frac{k}{h}\right)$$

Or, on sait que pour tout $x \in \mathbb{R}$, $1 - x \leqslant e^{-x}$. D'où,

$$1 - \prod_{k=1}^{n-1}\left(1 - \frac{k}{h}\right) \leqslant 1 - \prod_{k=1}^{n-1}\left(e^{k/h}\right)$$

Et,

$$1 - \prod_{k=1}^{n-1}\left(e^{k/h}\right) = 1 - \exp\left(-\frac{1}{h}\sum_{k=1}^{n-1} k\right) = 1 - \exp\left(-\frac{n(n-1)}{2h}\right) \leqslant 1 - \exp\left(-\frac{n^2}{2h}\right)$$

Par croissance de l'exponentielle.
Donc,

$$P \leqslant 1 - \exp\left(-\frac{n^2}{2h}\right) \tag{2.2}$$

On note N le plus petit nombre de valeur que l'on doit choisir pour obtenir une collision sur $\mathcal{H}$. Le point précédent assure alors que la probabilité d'en trouver effectivement une vaut au moins P.

L'équation 2.2 assure alors que

$$N \geqslant \sqrt{2h \ln\left(\frac{1}{1-P}\right)}$$

Or $h = 2^{\ell}$.

Finalement, la complexité temporelle de l'algorithme Rho de Pollard est en $\mathcal{O}(2^{\ell/2})$

CQFD

III. Algorithme Rho de Pollard

III.1. Description

On généré aléatoirement une empreinte `h0`.
On parcourt les suites `u` et `v` jusqu'à trouver une collision. Chaque itération effectue 3 appels à `f`. On note `j0` l'indice trouvé
On parcourt les suites `u` et `w` jusqu'à trouver une collision. Chaque itération effectue 2 appels à `f`. On note `i0` l'indice trouvé.
`R(v[i0-1])` et `R(w[i0-1])` constituent une collision pour `H`

III.2. Code source

```
1  #
2  # Created in 2020 by STCHEPINSKY Nathan
3  #
4  # Copyright (c) 2021 STCHEPINSKY Nathan. All rights reserved.
5  #
6
7  import random
8  import sys
9  import string
10 import numpy as np
11 import hashlib
12
13
14 class Rho_pollard:
15     """
16         Class definissant l'agorithme rho de pollard caractérisé par :
```

```
17          - Nombre de bits du hash
18          - Le nombre de caractère maximum du mot aléatoire initial
19          - L'affichage - ou non - des différents
20      """
21      def __init__(self, nb_bits, len_max_word, is_printing_hash, diff_hash = False, already_found
        ↪  = []):
22          self.len_max_word = len_max_word
23          self.nb_bits = nb_bits
24          self.is_printing_hash = is_printing_hash
25          self.diff_hash = diff_hash # Vrai : on ne renvoie que des collisions jamais trouvées
26          self.already_found = already_found
27
28      def message_generator(self):
29          '''
30          PRIVATE FUNCTION
31          Generate a random messages with a maximum length of max_length
32          '''
33          word_len = random.randint(1,self.len_max_word)
34          message = ''.join([random.choice(string.ascii_letters + string.digits) for y in
            ↪  range(word_len)])
35          return message
36
37
38      def H(self,hash):
39          '''
40          H function.
41          '''
42          #return md5.algo(hash, NB_BITS) # si on utilise notre propre hashage
43          return hashlib.md5(hash.encode()).hexdigest()[:self.nb_bits]
44
45      def R(self,hash):
46          '''
47          PRIVATE FUNCTION
48          R function. Identity
49          '''
50          #return self.H(hash)
51          return hash
52
53      def f(self,hash) :
54          '''
55          PRIVATE FUNCTION
56          f function. f(x) = H(R(x))
57          '''
58          return self.H(self.R(hash))
59
60
61
62      def search_j0(self,h0):
63          '''
64              PRIVATE FUNCTION
65              Trouve la première valeur j_0 telle que h_i = h'_i
66          '''
67          h = self.f(h0)
68          h_prime = self.f(self.f(h0))
69          if self.is_printing_hash:
70              print("h=", h)
71              print("h'", h_prime)
72          j = 1
73          while h != h_prime :
```

```
74              h = self.f(h)
75              h_prime = self.f(self.f(h_prime))
76              if self.is_printing_hash:
77                  print("h=", h)
78                  print("h'", h_prime)
79              j+=1
80              if j%100000 == 0:
81                  print(j, " hashs generated (" + str(self.nb_bits) + ")")
82          print("\n###\n j0 = ", j, "\n###\n")
83          return (h,j)
84
85      def search_i0(self,h0, h0_pprime):
86          '''
87              Trouve la première valeur j_0 telle que h_i = h''_i
88          '''
89
90          h = self.f(h0)
91          print("h=", h)
92          h_pprime = self.f(h0_pprime)
93          old_h = h0
94          old_h_pprime = h0 # correspondent a h_{i-1}
95          print("h''=", h_pprime)
96          i = 1
97          while h != h_pprime :
98              old_h = h
99              old_h_pprime = h_pprime
100             h = self.f(h)
101             h_pprime = self.f(h_pprime)
102             if self.is_printing_hash:
103                 print("h=", h)
104                 print("h''=", h_pprime)
105             i+=1
106             if i%100000 == 0:
107                 print(i, " hashs generated (" + str(self.nb_bits) + " bits)")
108         print("\n###\n i0 = ", i, "\n###\n")
109         print("old_h =", old_h)
110         print("old_h' =", old_h_pprime)
111
112         return (i, old_h, old_h_pprime)
113
114     def main(self):
115         i0 = 0
116         j0=0
117         while i0 == 0 : # Tant que i0 = 0, on recommence avec un autre h0
118             h0 = self.message_generator()
119             print("\n\nh0=", h0)
120             (h_pprime0,j0) = self.search_j0(h0)
121             (i0, old_hi0,old_h_pprime_i0) = self.search_i0(h0, h_pprime0)# old_hi0 et
                ↪  old_h_pprime_i0 correspondent à h_{i0-1} et    h''_{i0-1}
122         clear1 = self.R(old_hi0)
123         clear2 = self.R(old_h_pprime_i0)
124         print("\n\n\n ##################\n Collision trouvée\n\n H(",clear1, ")=H(", clear2,
            ↪  ")\n##################" )
125         return (clear1,clear2,i0+j0)
```

Chapitre 3

Construction d'une fonction de hachage à partir de l'interpolation Lagrangienne

I. Construction et formalisation de la fonction hachage

I.1. Motivations

Théorème 1 (Polynôme interpolateur de Lagrange)

Pour tout $(x_0, \dots, x_n) \in \mathbb{R}^{n+1}$ deux à deux distincts et pour tout $(y_0, \dots, y_n) \in \mathbb{R}^{n+1}$, le polynôme

$$L(X) = \sum_{j=0}^{n} y_j \left(\prod_{i \in [\![1,n]\!] \setminus \{j\}} \frac{X - x_i}{x_j - x_i} \right)$$

est l'unique polynôme de degré au plus n qui interpole $((x_0, y_0), \dots, (x_n, y_n))$.

Le polynôme interpolateur de Lagrange apporte d'intéressantes **qualités** que nous cherchons à exploiter :

— Le degré du polynôme est **majorée par** n.
↪ Cette propriété est intéressante de par sa stabilité. C'est une qualité essentielle pour la construction d'une fonction de hachage, dont on doit **contrôler la taille**.

— Les points $((x_0, y_0), \dots, (x_n, y_n)) \in \left(\mathbb{R}^2\right)^n$ à interpoler **interviennent au sein de chacuns des coefficients** du polynôme.
↪ Cette propriété est **fondamentale**. Afin de de garantir l'**effet avalanche** de la fonction de hachage, et rendre la fonction **resistante à la seconde pré-image** qui serait vulnérable à une attaque par omission ou permutation de lettres.

En revanche, il présente des **défauts**, qu'il nous faut corriger.

— La fonction Lagrangienne est **bijective**.
↪ Une fonction de hachage doit être **fortement non injective**. Il nous faut alors appliquer des opérations *a posteriori* au polynôme pour n'en garder que l'injectivité. *En pratique, nous utiliserons l'opération* `modulo` *sur les coefficients du polynôme.*

I.2. Formalisation mathématique

Dans ce document $n \in \mathbb{N}^*$ désigne le nombre de caractères composant le haché désiré.

Soit Σ l'alphabet codable en `ASCII`.
Soit $\Sigma' \subset \mathbb{N}$ l'alphabet Σ codé en `ASCII`.
Soit $\varphi_\Sigma : \Sigma^* \to \left(\Sigma'\right)^*$ la bijection associant à chaque mot de Σ son mot associé dans Σ'.

L'application $|\ |:\left(\Sigma'\right)^* \to \mathbb{N}$ désigne la taille d'un mot de Σ'

$S \in \Sigma^* \setminus \{\varepsilon\}$ désigne la chaîne de caractère à hacher.
Il existe alors $m \in \left(\Sigma'\right)^*$ tel que $m = \varphi_\Sigma(S)$. m désigne en fait S, codé en `ASCII`.

— Si $|m| < n$: L'entrée est bourrée pour atteindre une taille suffisante, *i.e.* n, selon le principe suivant :

Algorithme de bourrage inspiré du renforcement de Merkle-Damgard
Le bourrage, inspiré du bourrage de Merkle-Damgard, consiste ici à ajouter un 2, puis une suite de 1 à m jusqu'à ce que, $|m| = n - 1$. La dernière lettre ajoutée à m est $256+|S|$. Autrement dit, le mot bourrage de m noté m_B est défini tel que,

$$m_B \overset{\text{déf.}}{=} m.\{2\}.\{1\}^{n-1-|m|}.\{256 + |S|\}$$

où . désigne l'opération de *concaténation*. Par commodité, nous appellerons m, le mot m_B.

Cela garanti de ne pas pouvoir créer une collision avec un clair commençant par S et dont les $n - |S|$ dernières lettres sont les équivalents ASCII de 1. *En effet le code ASCII d'une lettre ne peut pas excéder 256*

Nous noterons que dans le cadre de notre fonction de hachage, il est plus profitable d'utiliser des entiers strictement positifs. En effet, les 0 ont un impact très fort sur le polynôme, en annulant le produit qui leur est associé. Ainsi, bourrer avec une suite nulle engendre un très haut risque de grande perte d'informations, d'autant plus grande que la taille du mot intial serait petite.

On note enfin, $(\ell_0, \ldots, \ell_{|m|-1}) \in \left(\Sigma'\right)^{|m|}$ les lettres de m.
On construit ainsi $L \in \mathbb{K}_{n-1}[X]$, le polynôme suivant :

Formule 1 (Polynôme de hachage)

$$L = \sum_{k=0}^{|m|-1} \ell_k \prod_{p=0}^{k} (X - \ell_p) \bmod X^{n-1}$$

On pose alors $(\lambda_0, \ldots, \lambda_{n-1})$ les coefficients de L. On encode finalement $\lambda_0 \ldots . \lambda_{n-1}$ sur l'alphabet l'alphabet latin muni des 10 premiers entiers. On note H cet encodage.

H est le haché de S.

Remarque 1 L'empreinte est donc une séquence de n caractères codés sur notre alphabet donc sur 5 bits.

Remarque 2 En pratique, comme l'on peut s'y attendre, les coefficients polynomiaux manipulés sont très grands devant 1 (de l'ordre de 10^{19} pour les plus grands). Par soucis de mémoire et de gestion de ces entiers (qui seraient par ailleurs impossible par n'importe quel langage compilé) les coefficients seront congrus à 100 003 à chaque sommation afin de limiter sur taille. Cet entier arbitraire a été choisi pour ne pas trop limiter les coefficients et éviter la trop grand apparitions de coefficients nuls, qui n'ont que très peu d'impact sur le polynôme et donc sur le haché final.

I.3. Points clés de sécurité de la fonction de hachage

Cette partie traite de la justification des choix faits dans la Formalisation mathématique de la fonction d'un point de vu tout à fait théorique, en amont de toute considération pratique d'une implémentation effective, garantissant les points clés de robustesse de la fonction.

1. **Résistance préimage**

   Cette propriété tient à la capacité d'établir une relation entre le haché et le clair. Une telle relation n'existe pas avec notre fonction de hachage **en l'absence de connaissance de la taille du clair**. Si tel est est toute fois le cas, on obtient en effet n équations non linéaires à $|m| \geqslant n$ inconnues en raisonnant sur les coefficients du polynôme *(inconnues à une congruence modulo 36 près dû à la conversion des coefficients en caractères)*. Une attaque par length extension [1] est cependant *a priori* inenvisageable, etant donné le système non linéaire qu'il faudrait résoudre.

   Remarque *La limite de la taille des coefficients à chaque sommation opérée par la congruence modulo 10 003 (cf remarque 2) suffit à rendre caduque une quelconque recherche de relation entre les lettres du haché et les lettres du clair. Il faudrait en effet, considérer tous les termes de toutes les relations congrus à 0 modulo 100x 003 à chaque sommation.*

   Cependant cette opération n'est présente que pour une raison pratique et la fonction de hachage se veut être robuste à la préimage, sans toute considération purement pratique. Ce gain est toute fois non négligeale et achève la démonstration de la résistance préimage de la fonction.

I.4. Complexité

I.4.a. Complexité temporelle

Propriété 6 (Complexité temporelle de la fonction de hachage)

La complexité temporelle de cet algorithme est en

$$\Theta(|m|^2) \text{ multiplications de polynômes de degré au plus } n-1$$

où $|m|$ désigne la taille du mot initial *(bourré si nécessaire)*.

Remarque La complexité de l'algorithme est trop élevée, car trop dépendante de la taille de mot initial, qui est de taille quelconque. Nous conviendrons cependant que cette complexité est acceptable dans le cadre de notre étude, étant du fait de la construction inspirée du Polynôme de Lagrange qui l'impose. En effet, la résistance aux collisions de notre fonction est entièrement basée sur l'impact de chacun des coefficients sur le polynôme final. Il était donc attendu que la complexité temporelle en soit fortement dépendante.

I.4.b. Complexité spatiale

Propriété 7

La complexité spatiale de cet algorithme est en

$$\Theta(|m|)$$

Remarque En effet nous stockons au plus un polynôme de degré $n-1$.

1. https://en.wikipedia.org/wiki/Length_extension_attack

Taille du message *(nombre de caractère)*	**Temps de hachage** [3]
8 char	5,7 ms
20 char	5,7 ms
100 char	130 ms
200 char	636 ms
500 char	4.8 s
1000 char	20 s
5000 char	8 min 38 s

TABLE 3.1 – Moyenne de temps d'exécutions de l'interpolation Lagrangienne en fonction de la taille du clair et de la taille du haché.

II. Résultats

II.1. La fonction de hachage

La fonction de hachage semble être fonctionnelle et répondre à nos exigences premières théoriques. Nous proposons dans cette partie d'en étudier les résultats pratiques, venant confirmer ou infirmer nos estimations.

1. **Effet avalanche**
   L'effet avalanche, au coeur de la résistance aux collisions, semble être respecté. Par exemple [2], pour 256 bits on a

   ```
   Lagrange.hash("The quick brown fox jumps over the lazy dog") =
                   x8d7ajb16h0o4rdca2bbss9kniuotk6z

   Lagrange.hash("The quick brown fox jumps over the lazy dog.") =
                   v0jg3vp94x8qnxjqmzt8wwkonea12kb8

   Lagrange.hash("") = u63vej0nfy85t0e996kjq2y6ygyhyezx
   ```

   Nous observons de plus que le bourrage semble est fonctionnel en l'absence de redondance manifeste de caractère et de corrélation entre les différents hachés. Nous mènerons une étude plus poussée de cette propriété lors de l'attaque du haché lagrangien par l'algorithme Rho de Pollard.

2. **Temps d'exécution**

   La complexité asymptotique de notre fonction est suffisante dans le cadre de notre étude. L'attaque par Rho de Pollard n'est en effet réalisable, sur une machine standard [4], que pour des haché n'excédent pas 12 caractères en sortie *(en a fortiori 12 caractères en entrée)*. La vitesse de hachage pour un tel haché - avant notre fonction - est de

   655 hachés/s

   C'est suffisant pour notre étude.

   Remarque À titre de comparaison le MD5 issu du module hashlib de python a une vitesse de 600 000 hachés/s. La différence est tout à fait normal. Le MD5 est naturellement implémenté en C qui est un langage compilé et nous savons que notre fonction n'a pas été choisie pour sa rapidité.

2. https://en.wikipedia.org/wiki/The_quick_brown_fox_jumps_over_the_lazy_dog
4. Tous les algorithmes sont exécutés par un processeur Intel Core i5 quadricœur à 1,4 GHz (Turbo Boost jusqu'à 3,9 GHz)

II.2. Attaque du haché lagrangien par l'algorithme Rho de Pollard

Afin d'étudier la résistance aux collisions de la fonction de hachage, nous l'avons attaqué avec l'algorithme Rho de Pollard. Cette attaque à 2 objectifs :

— Le premier est de vérifier la bonne uniformité de la fonction. La comparaison des différentes collisions obtenus permet de déterminer s'il existe une corrélation entre deux collisions, avec des caractères communs ou des relations entre les codes ASCII des différents coefficients. Elle permet aussi de localiser les collisions et de remarquer si certains clairs sont plus susceptibles de causer une collision ou non.

— Le second est d'étudier la résistance propre de la fonction à cette attaque, et la comparer avec d'autres fonctions déjà attaquées, telles que le MD5.

II.2.a. Collisions trouvées

Ci-après se trouve pour différentes tailles de haché, les collisions trouvées à l'aide de l'algorithme Rho de Pollard.

Clair 1	Clair 2	Haché commun	Nombre de hachés calculés	Temps de calcul
8m09e9	3vly7k	iepa24	50 385	147.49s
fzkp5d	74v5e2	b6llh4	41 587	130.55s
bvg69b	mbdcop	xb6nzv	49 940	148.17s
fs9scy	vk6has	bvqgun	55 926	162.54s
5fs8na	hb08s3	q13boe	42 502	133.18s

TABLE 3.2 – Liste de 5 collisions trouvées pour la fonction de hachage lagrangienne pour un haché de **30 bits** (6 caractères)

En moyenne, il faut calculer **48 068 hachés** durant **144.4 s** pour trouver une collision de 30 bits.

Clair 1	Clair 2	Haché commun	Nombre de hachés calculés	Temps de calcul
rniyr0m	z72m0kg	gjw3rc6	46 6841	1 428.8 s
ab6fja7	ecd5jh5	xmmrvnh	59 791	191.87 s
6ylmn99	v0t3kxc	moqx9ww	303 681	1 040.6 s
6p3w6n1	0mckuf3	a27cemc	362 256	1 170.4 s

TABLE 3.3 – Liste de 4 collisions trouvées pour la fonction de hachage lagrangienne pour un haché de **35 bits** (7 caractères)

En moyenne, il faut calculer **299 217 hachés** durant **974.7 s** pour trouver une collision de 35 bits.

On tire de ces données qu'il ne semble pas y avoir d'abérrations récurrente au sein du hachage lagrangien. Au contraire, il n'y a pas de corrélation directe, ni évidente entre les empreintes et les collisions. Il ne semble pas y avoir de plus de courts cycles au sein du hachage, puisqu'il faut calculer pas moins de 300 000 hachés pour trouver une collision de 35 bits.

II.2.b. Comparaison avec le MD5

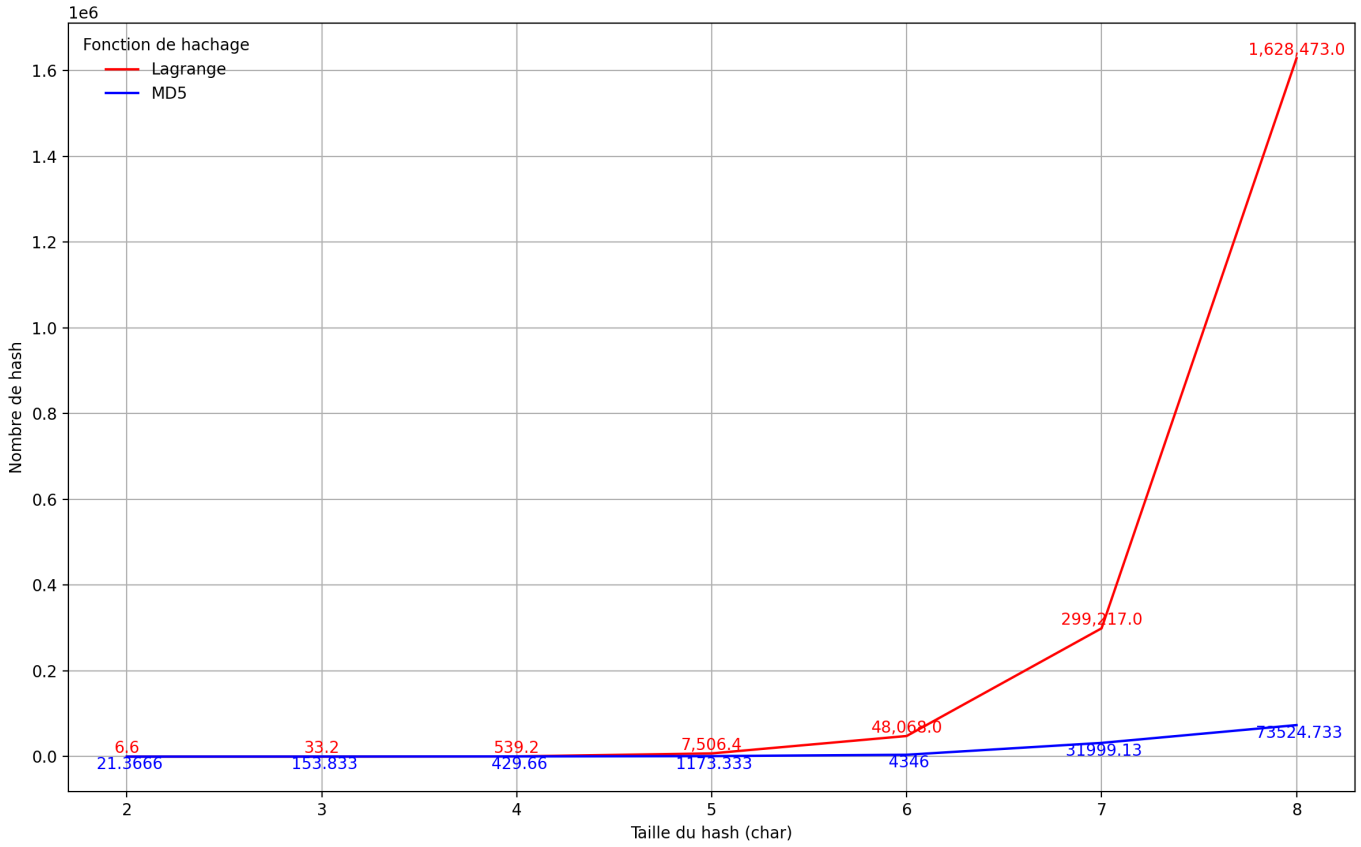

FIGURE 3.1 – Comparaison de la puissance de calcul nécessaire à la génération d'une collision de 2 à 8 caractères, entre le MD5 et le hachage Lagrangien

Comparé au MD5, à taille d'empreinte fixé, la résistance à l'algorithme Rho de Pollard, donc la résistance aux collisions, de la fonction de hachage basée sur l'interpolation lagrangienne semble être plus robuste pour que générer une collision de 8 caractère sur la fonction lagrangienne nécessite le calcul de près de 1 600 000 empreintes, tandis, qu'une collision de même taille sur MD5 ne coûte que le calcul de 73 000 empreintes. **La nouvelle fonction de hachage est donc au moins plus robuste que MD5 aux collisions.**

III. Algorithme de hachage

III.1. Pseudo-code

```
1   S = input() # Texte à hacher saisi par l'utilisateur
2   B = text_to_ASCII(S) # Encodage du texte en ASCII
3
4   Si length(B) < n alors
5
6       padding(B) # Bourrage d'après la méthode inspirée de Merkle-Damgard
7
8   Fin si
9
10  polynom = 0 # Pôlynome final
11  lastProd = 1 # On stock le dernier produit effectué pour minimiser les calculs
12
```

```
    Pour i=0 jusque length(B)-1 faire
        lastProd = (lastProd*(X-B[i]))

        polynom = (polynom + B[i]* lastProd) mod X^n # X désigne l'indéterminée

        coefficient(lastProd) = coefficient(lastProd) mod 10 003 # Les coefficents du
        ↪  polynômes sont congrus modulo 10 003
    Fin pour

    encoded_hash = coefficent(polynom)[0:length(polynom)-1] # On retire le coefficent
    ↪  dominant et on stock les coefficents dans un tableau
    hashed = decode(encoded_hash) # Encodage des coefficients sur l'alphabet latin muni
    ↪  des 10 premiers entiers naturels
```

III.2. Code source

5

```python
#
# Created on Mon Jan 25 2021 by STCHEPINSKY Nathan
#
# Copyright (c) 2021 STCHEPINSKY Nathan. All rights reserved.
#

import numpy as np
import sys
import random
import string
import time as t

class Lagrange:
    def __init__(self, length):
        """
            Initialise les objets

            length = taille du hash final
        """
        self.length = length
        self.alphabet =  string.ascii_lowercase + "1234567890"

    def encode(self,string): # Encodage en ascii
        encoded = [ord(c) for c in string]
        return encoded

    def decode(self,tab):
        """
            Decode sur notre alphabet
        """
        decoded = ""
        len_latin = len(self.alphabet)
        for e in tab:
            modulo  = int(e)
            decoded += self.alphabet[modulo%len_latin]

        return decoded

    def padding(self, bits):
```

5. La dernière version du code est disponible surhttps://github.com/DevnathanGithub

```
40          """
41              Padd le binaire avec la méthode de Merkle-Damgard + 1
42          """
43          init_len = len(bits)
44          if init_len < self.length : # on ne padd pas si on a suffisament de caractère.
45              if len(bits) == 0:
46                  bits.append(2)
47              if  len(bits)%(self.length) != 0:
48                  bits.append(2)
49              while len(bits)%(self.length)-1 != 0 :
50                  bits.append(1)
51              bits.append(init_len + 256) # Ce n'est pas une lettre attégnable car elle est au
                ↪  dessus du plus haut code ascii obtenable
52          return bits
53
54
55      def hash_lagrange(self,encoded_string):
56          """
57              Calcul le polynôme interpolateur de Lagrange
58          """
59          poly = np.poly1d(0)
60          max_degree = np.poly1d([1] + [0]*(self.length+1)) # X^n pour le modulo
61          if encoded_string == "":
62              raise TypeError("FATAL ERROR : [hash_lagrange func] Le mot à interpoler est vide")
63          last = np.poly1d(1)
64          for i in range(len(encoded_string)):
65              last = np.polymul(last, np.poly1d([1,-encoded_string[i]]))
66              poly = np.polyadd((encoded_string[i])*last, poly)
67              (_,poly) = np.polydiv(poly, max_degree) # on récupère le modulo
68              last = np.poly1d(np.mod(last,100003)) # Pour ne pas manipuler de coefficient trop
                ↪  grand On prend un nombre premier pour ne jamais avoir de modulo 0 qui est fatal.
69          return np.array(poly,dtype = int)[1:]# on a un polynôme unitaire de degré self.length,
            ↪  donc self.length + 1 coefficients
70
71      def hash(self,clear):
72          """
73              Fonction principale, appelant les différentes fonctions construisant l'algorithme
74          """
75          encoded = self.encode(clear) # On encode sur l'alphabet latin
76          padded  = self.padding(encoded) # On padd si besoin
77          hashed = self.hash_lagrange(padded)
78          decoded = self.decode(hashed)
79          return decoded
```

Chapitre 4

Conclusion

En définitive, ce document à permis de montrer qu'il est possible de créer une fonction de hachage dont la robustesse aux collisions est optimisée. À cet égard, le polynôme interpolateur de Lagrange fournit nombre de qualités autant intérressantes pour le hachage que fondamentales pour une telle robustesse. Toutefois ce gain en résistance aux collisions s'accompagne d'une perte non négligeable en perfomance du hachage, avec une complexité temporelle bien trop dépendante en la taille du message à hacher ce qui réduite fortement son champ d'action.

Chapitre 5

Bibliographie

[1] **Efficient Selective-ID Secure Identity-Based Encryption Without Random Oracles** -Dan Boneh & Xavier Boyen - *Computer Science DepartmentStanford University, USA*.
↪ *Définitions et propriétés cryptographiques fondamentales des fonctions de hachage. Insiste sur l'aspect cryptographique de telles fonctions, de leurs enjeux et leurs objectifs. Papier très abstrait.*
https://link.springer.com/chapter/10.1007%2F978-3-540-24676-3_14

[2] **Cryptographic Hash-Function Basics : Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance,and Collision Resistance** - Phillip Rogaway & homas Shrimpton - *University of California, USA*.
↪ *Définitions (et formalisation) des propriétés fondamentales des fonctions de hachage cryptographiques (cf. Introduction)*
https://www.iacr.org/archive/fse2004/30170373/30170373.pdf

[3] **The first collision for full SHA-1** - Marc Stevens & Elie Bursztein & Pierre Karpman & Ange Albertini & Yarik Markov - *CWI Amsterdam & Google Research*.
↪ *Première attaque par collision de SHA-1 et explication de l'attaque par usurpation de signature. Voir l'infographie et le papier associé.*
https://shattered.it/

[4] **Lecture Notes on Cryptography** - Shafi Goldwasser & Mihir Bellare - *2008*.
↪ *Définitions et propriétés des fonctions cryptographiques. Voir en particulier la partie* `I.2 Modern Encryption: A Computational Complexity Based Theory`. *Papier s'attardant particulèrement sur ce qui est considéré comme "pratiquement impossible", avec leur preuve probabiliste (applicqué à la factorisation et au problème du logarithme discret par exemple).*
https://cseweb.ucsd.edu/~mihir/papers/gb.pdf

[5] **Factorisation et logarithme discret : Méthodes exponentielles** - *Aix-Marseille Université*.
↪ *Application de Rho de Pollard à la factorisation et définition de l'algorithme de Floyd*
http://iml.univ-mrs.fr/~kohel/tch/M2-TAN/CM/Pollard_rho.pdf

[6] **Entretiens par visio-conférences** - Cécile Pierrot- *Chercheuse en cryptographie à l'INRIA*.
↪ *Plusieurs entretiens et partages de ressources concernant les pré-requis des fonctions de hachage cryptographiques et concernant la recherche de collisions basée à l'aide de l'algorithme Rho de Pollard.*
https://members.loria.fr/CPierrot/

[7] **Polynômes d'interpolation de Lagrange** - Jean-Louis Rouget - *2007*.
↪ *Définition et propriétés du polynôme interpolateur de Lagrange*
https://www.maths-france.fr/MathSpe/GrandsClassiquesDeConcours/Polynomes/PolynomesLagrange.pdf