Rho de Pollard

présenté
comme exigence partielle
du master CSI

par
theo bruyat
vladimir sarde
anass boussadean

avril 2022

Table des matières

1 Introduction et Définitions

Les recours à des solutions cryptographiques pour protéger les échanges et informations transitant électroniquement se sont largement généralisés. Ils permettent d'assurer la confidentialité, l'authenticité et l'intégrité d'un message. Dans ce document il sera question d'un des problèmes mathématiques sur lequel est basé différents protocoles de cryptographie : le problème du logarithme discret (PLD). Avant tout chose, prenons le temps de bien définir ce problème. Comme son nom l'indique il consiste à calculer un logarithme mais contrairement au cas réel habituel, le logarithme discret s'effectue dans un groupe avec des valeurs entières.

Définition 1.1. Soient $(G, .)$ un groupe et $a,\ b \in G$. Un entier k tel que $b^k = a$ défini un *logarithme discret* de a en base b. Nous écrirons $k = log_b(a)$.

Cette définition de logarithme se veut être la plus générale possible, mais en pratique le problème du logarithme discret se pose dans un cadre un peu plus restreint.

Définition 1.2. Soient G un groupe fini cyclique d'ordre q, généré par $g \in G$. Pour tout $h \in G$, le *problème du logarithme discret* sur G consiste à trouver le plus petit entier positif x tel que $g^x = h$. L'entier x est alors appelé le *logarithme discret de h en base g*. Nous le noterons $log_g(h)$.

L'intérêt de ce problème consiste dans le fait que le calcul d'un tel logarithme est particulièrement compliqué si le groupe est suffisamment grand. Cela garantit la protection des informations. Mais d'un autre côté sa fonction réciproque, l'exponentiation, ce calcul en un temps logarithme. Cela permet d'avoir un chiffrement et un déchiffrement rapide.

Ce problème mathématique date du début du XXème siècle, bien avant ces d'applications cryptographiques. Il apparaît pour la première fois dans ce domaine lors de l'étude des LFSR (de l'anglais linear feedback-shift registers). Mais le PLD prend toute son importance lors de la publication du protocole de Diffie-Hellman en 1976 ([DH76]). Ce protocole, basé sur le PLD, permet notamment d'échanger des clés privées (pour une cryptographie symétrique) de façon sécurisée. Il est encore aujourd'hui grandement utilisé. Depuis, d'autres protocoles basés sur le PLD ont été développé et sont encore utilisés. Nous pouvons par exemple citer le cryptosystème de ElGamal ([ElG85]) ou des systèmes de signatures digitales ([FP94]).

Néanmoins, notons que la difficulté de ce problème n'est pas démontrée. Notre confiance pour les méthodes s'appuyant sur le PLD provient du fait qu'il s'agit d'un problème étudié depuis longtemps et qu'encore aujourd'hui, aucune méthode de calcul véritablement performante ne semble avoir été trouvée.

Dans ce document nous nous concentrerons sur l'algorithme de Rho de Pollard qui permet résoudre le PLD de façon générale. Dans un premier temps nous étudierons le fonctionnement de

cet algorithme et sa complexité attendue. Puis nous introduirons la méthode dite des r-adding-walk qui permet de réduire le nombre d'itération en améliorant la génération d'aléatoire de l'algorithme. Suite à cela nous nous intéresserons à la méthode des points distingués qui permet notamment de paralléliser l'algorithme. Enfin, nous analyserons la méthode proposée par l'article [CHK08] permettant d'optimiser le temps des calculs utilisés dans cet algorithme.

Pour chacune de ces méthodes nous proposerons une implémentation concrète de l'algorithme que nous joindrons en annexe. L'algorithme de Rho de Pollard est généraliste, il permet de résoudre le PLD dans un groupe quelconque. Néanmoins, afin de rendre plus concrète son utilisation, nos implémentions se concentreront sur des groupes d'entiers de la forme $\mathbb{Z}/q\mathbb{Z}$. Nous effectuerons également une batterie de test pour chacune des implémentations afin d'observer l'efficacité de chaque optimisation présentée. Précisons aussi que ces implémentations seront codées en C pour obtenir des résultats plus fins.

2 Rho de Pollard

2.1 L'algorithme

Étudions le principe de l'algorithme de base de Rho de Pollard. Cet algorithme a été conçu par John M. Pollard en 1978. Il reprend le fonctionnement d'un premier algorithme dit Rho de Pollard aussi (du même mathématicien), qui, lui, servait à factoriser des entiers.

Commençons par réduire le contexte de recherche. Comme nous l'avions évoqué dans l'introduction nous travaillerons tout le long de ce document avec une groupe fini cyclique G, d'ordre q, généré par $g \in G$. Mais, les principales applications, et donc aussi notre implémentation, se font dans un cadre plus restreint. En pratique, nous considérons un grand premier p et le corps $\mathbb{F}_p$. Le groupe G correspond alors à un sous groupe multiplicatif d'ordre q de $\mathbb{F}_p$. De plus, nous rajoutons la condition que q soit aussi premier. Cette dernière condition est nécessaire au fonctionnement de l'algorithme (car une étape demande d'inverser modulo q). Or nous pouvons toujours nous ramener à un tel cas en utilisant l'algorithme de Pohlig-Hellman (voir les détails dans [PH78]).

L'algorithme de Pohlig-Hellman introduit une façon de calculer le logarithme discret dans un corps de cardinal $q = \prod_{i=1} p_i^{\alpha_i}$ en se ramenant à des PLD dans des sous-groupes d'ordre p_i. Plus précisément, l'algorithme découpe d'abord le problème dans des sous-groupes d'ordres $p_i^{\alpha_i}$ (en calculant des exponentiations). Puis, dans chaque sous groupe, sépare le calcul du logarithme discret en α_i calcul de logarithme discret dans un sous-groupe d'ordre p_i. Donc si nous notons c_q la complexité de résolution du PLD dans $\mathbb{F}_q$ alors la complexité est seulement $c_q = \mathcal{O}(\sum_i \alpha_i(log(q) + c_{p_i}))$. Cela explique pourquoi dans les applications un q premier est privilégié.

Maintenant que nous avons posé le contexte regardons véritablement l'algorithme de Rho de Pollard pour calculer $log_g(h)$. L'idée principale consiste à trouver une égalité de la forme $g^a h^b = g^{a'} h^{b'}$.

Proposition 2.1. *Avec les objets précédemment introduit, si $g^a h^b = g^{a'} h^{b'}$ avec $b \neq b'$, alors $log_g(h) = (b - b')^{-1}(a' - a) \ mod \ q$.*

Démonstration. Notons $log_g(h) = x$. Donc par définition $g^x = h$. Nous avons que :

$$\begin{aligned} & g^a h^b = g^{a'} h^{b'} \\ \implies & g^a (g^x)^b = g^{a'} (g^x)^{b'} \\ \implies & a + xb = a' + xb' \bmod q \\ \implies & x(b - b') = a' - a \bmod q \\ \implies & x = (a' - a)(b - b')^{-1} \bmod q \end{aligned}$$

Nous pouvons remarquer que $(b - b)^{-1}$ est bien inversible mod q, car q est premier et $b \neq b'$.

□

Cette proposition explicite donc pourquoi trouver une telle égalité permet de résoudre le PLD. Pour trouver cette égalité l'algorithme partitionne d'abord le groupe G en trois ensembles S_1, S_2, et S_3 de taille (environ) égale. Nous pouvons faire la partition de façon plutôt libre, une des seules règles est que $1 \notin S_2$. En pratique, la partition est donc faite avec une propriété facilement testable comme le reste modulo 3 par exemple. C'est donc de cette façon que nous l'avons implémenté, en suivant les règles : $x \in S_1 \iff x \equiv 1 \bmod 3$, $x \in S_2 \iff x \equiv 0 \bmod 3$, et $x \in S_3 \iff x \equiv 2 \bmod 3$.

Cette partition nous servira à définir une suite $(x_i)_{i \in \mathbb{N}}$ à valeur dans G, avec $x_0 = 1$ et telle que :

$$x_{i+1} = f(x_i) =: \begin{cases} hx_i, \text{ si } x_i \in S_1 \\ x_i^2, \text{ si } x_i \in S_2 \\ gx_i, \text{ si } x_i \in S_3 \end{cases} \tag{1}$$

Le but de cette fonction f est de simuler un aléa afin que la suite x_i parcours le groupe de façon aléatoire. L'idée étant de trouver deux éléments de la suite telle que $x_i = x_j$, $i \neq j$. En effet, les x_i sont construit comme des puissances de g et de h, donc une telle égalité donnera bien une égalité de la forme $g^a h^b = g^{a'} h^{b'}$. Pour pouvoir déduire le logarithme de notre algorithme il faut donc suivre comment évolue la décomposition de x_i sur g et h. Notons $x_i = g^{a_i} h^{b_i}$. Cela définie deux suites d'entiers $(a_i)_{i \in \mathbb{N}}$ et $(b_i)_{i \in \mathbb{N}}$ satisfaisant $a_0 = 0$, $b_0 = 0$ et $\forall i \geq 0$:

$$a_{i+1} = \begin{cases} a_i, \text{ si } x_i \in S_1 \\ 2a_i \bmod q, \text{ si } x_i \in S_2 \\ a_i + 1 \bmod q, \text{ si } x_i \in S_3 \end{cases} \tag{2}$$

$$b_{i+1} =: \begin{cases} b_i + 1, \text{ si } x_i \in S_1 \\ 2b_i \bmod q, \text{ si } x_i \in S_2 \\ b_i \bmod q, \text{ si } x_i \in S_3 \end{cases} \tag{3}$$

Nous avons donc une suite qui parcours notre groupe et dont nous connaissons la décomposition de chaque élément en puissances de g et de h. Le dernier problème auquel il faut répondre consiste à détecter l'égalité entre deux éléments du groupe. La solution naïve serait de stocker toute la suite et de comparer à chaque fois, mais cela ne serait pas du tout efficace (en terme de comparaisons et de mémoire). Donc l'algorithme original penche pour une autre méthode : l'algorithme du lièvre et de la tortue (ou de détection de cycle de Floyd). Cette méthode repose sur la proposition suivante.

Proposition 2.2. *Soit une fonction $f : G \to G$ avec G un ensemble fini de cardinal q. Définissons la suite $(a_i)_{i\in\mathbb{N}}$ tel que $a_0 \in S$ et $a_{i+1} = f(a_i)$. Alors il existe un indice m tel que $a_m = a_{2m}$*

Démonstration. Nous savons que G est fini, donc il existe un cycle (voir figure 1), ie il existe un λ tel que $\forall n \geq \lambda$, $a_{n+\mu} = a_n$.

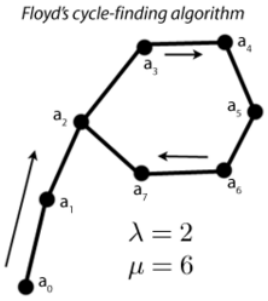

FIGURE 1 – Cycle

Posons $m = \mu(1 + \lfloor \lambda/\mu \rfloor)$. Nous avons que $m > \lambda$, de plus $2m - m = m = \mu(1 + \lfloor \lambda/\mu \rfloor)$. Donc la différence d'indice est bien un multiple de la longueur du cycle, ainsi $a_{2m} = a_m$. Il est également évident par construction qu'il s'agit de première fois où nous avons cette égalité.

Remarquons aussi que $\lambda < m \leq \lambda + \mu$. □

Ainsi pour détecter une égalité $x_i = x_j$, avec $i \neq j$, nous calculerons en pratique deux suites x_i, et x_{2i}. Nous stockerons alors à chaque étape uniquement le dernier couple (x_i, x_{2i}) en vérifiant si $x_i = x_{2i}$.

Remarquons au passage qu'il faut au moins $\lambda + \mu$ itérations pour obtenir une collision entre deux éléments de la suite. Donc, avec cette méthode de détection il faudra moins du double d'itérations (pour calculer x_m mais aussi x_2m), ainsi la complexité de détection est la même que celle

pour obtenir une collision. Cette méthode a donc le gros avantage de ne pas utiliser de mémoire car seul le couple (x_m, x_2m) est stocké. C'est notamment ce point qui rend l'algorithme de Rho de Pollard très intéressant en pratique par rapport à des algorithme du type Baby Step, Giant Step.

Au final l'algorithme original est celui qui suit. Vous pouvez le retrouvez notamment dans [MVOV18] (p.106) qui offre également des explications sur l'algorithme.

Algorithme 1 Rho de Pollard

Entrée: g générateur du groupe cyclique G d'ordre premier q, et un élément $h \in G$.
Sortie: Le logarithme discret $x = log_g(h)$

Prenons $x_0 \leftarrow 0$, $a_0 \leftarrow 0$, $b_0 \leftarrow 0$
Pour $i = 1, 2, ...$ **faire** :
    Calculer x_i, a_i, b_i grâce aux valeurs de x_{i-1}, a_{i-1}, b_{i-1} via les équations (1), (2), (3).
    De même calculer x_{2i}, a_{2i}, b_{2i} grâce aux valeurs de x_{2i-2}, a_{2i-2}, b_{2i-2}.
    Si $x_i = x_{2i}$ **alors**
        $r \leftarrow b_i - b_{2i}$
        Si r $= 0$ **alors**
            Renvoyer "Echec"
        Sinon :
            $x = r^{-1}(a_{2i} - a_i) \bmod q$
            Renvoyer x

Remarque : Cet algorithme échoue dans le cas où $b_i = b_{2i}$. Si cela arrive nous pouvons juste relancer l'algorithme avec une graine différente : $x_0 = g^{a_0}h^{b_0}$ où $a_0, b_0 \in [1, n-1] \subset \mathbb{N}$.

De plus cette situation est marginale. En effet, reprenons la démonstration de la proposition 2.1 nous avons que l'égalité des éléments de la suite est équivalente à : $a + xb = a' + xb' \bmod q$. Or si nous isolons b' nous obtenons $b' = x^{-1}(a - a') + b \bmod q$. Cette équation à une solution $b' \in G$ pour tout $a, a', b \in G$ et $b' = b$ si est seulement si $a' = a$. Donc si nous posons a, b (ie x_i fixé) il y a q choix pour a' (tel qu'il respecte l'équation) et le choix de a' fixe b'. Ainsi au final la probabilité que $b = b'$ est de $1/q$. En réalité le fait que l'algorithme itère un tirage à chaque itération va favoriser les couples où a' et b' sont proche. Mais malgré cela, les probabilités d'échec reste très faible.

2.2 Complexité

Attardons nous maintenant un peu sur la complexité de l'algorithme Rho de Pollard. Les comparaisons et l'étape finale sont négligeable. La partie longue de l'algorithme correspond au temps nécessaire pour itérer la fonction f avant d'obtenir une collision. Á chaque itération nous devons effectuer une multiplication dans $\mathbb{F}_p$.

Proposition 2.3. *Supposons que la fonction f définie par l'équation (1) se comporte comme une fonction aléatoire. Alors la complexité algébrique (dans $\mathbb{F}_p$) pour l'algorithme de Rho de Pollard est*

$\mathcal{O}(\sqrt{q})$. *Plus précisément, le nombre d'itération attendu est proche de* $\sqrt{\pi q/2} \approx 1.253\sqrt{q}$.

Nous nous prouverons pas cette proposition car sa démonstration, bien qu'accessible, nous éloignerai de notre sujet. Elle repose sur le paradoxe des anniversaires. Vous pourrez trouver tous les détails dans le début du chapitre 14 de [Gal12].

De plus, ce même livre comprend également une analyse plus poussée de la complexité attendue en utilisant cette fonction f et ce mode de détection des collisions (voir Heuristic 14.2.10). Elle est égale à $(3.093 + o(1))\sqrt{q}$ (opérations dans $\mathbb{F}_p$). De plus, l'algorithme utilise un espace mémoire négligeable.

Cette écart entre la complexité attendue et celle obtenue s'explique principalement par deux points : la fonction f ne simule pas parfaitement l'aléatoire, et la méthode de détection n'est pas optimale. Nous travaillerons donc sur ces deux points dans les deux prochaines parties !

Nous pouvons aussi remarquer que nous n'avons travaillé qu'en complexité algébrique dans $\mathbb{F}_p$. Or une multiplication dans $\mathbb{F}_p$ peut rapidement être coûteuse, notamment lors que p est de taille cryptographique. Ainsi il peut également être intéressant d'essayer de réduire cette complexité. C'est ce qui est étudier dans la partie 5 : Tag Tracing.

3 R-Adding-Walk

3.1 Fonction d'itération et motivation

Une fonction d'itération pour l'algorithme rhô de Pollard est considérer comme bonne si le nombre d'itération avant d'obtenir une collision est proche de $\sqrt{\pi\frac{q}{2}}$: la valeur attendue pour une fonction aléatoire.

En pratique la fonction d'itération de l'algorithme rho de Pollard n'est pas assez aléatoire dans le sens ou il faut en général plus de $\sqrt{\pi\frac{q}{2}}$ itération avant d'obtenir une collision (comme discutée partie 2.2). Ce problème nous pousse à chercher une nouvelle définition de la fonction d'itération, celle-ci devra se comporter le plus possible comme une fonction aléatoire pour pouvoir se rapprocher de la complexité théorique. Cette nouvelle définition devra de plus vérifier certaines contraintes.

Pour cela nous allons avoir besoin d'un définition.

Définition 3.1. Soit G un groupe d'ordre q un nombre premier, fixons g un générateur du groupe et h un élément de G dont on cherche le logarithme discret en base g.

Une fonction f est dite exponant traceable si il est possible d'exprimer la fonction sous la forme : $f(g^n h^m) = g^{f_g(n,m)} h^{f_h(n,m)}$ pour certaine fonction facilement calculable f_g et f_h.

Cette contrainte permet d'extraire le logarithme discret lorsqu'il y a collision non triviale. En effet, si nous avons $g^{n_1}h^{m_1} = g^{n_2}h^{m_2}$ nous avons vu dans la proposition 2.1 qu'il est possible d'extraire le logarithme discret de h.

Nous imposerons a notre fonction d'itération d'être facilement calculable et exponant traceable pour pouvoir rapidement extraire le logarithme discret.

3.2 Première Approche

Plaçons nous dans un groupe G d'ordre q un nombre premier, fixons g un générateur du groupe et h un élément de G dont on cherche le logarithme discret en base g finalement notons $(y_i)_{i\geq 0}$ la suite de nos pas d'itérations.

Nous cherchons à construire une fonction d'itération. Une première idée naïve serait de générer à chaque pas d'itération deux entier $n_i, m_i \mod q$ et de définir notre suite comme suit :

$$\begin{cases} y_0 = 1_G \\ y_{i+1} = y_i g^{n_i} h^{m_i} \end{cases}$$

La fonction est évidement exponent traceable, en effet par récurrence immédiate nous avons $y_i = g^n h^m$ ou $n = \sum_{k\leq n} n_i$ et $m = \sum_{k\leq n} m_i$ de sorte que l'une des contraintes que nous nous sommes fixé est vérifiée.

Le caractère aléatoire de cette fonction d'itération est entièrement déterminé pas la génération des entiers n_i, m_i. En effet, cela repose sur le lemme suivant :

Lemme 3.1. Pour tout $x \in G$, si on note $E_x = \textbf{card}\left\{(n,m) \in {}^{\mathbb{Z}}\!/_{q\mathbb{Z}} \mid g^n h^m = x\right\}$ alors $E_x = q$.

Démonstration. Pour tout $z \in G$ notons e_z l'unique élément de ${}^{\mathbb{Z}}\!/_{q\mathbb{Z}}$ tel que $g^{e_z} = z$.

Soit $n, m \in {}^{\mathbb{Z}}\!/_{q\mathbb{Z}}$ tel que $g^n h^m = x$.

$g^n h^m = x \Leftrightarrow n + e_h m = e_x \Leftrightarrow n = e_x - e_h m$ et $m = m$

Ainsi, Pour tout $x \in G$ il y exactement q couple (n,m) qui vérifient l'égalité $g^n h^m = x$ de sorte que E_x vaut q.

Comme nous venons de le voir, pour tout $x, y \in G$, $E_x = E_y$ de sorte que si l'on tire aléatoirement n, m dans ${}^{\mathbb{Z}}\!/_{q\mathbb{Z}}$ alors les événement $x = g^n h^m$ et $y = g^n h^m$ sont équiprobables.

□

En utilisant des techniques bien connues de génération d'entiers aléatoires nous avons une fonction d'itération avec de bonnes propriétés aléatoires. Malgré tout, générer à chaque étape deux entiers $\bmod q$ puis, calculer $g^{n_i} h^{m_i}$ est en pratique plutôt coûteux de sorte que cette méthode n'est jamais utilisé.

En conclusion, cette première approche ne vérifie pas complètement les contraintes que nous nous sommes fixées, malgré tout, tout n'est pas à jeter comme nous allons le voir dans la définition de r-adding walk.

3.3 Définition de r-adding walk

Définition 3.2. Soient $3 \leq r \leq 100$ un petit entier et soit $(T_i)_{i \leq r-1}$ une partition de G en r ensembles d'environ la même taille.

Pour tout $j \in 0, ..., r-1$ choisissions aléatoirement n_j, m_j deux entier modulo q puis, définissions $M_j = g^{n_j} h^{m_j}$.

Nous définissons un r-adding walk dont la fonction d'itération est notée f_r et la suite d'éléments associés $(y_i)_{i \geq 0}$ comme suit :

$$y_0 = 1_G$$
$$y_{i+1} = f_r(y_i) = y_i M_s \text{ si et seulement si } y_i \in T_s$$

On remarque que cette définition est applicable dans n'importe quel groupe d'ordre un nombre premier.

Pour que cette fonction soit facilement calculable il faut d'une part, étant donnée un élément y_i, savoir rapidement déterminer s tel que $y_i \in T_s$.

Une fois que les M_j et la partition $(T_i)_{i \leq r-1}$ fixée, la fonction est complètement déterministe. En pratique cette définition est un bon compromis entre bonne propriété aléatoire et temps de calcul.

3.4 Différence la définition initiale de Pollard

Soit T_0, T_1, T_2 une partitions de G. La fonction d'itération de rhô de Pollard notée (f_P) est définit comme suit :

$$f_P(y) = \begin{cases} gy \text{ si } y \in T_0 \\ y^2 \text{ si } y \in T_1 \\ hy \text{ si } y \in T_2 \end{cases}$$

Nous remarquons que c'est presque un 3-adding walk à la différence près que lorsque $y_i \in T_1$, pour obtenir y_{i+1} nous multiplions pas y_i par un éléments fixé à l'avance mais nous faisons le carré de y_i pour avoir le prochain élément de notre suite.

3.5 Implémentions et partie pratique

Soient p, q deux nombre premiers.

En pratique nous nous sommes placés dans un sous groupe $G =< g >$ de $\left(\mathbb{Z}/p\mathbb{Z}\right)^\times$ d'ordre q, nous avons fait notre l'implémentation en C et, pour travailler sur de grand entier, nous avons utilisé la bibliothèque gmplib. Pour générer les entier n_i, m_i, qui rappelons, le sont définit modulo q, nous avons utilisé le générateur aléatoire fournit par la bibliothèque gmplib avec pour graine le temps de la machine. Finalement nous avons assez naturellement choisit l'ensemble des entier inférieurs à p comme système de représentant.

Il nous reste à détailler notre choix de partition $(T_i)_{i \leq r-1}$ de G qui rappelons le est tel que les T_i sont d'environ la même taille et pour tout $y \in G$ il est facile de trouver s tel que $y \in T_s$. Pour cela nous avons choisi de définir $s = y \mod r$ comme conseillé dans l'article [CHK08] et nous avons testé différentes valeurs de r pour finalement choisir $r = 20$ durant nos tests.

4 Points Distingués et Parallélisation

4.1 Motivation

L'algorithme initiale rho de Pollard utilise l'algorithme de détection de cycle de floyd pour détecter une collision. L'inconvenant de cette méthode est qu'elle ne permet pas une parallélisation de l'algorithme. C'est pour cette raison que nous sommes poussés à chercher une autre méthode de détection de collision qui cette fois permet une parallélisation efficace de l'algorithme. L'idéal étant que si nous disposons d'une machine ayant n processeur, l'algorithme doit aller n fois plus rapidement.

4.2 Points Distingués détection de collision

Pour pouvoir paralléliser l'algorithme nous allons utiliser une table en mémoire et définir certains points du groupe comme étant distingués puis, à chaque fois qu'un point distingué est le résultat d'un pas d'itération, nous allons le stocker dans la table. Si le point est déjà présent dans la table nous avons notre collision et nous pouvons extraire le logarithme discret.

Pour fixer les idées, notons que en pratique nous définissions un point comme étant distingué si et seulement si son représentant canonique commence par un certain nombre de zéros.

Avec cette méthode nous pouvons avoir n processus qui calculent indépendemment leurs pas d'itération et qui, dès qu'ils tombent sur un point distingué, le mettent dans la table commune ainsi que les exposants associés.

Lorsqu'un processus veut mettre un point dans la table il doit : réserver la ressource puis, mettre le point dans la table et finalement libérer la ressource. Lorsque la ressource n'est pas disponible parce qu'un autre processus est entrain de l'utiliser le processus attend qu'elle se libère. En particulier, si l'on définit trop de points comme étant distingué alors les processus risquent de se bloquer entre eux et nous perdrons en efficacité.

Si nous notons θ la proportion de points distingués dans le groupe G, le nombre de pas d'itération moyen a faire au total avant de pouvoir extraire le logarithme discret sera de $\sqrt{\pi\frac{q}{2}} + \frac{1}{\theta}$. Cela correspond au temps qu'il faut pour obtenir une collision : $\sqrt{\pi\frac{q}{2}}$, plus un surcoût de $\frac{1}{\theta}$ pour que l'algorithme s'en rende compte étant donné que les collisions ne sont détectées que sur des points distingués. En particulier, si le nombre de points distingués est trop faible alors on risque d'avoir un surcoût $\frac{1}{\theta}$ important.

Voici une illustration de ce phénomène. Nous voyons dans cette illustration quatre processus calculer leurs "trail" de points puis, une collision arrivant sur le points x_3 et x'_2 n'est détecter que sur les points distingué x_5 et x'_4 [VOW99].

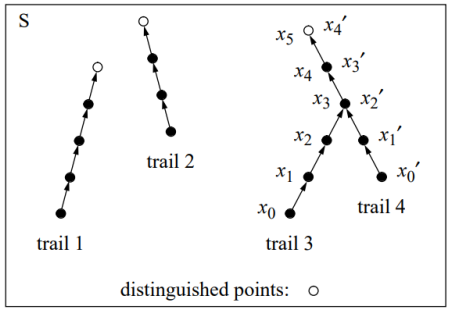

FIGURE 2 – Parallelized Collision Search

4.3 r-adding walk et Parallélisation

La fonction d'itération r-adding walk est particulièrement adaptée à la méthode des points distingués. En effet, au lieu de générer les (M_j) et leurs exposants n_j, m_j pour chaque processus nous pouvons se contenter de les générer une seul fois et mettre cette ressource en commun à tous les processus, comme aucune modification des (M_j) n'est faite durant les pas d'itérations, les processus peuvent lire les (M_j) simultanément (sans avoir besoin de réserver la ressource). Finalement pour que deux processus ne calculent pas une suite de points identiques il suffit que chaque processus commence son r-adding walk avec un points initial y_0 diffèrent.

4.4 Implémentions et partie pratique

Soient p, q deux nombres premiers.

En pratique nous nous sommes placés dans un sous groupe $G =< g >$ de $\left(\mathbb{Z}/p\mathbb{Z}\right)^{\times}$ d'ordre q, nous avons fait notre l'implémentation en C et, pour travailler sur de grands entiers, nous avons utilisé la bibliothèque gmplib. Pour générer les entier n_i, m_i qui, rappelons le, sont définit modulo q, nous avons utilisé le générateur aléatoire fournit par la bibliothèque gmplib avec pour graine le temps de la machine. Finalement nous avons assez naturellement choisit l'ensemble des entier inférieur à p comme système de représentant.

Dans cette section nous détaillons les choix fait pour la définition de points distingués et la table

.

Pour la table il nous fallait une structure de données ayant une complexité efficace pour l'insertion et la recherche d'un élément. Notre choix s'est porté sur les arbres AVL qui sont des arbres de recherche équilibrés. L'insertion et la recherche dans cette structure de données ont pour complexité $\mathcal{O}(\log n)$ ou n est le nombre d'éléments dans la table. Nous aurions pu utiliser une table de hachage qui en théorie a une meilleur complexité pour les opérations qui nous intéressent ou encore un autre type d'arbre équilibré. Notre choix a principalement été motivé par la difficulté d'implantation et notre familiarité avec ces structures de données.

Nous avons choisit l'ensemble des points distingués comme étant l'ensemble des points dont le représentant commence avec un certain nombre de zéros ; cette propriété étant facilement vérifiable. En pratique cette propriété se comporte bien, en effet, de façon empirique nous remarquons que si l'on impose aux points distingués de commencer avec n zéros et qu'en conséquence il y a m points distingué alors, imposer aux points distingués de commencer avec $n + 1$ zéros implique qu'il y a $\frac{m}{2}$ points distingués. Cette dernière observation nous indique que les points distingués sont bien repartis dans le groupe. Cela nous permet de moduler facilement le nombre d'éléments ajoutés dans la table durant l'exécution de l'algorithme. En effet, notons θ la proportion de points distingués dans le groupe lorsque nous définissons un points distingué comme étant un points dont le représentant commence avec n zéros. Nous savons que l'algorithme va détecter une collision après en moyenne

$\sqrt{\pi\frac{q}{2}}+\frac{1}{\theta}$ itérations, comme les points distingués sont bien répartis dans le groupe on peut s'attendre à avoir dans la table $\theta\sqrt{\pi\frac{q}{2}}+1$: le nombre de points distingués parmi les itérations (en moyenne).

5 Tag Tracing

Nos premières parties résument les principales optimisations classiques de la méthode Rho de Pollard. Mais pour aller plus loin et accélérer encore cette approche nous allons étudier la stratégie présenté dans l'article [CHK08] : *Tag Tracing*. Cette optimisation vise à réduire le temps de chaque itération. Dans cette partie nous étudierons donc d'abord l'idée et la stratégie de cette optimisation. Nous regarderons également quelles sont les modifications à mettre en place pour adapter notre algorithme à ce mode de calcul. Puis nous donnerons une description plus précise et mathématiques de la construction proposée. Enfin, quelques résultats sur les temps de calculs serons analysés à la fin de cette partie.

5.1 Idée et premières définitions

Considérons l'algorithme Rho de Pollard implémenté avec une r-adding-Walk et parallélisé avec la méthode des points distingués. Nous gardons les mêmes notations que précédemment : $< g >= G \subset \mathbb{F}_p$, $log_g(h) = x$. De plus nous notons $S = \{0, 1, ..., r-1\}$ l'ensemble d'indices et $M = \{M_s = g^{m_s}h^{n_s}\}$ l'ensemble des multiplicateurs associés à ces indices. Nous noterons également $s : G \to S$ la fonction qui à un élément du groupe $g \in G$ lui associe son indice dans la r-adding-walk. Nous redéfinirons cette dernière différemment que précédemment.

Rappelons qu'à chaque itération de Rho de Pollard, la fonction s détermine $s(g_i) =: s_i$ l'indice de l'élément obtenu à la précédente itération, puis la fonction f calcul $g_{i+1} = f(g_i) = g_i M_{s_i}$. S'il s'agit d'un point distingué, g_{i+1} est alors stocké dans la table.

L'idée de tag tracing est d'éviter de calculer la multiplication $f(g_i) = g_i M_{s_i}$ à chaque étape. Il s'agit en effet de l'opération la plus coûteuse dans l'exécution de Rho de Pollard car les entiers manipulés sont de taille cryptographiques. Pour cela, la méthode proposée consiste à calculer uniquement l'indice du prochain élément pour continuer les itérations. Un élément est alors calculé entièrement uniquement s'il a une chance d'être un point distingué. Regardons de plus près comment cela peut être réalisé.

Définition 5.1. Fixons un entier $l \in \mathbb{N}$. Nous définissons l'ensemble M_l par $M_l =: (M \cup \{1\})^l$. Cet ensemble est constitué de tous les produits d'au plus l éléments de M.

Cette table sera pré-calculé au début de l'algorithme. L'idée derrière ce pré-calcul est que ces produits apparaîtront souvent lors des itérations de f. Ainsi plutôt que de répéter plusieurs fois les

mêmes multiplications, nous préférons prendre un peu de temps et d'espace au début de l'algorithme pour stocker tous ces résultats.

Bien sûr selon la taille de l et r le temps de calcul et l'espace de cette table peuvent rapidement exploser. Vous pouvez vous référer aux lemmes 1 et 2 de [CHK08] pour plus de détails sur la complexité de calcul de cette table. Les auteurs y décrivent également une méthode de calcul efficace de la table dont nous nous sommes servi lors de l'implémentation de notre algorithme utilisant tag tracing.

Notons également que les produits sont stockés avec leurs exposants dans la table afin de conserver la propriété d'exponant traceable de f.

Maintenant que notre table est définie, nous pouvons vraiment nous intéresser à la méthode d'optimisation en elle même.

Définition 5.2. Posons T un ensemble, dit de *Tag*, et définissons trois fonctions :

$$\tau : G \to T$$
$$\tilde{\tau} : G \times M_l \to T \cup \{\text{échec}\}$$
$$\sigma : T \to S = \{0, 1, ..., r-1\}$$

Nous nommerons τ la fonction de tag.

Enfin nous définissons la fonction d'indice $s : G \to S$ comme étant $s = \sigma \circ \tau$ et la fonction d'indice pour le produit par $\tilde{s} = \sigma \circ \tilde{\tau} : G \times M_l \to S \times \{\text{échec}\}$. Ces fonctions sont choisies telle que :

1. La fonction d'indice s est surjective et les ensembles de pré-images pour chaque indice possèdent des cardinaux environ égaux.

2. Lorsque $\tilde{s}(g, M) \in S$ nous avons que $\tilde{s}(g, M) = s(g \times M)$.

Considérons pour l'instant ces objets de façon théorique, sans leurs implémentations concrètes. Intuitivement l'ensemble T est un ensemble intermédiaire pour le calcul de l'indice d'un élément. Lors d'une itération normal nous utilisons la fonction s qui ramène d'abord l'entrée dans T puis ensuite calcul son indice. Ce que nous souhaitons faire dans cette partie est définir une fonction $\tilde{\tau}$ qui permet de calculer, dans la plupart des cas, l'indice d'un élément multiplié par un coefficient.

Par définition $\tilde{\tau}$ nous permettrait d'éviter d'effectuer la multiplication. En effet considérons un élément g_i, au lieu de calculer $g_{i+1} = g_i M_{s_i}$ nous pouvons calculer :

$$\tilde{s}(g_i, M_{s_i}) = s(g_i \times M_{s_i}) = s(g_{i+1}) = s_{i+2}$$

Donc $g_{i+2} = g_i \times M_{s_i} \times M_{s_{i+1}}$. Á nouveau nous ne recalculons pas la multiplication et utilisons $\tilde{s}$ pour calculer s_{i+3}. Cela est possible car $M_{s_i} \times M_{s_{i+1}} \in M_l$. Nous itérons ce processus k fois. Nous avons alors $g_{i+k} = g_i \times M_{s_i} \times ... \times M_{s_{i+k}}$.

Lorsque finalement $\tilde{s}$ échoue, ou que $k = l$ nous calculons complètement g_{i+k}. Cela ne prend qu'une multiplication car le produit $M_{s_i} \times ... \times M_{s_{i+k}}$ est déjà pré-calculé dans la table M_l (car $k \leq l$).

Bien sûr nous espérons que le temps de calcul de $\tilde{\tau}(g, M)$ soit plus court que celui nécessaire au calcul de $g \times M$. L'idée est qu'intuitivement calculer le produit en entier nous donne toutes les informations sur l'élément suivant, mais qu'il est possible de déterminer l'indice avec seulement une partie de ces informations via l'intermédiaire de $\tilde{\tau}$.

Comme nous l'avons mentionné, nous utiliserons ici aussi la méthode des points distingués pour repérer une collision. Pour définir un point distingué nous partons du même principe que celui décrit dans la partie 4 : un certain nombre des bits de poids fort sont à zéro. Mais nous ajoutons à cela une condition supplémentaire, que nous décrirons plus en détails après, afin qu'un point g ne puisse être distingué que si $\tilde{\tau}(g', M') = \textit{échec}$, $\forall g', M'$ tel que $g = g'M'$.

Ainsi seul les points déjà calculés entièrement peuvent être distingués. En effet, si nous arrivons sur un point distingué c'est que $\tilde{\tau}$ a renvoyé *échec* juste avant. Donc nous n'avons pas à calculer de produits supplémentaire.

5.2 Tag Tracing dans $\mathbb{F}_p$

Pour l'instant, nous sommes restés très général dans nos définitions. Ainsi nous avons écrit une stratégie qui peut s'appliquer au calcul du logarithme discret dans n'importe quel groupe. Intéressons nous maintenant à l'implémentation concrète de cette méthode dans notre cas $G =< g > \subset \mathbb{F}_p^\times$ où G est d'ordre q.

Avant toute chose nous allons avoir besoin d'introduire plusieurs variables. L'ensemble d'indices S sera de cardinal r et reprenons M_l avec l et r choisis afin de rendre le temps calcul de M_l acceptable. Définissons l'ensemble $T =: \{0, 1, .., t-1\}$ avec $t = r \times b$ (un multiple de r). Prenons ϵ un entier positif et posons $d = \lceil log_\epsilon p \rceil$. Choisissons un entier $\omega' \geq d(\epsilon - 1) + 1$. Enfin nous noterons $\omega = t\omega'$ et supposerons que $\omega < p^{1/3}$.

Le choix optimal de ces paramètres dépend de nombreux facteurs comme la taille des entiers p et q ou la puissance de la machine sur laquelle tourne l'algorithme. L'article [CHK08] propose des choix possibles. Ce sont avec ces propositions que notre algorithme est implémenté. La figure 3 (provenant également de l'article) reprend les tailles conseillées. Elle permet de mieux comprendre d'où proviennent les résultats de cette section.

La suite de cette partie est relativement technique. Nous ne reviendrons donc pas sur tous les détails mathématiques ici. En effet, l'article [CHK08] contient déjà des démonstrations complètes et précises. Ainsi, dans ce rapport, nous rappellerons les principales définitions et résultats mais

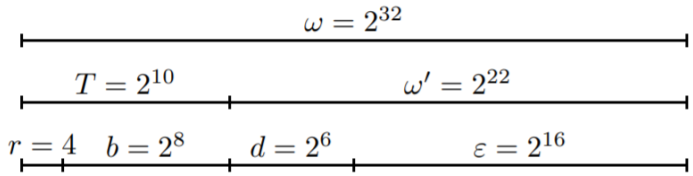

FIGURE 3 – Paramètres

nous nous contenterons de présenter leur aspect intuitif. Le but étant de comprendre l'idée derrière ces résultats, tout en ayant la possibilité de se référer à l'article pour les détails techniques.

Définition 5.3. Soit un entier $B > p^{2/3}$ tel que $0 \leq \omega B - p \leq B^{1/2}$. Par exemple $B = \lceil p/\omega \rceil$. La fonction tag $\tau : G \to T$ peut être définie par :

$$\tau(g) = \left\lfloor \frac{g \bmod p}{\omega' B} \right\rfloor$$

De plus nous pouvons définir $\sigma : T \to S$ par $\sigma(x) = \lfloor x/b \rfloor$.

Ceci nous donne également s car $s = \sigma \circ \tau$.

Proposition 5.1. *Supposons que les éléments de G soient uniformément distribués sur $\mathbb{F}_p$. Nous avons que la fonction d'indice s est environ de pré-image uniforme.*

Cette propriété découle du choix effectué pour la taille de l'entier B.

Nous pouvons toujours supposer que les éléments de G sont uniformément distribué. Donc cette proposition nous assure que la fonction s telle que définie ici respecte bien la première conditions que nous avions énoncé lors de sa définition théorique (5.2).

Nous allons maintenant définir la fonction auxiliaire $\tilde{\tau} : G \times M_l \to T \cup \{\text{échec}\}$. Nous passerons d'abord par une fonction intermédiaire $\tilde{\tilde{\tau}}$.

Soient $x, y \in \mathbb{F}_p$, nous pouvons toujours écrire (écriture en base ϵ) :

$$x = \sum_{i=0}^{d-1} x_i \epsilon^i \quad (0 \leq x_i < \epsilon)$$

et, $\forall\ 0 \leq i \leq d-1$ nous pouvons écrire (division euclidienne) :

$$\epsilon^i y \bmod p = \hat{y}_i B + \check{y}_i \quad (0 \leq \hat{y}_i < \frac{p-1}{B},\ 0 \leq \check{y} < B)$$

Définition 5.4. En utilisant les notations ci dessus nous pouvons définir :

$$\tilde{\tilde{\tau}}(x,y) = \left\lfloor \frac{\sum_{i=0}^{d-1} x_i \hat{y}_i \bmod \omega}{\omega'} \right\rfloor$$

.

Le but de cette fonction est d'approcher τ lorsque qu'elle est évaluée en un produit. Le prochain résultat nous confirme que c'est effectivement le cas.

Lemme 5.1. Soient $x, y \in \mathbb{F}_p$, nous avons que $\tau(xy) = \tilde{\tilde{\tau}}(x,y)$ ou $\tilde{\tilde{\tau}}(x,y)+1$, sauf si $\tilde{\tilde{\tau}}(x,y) = t-1$.

Ce lemme est démontré rigoureusement dans l'article [CHK08] (lemme 4). Mais essayons d'approcher intuitivement pourquoi cela fonctionne.

Démonstration. L'idée est d'approcher $\tau(xy)$ en ignorant les morceaux qui ne contribue que peu.

$xy = (\sum_{i=0}^{d-1} x_i \hat{y}_i)B + \sum_{i=0}^{d-1} x_i \check{y}_i \bmod p$

Comme la fonction τ divise par B nous pouvons retirer le B du premier terme. Nous pouvons également ignorer le deuxième terme car le choix des tailles des paramètres nous permet de nous assurer qu'il ne contribuera presque pas au résultat final.

De plus xy est considéré mod p. Mais comme nous divisons par B et que $\omega \times B \approx p$ nous considérons le résultat mod ω.

Enfin nous divisons par ω' dans τ et $\tilde{\tilde{\tau}}$ donc cela n'influe pas.

Au final nous avons bien une bonne approximation de $\tau(xy)$.

Si $\tilde{\tilde{\tau}} = t-1$ alors le résultat peut être mal défini. Il s'agit d'un problème de modulo non réduit. De plus, nous voulons que ce cas soit considéré comme un échec comme car il pose problème pour le calcul d'indice. □

Remarquons également que le calcul de $\tilde{\tilde{\tau}}$ va être bien plus rapide que celui de τ comme espéré. En effet, au lieu de calculer un produit mod p, nous cassons notre problème en plusieurs multiplications. Nous éliminons les morceaux inutiles et réalisons le reste des multiplications mod ω, or $\omega << p$.

Nous pouvons maintenant définir $\tilde{\tau}$.

Définition 5.5. Nous définissons $\tilde{\tau} : G \times M_l \to T \cup \{\text{échec}\}$ comme ceci :

$$\tilde{\tau}(g,M) = \begin{cases} \text{échec} & \text{si } \tilde{\tilde{\tau}}(g,M) \bmod b \text{ est soit } b-1 \text{ ou } b-2 \\ \tilde{\tilde{\tau}}(g,M) & \text{sinon} \end{cases}$$

Proposition 5.2. *Lorsque $\tilde{\tau}(g,M) \in T$ et donc $\tilde{s}(g,M) \in S$, nous avons que $s(gM) = \tilde{s}(g,M)$ et $\tau(gM) \bmod b \neq b-1$.*

Cette proposition découle directement de la définition et du lemme précédent (voir détails dans [CHK08]). L'idée derrière est que nous connaissons τ à un près (car la définition nous assure que $\tilde{\tilde{\tau}} \neq T-1$). Ici nous divisons par b donc le seul moment où deux cas se présentent, c'est lorsque $\tilde{\tilde{\tau}} = b-1 \bmod b$. Or la définition de $\tilde{\tau}$ empêche cela. Ainsi $\tilde{s}$ calcul bien l'indice du produit (il s'agissait de la deuxième condition imposée dans la définition 5.2).

De plus nous refusons également les cas où $\tilde{\tilde{\tau}} = b-2 \bmod b$ afin de s'assurer que $\tau \neq b-1 \bmod b$. Cela nous permet de poser la définition de points distingués suivante.

Définition 5.6. Dans cette partie, nous définissons les points distingués comme étant les points $g \in G$ tel que $\tau(g) = b-1 \bmod b$, et g possède un certain nombre de bits de poids fort à zéro.

Nous pouvons remarquer que la dernière proposition nous permet bien de vérifier que les points distingués sont nécessairement des cas où la fonction $\tilde{\tau}$ avait échoué. C'est ce que nous espérions précédemment. Ainsi seul les points déjà calculés entièrement peuvent être distingués.

5.3 Temps de calcul

Nous avons déjà discuté (après le lemme 5.1) de pourquoi intuitivement le calcul de $\tilde{\tilde{\tau}}$ et donc de $\tilde{\tau}$ est plus rapide que le calcul du produit entier. De plus, une étude complète de la complexité du programme est offerte dans la partie 5 de l'article. Nous ne rentrerons donc pas plus dans les détails de ce genre de calcul ici.

La conclusion de ces calculs est qu'il est possible de remplacer la multiplication modulo p, normalement nécessaire à une itération, par une opération de complexité linéaire (en la taille de p).

Nous pouvons quand même remarquer que pour arriver à ce résultat, une modification est effectuée par rapport à la méthode présentée. Dans ce que nous avons décrit, les paramètres étaient fixées au début. Or il est possible d'optimiser encore les calculs en utilisant plusieurs jeux de paramètres pour essayer de maximiser le nombre de calcul fait avec le ω le plus petit possible (car les multiplications de $\tilde{\tilde{\tau}}$ sont faites modulo ω). Cette méthode consiste à essayer de calculer un indice avec tag tracing et un ω petit. Puis si cela renvoie un échec alors nous essayons de calculer un indice avec un ω légèrement plus grand, et ainsi de suite. Le produit n'est calculé entièrement que si nous avons obtenu un échec pour chaque jeu de paramètre. Cette méthode nécessite plusieurs jeux de paramètres ainsi que plusieurs tables mais repose fondamentalement sur ce qui a été décrit au dessus.

L'article propose également des comparatifs concrets des temps obtenus avec tag tracing dans la partie 4.5. Nous pouvons y voir que leur implémentation permet de multiplier par 11 la vitesse d'exécution de l'algorithme par rapport à celui utilisant uniquement une r-adding-walk et les points distingués (méthode présentée dans nos parties 3 et 4). De plus, une itération en elle même peut être en moyenne jusqu'à 15 fois plus rapide avec la méthode de tag tracing.

6 Résultat

Dans un premier temps, pour avoir des jeux de test cohérents et aléatoires, nous avons codé une fonction en python permettant de nous donner g, p et q. Pour ce faire, notre fonction prend en entrée le nombre de bits de q et le nombre de bits de p afin que tant que p n'est pas un nombre premier, nous prenons un q aléatoire de sa taille ainsi qu'un a de taille taille de p - taille de q. Enfin nous donnons à p la valeur de qa+1. Pour g, nous calculons b^a mod p avec $b = 2$ pour commencer. Ensuite, tant que g vaut 1 nous incrémentons b et nous recalculons h. Enfin, nous retournons nos trois variables.

Maintenant que nous avons notre générateur de paramètres aléatoires, nous allons pouvoir tester nos fonctions avec ceux-ci. Nous avons fait varier la taille de q afin d'observer la robustesse de nos programmes dans la limite de temps raisonnable. Nous testons chaque paramètre 5 fois sur chaque programme. Tous nos programmes seront codés en C en utilisant la bibliothèque GMP pour les grands ensembles et disponibles en annexe.

Pour commencer, attardons nous sur le premier programme : Rho de Pollard

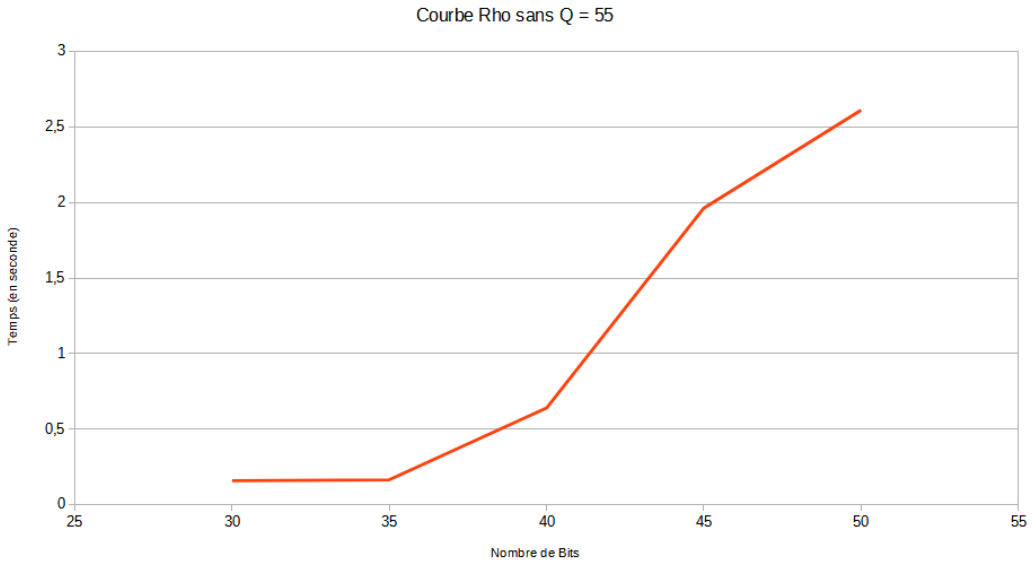

FIGURE 4 – Courbe des temps de Rho

Nous pouvons observer une croissance des temps plutôt linéaire tant que le nombre de bits de q ne dépasse pas 50. Mais lorsque nous passons à 55 bits nous obtenons le graphique 5.

L'exécution du programme qui se faisait en quelques secondes se fait maintenant en plusieurs minutes en moyenne. C'est lorsque nous passons à 55 bits que nous voyons la vraie courbe qui est

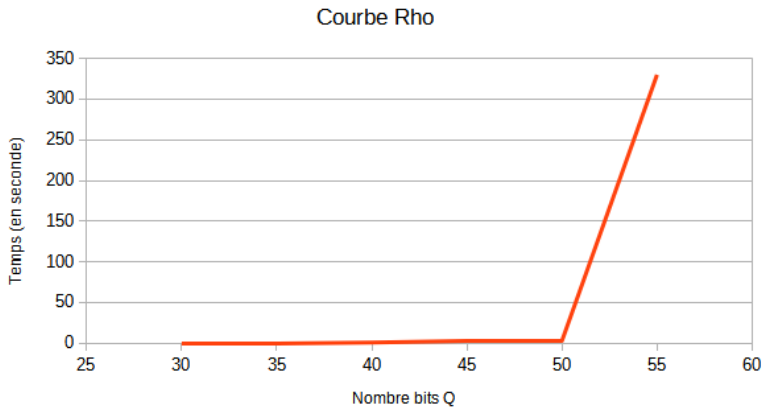

FIGURE 5 – Courbe des temps de Rho

bien exponentielle.

Remarque : Nos graphiques sont produit avec les moyennes des temps obtenus. Ces résultats sont néanmoins significatifs car les écarts types sont suffisant petit comme l'illustre le diagramme 6.

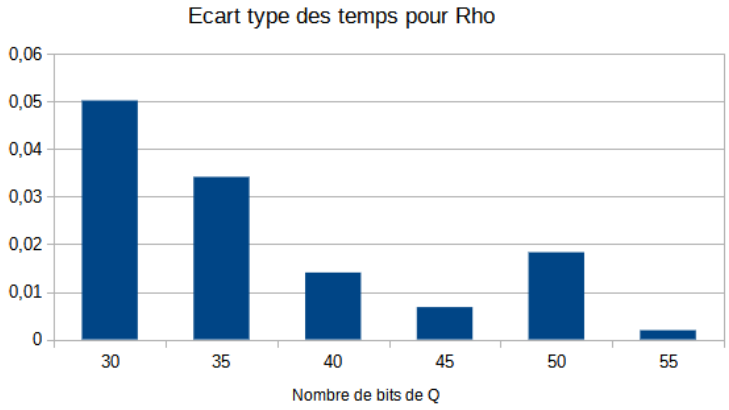

FIGURE 6 – Diagramme de l'écart type divisé par le temps d'exécution totale en fonction de la taille de q.

Ensuite, nous allons nous intéresser à rAddingWalk. Le graphique 7 représente les temps de calculs en fonction du nombre de bits de q pour le programme utilisant r-adding walk.

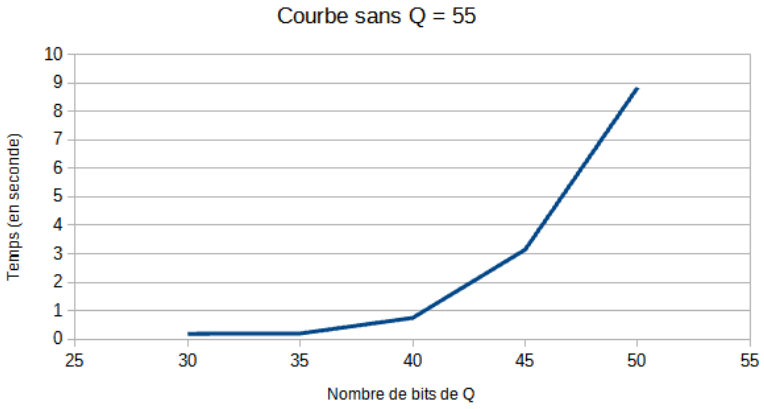

FIGURE 7 – Courbe des temps de rAddingWalk

Comme précédemment, nous pouvons voir le temps augmenter avec l'augmentation de la taille de q. L'exécution ne prend que quelques secondes jusqu'à $q = 50$ bits.

Mais comme pour Rho de Pollard, lorsque nous augmentons à 55 bits, nous voyons la courbe exponentielle et nous passons à plusieurs minutes voir dizaines de minutes comme l'illustre le graphique 8.

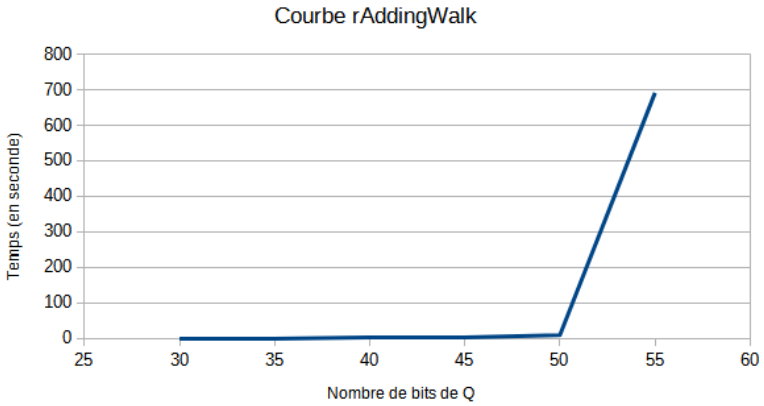

FIGURE 8 – Courbe des temps de rAddingWalk

Enfin, nous avons optimisé Rho de Pollard afin de pouvoir le paralléliser avec la méthode des points distingués. Nous avons exécuté du programme sur les 20 coeurs de la machine et stocké les points distingués dans un tableau commun. Nous allons maintenant étudier les temps réels puis les temps utilisateurs des tests réalisés. Les différents résultats obtenus sont représenté par les graphiques 9, 10, 11, et 12.

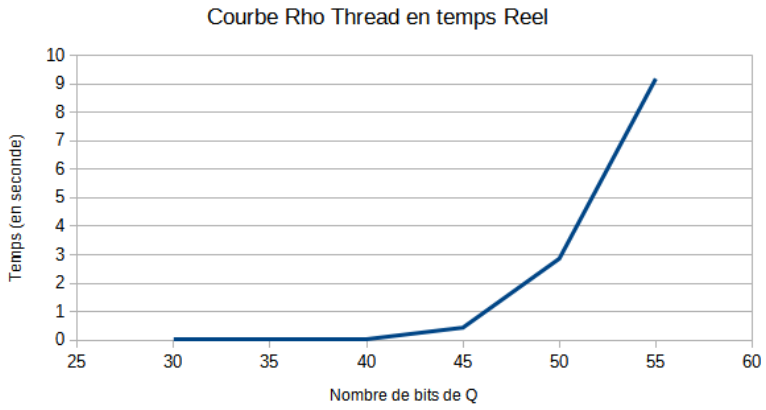

FIGURE 9 – Courbe des temps de Thread en temps réel

De même que pour les autres courbes, nous avons ici une augmentation jusqu'à 50 bit, puis celle ci s'accélère en 55. Mais cela reste toujours sous l'ordre des secondes. Lorsque nous augmentons à 60 et plus, nous obtenons le graphique 10.

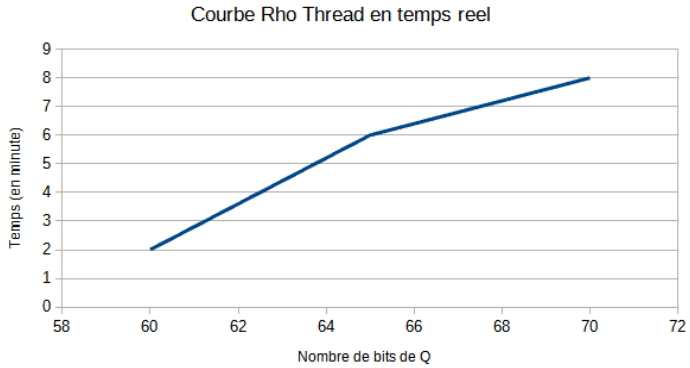

FIGURE 10 – Courbe des temps de Thread en temps réel

Les temps augmentent toujours mais de manière beaucoup moins forte. À partir de 60 bits, nous passons sous l'ordre des minutes mais cela reste encore calculable en temps raisonnable.

Mais si nous regardons le temps utilisateur, qui est le temps qu'aurait du mettre le programme si nous le lancions que sur un seul coeur, nous pouvons voir que cela aurait pris une toute autre mesure (graphique 11).

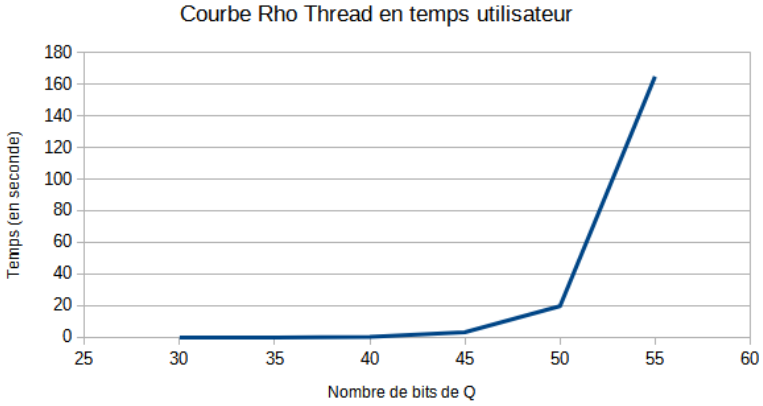

FIGURE 11 – Courbe des temps de Thread en temps utilisateur

D'abord nous voyons toujours la même courbe qui augmente d'un coup à 55 bits pour environ 2min30. Mais dès que nous dépassons cela nous obtenons des temps beaucoup plus grands. Voir graphique 12.

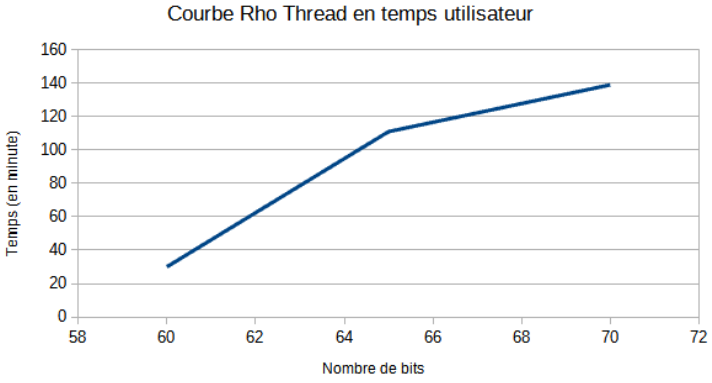

FIGURE 12 – Courbe des temps de Thread des points distingués en temps utilisateur

Ici, nous sommes à, en moyenne, 140 min soit 2h30 d'exécution du programme sur un seul coeur. Nous en déduisons donc que paralléliser son programme est nécessaire lorsque nous voulons travailler avec de très grands entiers.

Intéressons nous enfin au nombre de points distingués stockés dans la table. Nous utiliserons pour ce test 24 coeurs avec les mêmes paramètres. Seul le nombre de points distingués va évoluer. Cela est représenté par le graphique 13.

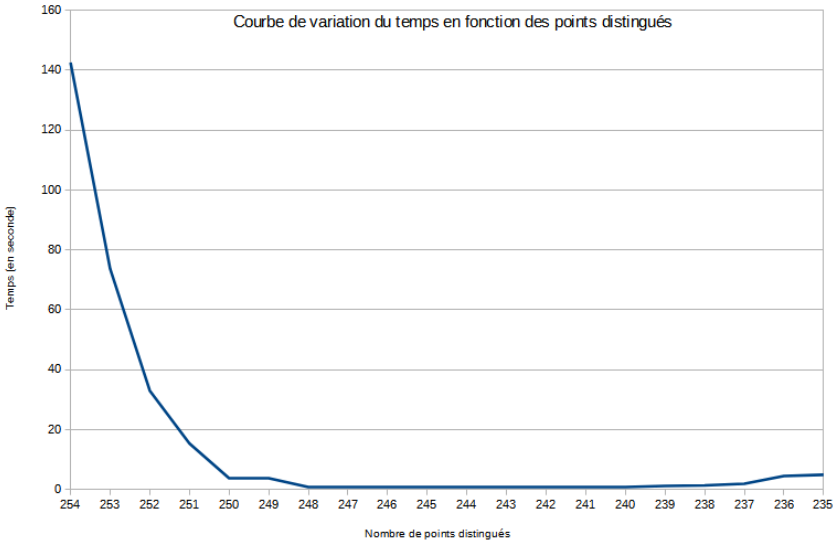

FIGURE 13 – Courbe de variations du temps en fonctions du nombres des points distingués

Nous avons du tester plusieurs paramètres afin de voir quel nombre de points distingués était le plus optimal. En effet, si nous en stockons trop, cela prend de la place en mémoire et les threads se bloquent entre elles en essayant de remplir la même table de points distingués. Mais si nous n'en stockons pas assez, nous ne trouvons pas de collision.

Le graphique 13 représente ces tests pour un q de taille 48 bits. Nous y observons en effet une courbe qui a un minimum en temps et qui remonte si nous stockons trop ou pas assez de points. Dans cet exemple, nous pouvons observer que le plus optimal serait de prendre une valeur comprise entre 242 et 247. Dans ce graphique, nous avions utilisé un p de 256 bits et la valeur 245 signifie par exemple que nous stockons tous les points dont les 11 premiers bits sont à zéro.

Intéressons nous maintenant aux résultats de l'implémentation de Tag Tracing (disponible en annexe). Nous ne possédons malheureusement pas de tests aussi complets sur cet algorithme. En

effet, celui-ci repose sur énormément de paramètres différents et nous n'avons pas réussi à extraire des données suffisamment marquantes dans le temps qui nous était imparti. Néanmoins nous avons tout de même obtenu des résultats significatifs avec les paramètres conseillés dans l'article [CHK08].

Nous avons notamment comparé le temps moyen d'une itération de Tag Tracing à celui d'une itération de l'algorithme Points Distingués. Nous avons obtenu qu'en moyenne l'itération de Tag Tracing se faisait 4 fois plus rapidement (avec les options de compilation optimisés, ie -o2).

Dans sa globalité, notre algorithme utilise un temps équivalent à celui de points distingués. Cela s'explique par le fait que Tag Tracing réalise en moyenne plus d'itérations dans nos tests. Cela provient probablement de la définition de points distingués. En optimisant les conditions pour être un points distingués, Tag Tracing devrait retrouver un nombre d'itérations comparable.

Remarquons également que ces résultats sont moins impressionnant que ceux présentés dans [CHK08]. Il y a, à priori, deux principales raisons à cela. Nous avons déjà évoqué la première : il est compliqué de trouver un jeu de paramètres parfaitement optimisé pour une machine et des tailles d'entiers données. De plus, dans la dernière partie de l'article, les auteurs proposent une nouvelle optimisation : une approche incrémentale. Nous l'avons décrit dans la partie 5.3 de ce rapport. Cette optimisation, rend relativement compliquée l'implémentation, car elle suppose de stocker plusieurs tables, plusieurs jeux de paramètres, etc .. Mais, elle permet d'accélérer encore l'algorithme en rendant le calcul de $\tilde{\tau}$ de complexité linéaire.

Remarque : Pour implémenter Tag Tracing nous avons utilisé une table construite de la même façon que dans l'article. De plus, nos grands entiers sont manipulés via la bibliothèque gmp. Nous avons exploité cette implémentation pour tenter d'accélérer le calcul de $\tilde{\tilde{\tau}}$ car les fonctions classiques de gmp n'étaient pas assez performantes.

7 Conclusion

Tout au long de ce rapport, nous nous sommes concentré sur l'algorithme de Rho de Pollard de sa forme original à ses formes les plus optimisées. Nous avons essayé de résumer à chaque fois le fonctionnement de l'algorithme et les idées derrières chaque optimisations présentées. Chacune d'entre elles visaient à optimiser un aspect différent de l'algorithme. La première, r-adding-walk, améliore l'aléa de notre fonction afin de réduire le nombre d'itérations. La deuxième, les points distingués, permet la parallélisation des calculs. La troisième, tag tracing, réduit le temps de calcul de chaque itération en remplaçant la multiplication nécessaire à l'itération par une opération de complexité linéaire.

Pour chacune de ces optimisations, une implémentation est proposée en annexe. Les résultats de ces algorithmes ont été étudié dans notre partie 6. Ils permettent de souligner l'efficacité de toutes

ces améliorations. L'article [CHK08] propose également un comparatif (dans leur partie 4.5) entre la première version de Rho de Pollard et la dernière (incluant tag tracing). Celui-ci révèle un temps de calcul diviser par un facteur 14 en moyenne. Il s'agit donc bien d'optimisations conséquentes ayant un impact concret sur les temps d'utilisations.

Malgré cela, la méthode Rho de Pollard reste d'une complexité $\mathcal{O}(\sqrt{p})$. Elle ne permet donc pas de casser le PLD lorsque p est de taille cryptographique. Cela est aussi illustré par les temps obtenus lors de nos tests. En effet, les optimisations ne font que repousser le moment où la taille de p fait exploser les temps de calculs.

Pour des raisons pratiques nous nous sommes restreint dans ce travail à l'étude de Rho de Pollard dans des sous-groupes cycliques du corps $\mathbb{F}_p$. Nous pouvons noter que, pour ce type de groupe, il existe des algorithmes plus efficaces. Le meilleur actuellement, pour les corps finis de caractéristique première très grande, est celui du crible algébrique. Il est de complexité sous exponentielle.

Néanmoins, le grand avantage de l'algorithme Rho de Pollard est qu'il est généralisable à tout les groupes cycliques. En effet, les algorithmes plus performant se limitent à l'étude des corps de nombres. Ainsi il reste très intéressant d'optimiser ce genre d'algorithmes généralistes. Remarquons également que toutes les optimisations présentées sont elles aussi généralisable à l'étude du logarithme dans n'importe quel groupe cyclique. Seule la dernière, tag tracing, demande un travail supplémentaire pour être implémentée dans un groupe précis. Les auteurs de l'article évoquent d'ailleurs leur souhait de travailler à généraliser cette méthode pour qu'elle soit implémentable dans le cadre des courbes elliptiques ou hyperelliptiques.

Références

[CHK08] Jung Hee Cheon, Jin Hong, and Minkyu Kim. Speeding up the pollard rho method on prime fields. In International Conference on the Theory and Application of Cryptology and Information Security, pages 471–488. Springer, 2008.

[DH76] WHITFIELD DIFFIE and MARTIN E HELLMAN. New directions in cryptography. IEEE TRANSACTIONS ON INFORMATION THEORY, 22(6), 1976.

[ElG85] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. IEEE transactions on information theory, 31(4) :469–472, 1985.

[FP94] NIST FIPS-PUB. 186. Digital Signature Standard (DSS). Retrieved [May 2015] from csrc. nist. gov/publications/fips/archive/fips186-2/fips186-2. pdf, 1994.

[Gal12] Steven D Galbraith. Mathematics of public key cryptography. Cambridge University Press, 2012.

[MVOV18] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. Handbook of applied cryptography. CRC press, 2018.

[PH78] Stephen Pohlig and Martin Hellman. An improved algorithm for computing logarithms over gf (p) and its cryptographic significance (corresp.). IEEE Transactions on information Theory, 24(1) :106–110, 1978.

[VOW99] Paul C Van Oorschot and Michael J Wiener. Parallel collision search with cryptanalytic applications. Journal of cryptology, 12(1) :1–28, 1999.

A Implémentation du Générateur d'Exemples

```
1
2  def generate_exemple(size1,size2):
3      p=1
4      k=size2-size1
5      g=2
6      while not is_prime(p):
7          q=random_prime((1<<size1)-1,lbound=(1<<(size1-1)))
8          a=randint(1<<(k),1<<(k+1)-1)
9          p=q*a+1
10     while True:
11         h=power_mod(g,a,p)
12         if h != 1:
13             break
14         g=g+1
15     return p,q,h
```

B Implémentation de Rho de Pollard

```c
// Pour compiler : gcc -g -Wall TER.c -lgmp

#include <gmp.h>
#include <stdio.h>
#include <assert.h>
#include <stdbool.h>
#include <time.h>
#include <stdlib.h>
#include <math.h>

bool is_generated = false;

void f_roh(mpz_t *Y, mpz_t *A, mpz_t *B, mpz_t *g, mpz_t *h, mpz_t *n, mpz_t *p)
{

    unsigned long int s;
    s = mpz_fdiv_ui(*Y, 3);
    if (s == 1)
    {
        mpz_add_ui(*B, *B, 1);
        mpz_mod(*B, *B, *n); // exposant tracing

        mpz_mul(*Y, *Y, *h); // on fait  i t  r   Y

        mpz_mod(*Y, *Y, *p);
    }
    if (s == 0)
    {
        mpz_mul_ui(*A, *A, 2);
        mpz_mod(*A, *A, *n);

        mpz_mul_ui(*B, *B, 2);
        mpz_mod(*B, *B, *n);

        mpz_mul(*Y, *Y, *Y);
        mpz_mod(*Y, *Y, *p);
    }
    if (s == 2)
    {
        mpz_add_ui(*A, *A, 1);
        mpz_mod(*A, *A, *n);

        mpz_mul(*Y, *Y, *g);
        mpz_mod(*Y, *Y, *p);
    }
}



int main(int argc, char *argv[])
{
    char temp_g[1000];
    char temp_h[1000];
    char temp_p[1000];
    char temp_n[1000];

    FILE *input = fopen("input.txt", "r+");
```

```
58
59      if (input == NULL)
60      {
61          printf("file is NULL\n");
62          exit(EXIT_FAILURE);
63      }
64
65      fscanf(input, "%s %s %s %s", &temp_g, &temp_h, &temp_p, &temp_n);
66
67
68      mpz_t g; // g nerateur G
69      mpz_t h;
70      mpz_t p;
71      mpz_t n; // ordre groupe G
72
73      mpz_init(g);
74      mpz_set_str(g, temp_g, 10);
75
76      mpz_init(h); // on initialise les entiers a 0
77      mpz_set_str(h, temp_h, 10);
78      mpz_init(p);
79      mpz_set_str(p, temp_p, 10);
80      mpz_init(n);
81      mpz_set_str(n, temp_n, 10);
82      // fin de l'initialisation des entiers
83
84      // debut de la declaration des variables
85      mpz_t Y_EVEN, Y; // variables iteratives
86      mpz_t A;
87      mpz_t A_EVEN;
88      mpz_t B;
89      mpz_t B_EVEN;
90      mpz_t r;
91
92      mpz_init(Y); // la variable qu'on it re
93      mpz_set_ui(Y, 1);
94      mpz_init(Y_EVEN);       // pour les it rations 2i
95      mpz_set_ui(Y_EVEN, 1); // initialisation des entiers a 0
96      mpz_init(A);            // exposant
97      mpz_set_ui(A, 0);
98      mpz_init(A_EVEN);
99      mpz_set_ui(A_EVEN, 0);
100     mpz_init(B); // exposant
101     mpz_set_ui(B, 0);
102     mpz_init(B_EVEN);
103     mpz_set_ui(B_EVEN, 0);
104     mpz_init(r);
105     mpz_set_ui(r, 0);
106
107     while (1)
108     {
109         f_roh(&Y, &A, &B, &g, &h, &n, &p);
110         f_roh(&Y_EVEN, &A_EVEN, &B_EVEN, &g, &h, &n, &p);
111         f_roh(&Y_EVEN, &A_EVEN, &B_EVEN, &g, &h, &n, &p);
112
113         if (mpz_cmp(Y, Y_EVEN) == 0)
114         {
115
116             mpz_sub(r, B, B_EVEN);
```

```
            // r = (B - B_EVEN);

            mpz_mod(r, r, n);
            // r = r % n;

            if (mpz_cmp_si(r, 0) == 0)
            {
                return -1;
            }
            mpz_sub(A_EVEN, A_EVEN, A);
            mpz_invert(r, r, n);
            mpz_mul(r, r, A_EVEN);
            mpz_mod(r, r, n);

            printf("r = ");
            mpz_out_str(stdout, 10, r);
            printf("\n");
            return 1;
        }
    }

    mpz_clear(g);
    mpz_clear(h);
    mpz_clear(p); // liberation de l'espace
    mpz_clear(Y);
    mpz_clear(Y_EVEN);
    mpz_clear(A_EVEN);
    mpz_clear(A);
    mpz_clear(B);
    mpz_clear(B_EVEN);
    mpz_clear(n);
    free(list_exp);
    free(list_M);
    fclose(input);
    return 0;
}
```

C Implémentation des r-adding walk

```c
// Pour compiler : gcc -g -Wall TER.c -lgmp

#include <assert.h>
#include <gmp.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 20

#define INIT(name, value) \
  mpz_t name;             \
  mpz_init(name);         \
  mpz_set_str(name, value, 10);

#define PRINT(var)              \
  mpz_out_str(stdout, 10, var); \
  printf("\n");

typedef struct
{
  mpz_t m; /*exposant de  g*/
  mpz_t n; /*exposant de  h*/
} couple;

void f(mpz_t *y, mpz_t *a, mpz_t *b, mpz_t *g, mpz_t *h, mpz_t *q, mpz_t *p,
       couple *listExposant, mpz_t *listM)
{
  /* on g n re une partition de l'ensemble en N cas */
  unsigned long int s = mpz_fdiv_ui(*y, N);
  mpz_add(*b, *b, listExposant[s].n);
  mpz_mod(*b, *b, *q);
  mpz_add(*a, *a, listExposant[s].m);
  mpz_mod(*a, *a, *q);
  mpz_mul(*y, *y, listM[s]);
  mpz_mod(*y, *y, *p);
}

int main()
{
  char temp_g[1000];
  char temp_h[1000];
  char temp_p[1000];
  char temp_q[1000];

  FILE *input = fopen("input.txt", "r+");

  if (input == NULL)
  {
    printf("file is NULL\n");
    exit(EXIT_FAILURE);
  }

  fscanf(input, "%s %s %s %s", &temp_g, &temp_h, &temp_p, &temp_q);

  mpz_t g; // g nerateur G
```

```
58  mpz_t h;
59  mpz_t p;
60  mpz_t q; // ordre groupe G
61  mpz_init(g);
62  mpz_set_str(g, temp_g, 10);
63
64  mpz_init(h); // on initialise les entiers a 0
65  mpz_set_str(h, temp_h, 10);
66  mpz_init(p);
67  mpz_set_str(p, temp_p, 10);
68  mpz_init(q);
69  mpz_set_str(q, temp_q, 10);
70
71  // debut de la declaration des variables
72  INIT(y, "1")
73  INIT(y_even, "1")
74  INIT(a, "0") /* exposant de g */
75  INIT(a_even, "0")
76  INIT(b, "0") /* exposant de h */
77  INIT(b_even, "0")
78  INIT(r, "0") /* r for result */
79
80  /* generation des ms et ns de fa on al atoire */
81  gmp_randstate_t state;
82  gmp_randinit_mt(state);
83  gmp_randseed_ui(state, time(NULL));
84  couple listExposant[N]; /* from 0 to N-1: c'est un N-adding walk */
85  mpz_t listM[N];
86  INIT(g_init, "0")
87  INIT(h_init, "0")
88
89  for (int i = 0; i < N; i++)
90  {
91    mpz_init(listExposant[i].m);
92    mpz_urandomm(listExposant[i].m, state, q); /* defined mod q */
93    mpz_init(listExposant[i].n);
94    mpz_urandomm(listExposant[i].n, state, q);
95    mpz_powm(g_init, g, listExposant[i].m, p);
96    mpz_powm(h_init, h, listExposant[i].n, p);
97    mpz_init(listM[i]);
98    mpz_mul(listM[i], g_init, h_init);
99    mpz_mod(listM[i],listM[i],p);
100 }
101 mpz_clear(g_init);
102 mpz_clear(h_init);
103 gmp_randclear(state);
104
105 /* application de la fonction tant qu'il n'y a pas de collision */
106 do
107 {
108   f(&y, &a, &b, &g, &h, &q, &p, listExposant, listM);
109   f(&y_even, &a_even, &b_even, &g, &h, &q, &p, listExposant, listM);
110   f(&y_even, &a_even, &b_even, &g, &h, &q, &p, listExposant, listM);
111 } while (mpz_cmp(y, y_even) != 0);
112
113 /* Extraction du resultat , on veut : (a_even-a)(b-b_even)^(-1) (mod q)  */
114 mpz_sub(r, b, b_even); /* r = (b - b_even) (mod q) */
115 mpz_mod(r, r, q);      /* r = (b - b_even) (mod q) */
116
```

```
  /* test si (b - b_even) est inversible (mod q) */
  if (mpz_cmp_si(r, 0) == 0)
  {
    printf("ECHEC \n");
    goto end;
  }

  mpz_invert(r, r, q);        /* r= (b-b_even)^(-1) (mod q) */
  mpz_sub(a_even, a_even, a); /* a_even = a_even-a (mod q) */
  mpz_mul(r, r, a_even);
  mpz_mod(r, r, q);

  printf("r = ");
  PRINT(r)

  /* liberation de la m moire */
end:
  for (int i = 0; i < N; i++)
  {
    mpz_clear(listExposant[i].m);
    mpz_clear(listExposant[i].n);
    mpz_clear(listM[i]);
  }
  mpz_clear(g);
  mpz_clear(h);
  mpz_clear(p);
  mpz_clear(y);
  mpz_clear(y_even);
  mpz_clear(a_even);
  mpz_clear(a);
  mpz_clear(b);
  mpz_clear(b_even);
  mpz_clear(q);
  mpz_clear(r);
  fclose(input);

  return EXIT_SUCCESS;
}
```

D Implémentation des Points Distingués

```
// Pour compiler : gcc -g -Wall TER.c -lgmp

#include <assert.h>
#include <gmp.h>
#include <pthread.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <time.h>

#include "set.h"

#define SIZE_P 1024
#define N 24
#define SIZE_DISTINGUE 1017
#define N_THREADS 20
#define I_THETA \
  (1 << (SIZE_P - SIZE_DISTINGUE)) // Nb de bit de p -size distingu

#define INIT(name, value) \
  mpz_init(name);         \
  mpz_set_str(name, value, 10);

#define PRINT2(var)             \
  mpz_out_str(stdout, 10, var); \
  printf("\n");

typedef struct
{
  mpz_t m; /*exposant de  g*/
  mpz_t n; /*exposant de  h*/
} couple;

/* compteur de la taille du l'arbre */

int cpt = 0;

/* the struct shared by all threads*/
typedef struct
{
  node *set;                    /*Ensemble ou les thread mettent les    lments   */
  couple collision_1;           /* element 1 de la collision */
  couple collision_2;           /* element 2 de la collision */
  pthread_mutex_t lock_set;     /* lock pour la table (set) et collision1/2 */
  pthread_mutex_t lock_signal; /* lock pour la condition variable et
                                  collision_1/collision_2 */
  pthread_cond_t signalFinish; /* indique quand la collision a   t   trouv   */
  mpz_t p;
  mpz_t q;
  mpz_t g;
  mpz_t h;
  mpz_t listM[N];
  couple listExposant[N];
} thread_struct;

void f(mpz_t *y, mpz_t *a, mpz_t *b, mpz_t *g, mpz_t *h, mpz_t *q, mpz_t *p,
```

```
58        couple *listExposant, mpz_t *listM)
59 {
60   /* on g n re une partition de l'ensemble en N cas */
61   unsigned long int s = mpz_fdiv_ui(*y, N);
62   mpz_add(*b, *b, listExposant[s].n);
63   mpz_mod(*b, *b, *q);
64   mpz_add(*a, *a, listExposant[s].m);
65   mpz_mod(*a, *a, *q);
66   mpz_mul(*y, *y, listM[s]);
67   mpz_mod(*y, *y, *p);
68 }
69
70 void *f_dinstingue(thread_struct *s_struct)
71 {
72   /* INIT */
73   mpz_t y, a, b, n1, m1, tmp1, tmp2;
74   mpz_t *listExposant, listM;
75   mpz_init(tmp1);
76   mpz_init(tmp2);
77   mpz_init(a);
78   mpz_init(b);
79   mpz_init(y);
80   mpz_init(n1);
81   mpz_init(m1);
82   gmp_randstate_t state;
83   gmp_randinit_mt(state);
84   node *node_Tmp;
85   int borne = 20 * I_THETA;
86
87 start:
88   mpz_urandomm(n1, state, s_struct->q);
89   mpz_urandomm(m1, state, s_struct->q);
90   mpz_powm(tmp1, s_struct->g, m1, s_struct->p); /* tmp1 = g^m1 (mod p)*/
91   mpz_powm(tmp2, s_struct->h, n1, s_struct->p); /* tmp1 = h^n1 (mod p)*/
92   mpz_mul(y, tmp1, tmp2);
93   mpz_set(a, m1);
94   mpz_set(b, n1);
95
96   for (int i = 0; i < borne; i++)
97   {
98     /*faire un pas avec la fonction f */
99     f(&y, &a, &b, &(s_struct->g), &(s_struct->h), &(s_struct->q),
100      &(s_struct->p), s_struct->listExposant, s_struct->listM);
101    if (mpz_sizeinbase(y, 2) <= SIZE_DISTINGUE)
102    {
103      pthread_mutex_lock(&s_struct->lock_set);
104      node_Tmp = search(s_struct->set, y);
105      if (node_Tmp != NULL)
106      {
107        if (mpz_cmp(b, node_Tmp->n) == 0)
108        {
109          pthread_mutex_unlock(&s_struct->lock_set);
110          goto start;
111        }
112        mpz_set(s_struct->collision_1.m, a);
113        mpz_set(s_struct->collision_1.n, b);
114        mpz_set(s_struct->collision_2.n, node_Tmp->n);
115        mpz_set(s_struct->collision_2.m, node_Tmp->m);
116        pthread_mutex_lock(&s_struct->lock_signal);
```

```
        pthread_cond_broadcast(&s_struct->signalFinish);
        pthread_mutex_unlock(&s_struct->lock_signal);
        return NULL;
      }
      s_struct->set = insert(s_struct->set, y, b, a);
      cpt++;
      pthread_mutex_unlock(&s_struct->lock_set);
    }
  }
  goto start;
  return NULL;
}

int main(int argc, char *argv[])
{

  bool echec = false;
  FILE *in = fopen("input.txt", "r");


  if (in == NULL)
  {
    printf("Error open file IN\n");
    exit(EXIT_FAILURE);
  }

  char temp_g[10000];
  char temp_h[10000];
  char temp_p[10000];
  char temp_q[10000];

    fscanf(in, "%s %s %s %s", &temp_g, &temp_h, &temp_p, &temp_q);

    thread_struct sharedStruct; /* shared struct */
    sharedStruct.set = NULL;
    /* g nerateur g de G */
    INIT(sharedStruct.g, temp_g)
    /* h tel que l'on cherche log_g(h) */
    INIT(sharedStruct.h, temp_h)
    /* l'ordre du grand groupe */
    INIT(sharedStruct.p, temp_p)
    /* ordre du sous-groupe G */
    INIT(sharedStruct.q, temp_q)

    /* generation des ms et ns de fa on al atoire */
    gmp_randstate_t state;
    gmp_randinit_mt(state);
    gmp_randseed_ui(state,time(NULL));
    mpz_t g_init;
    mpz_t h_init;
    INIT(g_init, "0")
    INIT(h_init, "0")

    for (int i = 0; i < N; i++)
    {
      mpz_init(sharedStruct.listExposant[i].m);
      mpz_urandomm(sharedStruct.listExposant[i].m, state,
                   sharedStruct.q); /* defined mod q */
      mpz_init(sharedStruct.listExposant[i].n);
```

```
176       mpz_urandomm(sharedStruct.listExposant[i].n, state, sharedStruct.q);
177       mpz_powm(g_init, sharedStruct.g, sharedStruct.listExposant[i].m,
178                sharedStruct.p);
179       mpz_powm(h_init, sharedStruct.h, sharedStruct.listExposant[i].n,
180                sharedStruct.p);
181       mpz_init(sharedStruct.listM[i]);
182       mpz_mul(sharedStruct.listM[i], g_init, h_init);
183       mpz_mod(sharedStruct.listM[i],sharedStruct.listM[i],sharedStruct.p);
184     }
185
186     mpz_clear(g_init);
187     mpz_clear(h_init);
188     gmp_randclear(state);
189     /**/
190
191     /* INIT collision*/
192     mpz_init(sharedStruct.collision_1.n);
193     mpz_init(sharedStruct.collision_2.n);
194     mpz_init(sharedStruct.collision_1.m);
195     mpz_init(sharedStruct.collision_2.m);
196
197     /* creation de 2 thread qui calcul des points distingu  */
198     pthread_t tid[N_THREADS];                          /* thread identifiers */
199     pthread_mutex_init(&sharedStruct.lock_set, NULL); /* mutex lock for set */
200     /* the main thread waits on this variable to shut down everything
201     (has to be linked with a mutex) */
202     pthread_mutex_init(&sharedStruct.lock_signal, NULL);
203     pthread_cond_init(&sharedStruct.signalFinish, NULL);
204     /*locking the threads while creating them*/
205     pthread_mutex_lock(&sharedStruct.lock_signal);
206     pthread_mutex_lock(&sharedStruct.lock_set);
207
208
209     for (int i = 0; i < N_THREADS; i++)
210     {
211
212       if (pthread_create(&tid[i], NULL, f_dinstingue, &sharedStruct) != 0)
213       {
214         fprintf(stderr, "Unable to create thread number : %d\n", i);
215         exit(EXIT_FAILURE);
216       }
217     }
218
219     pthread_mutex_unlock(&sharedStruct.lock_set);
220     pthread_cond_wait(&sharedStruct.signalFinish, &sharedStruct.lock_signal);
221     for (int i = 0; i < N_THREADS; i++)
222     {
223       pthread_cancel(tid[i]);
224     }
225
226     printf("HEIGHT OF THE TREE : %d | NB OF ELEMENT ADDED : %d\n",
227            sharedStruct.set->height, cpt);
228
229     /* Extraction du resultat , on veut : (a_even-a)(b-b_even)^(-1) (mod q)  */
230     mpz_t r;
231     mpz_init(r);
232     mpz_sub(r, sharedStruct.collision_1.n,
233             sharedStruct.collision_2.n); /* r = (b - b_even) (mod q) */
234     mpz_mod(r, r, sharedStruct.q);       /* r = (b - b_even) (mod q) */
```

```
235
236     /* test si (b - b_even) est inversible (mod q) */
237     if (mpz_cmp_si(r, 0) == 0)
238     {
239       printf("ECHEC \n");
240       echec = true;
241       // goto end;
242     }
243
244     if (!echec)
245     {
246       mpz_invert(r, r, sharedStruct.q); /* r= (b-b_even)^(-1) (mod q) */
247       mpz_sub(sharedStruct.collision_2.m, sharedStruct.collision_2.m,
248               sharedStruct.collision_1.m); /* a_even = a_even-a (mod q) */
249       mpz_mul(r, r, sharedStruct.collision_2.m);
250       mpz_mod(r, r, sharedStruct.q);
251
252       /* liberation de la m moire */
253
254       mpz_clear(sharedStruct.g);
255       mpz_clear(sharedStruct.h);
256       mpz_clear(sharedStruct.p);
257       mpz_clear(sharedStruct.q);
258       deleteTree(sharedStruct.set);
259       for (int i = 0; i < N; i++)
260       {
261         mpz_clear(sharedStruct.listExposant[i].m);
262         mpz_clear(sharedStruct.listExposant[i].n);
263         mpz_clear(sharedStruct.listM[i]);
264       }
265     }
266     else
267     {
268       for (int i = 0; i < N; i++)
269       {
270         mpz_clear(sharedStruct.listExposant[i].m);
271         mpz_clear(sharedStruct.listExposant[i].n);
272         mpz_clear(sharedStruct.listM[i]);
273       }
274     }
275
276   fclose(in);
277   return EXIT_SUCCESS;
278 }
```

E Implémentation de Tag Tracing

E.1 Le header

```
#ifndef TAGTRACING_H
#define TAGTRACING_H

#include <gmp.h>
#include <pthread.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <time.h>

#define D 64
#define B_p 256
#define EPSILON 65536
#define OMEGA_p 4194304
#define OMEGA 4294967296
#define T 1024

typedef struct {
  mpz_t m; /*exposant de  g  m <-> g */
  mpz_t n; /*exposant de  h  n <-> h */
  mpz_t M; /*  Ms=g^m*h^n */
} element;

typedef struct {
  mpz_t m; /*exposant de  g  m <-> g */
  mpz_t n; /*exposant de  h  m <-> h */
} couple;

typedef struct tree_M {
  element value;
  unsigned long int v[D];  // tableau
  struct tree_M**
      childrens;      /*  ces enfant sous la forme de liste de r f rerence ces
                          enfants (donc ** pour dire list de r f rence) */
  int size_childrens; /* nombre d'noeuds*/
} treeM_l;

typedef struct s_linked_list {
  unsigned int value;
  struct s_linked_list* next;
} linked_list;

treeM_l* generate_tree(int r, int l, element* listM, mpz_t q, mpz_t p);

void delete_tree(treeM_l* Tree);

unsigned long int tau__(mpz_t g, unsigned long int* M_v);
unsigned long int tau_(mpz_t g, unsigned long int* M_v);
unsigned long int tau(mpz_t g, mpz_t B, mpz_t p, mpz_t tmp);

linked_list* add_to_list(unsigned long int s, linked_list* head);

void delete_list(linked_list* head);

```

```
56 treeM_l* find_node(treeM_l* root, linked_list* head);
57
58 #endif /* MODULE_H */
59
60 /*** end of file ***/
```

E.2 Le Génération de la table

```
1  #include "tagTracing.h"
2
3  treeM_l* build_aux(mpz_t q, mpz_t p, element M_prev, int r_curr, int l_curr,
4                     int r, element* listM, mpz_t B) {
5    if (l_curr == 0) {
6      return NULL;
7    }
8    /* allocation du noeud */
9    treeM_l* node = malloc(sizeof(treeM_l));
10
11   /* Construction du noeud */
12
13   /* first init value */
14   mpz_init(node->value.M);
15   mpz_init(node->value.m);
16   mpz_init(node->value.n);
17   mpz_mul(node->value.M, listM[r_curr].M, M_prev.M);
18   mpz_mod(node->value.M, node->value.M, p);
19
20   mpz_add(node->value.m, listM[r_curr].m, M_prev.m);
21   mpz_mod(node->value.m, node->value.m, q);
22
23   mpz_add(node->value.n, listM[r_curr].n, M_prev.n);
24   mpz_mod(node->value.n, node->value.n, q);
25   /* second init V as defined in the article */
26
27   mpz_t tmp;  // tmp vaut epsilon^i dans la boucle for
28   mpz_init(tmp);
29   mpz_set_ui(tmp, 1UL);
30   mpz_t tmp2;
31   mpz_init(tmp2);
32
33   for (int i = 0; i < D; i++) {
34     /* compute floor((eplsion^i  * M (mod p) ) /B) */
35     mpz_mul(tmp2, node->value.M, tmp);
36     mpz_mod(tmp2, tmp2, p);
37     mpz_fdiv_q(tmp2, tmp2, B);
38     node->v[i] = mpz_get_ui(tmp2);
39     /* update tmp */
40     mpz_mul_si(tmp, tmp, EPSILON);
41     mpz_mod(tmp, tmp, p);
42   }
43   mpz_clear(tmp);
44   mpz_clear(tmp2);
45
46   /* third the childrens nodes */
47   int size_c = r - r_curr;
48   node->size_childrens = size_c;
49   /* allocate the array of reference to the childrens */
50   node->childrens = malloc(size_c * sizeof(treeM_l*));
51   for (int i = r_curr; i < r; i++)
52     node->childrens[i - r_curr] =
```

```
        build_aux(q, p, node->value, i, l_curr - 1, r, listM, B);

  return node;
}

treeM_l* generate_tree(int r, int l, element* listM, mpz_t p, mpz_t q) {
  treeM_l* root = malloc(sizeof(treeM_l));
  mpz_init(root->value.M);
  mpz_init(root->value.n);
  mpz_init(root->value.m);
  mpz_set_ui(root->value.M, 1UL);
  /* init B */
  mpz_t B;
  mpz_init(B);
  mpz_cdiv_q_ui(B, p, OMEGA);
  // printf("B : ");
  // mpz_out_str(stdout, 10, B);
  // printf("\n");
  /* the array */
  root->childrens = malloc(r * sizeof(treeM_l*));
  root->size_childrens = r;

  for (int i = 0; i < r; i++)
    root->childrens[i] = build_aux(q, p, root->value, i, l, r, listM, B);
  mpz_clear(B);
  return root;
}

void delete_tree(treeM_l* node) {
  if (node == NULL) return;
  for (int i = 0; i < node->size_childrens; i++)
    delete_tree(node->childrens[i]);
  mpz_clear(node->value.M);
  mpz_clear(node->value.m);
  mpz_clear(node->value.n);

  free(node->childrens);
}

unsigned long int tau__(mpz_t g, unsigned long int* M_v) {
  unsigned long long int k = 0;
  unsigned long int t;
  unsigned long int x;

  for (int j = 0; j < g->_mp_size; j++) {
    t = g->_mp_d[j];
    for (int w = 0; w < 4; w++) {
      x = t % EPSILON;
      k = (k + (x * M_v[4 * j + w]) % OMEGA) % OMEGA;
      t = t >> 16;
    }
  }
  return k / OMEGA_p;
}

unsigned long int tau_(mpz_t g, unsigned long int* M_v) {
  unsigned long int k = tau__(g, M_v);
  return (k % B_p == B_p - 1 || k % B_p == B_p - 2) ? B_p : k;
}
```

```

linked_list* add_to_list(unsigned long int s, linked_list* head) {
  linked_list* new_node = malloc(sizeof(linked_list));
  new_node->value = (s);
  if (head == NULL || s <= head->value) {
    new_node->next = head;
    return new_node;
  }
  linked_list* current_node = head;
  while (current_node->next != NULL &&
         current_node->next->value < new_node->value) {
    current_node = current_node->next;
  }
  new_node->next = current_node->next;
  current_node->next = new_node;
  return head;
}

void delete_list(linked_list* head) {
  linked_list* tmp;
  while (head != NULL) {
    tmp = head->next;
    free(head);
    head = tmp;
  }
}

treeM_l* find_node(treeM_l* root, linked_list* head) {
  int curr_r = 0;
  int index;
  while (head != NULL) {
    index = head->value - curr_r;
    root = root->childrens[index];
    curr_r = head->value;
    head = head->next;
  }
  return root;
}

unsigned long int tau(mpz_t g, mpz_t B, mpz_t p, mpz_t tmp) {
  mpz_mod(tmp, g, p);
  mpz_fdiv_q(tmp, tmp, B);
  unsigned long int k = mpz_get_ui(tmp);
  return k;
}
```

E.3 Tag Tracing

```
// Pour compiler : gcc -g -Wall TER.c -lgmp

#include <assert.h>
#include <gmp.h>
#include <pthread.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <time.h>

#include "set.h"
```

```c
#include "tagTracing.h"

#define SIZE_P 1024
#define R 7
#define L 20
#define SIZE_DISTINGUE 1023

#define N_THREADS 1
#define I_THETA \
  (1 << (SIZE_P - SIZE_DISTINGUE))  // Nb de bit de p -size distingu

#define INIT(name, value) \
  mpz_init(name);         \
  mpz_set_str(name, value, 10);

#define PRINT2(var)             \
  mpz_out_str(stdout, 10, var); \
  printf("\n");

/* compteur de la taille du l'arbre */

int cpt = 0;
unsigned long int itter = 0;

/* the struct shared by all threads*/
typedef struct {
  node *set;                /* Ensemble ou les thread mettent les    lments   */
  couple collision_1;       /* element 1 de la collision */
  couple collision_2;       /* element 2 de la collision */
  pthread_mutex_t lock_set; /* lock pour la table (set) et collision1/2 */
  pthread_mutex_t lock_signal; /* lock pour la condition variable et
                                  collision_1/collision_2 */
  pthread_cond_t signalFinish; /* indique quand la collision a   t   trouv   */
  mpz_t p;
  mpz_t q;
  mpz_t g;
  mpz_t h;
  element listM[R];
  treeM_l *root;
} thread_struct;

void *f_dinstingue(thread_struct *s_struct) {
  /* INIT */
  mpz_t y, a, b, n1, m1, tmp1, tmp2, tmp3;
  mpz_init(a);
  mpz_init(b);
  mpz_init(y);
  gmp_randstate_t state;
  gmp_randinit_mt(state);
  node *node_Tmp;
  int borne = 1000000;  // just a definition

  mpz_init(n1);
  mpz_init(m1);
  mpz_init(tmp1);
  mpz_init(tmp2);
  mpz_init(tmp3);
  // new
  linked_list *l;
```

```
72  treeM_l *node_curr;
73  unsigned long int s, tag;
74  mpz_t B;
75  mpz_init(B);
76  mpz_cdiv_q_ui(B, s_struct->p, OMEGA);
77  mpz_mul_ui(B, B, OMEGA_p);
78
79  while (true) {
80    mpz_urandomm(n1, state, s_struct->q);
81    mpz_urandomm(m1, state, s_struct->q);
82    mpz_powm(tmp1, s_struct->g, m1, s_struct->p); /* tmp1 = g^m1 (mod p)*/
83    mpz_powm(tmp2, s_struct->h, n1, s_struct->p); /* tmp1 = h^n1 (mod p)*/
84    mpz_mul(y, tmp1, tmp2);
85    mpz_mod(y, y, s_struct->p);
86    // mpz_mod(y, y, s_struct->p);
87    tag = tau(y, B, s_struct->p, tmp3);
88    mpz_set(a, m1);
89    mpz_set(b, n1);
90    for (int i = 0; i < borne; i++) {
91      /* faire un tag tracing */
92      l = NULL;
93      node_curr = s_struct->root;
94      s = tag / B_p;
95      for (int j = 0; j < L; j++) {
96        itter++;
97        l = add_to_list(s, l);
98        node_curr = find_node(s_struct->root, l);
99        tag = tau__(y, node_curr->v);
100       if (tag % B_p == B_p - 1 || tag % B_p == B_p - 2) break;
101       s = tag / B_p;
102       // printf("the value of j : %d\n", j);
103     }
104     delete_list(l);
105     mpz_add(b, b, node_curr->value.n);
106     mpz_mod(b, b, s_struct->q);
107     mpz_add(a, a, node_curr->value.m);
108     mpz_mod(a, a, s_struct->q);
109     mpz_mul(y, y, node_curr->value.M);
110     mpz_mod(y, y, s_struct->p);
111     tag = tau(y, B, s_struct->p, tmp3);
112
113
114     if (tag % B_p == B_p - 1 && mpz_sizeinbase(y, 2) <= SIZE_DISTINGUE) {
115       pthread_mutex_lock(&s_struct->lock_set);
116
117       node_Tmp = search(s_struct->set, y);
118       if (node_Tmp != NULL) {
119         if (mpz_cmp(b, node_Tmp->n) == 0) {
120           pthread_mutex_unlock(&s_struct->lock_set);
121           break;
122         }
123         mpz_set(s_struct->collision_1.m, a);
124         mpz_set(s_struct->collision_1.n, b);
125         mpz_set(s_struct->collision_2.n, node_Tmp->n);
126         mpz_set(s_struct->collision_2.m, node_Tmp->m);
127         pthread_mutex_lock(&s_struct->lock_signal);
128         pthread_cond_broadcast(&s_struct->signalFinish);
129         pthread_mutex_unlock(&s_struct->lock_signal);
130         return NULL;
```

```
        }
        s_struct->set = insert(s_struct->set, y, b, a);
        cpt++;
        pthread_mutex_unlock(&s_struct->lock_set);
      }
    }
  }
  return NULL;
}

int main(int argc, char *argv[]) {
  FILE *in = fopen("input.txt", "r");
  if (in == NULL) {
    printf("Error open file IN\n");
    exit(EXIT_FAILURE);
  }

  // while (!feof(in))
  //{
  char temp_g[1000];
  char temp_h[1000];
  char temp_p[1000];
  char temp_q[1000];

  // while (!feof(in))
  //{
  struct timeval start, end;

  fscanf(in, "%s %s %s %s", temp_g, temp_h, temp_p, temp_q);
  thread_struct sharedStruct; /* shared struct */
  sharedStruct.set = NULL;
  /* g nerateur g de G */
  INIT(sharedStruct.g, temp_g)
  /* h tel que l'on cherche log_g(h) */
  INIT(sharedStruct.h, temp_h)
  /* l'ordre du grand groupe */
  INIT(sharedStruct.p, temp_p)
  /* ordre du sous-groupe G */
  INIT(sharedStruct.q, temp_q)

  PRINT2(sharedStruct.h)
  // PRINT2(sharedStruct.p)
  // PRINT2(sharedStruct.q)

  /* generation des ms et ns de fa on al atoire */
  gmp_randstate_t state;
  gmp_randinit_mt(state);
  unsigned long int seed = time(NULL);
  gmp_randseed_ui(state, seed);
  mpz_t g_init;
  mpz_t h_init;
  mpz_init(g_init);
  mpz_init(h_init);
  for (int i = 0; i < R; i++) {
    mpz_init(sharedStruct.listM[i].m);
    mpz_urandomm(sharedStruct.listM[i].m, state,
                 sharedStruct.q); /* defined mod q */
    mpz_init(sharedStruct.listM[i].n);
    mpz_urandomm(sharedStruct.listM[i].n, state, sharedStruct.q);
```

```
    mpz_powm(g_init, sharedStruct.g, sharedStruct.listM[i].m, sharedStruct.p);
    mpz_powm(h_init, sharedStruct.h, sharedStruct.listM[i].n, sharedStruct.p);
    mpz_init(sharedStruct.listM[i].M);

    mpz_mul(sharedStruct.listM[i].M, g_init, h_init);
    mpz_mod(sharedStruct.listM[i].M, sharedStruct.listM[i].M, sharedStruct.p);
    printf("here is M%d : ", i);
    PRINT2(sharedStruct.listM[i].M)
    printf("\n");
  }

  mpz_clear(g_init);
  mpz_clear(h_init);
  gmp_randclear(state);

  /* INIT DE M_L */
  sharedStruct.root =
      generate_tree(R, L, sharedStruct.listM, sharedStruct.p, sharedStruct.q);
  printf("the table has been generated \n");

  /* INIT collision*/
  mpz_init(sharedStruct.collision_1.n);
  mpz_init(sharedStruct.collision_2.n);
  mpz_init(sharedStruct.collision_1.m);
  mpz_init(sharedStruct.collision_2.m);

  /* creation de N_THREADS thread qui calcul des points distingu  */
  pthread_t tid[N_THREADS];                          /* thread identifiers */
  pthread_mutex_init(&sharedStruct.lock_set, NULL); /* mutex lock for set */
  /* the main thread waits on this variable to shut down everything
  (has to be linked with a mutex) */
  pthread_mutex_init(&sharedStruct.lock_signal, NULL);
  pthread_cond_init(&sharedStruct.signalFinish, NULL);
  /*locking the threads while creating them*/
  pthread_mutex_lock(&sharedStruct.lock_signal);
  pthread_mutex_lock(&sharedStruct.lock_set);

  for (int i = 0; i < N_THREADS; i++) {
    if (pthread_create(&tid[i], NULL, f_dinstingue, &sharedStruct) != 0) {
      fprintf(stderr, "Unable to create thread number : %d\n", i);
      exit(EXIT_FAILURE);
    }
  }

  gettimeofday(&start, NULL);
  pthread_mutex_unlock(&sharedStruct.lock_set);
  pthread_cond_wait(&sharedStruct.signalFinish, &sharedStruct.lock_signal);
  for (int i = 0; i < N_THREADS; i++) {
    pthread_cancel(tid[i]);
  }

  printf("HEIGHT OF THE TREE : %d | NB OF ELEMENT ADDED : %d\n",
         sharedStruct.set->height, cpt);

  /* Extraction du resultat , on veut : (a_even-a)(b-b_even)^(-1) (mod q)  */
  mpz_t r;
  mpz_init(r);
  mpz_sub(r, sharedStruct.collision_1.n,
          sharedStruct.collision_2.n); /* r = (b - b_even) (mod q) */
```

```
  mpz_mod(r, r, sharedStruct.q);       /* r = (b - b_even) (mod q) */

  /* test si (b - b_even) est inversible (mod q) */
  if (mpz_cmp_si(r, 0) == 0) {
    printf("ECHEC \n");
    goto clear;
  }

  mpz_invert(r, r, sharedStruct.q); /* r= (b-b_even)^(-1) (mod q) */
  mpz_sub(sharedStruct.collision_2.m, sharedStruct.collision_2.m,
          sharedStruct.collision_1.m); /* a_even = a_even-a (mod q) */
  mpz_mul(r, r, sharedStruct.collision_2.m);
  mpz_mod(r, r, sharedStruct.q);

  printf("r = ");
  PRINT2(r)

  /* liberation de la m moire */
clear:
  mpz_clear(sharedStruct.g);
  mpz_clear(sharedStruct.h);
  mpz_clear(sharedStruct.p);
  mpz_clear(sharedStruct.q);
  mpz_clear(r);
  mpz_clear(sharedStruct.collision_1.n);
  mpz_clear(sharedStruct.collision_1.m);
  mpz_clear(sharedStruct.collision_2.n);
  mpz_clear(sharedStruct.collision_2.m);
  deleteTree(sharedStruct.set);
  for (int i = 0; i < R; i++) {
    mpz_clear(sharedStruct.listM[i].m);
    mpz_clear(sharedStruct.listM[i].n);
    mpz_clear(sharedStruct.listM[i].M);
  }
  delete_tree(sharedStruct.root);

  gettimeofday(&end, NULL);
  long int temps = ((end.tv_sec * 1000000 + end.tv_usec) -
                    (start.tv_sec * 1000000 + start.tv_usec));
  printf("temps = %ld\n", temps);
  printf("here is the number of itteration : %ld", itter);

  fclose(in);
  return EXIT_SUCCESS;
}
```