Logarithme discret en cryptographie

Ce chapitre introductif présente le problème du calcul du logarithme discret, qui est un problème central en cryptographie à clé publique. Dans ce chapitre, nous donnons un aperçu plutôt bref aussi bien des aspects cryptographiques que des aspects cryptanalytiques liés au calcul du logarithme discret. Nous partons d'un exemple d'application qui motive l'intérêt cryptographique pour étudier ce problème. Ensuite, nous définissons d'une manière formelle le problème du logarithme discret et les notions qui lui sont liées. Nous détaillons les propriétés requises pour les groupes utilisés ainsi que les applications dans lesquelles intervient ce problème. Dans un second temps, nous passerons à la cryptanalyse du problème du logarithme discret, en ciblant principalement les algorithmes de *calcul d'index* qui résolvent le logarithme discret sur les corps finis.

Sommaire

1.1 Problème de l'échange de clés

La clé est un outil central en cryptographie pour pouvoir mettre en place un protocole qui protège la communication entre deux parties. Dans le cadre de la cryptographie symétrique (également appelée cryptographie à clé secrète), les deux parties, que l'on appelle habituellement Alice et Bob, sont amenées à établir une clé secrète commune. On appelle ce problème le problème de l'échange de clé. Par le passé, les valises diplomatiques ou d'autres formes de canaux « sécurisés » ont été utilisées pour pouvoir transmettre la clé. De nos jours, il est pertinent de supposer qu'Alice et Bob n'ont accès qu'à un canal de communication non protégé. Nous supposons par ailleurs que les deux parties sont authentifiées de sorte à prévenir des attaques de type *Homme de milieu* (*Man in the middle* en anglais).

Diffie et Hellman ont proposé en 1976 une solution qui répond au problème de la création d'une clé commune [DH76]. De nos jours, le protocole d'échange de clés Diffie-Hellman est intégré dans différents navigateurs web et fait partie de plusieurs protocoles de sécurisation des échanges sur Internet tels que TLS (Transport Layer Security).

Le schéma de la figure 1.1 détaille le protocole qui permet à Alice et Bob de créer communément une clé. Une des deux parties, ici Alice, commence par choisir un groupe cyclique G de cardinal n, noté multiplicativement et engendré par g. Nous utilisons la notation $G = \langle g \rangle$ pour préciser que G est engendré par g. Alice choisit un entier $k_A \in [0, n-1]$, calcule g^{k_A} et l'envoie à Bob. Idem, Bob choisit un entier $k_B \in [0, n-1]$, calcule g^{k_B} et l'envoie à Alice. À la fin de l'échange, les deux parties peuvent alors construire la même clé $K = (g^{k_A})^{k_B} = (g^{k_B})^{k_A}$.

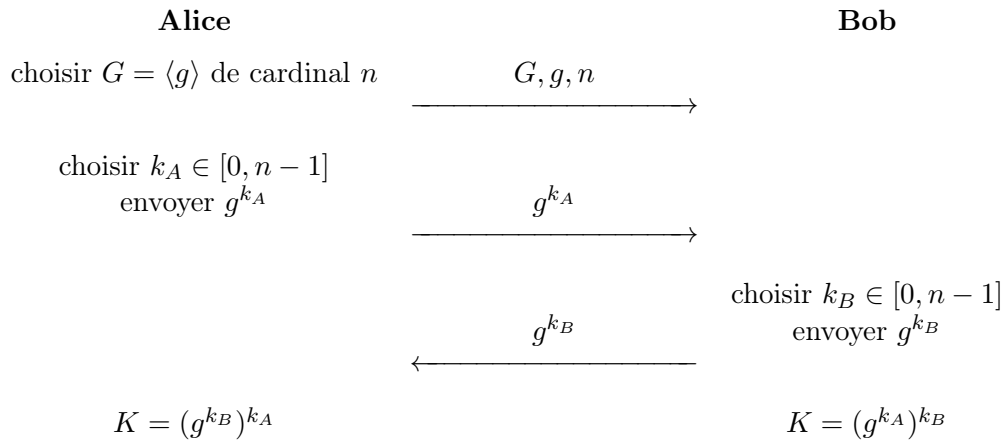


FIGURE 1.1 – Échange de clés Diffie-Hellman.

Un attaquant, qui écoute sur le canal, voit passer G, g, n, g^{k_A} et g^{k_B}. Il espère pouvoir deviner K à partir de ces observations. On appelle ce problème le problème Diffie-Hellman. Pour être en mesure de calculer K, l'attaquant pourrait espérer pouvoir calculer k_A ou k_B à partir de g^{k_A} et g^{k_B}, respectivement. Ce calcul correspond à une résolution du problème du logarithme discret que nous allons à présent définir plus en détail.

1.2 Problème du logarithme discret

1.2.1 Définitions

Reprenons le groupe cyclique G de cardinal n et engendré par g. L'opération d'exponentiation discrète est définie dans le groupe comme suit.

Définition 1.1 (Exponentiation discrète)**.** *Étant donnés le groupe G, un entier k dans $[0, n-1]$ et un élément $g \in G$, élever g à la puissance k consiste à calculer $g^k = \underbrace{g \times g \times \cdots \times g}_{k \text{ fois}}$.*

On note que l'exponentiation discrète, en utilisant la méthode d'exponentiation binaire, se calcule en $O(\log n)$ opérations dans G.

Le logarithme discret correspond à la réciproque de l'exponentiation. Le logarithme discret d'un élément du groupe est défini comme suit.

Définition 1.2 (Logarithme discret et problème du logarithme discret)**.** *Étant donnés un groupe G cyclique engendré par g et un élément h dans le groupe, le logarithme discret de h, en base g, qu'on note $\log_g h$, est l'unique entier k dans $[0, n-1]$ tel que*

$$h = g^k. \tag{1.1}$$

Le problème du logarithme discret consiste à calculer $k = \log_g h$ à partir de h et g.

Dès lors que la loi de groupe est calculable en temps polynomial en la taille des entrées, le problème de l'exponentiation discrète est dit « facile », c'est-à-dire, qu'il existe des algorithmes qui permettent de le résoudre en temps « court » avec des ressources de calcul limitées. Le problème inverse, celui du logarithme discret, est « difficile » dans certains groupes, c'est-à-dire nécessite un temps de calcul « long » même en utilisant un grand nombre de ressources. Les notions de « court » et « long » qu'on utilise ici sont évidemment relatives et seront clarifiées dans la suite quand nous allons préciser les complexités des opérations.

L'asymétrie entre l'exponentiation et le logarithme discret est utilisée en cryptographie à clé publique pour construire des *fonctions à trappe*, c'est-à-dire, des fonctions qui peuvent être aisément calculées lorsque la trappe est connue et dures à inverser sans connaissance de la trappe.

Dans l'exemple de l'échange de clés Diffie-Hellman, que nous avons précédemment évoqué, la sécurité du protocole repose sur la difficulté pour une troisième partie de pouvoir calculer des logarithmes discrets à partir des données échangées. Cette difficulté est formalisée dans la définition suivante.

Définition 1.3 (Problème Diffie-Hellman calculatoire (CDH))**.** *Étant donné le groupe G engendré par g, le problème CDH consiste à calculer $g^{k_A k_B}$ à partir de g, g^{k_A} et g^{k_B}.*

Il est clair que résoudre le problème du logarithme discret permettra à l'attaquant de résoudre le problème CDH. La réciproque est moins triviale [Mau94, MW96].

1.2.2 Groupes proposés

Les groupes proposés pour les applications cryptographiques doivent satisfaire les propriétés suivantes :

— les éléments du groupe peuvent être représentés à l'aide de $O(\log n)$ bits ;

— l'arithmétique dans le groupe G doit être efficace, en particulier l'exponentiation doit pouvoir être faite avec une complexité au plus polynomiale en la taille des entrées ;
— le problème du logarithme discret doit être aussi difficile que possible. Idéalement, la complexité des meilleurs algorithmes qui permettent de le résoudre devrait être exponentielle en la taille des entrées.

Il existe des groupes qui ne répondent pas à ces critères, tels que les groupes additifs $(\mathbb{Z}/n\mathbb{Z}, +)$, vu que le problème du logarithme discret dans ces groupes se résout avec l'algorithme d'Euclide, en temps polynomial en la taille des entrées.

Parmi les bons groupes au sens de la difficulté du problème du logarithme discret, nous trouvons les groupes multiplicatifs des corps finis $\mathbb{F}_{p^k}$ et les groupes des points de courbes elliptiques définies sur des corps finis. Ces deux familles de groupes satisfont les deux premières propriétés. La difficulté de résoudre le problème du logarithme discret varie en fonction du type de groupe. En effet, pour les corps finis, il existe des algorithmes pour résoudre le problème du logarithme discret avec une complexité sous-exponentielle ou quasi-polynomiale en la taille du corps, que nous allons détailler dans la section 1.4. Pour les courbes elliptiques, il existe aujourd'hui des algorithmes sous-exponentiels de cryptanalyse pour des familles particulières de courbes. Toutefois, il n'existe aucun algorithme de complexité sous-exponentielle pour calculer le logarithme discret sur une courbe elliptique générique.

1.2.3 Applications

Des primitives cryptographiques qui reposent sur la difficulté du logarithme discret ont été développées pour l'échange de clés, pour le chiffrement ou pour la signature.

Cryptosystème d'ElGamal. ElGamal a proposé en 1985 un système de chiffrement et de signature, reposant sur le problème du logarithme discret dans un corps fini [ElG85]. L'algorithme de signature est intégré dans la suite GnuPG (GNU Privacy Guard).

La signature DSA. L'algorithme DSA (Digital Signature Algorithm) est l'algorithme de signature standardisé par le NIST par le biais du standard DSS (Digital Signature Standard). Dans cet algorithme, un corps fini premier $\mathbb{F}_p$ est utilisé. Néanmoins, les opérations de calcul ne sont pas effectuées dans ${\mathbb{F}_p}^\times$, mais dans un sous-groupe d'ordre premier q qui divise $p-1$. Nous reviendrons dans la sous-section 1.4.1 sur cette idée de calculer dans un sous-groupe plutôt que dans tout le corps.

Les couplages.

Définition 1.4 (Couplage)**.** *Soient G_1, G_2 et G_T trois groupes cycliques de même cardinal. On note G_1 et G_2 additivement et G_T multiplicativement. Un couplage est une application bilinéaire*

$$e : G_1 \times G_2 \to G_T \tag{1.2}$$

qui vérifie les propriétés suivantes

1. *Non-dégénérescence :*

$$\begin{cases} \forall P \in G_1, \exists Q \in G_2 \text{ tel que } e(P,Q) \neq 1_{G_T} \\ \forall Q \in G_2, \exists P \in G_1 \text{ tel que } e(P,Q) \neq 1_{G_T} \end{cases}$$

2. *Bilinéarité :* $\forall P, P' \in G_1$ *et* $Q, Q' \in G_2$,

$$\begin{cases} e(P+P',Q) &= e(P,Q) \times e(P',Q) \\ e(P,Q+Q') &= e(P,Q) \times e(P,Q') \end{cases}$$

3. *Calculabilité : il existe un algorithme efficace pour calculer* e.

Les couplages ont été introduits en cryptographie par Menezes, Okamoto et Vanstone [MOV93] et par Frey et Rück [FR94] dans le cadre des attaques sur les courbes elliptiques supersingulières. Par la suite, différents systèmes cryptographiques qui reposent sur les couplages ont été proposés, parmi lesquels on peut citer l'échange de clés tripartite en un tour introduit par Joux [Jou00, Jou04], le chiffrement fondé sur l'identité de Boneh et Franklin [BF01] et de Sakai, Ohgishi et Kasahara [SOK00], le schéma de traçage de traîtres de Mitsunari, Sakai et Kasahara [MSK02] ou encore la signature courte de Boneh, Lynn et Shacham [BLS01]. Une étude d'un grand nombre de systèmes cryptographiques reposant sur les couplages a été réalisée par Dutta, Barua et Sarkar dans [DBS04].

En cryptographie à base de couplage, les groupes G_1 et G_2 correspondent à des groupes de points de courbes elliptiques ; le groupe G_T est un groupe multiplicatif d'un corps fini. La sécurité des systèmes utilisant les couplages nécessite que le problème du logarithme discret soit difficile dans les trois groupes.

1.2.4 Intérêt pour les logarithmes discrets

Le choix des cryptographes d'utiliser des systèmes reposant sur la difficulté du logarithme discret est motivé par plusieurs raisons. D'abord, ces systèmes répondent aux exigences de « bons » systèmes cryptographiques en terme de taille des clés et de niveau de sécurité atteint. De plus, ces systèmes représentent des alternatives aux systèmes dérivés de RSA qui reposent sur le problème de la factorisation d'entiers. Un troisième argument en faveur de l'étude des logarithmes discrets est que les couplages ne peuvent pas être réalisés avec RSA et ses variantes.

1.3 Algorithmes génériques

Dans un premier temps, nous allons voir certains algorithmes génériques de calcul de logarithme discret, c'est-à-dire des algorithmes qui s'appliquent sur n'importe quel groupe et qui ne tiennent pas compte des spécificités du groupe. Ces algorithmes sont plus efficaces que la recherche exhaustive qui a une complexité en $O(n)$, avec n le cardinal du groupe. Leurs complexités restent néanmoins exponentielles en la taille des entrées. Nechaev et Shoup ont même prouvé qu'il n'existe pas d'algorithme générique qui ait une complexité meilleure que $\Omega(\sqrt{n})$ lorsque n est premier [Nec94, Sho97]. Plus loin, dans la sous-section 1.4.1, nous allons voir une classe d'algorithmes non génériques qui sont spécifiques aux corps finis et qui permettent d'atteindre des complexités plus petites.

1.3.1 Baby-step–Giant-step

Cet algorithme a été inventé par Shanks en 1971 [Sha71]. En reprenant les notations précédentes, l'algorithme calcule le logarithme discret k d'un élément h de $G = \langle g \rangle$. L'idée est d'écrire k sous la forme de $c \times u + d$, pour $u = \lceil \sqrt{n} \rceil$. Le couple (c, d) est obtenu en trouvant une collision entre deux listes, une dite des « pas de bébé » et l'autre dite des « pas de géant ».

L'algorithme procède comme suit :

— on calcule $\mathcal{G} = \{g^d \text{ pour } d \in [0, u]\}$
— on calcule $\mathcal{B} = \{h(g^{-u})^c \text{ pour } c \in [0, n/u]\}$ et on s'arrête si $h(g^{-u})^c \in \mathcal{G}$.
— le logarithme de h est alors $c \times u + d$.

Le temps de calcul de l'algorithme *Baby-step–Giant-step* est dominé par le calcul des deux listes. C'est un algorithme déterministe qui a une complexité en temps en $O(\sqrt{n})$ en utilisant un espace mémoire en $O(\sqrt{n})$.

1.3.2 Pollard rho

L'algorithme *rho* a été introduit par Pollard en 1978 pour résoudre le problème du logarithme discret [Pol78] en adaptant l'algorithme du même nom pour la factorisation des entiers [Pol75]. C'est pourquoi il est assez commun de trouver dans la littérature la nomenclature *rho pour les logarithmes* pour distinguer cet algorithme de celui pour la factorisation.

Pour calculer le logarithme d'un élément h, l'algorithme *rho* repose sur la recherche de collisions dans les termes d'une séquence obtenue en appliquant une fonction pseudo-aléatoire f de G dans G sur des éléments de la forme $x_i = g^{a_i} h^{b_i}$.

Comme G est fini et que f est déterministe, la séquence obtenue contient un cycle. On s'attend à ce que la taille du cycle soit en $O(\sqrt{n})$. La recherche de collision est effectuée selon l'algorithme de Floyd pour la détection de cycle [Flo67]. Quand une collision est trouvée, nous avons une égalité de la forme $g^{a_1} h^{b_1} = g^{a_2} h^{b_2}$, pour des entiers connus a_1, b_1, a_2 et b_2. Cette égalité nous donne l'équation suivante :

$$(b_2 - b_1)k \equiv (a_1 - a_2) \pmod{n}$$

Si cette équation admet une solution (c'est-à-dire si $\text{pgcd}(b_2 - b_1, n) = 1$), alors k correspond au logarithme discret de h. Sinon, nous recommençons le calcul avec de nouvelles valeurs initiales pour les variables a_0 et b_0.

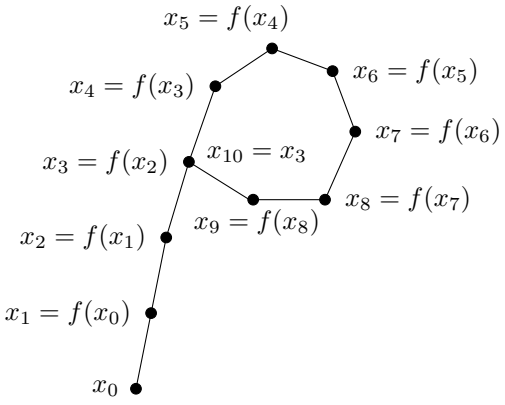


FIGURE 1.2 – Orbite typique de l'algorithme *rho* de Pollard.

L'algorithme *rho* de Pollard est probabiliste avec une complexité temps de $O(\sqrt{n})$ et une complexité mémoire de $O(\log n)$. C'est à ce jour l'algorithme générique le plus efficace en terme de temps et d'espace. Il existe un algorithme déterministe cousin de l'algorithme *rho*, souvent appelé l'algorithme *lambda* de Pollard, qui n'a pas une meilleure complexité, mais qui se parallélise mieux et qui est utile lorsque le logarithme recherché se trouve dans un intervalle limité [Pol78].

1.3.3 Pohlig-Hellman

L'algorithme de *Pohlig-Hellman*, parfois désigné aussi par méthode de *réduction de Pohlig-Hellman* permet de réduire le calcul du logarithme discret dans le groupe G en calculs de logarithmes discrets dans des groupes plus petits lorsque le cardinal de G n'est pas premier [PH78]. Supposons que nous connaissons la factorisation du cardinal du groupe G : $n = \prod_{i=1}^{r} p_i^{e_i}$, où $p_1, \ldots, p_r$ sont des nombres premiers et $e_1, \ldots, e_r$ sont les exposants correspondants. L'algorithme procède comme suit. Pour chaque i dans $\{1, \ldots, r\}$, on effectue e_i calculs de logarithme discret dans un sous-groupe isomorphe à $(\mathbb{Z}/p_i\mathbb{Z})$ (d'ordre p_i). Par la suite, le logarithme discret dans G peut être obtenu à partir des logarithmes calculés avec la reconstruction du théorème des restes chinois.

Le coût de la reconstruction, polynomial, est petit devant le coût du calcul du logarithme discret dans un sous-groupe. Par conséquent, la complexité de la résolution devient celle de la résolution du logarithme discret dans le plus grand sous-groupe d'ordre premier. Ainsi, combiner la méthode de *Pohlig-Hellman* avec la méthode *rho* de Pollard, va permettre de calculer des logarithmes discrets avec une complexité en $O\left(\sqrt{p}\right)$, où p est le plus grand diviseur premier de l'ordre n.

Le groupe G peut aussi avoir une structure particulière. Dans ces cas, exploiter la structure du groupe pour calculer le logarithme discret peut s'avérer une meilleure stratégie que d'appliquer la réduction de *Pohlig-Hellman*.

1.4 Logarithme discret dans les corps finis

1.4.1 Cadre du problème

Dans la suite de ce document, nous allons nous concentrer sur les groupes multiplicatifs des corps finis $\mathbb{F}_q$, où q est une puissance d'un nombre premier qu'on écrit sous la forme $q = p^k$. Suivant comment se comparent p et k, nous distinguons les corps de petite, moyenne ou grande caractéristique (voir figure 1.5) :

— Petite caractéristique : ce cas correspond à $\log p$ petit devant k.
— Moyenne caractéristique : ce cas correspond à $\log p$ de même ordre que k.
— Grande caractéristique : ce cas correspond à k petit devant $\log p$.

Nous allons aussi être amenés à considérer des cas plus particuliers tels que les corps premiers $\mathbb{F}_p$ ou les corps binaires $\mathbb{F}_{2^k}$.

À ce jour, le problème du logarithme discret dans les groupes multiplicatifs des corps finis est modérément difficile. Depuis quelques années, ce problème est la cible de plusieurs améliorations théoriques et pratiques.

Dans notre étude de la difficulté du logarithme discret dans les corps finis, l'objectif est de calculer le logarithme discret non pas vraiment dans le groupe $\mathbb{F}_{p^k}^{\times}$, mais dans un sous-groupe premier d'ordre suffisamment grand, qui est souvent le plus grand sous-groupe d'ordre premier de $\mathbb{F}_{p^k}^{\times}$. En effet, nous avons vu avec la technique de réduction de *Pohlig-Hellman* que résoudre le logarithme discret dans $\mathbb{F}_{p^k}^{\times}$ correspond à le résoudre dans son plus grand sous-groupe. Dans le contexte des applications, il est cryptographiquement pertinent de reposer sur la sécurité dans un sous-groupe suffisamment grand plutôt que de faire les calculs dans le groupe multiplicatif. Ainsi, si on cherche un « bon » corps fini $\mathbb{F}_{p^k}$, on va vérifier si $p^k - 1$ a un « grand » facteur premier, ce qui ne doit pas forcément être interprété comme le souhait que $(p^k - 1)/2$ soit premier. Par

« grand », nous voulons préciser qu'il doit résister à une attaque d'un algorithme générique, en l'occurrence Pollard rho.

1.4.2 Algorithmes de calcul d'index

Les groupes multiplicatifs des corps finis font partie des groupes pour lesquels les algorithmes de *calcul d'index* peuvent être utilisés pour résoudre le problème du logarithme discret. Ces algorithmes sont des adaptations des algorithmes de combinaison de congruences pour la factorisation des entiers. Le terme *calcul d'index* désigne la famille de ces algorithmes. En effet, ces algorithmes partagent un même schéma général, même si les détails relatifs aux étapes successives de ces algorithmes diffèrent et qu'ils n'utilisent pas toujours les mêmes outils et objets mathématiques.

Les algorithmes de calcul d'index résolvent le logarithme discret en un temps sous-exponentiel. Il est commun d'exprimer les complexités sous-exponentielles de ces algorithmes par le biais de la fonction sous-exponentielle $L_n(\alpha, c)$, introduite par Pomerance [Pom82].

Définition 1.5 (Fonction sous-exponentielle)**.** *Étant donnés les paramètres* $0 \leq \alpha \leq 1$ *et* $c > 0$*, la fonction sous-exponentielle est définie par*

$$L_n(\alpha, c) = \exp\left(c(1 + o(1))(\log n)^{\alpha}(\log\log n)^{1-\alpha}\right). \tag{1.3}$$

Le paramètre α est le paramètre principal de cette notation. Lorsque $\alpha = 0$, nous retrouvons une complexité polynomiale en la taille des entrées, c'est-à-dire en $O\left((\log n)^c\right)$. Si $\alpha = 1$, la complexité est exponentielle en la taille des entrées. Pour des valeurs de α intermédiaires, il s'agit de complexité sous-exponentielle.

L'idée principale des algorithmes de calcul d'index est de « construire » le logarithme discret d'un élément arbitraire à partir d'un ensemble de logarithmes discrets connus. La structure générale de ces algorithmes s'appuie sur trois étapes principales :

— l'étape de *crible*,
— l'étape d'*algèbre linéaire*,
— et l'étape de *logarithme individuel*.

La première étape cherche des relations linéaires entre les logarithmes discrets d'éléments appartenant à un sous-ensemble du groupe ; ce sous-ensemble est appelé *base de facteurs*. Quand suffisamment de relations ont été trouvées, les logarithmes des éléments de la base de facteurs peuvent être calculés pendant la deuxième phase en résolvant un système linéaire. Enfin, la troisième étape déduit le logarithme discret de n'importe quel élément du groupe à partir des logarithmes déjà calculés.

Afin de mieux clarifier les étapes des algorithmes de calcul d'index, nous allons commencer par présenter l'algorithme d'Adleman, le premier algorithme publié de calcul d'index dans le contexte du logarithme discret. Le choix de cet algorithme se justifie de par sa simplicité, mais aussi du fait que les algorithmes qui lui ont succédé peuvent être considérés comme des évolutions ou des optimisations de cet algorithme.

1.4.3 Algorithme d'Adleman

Cet algorithme a été inventé par Adleman en 1979 [Adl79]. Ici, nous présentons l'algorithme pour un groupe multiplicatif d'un corps fini premier $(\mathbb{F}_p)^{\times}$ engendré par g avec p un nombre premier. Néanmoins, la méthode a été généralisée à n'importe quel type de corps fini $\mathbb{F}_{p^k}$ [AD93]. L'algorithme est composé de 4 étapes :

Choix de la base de facteurs

La base de facteurs doit être un petit ensemble d'éléments du groupe tel qu'une grande proportion des éléments du groupe ont des facteurs premiers qui appartiennent tous à cette base. Ceci nous amène à définir la propriété de « friabilité ». Cette notion concerne aussi bien les entiers que les polynômes.

Définition 1.6 (Entier friable)**.** *Un entier est dit B-friable si tous ses facteurs premiers sont inférieurs à B.*

Définition 1.7 (Polynôme friable)**.** *Un polynôme est dit* B-friable *si sa factorisation en polynômes irréductibles ne comprend que des polynômes dont les degrés sont inférieurs ou égal à B.*

La borne B est appelée *borne de friabilité*. Dans l'algorithme d'Adleman, nous avons uniquement la friabilité des entiers. Une fois une borne de friabilité B choisie, la *base de facteurs* correspond à l'ensemble $\{-1\} \cup \{2, 3, \ldots, p_r\}$ contenant $r+1$ nombres premiers, tous inférieurs ou égaux à B.

Crible

Cette étape est aussi connue sous l'appellation de *collecte de relations*. Nous cherchons au moins $r+1$ relations linéairement indépendantes entre des éléments de la base de facteurs et des puissances de g, de la forme

$$g^x \equiv (-1)^{e_0} 2^{e_1} \ldots {p_r}^{e_r} \pmod{p}. \tag{1.4}$$

Nous avons besoin de tester la friabilité d'un élément et, si l'élément est friable, de pouvoir trouver sa factorisation. La factorisation d'un élément friable nous permet d'établir une relation. N'importe quel algorithme qui fournit un test de friabilité et la factorisation peut être utilisé. À titre d'exemple, nous pourrons mentionner l'algorithme *Trial division* ou des algorithmes qui sont plus rapides dans le cas d'une factorisation générale, en l'occurrence l'algorithme ECM (Elliptic Curve Method [Len87, BBB+12]) ou l'algorithme du crible quadratique [Pom84]. Toutefois, l'algorithme ECM est le plus efficace quand l'élément a de petits facteurs, ce qui est justement le cas lorsque celui-ci est friable.

Algèbre linéaire

En appliquant l'opération $\log_g$ sur l'équation 1.4, nous obtenons une relation qui implique les logarithmes des éléments de la base de facteurs et qui s'écrit sous la forme suivante

$$x \equiv e_0 \log_g(-1) + e_1 \log_g 2 + \cdots + e_r \log_g p_r \pmod{p-1}. \tag{1.5}$$

Nous formons avec les relations précédentes un système linéaire modulo $p-1$ et où les inconnues sont les logarithmes en base g des éléments de la base de facteurs. Le fait que nous disposions d'un nombre suffisant de relations va nous garantir de pouvoir calculer une solution du système. Résoudre le système correspond à calculer un élément non trivial du noyau de la matrice qui correspond au système. Une fois le système résolu, nous connaissons le logarithme discret de chaque élément de la base de facteurs.

Logarithme individuel

Cette dernière étape est aussi connue sous le nom de *descente*. Elle consiste à obtenir le logarithme discret de n'importe quel élément h du groupe à partir des logarithmes déjà calculés.

Nous cherchons un élément friable sur la base de facteurs de la forme $g^x h$, pour un x arbitraire. Quand cet élément est trouvé, nous pouvons alors écrire :

$$g^x h \equiv (-1)^{f_0} 2^{f_1} \dots {p_r}^{f_r} \pmod{p}. \tag{1.6}$$

Le logarithme discret k de h est alors calculé comme suit :

$$k = (f_0 \log_g(-1) + f_1 \log_g 2 + \dots + f_r \log_g p_r - x) \bmod (p-1). \tag{1.7}$$

Remarques

Bien choisir la taille de la base de facteurs est important. En effet, avec une base petite, les éléments friables sont plus rares et la tâche pour les trouver va être coûteuse. Si la base est grande, il faudra chercher beaucoup de relations et le système linéaire à résoudre sera plus grand.

Les étapes de *choix de la base de facteurs*, de *crible* et d'*algèbre linéaire* ne dépendent pas de l'élément h. Par conséquent, si nous sommes amenés à calculer plusieurs logarithmes discrets, seule l'étape de *logarithme individuel* est calculée plusieurs fois ; les autres étapes peuvent être donc considérées comme du pré-calcul.

En équilibrant les temps de calcul des phases de *crible* et d'*algèbre linéaire*, qui sont les étapes les plus coûteuses, la complexité totale de l'algorithme d'Adleman est égale en $L_p(\frac{1}{2}, \sqrt{2})$.

L'algorithme d'Adleman a été le précurseur en permettant de résoudre le problème du logarithme discret dans les corps premiers $\mathbb{F}_p$, avec une complexité en $L_p(\frac{1}{2}, \sqrt{2})$. Plus tard, différentes variantes de cet algorithme sont apparues et ont permis de le généraliser vers les autres corps et d'améliorer la complexité théorique. De nos jours, dans le cas général des corps finis $\mathbb{F}_{p^k}$, en fonction de comment se compare k par rapport à $\log p$, nous aurons des « régions » ; dans chacune d'entre elles, un certain algorithme de calcul d'index avec une certaine complexité est le plus efficace. Ainsi, la difficulté du problème du logarithme discret varie en fonction de la « région » (voir figure 1.5). Tous ces algorithmes sont des algorithmes cousins de l'algorithme NFS pour la factorisation.

Dans le prochaine section, nous allons présenter l'algorithme du crible du corps de fonctions (FFS pour Function Field Sieve) et l'algorithme du crible algébrique (NFS pour Number Field Sieve). Le choix de ces deux algorithmes parmi le spectre des algorithmes de calcul d'index est principalement motivé par le fait que les systèmes linéaires considérés dans les travaux exposés dans ce mémoire sont issus de calculs effectués avec ces deux algorithmes.

Nous n'allons pas fournir une description complète des algorithmes FFS et NFS, mais plutôt discuter de leurs spécificités par rapport au schéma de l'algorithme d'Adleman et les objets mathématiques considérés dans ces algorithmes.

1.4.4 Crible du corps de fonctions et crible algébrique

Crible du corps de fonctions pour les corps $\mathbb{F}_{p^k}$ de petite caractéristique

Les corps de petite caractéristique, en particulier de caractéristique 2, sont d'une grande importance du fait que l'arithmétique avec des petits corps de base, en particulier $\mathbb{F}_2$, peut

être implémentée efficacement. Un corps $\mathbb{F}_{p^k}$ peut être représenté de plusieurs façons. Ici, nous représentons le corps comme $\mathbb{F}_p[t]/\varphi(t)$, avec $\varphi(t)$ un polynôme irréductible de $\mathbb{F}_p[t]$ de degré k. Le polynôme $\varphi(t)$ est appelé polynôme de définition.

L'algorithme FFS [Adl94, AH99, JL02] est apparu en 1994, après l'algorithme de Coppersmith [Cop84] qui a permis de descendre en dessous de $L(\frac{1}{2})$ pour le cas particulier des corps de caractéristique 2, avec la complexité $L_{2^n}(\frac{1}{3}, 1.41)$. FFS s'applique aussi bien au cas de la caractéristique 2, qu'au cas de la caractéristique différente de 2, avec la complexité $L_{p^k}(\frac{1}{3}, 1.53)$. FFS a été longtemps considéré comme l'algorithme le plus efficace dans les corps de petite caractéristique. À partir de la fin de 2012, une série d'améliorations ont réduit dans un premier temps la complexité à $L_{p^k}(\frac{1}{4})$ [Jou13a, Jou13b, GGM+13]. Par la suite, un nouvel algorithme, inventé par Barbulescu, Gaudry, Joux et Thomé, a montré qu'il était possible de calculer des logarithmes discrets dans des corps de petite caractéristique en un temps quasi-polynomial qui s'exprime en $(\log p^{dk})^{O(\log\log p^{dk})}$, pour un petit entier d choisi d'une manière appropriée [BGJ+14].

L'algorithme FFS conserve les même étapes que l'algorithme d'Adleman, et y ajoute une étape dite de *Sélection polynomiale*. La tâche de la sélection polynomiale consiste à déterminer un couple de polynômes à deux variables $f, g \in \mathbb{F}_p[t][x]$, tels que leur résultant $\mathrm{Res}_x(f, g)$ contienne dans sa factorisation un polynôme irréductible de degré k. Ce polynôme irréductible correspond au polynôme $\varphi(t)$. Le choix des polynômes f et g est d'une part utile pour la représentation du corps, mais aussi intervient dans la recherche de relations et dans le logarithme individuel.

Le crible pour FFS adopte une autre stratégie que l'algorithme d'Adleman pour la collecte de relations. L'idée est de représenter le corps fini de deux façons différentes, comme représenté dans le diagramme de la figure 1.3. On prend un élément ϕ dans $\mathbb{F}_p[t][x]$ qui s'écrit sous la forme $\phi(t, x) = a(t) - b(t)x$. Ce élément est envoyé dans les deux corps de fonctions $\mathbb{F}_p[t][x]/f(x)$ et $\mathbb{F}_p[t][x]/g(x)$. Le diagramme étant commutatif, ceci permet d'obtenir une égalité dans $\mathbb{F}_{p^k}$. Comme pour n'importe quel algorithme de calcul d'index, il nous faut définir un critère de friabilité pour obtenir des relations qui ne font intervenir que des éléments appartenant à une certaine base de facteurs. FFS définit un critère de friabilité dans les deux ensembles ; plus précisément, on cherche des éléments ϕ tels que les résultants $\mathrm{Res}_x(\phi, f)$ et $\mathrm{Res}_x(\phi, g)$, où $\phi(t, x) = a(t) - b(t)x$, soient friables par rapport à une borne. Le crible est alors sensible au choix des polynômes f et g, en particulier à la taille de leurs coefficients. Le diagramme de la figure 1.3 résume les structures mathématiques impliquées.

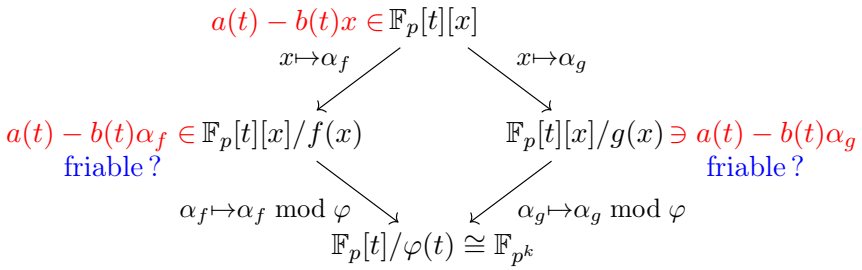


FIGURE 1.3 – Diagramme de crible pour FFS.

À l'issue de la phase de crible, nous retrouvons une phase d'algèbre linéaire, puis une phase dite de « descente » destinée au calcul d'un logarithme arbitraire. Ces phases sont très proches des procédures que nous avons décrites pour l'algorithme d'Adleman.

Crible algébrique pour les corps premiers $\mathbb{F}_p$

Les corps $\mathbb{F}_{p^k}^\times$, où p est grand, forment le domaine où le crible algébrique s'applique. Dans le cas particulier des corps premiers, ceci reste valide. L'algorithme du crible algébrique est une adaptation de l'algorithme du même nom pour la factorisation et hérite aussi de la structure générale d'un algorithme de calcul d'index pour le calcul des logarithmes discrets.

La première version de l'algorithme du crible algébrique pour le logarithme discret a été publiée par Gordon en 1993 [Gor93], puis plusieurs évolutions ont été proposées [SWD96, JL03]. Aujourd'hui, il a une complexité en $L_{p^k}(\frac{1}{3}, 1.9)$ [BP14, Sch00]. NFS est resté l'algorithme le plus efficace en grande caractéristique. Il existe une variante de NFS, appelée SNFS (Special Number Field Sieve), qui s'applique lorsque p est d'une forme spéciale. L'algorithme SNFS réduit la complexité de la résolution à $L_{p^k}(\frac{1}{3}, 1.53)$ [JP13, Sem02].

Le crible pour NFS suit la même approche que l'algorithme FFS, en utilisant un diagramme commutatif pour avoir deux représentations du corps $\mathbb{F}_p$. NFS choisit deux polynômes irréductibles f et g, à coefficients entiers, qui partagent une racine commune m dans $\mathbb{F}_p$. On suppose que g est degré 1. On associe aux polynômes f et g les deux anneaux $\mathbb{Z}[m]$ et $\mathbb{Z}[\alpha]$, où α est la racine de f dans le corps de nombres défini par f. Le crible cherche des polynômes de $\mathbb{Z}[x]$ de la forme $\phi = a - bx$ et tels que leurs résultants $\mathrm{Res}(\phi, f)$ et $\mathrm{Res}(\phi, g)$ soient friables. Dans le diagramme de la figure 1.4, nous schématisons ce procédé de collecte de relations. Dans le coté gauche, dit coté « rationnel », intervient une factorisation d'un nombre rationnel $a - bm$, alors que dans le coté gauche, dit coté « algébrique », il s'agit de décomposer l'idéal $(a - b\alpha)\mathcal{O}_{\mathbb{Q}(\alpha)}$, où $\mathcal{O}_{\mathbb{Q}(\alpha)}$ est l'anneau des entiers du corps de nombre $\mathbb{Q}(\alpha)$ défini par le polynôme f.

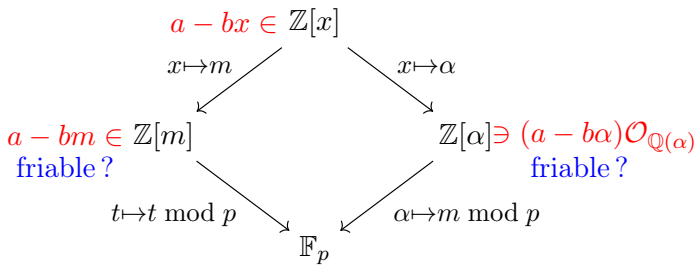


FIGURE 1.4 – Diagramme de crible pour NFS.

Une fois les relations collectées, on résout un système linéaire. À la fin, une descente suivant le schéma habituel est effectuée.

Comparaison des deux algorithmes de crible

Les algorithmes FFS et NFS partagent beaucoup de similitudes. L'algorithme FFS est plus rapide que l'algorithme NFS (voir les complexités dans la sous-section 1.4.6). En effet, lors de la sélection du polynôme f, il y a moins de contraintes avec l'algorithme FFS qu'avec l'algorithme NFS. Par conséquent, on arrive avec FFS à prendre un polynôme avec des coefficients petits, alors que ce n'est pas le cas avec l'algorithme NFS.

Les algorithmes FFS et NFS ne considèrent pas les mêmes objets mathématiques. Pendant le crible, FFS considère la factorisation des polynômes, alors que NFS considère la factorisation des entiers. Par conséquent, le test de friabilité de NFS est sous-exponentiel, alors que celui de

FFS est polynomial. Une autre différence majeure entre ces deux algorithmes concerne l'algèbre linéaire et plus spécifiquement, la nature des coefficients présents dans les systèmes linéaires. En effet, les systèmes linéaires issus de FFS contiennent des coefficients qui correspondent à des exposants et sont par conséquent de taille petite (inférieur à un mot machine), alors que les systèmes linéaires issus de NFS contiennent en outre des coefficients dont la taille est proche de celle de p. Ces coefficients plus grands correspondent aux caractères ℓ-adiques définis par Schirokauer [Sch93]. La présence de ces coefficients fait qu'à taille de système linéaire égale, l'algèbre linéaire correspondant à NFS est plus difficile qu'une algèbre linéaire correspondant à FFS. Nous reviendrons sur ce point plus loin dans la sous-section 3.2.1.

1.4.5 Problèmes difficiles

Dans cette sous-section, nous présentons les étapes calculatoirement difficiles dans les calculs des algorithmes de calcul d'index. Dans les algorithmes de calcul d'index apparus avant 2012, et dont la complexité est en $L(\frac{1}{3})$, les étapes de *crible* et d'*algèbre linéaire* sont les plus coûteuses en temps et en ressources. Dans les nouveaux algorithmes apparus après 2012 et dont la complexité est meilleure que $L(\frac{1}{3})$, l'étape de *logarithme individuel* devient aussi une étape coûteuse.

Crible

Le crible cherche des éléments qui vérifient une ou plusieurs conditions de friabilité par rapport à une ou plusieurs bornes données. Nous avons déjà vu avec les deux algorithmes FFS et NFS que cette étape est spécifique à chaque algorithme.

Pour pouvoir explorer l'espace des éléments à tester, il est commun d'utiliser les *réseaux euclidiens* (*lattices* en anglais) pour réduire l'espace de candidats potentiels [Pol93]. D'autres techniques plus spécifiques telles que les *special-*$\mathfrak{q}$ sont utilisées dans le contexte de l'algorithme FFS. Des informations sur les techniques d'accélération du crible pour l'algorithme FFS, ainsi que sur leurs apports en pratique peuvent être consultés dans [DGV13].

L'étape du crible est coûteuse. Dans les calculs récents de logarithmes discrets, le crible collecte des milliards d'éléments. Toutefois, cette étape possède l'avantage de bien se paralléliser. La recherche des relations peut être distribuée sur un grand nombre de machines. En effet, le grand espace dans lequel nous criblons peut être divisé en plusieurs sous-espaces, qui sont explorés en parallèle.

Algèbre linéaire

L'algèbre linéaire issue des algorithmes de calcul d'index consiste à résoudre un système d'équations linéaires homogènes. Le système est « grand » et « creux ». Par « creux », nous voulons dire que le nombre de coefficients non nuls dans une ligne est très faible. Pour donner un ordre de grandeur, il s'agit de systèmes qui contiennent des centaines de milliers voire des millions d'équations et où une équation contient uniquement quelques centaines de coefficients. Le système est défini dans $\mathbb{Z}/\ell\mathbb{Z}$, où ℓ est un grand nombre premier, qui correspond à l'ordre du groupe multiplicatif ou du sous-groupe multiplicatif dans lequel on regarde le logarithme discret.

L'étape d'algèbre linéaire a toujours été considérée comme le goulot d'étranglement des calculs de logarithme discret. On pourrait citer l'exemple du calcul mené par Gordon et McCurley dans $\mathbb{F}_{2^{503}}$ qui n'a pas abouti parce qu'ils n'arrivaient pas à résoudre le système linéaire [GM93]. L'algèbre linéaire est plus difficile dans le contexte du logarithme discret que dans le contexte

de la factorisation d'entiers. La différence est que, pour la factorisation, les systèmes sont définis sur $\mathbb{F}_2$, alors que pour le logarithme discret, les systèmes sont définis sur des grands corps finis. Les différents défis liés à cette algèbre linéaire sont la taille de la matrice, son caractère creux qui doit être bien exploité et l'arithmétique sur le grand corps fini. Pour faire face à ces défis, il est pertinent de pouvoir créer du parallélisme dans les calculs d'algèbre linéaire, car contrairement à l'étape précédente, il n'y a pas un parallélisme direct qui se dégage de la résolution des systèmes linéaires creux. Dans une grande partie de ce mémoire, nous allons détailler les défis et exposer différents approches et solutions utilisées pour accélérer cette étape.

Logarithme individuel

L'objectif de cette étape est d'exprimer l'élément dont on calcule le logarithme discret en fonction des éléments de la base de facteurs. L'approche consiste à exprimer un élément « grand » en fonction de plus petits éléments.

Comme les objets mathématiques varient entre les algorithmes de calcul d'index, il est difficile de décrire un schéma général de l'étape de *logarithme individuel*. Si on se place dans le cadre de l'algorithme FFS, la descente correspond à exprimer un polynôme de degré D avec des polynômes de degré $\sqrt{BD}$, où B est la borne de friabilité et à trouver une relation entre leurs logarithmes. Différentes stratégies de « descente » sont utilisées et combinées, telles que les fractions continues ou les bases de Gröbner.

Sélection polynomiale et filtrage

Dans les chantiers d'optimisation des algorithmes et des implémentations de calcul de logarithme discret, il est aussi important de mentionner les étapes de *sélection polynomiale* et de *filtrage*. Ces étapes ne sont pas calculatoirement difficiles, mais leurs résultats ont des impacts importants sur l'efficacité des étapes suivantes dans l'algorithme. La *sélection polynomiale* cherche des polynômes avec certaines propriétés qui influencent la difficulté de la recherche des relations [JL02, Bar13]. L'étape de *filtrage* agit sur les relations trouvées dans le *crible* de sorte à obtenir le plus petit système linéaire à résoudre [Bou13].

1.4.6 Panel des algorithmes de calcul d'index et leurs domaines de validité

Nous avons précédemment mentionné que, pour un corps fini donné $\mathbb{F}_{p^k}$, la taille de la caractéristique p va conditionner le choix d'un certain algorithme de calcul d'index. Plus précisément, les limites des régions de validité des algorithmes sont définies en fonction de comment se comparent k et p (voir figure 1.5).

p « petit »

On rappelle que ce cas correspond à $\log p$ petit devant k, exemple typique, $\mathbb{F}_{2^k}$. Avant 2012, FFS était l'algorithme le plus efficace dans cette région avec la complexité $L_{p^k}(\frac{1}{3}, 1.53)$. Actuellement, l'algorithme QPA (Quasi Polynomial Algorithm) de Barbulescu, Gaudry, Joux et Thomé, dont la complexité permet de calculer le logarithme discret avec une complexité quasi polynomiale. Cet algorithme et ses variantes sont les plus efficaces lorsque l'extension k a une forme particulière [BGJ+14]. Le cas où l'extension k est première représente le cas le plus difficile pour les nouveaux algorithmes et où FFS reste compétitif avec eux.

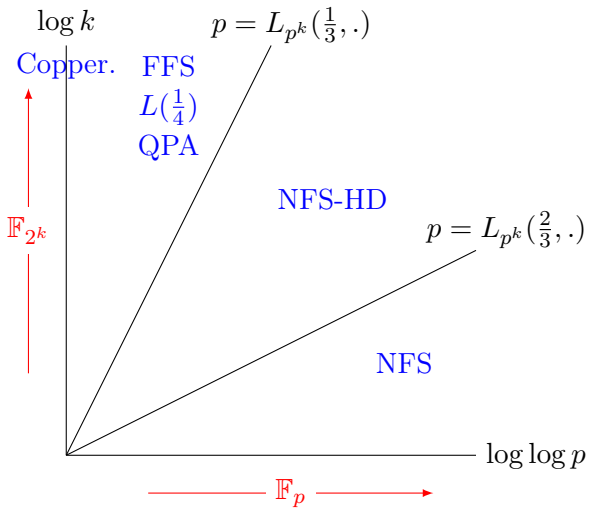


FIGURE 1.5 – Les domaines des algorithmes de calcul d'index pour $\mathbb{F}_{p^k}^\times$.

p « grand »

On rappelle que ce cas correspond à k petit devant $\log p$, exemple typique, $\mathbb{F}_p$. À ce jour, l'algorithme le plus efficace dans cette région est l'algorithme du crible algébrique (NFS), avec une complexité en $L_{p^k}(\frac{1}{3}, 1.9)$ [BP14]. La variante SNFS réduit la complexité à $L_{p^k}(\frac{1}{3}, 1.53)$ [JP13, Sem02].

p « moyen »

Entre les deux régions précédentes, nous trouvons le cas où $\log p$ et k sont de même ordre de grandeur. Dans cette région, une variante de l'algorithme du crible algébrique, dite NFS-HD, s'applique. Ce cas est connu pour être le plus difficile. Actuellement, la complexité de l'algorithme est descendue à $L_{p^k}(\frac{1}{3}, 2.24)$ [BP14, BGG+15].

1.4.7 Records de calcul de logarithme discret

Dans la table 1.1, nous recensons les derniers records de calcul de logarithme discret dans chaque domaine. Nous précisons aussi les données relatives aux records précédents : ainsi le lecteur verra quel est le « surcoût » de passer d'un calcul à un autre. Ce surcoût peut être en terme d'heures CPU si le même algorithme a été utilisé, mais peut aussi correspondre à l'utilisation d'un algorithme plus récent et plus efficace.

De l'observation de la table, il apparaît que le problème du logarithme discret dans les corps finis de petite caractéristique est devenu le moins difficile, grâce aux récentes améliorations théoriques. Dans le cas de la caractéristique 2, où l'extension est composée, le record est de 9234 bits, calculé avec l'algorithme $L(\frac{1}{4})$. Dans le cas où l'extension est un nombre premier, ce qui présente une difficulté plus importante, le record actuel est de 1279 bits, calculé avec une approche qui combine l'algorithme QPA et l'algorithme $L(\frac{1}{4})$, alors que le record précédent était de 809 bits calculé avec l'algorithme FFS. À ce jour, on situerait autour de 900 bits le seuil à partir duquel les nouveaux algorithmes battent FFS pour le cas où l'extension est un nombre premier. Pour la caractéristique 3, le record est de 3796 bits, calculé avec une variante de l'algorithme QPA.

Groupe	Date	Taille (bits)	Algorithme	Coût (h CPU)	Auteurs
$\mathbb{F}_{2^n}^\times$	01/2014	9234	$L(\frac{1}{4})$	398 k	R. Granger, T. Kleinjung et J. Zumbrägel [GKZ14]
	05/2013	6168	$L(\frac{1}{4})$	550	A. Joux [Jou13c]
$\mathbb{F}_{2^p}^\times$	10/2014	1279	QPA & $L(\frac{1}{4})$	35 k	T. Kleinjung [Kle14]
	04/2013	809	FFS	19.3 k	Équipe Caramel [BBD+13]
$\mathbb{F}_{3^n}^\times$	09/2014	3796	QPA	8.6 k	A. Joux et C. Pierrot [JP14]
	01/2014	1303	$L(\frac{1}{4})$	920	G. Adj, A. Menezes, T. Oliveira et F. Rodríguez-Henríquez [AMO+14]
p moyen	01/2013	1425	FFS	32 k	A. Joux [Jou13d]
	12/2012	1175	FFS	32 k	A. Joux [Jou12]
$\mathbb{F}_{p^2}^\times$	06/2014	529	NFS	1.92 k	R. Barbulescu, P. Gaudry, A. Guillevic et F. Morain [BGG+14]
$\mathbb{F}_p^\times$	06/2014	596	NFS	1.13 M	C. Bouvier, P. Gaudry, L. Imbert, H. Jeljeli et E. Thomé [BGI+14]
	02/2007	530	NFS	29 k	T. Kleinjung [Kle07]

TABLE 1.1 – Les records de calcul de logarithme discret sur les corps finis.

Les records pour les corps finis de moyenne caractéristique et à grande caractéristique sont respectivement de 1425 et de 596 bits. Ceci est dû au fait que dans ces régions, on ne connaît pas d'algorithmes qui ont une meilleure complexité que $L(\frac{1}{3})$.

Dans le chapitre 7, nous détaillerons deux calculs ; un premier relatif à un record en petite caractéristique sur une extension première et le second correspond à un record en grande caractéristique.

1.5 Conclusion

À travers ce chapitre, nous avons donné un aperçu du problème du logarithme discret dans les groupes pour lesquels il y a un intérêt cryptographique. Il apparaît que, depuis l'apparition du protocole d'échange de clé Diffie-Hellman il y a plus que trois décennies, le logarithme discret était l'objet d'une part d'avancées algorithmiques majeures mais aussi d'optimisations pratiques au niveau des implémentations. Aujourd'hui, ce problème est encore considéré comme difficile pour les groupes multiplicatifs des corps finis de moyenne et de grande caractéristique et pour les groupes additifs des courbes elliptiques et des courbes hyperelliptiques [Die11].

Chapitre 2

Calcul à hautes performances (HPC)

Ce chapitre est le chapitre « technologique » du mémoire, dans lequel nous allons introduire et décrire les concepts et les mécanismes liés aux architectures que nous allons utiliser pour accélérer le calcul d'algèbre linéaire. Dans un premier temps, nous allons répondre à la question : quelles sont les architectures pensées pour le calcul parallèle et qui répondent aux exigences des calculs que nous considérons ? Par la suite, nous allons décrire le modèle de programmation CUDA, qui permet d'utiliser les processeurs graphiques (GPU) NVIDIA pour accélérer une application. La dernière partie du chapitre sera consacrée à l'exécution du calcul sur un cluster. Plus spécifiquement, nous allons discuter les aspects liés aux communications, lorsque les nœuds de calcul sont des CPUs ou des GPU.

Sommaire

2.1 Calcul à hautes performances et algèbre linéaire

2.1.1 Architectures pensées pour le calcul parallèle

Avant l'apparition des premières plate-formes de calcul parallèle, un programme résolvant un problème correspondait à un calcul *séquentiel*. Dans ce cadre, le programme est composé d'une série d'instructions, exécutées sur une seule ressource, typiquement un processeur d'une machine mono-cœur. Une seule instruction est exécutée à la fois sur la ressource. Plus tard, les architectures ont permis la résolution *parallèle* du calcul, c'est-à-dire l'utilisation simultanée de plusieurs ressources. Le problème est alors divisé en différentes parties qui peuvent être résolues en parallèle. Chaque partie est transformée en une suite d'instructions. Chaque instruction est exécutée sur une ressource.

Aujourd'hui, nous trouvons un large ensemble d'architectures matérielles adaptées à la résolution parallèle, parmi lesquelles :

— Les processeurs multi-cœurs : ce sont les processeurs dont la puce contient au moins deux unités d'exécution (cœurs). Les configurations typiques sont 4, 6 ou 8 cœurs par puce.
— Les processeurs *many-cœurs* : ce sont des architectures multi-cœurs où le nombre de cœurs est de l'ordre de dizaines à quelques centaines, par exemple la famille des coprocesseurs Intel Xeon Phi.
— Les processeurs vectoriels : ce sont des processeurs pourvus de fonctionnalités de parallélisme de données ; c'est-à-dire qu'ils peuvent exécuter en parallèle une même instruction sur des données distinctes.
— Les processeurs graphiques (GPU) : les processeurs graphiques sont intégrés dans les cartes graphiques pour accélérer les traitements graphiques qui sont généralement hautement parallèles (*embarrassingly parallel*).
— Les circuits logiques configurables : ce sont des architectures reconfigurables, à l'image des FPGA, où le programmeur configure le circuit à l'aide de blocs logiques et de ressources de routage. Ces architectures permettent d'exploiter le parallélisme matériel.
— Les clusters de calcul : un cluster de calcul est un ensemble de machines interconnectées par un réseau de communication, généralement à haut débit.

Les architectures que nous avons décrites dans la liste précédente, qui n'est pas exhaustive, se distinguent de par le niveau du parallélisme qu'elles exploitent :

— instruction ;
— donnée ;
— mémoire ;
— tâche.

Les architectures parallèles sont généralement classées selon l'organisation des données, des processus et des instructions. Flynn a introduit dans sa taxonomie [Fly72] 4 modes d'exécution possibles :

— SISD (Single Instruction Single Data) : Ce mode correspond au fonctionnement traditionnel d'une machine mono-processeur qui n'exploite aucun parallélisme.
— SIMD (Single Instruction Multiple Data) : Une même instruction est effectuée en parallèle sur des données différentes. On trouve ce mode dans les processeurs vectoriels.
— MISD (Multiple Instructions Single Data) : Plusieurs instructions agissent en parallèle sur une même donnée. Il n'existe pas beaucoup d'architectures qui implémentent cette catégorie.

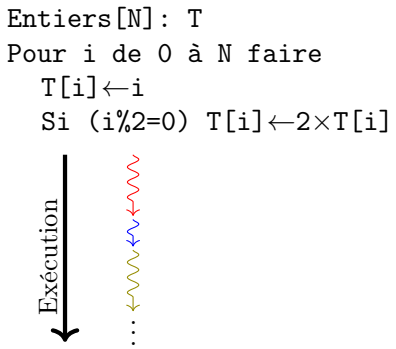
— MIMD (Multiple Instructions Multiple Data) : Ce mode est utilisé dans les machines multi-processeurs, où plusieurs processeurs exécutent différentes instructions sur différentes données.

Plus tard, cette classification a été complétée par les deux sous-catégories suivantes du mode MIMD :

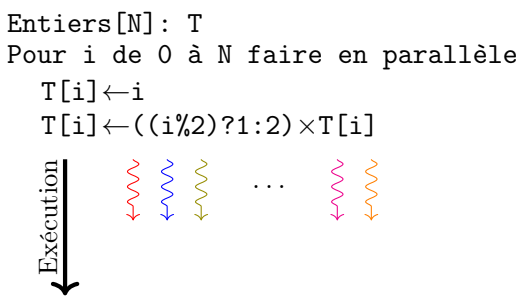
— SPMD (Single Program Multiple Data) : Un même programme (code) est exécuté par les processeurs, mais les instructions effectuées par chaque processeur peuvent varier. On trouve ce mode dans les processeurs graphiques.

— MPMD (Multiple Programs Multiple Data) : On rencontre ce mode par exemple sur un processeur bi-cœur, où on exécute un programme différent sur chaque cœur.

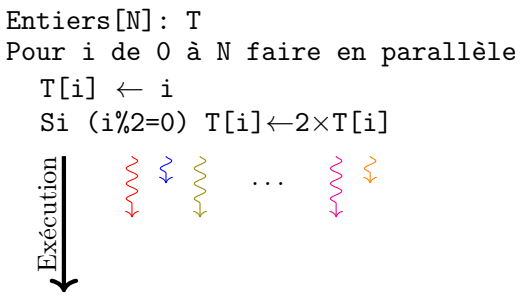
Dans la figure 2.1, nous illustrons le fonctionnement des modes SISD, SIMD et SPMD sur des programmes (écrits en pseudo-code), où on traite un tableau d'entiers. Dans la sous-figure 2.1a, il y a un traitement purement séquentiel (SISD). Dans la sous-figure 2.1b, les mêmes instructions sont effectuées en parallèle sur les éléments du tableau (SIMD). Dans la sous-figure 2.1c, le même programme n'exécute pas les mêmes instructions, parce que nous avons ajouté un test (SPMD).



(a) SISD



(b) SIMD



(c) SPMD

FIGURE 2.1 – Modes d'exécution : SISD, SIMD et SPMD.

2.1.2 Architectures appropriés à notre type d'algèbre linéaire

Dans cette section, nous allons réfléchir sur les caractéristiques que doit satisfaire une architecture appropriée à notre contexte applicatif d'algèbre linéaire. Nous avons mentionné à la sous-section 1.4.5 que notre problème est la résolution d'un système linéaire grand et creux sur un grand corps fini (voir section 3.2 pour plus de détails sur les caractéristiques des entrées). La

brique de base du calcul d'algèbre linéaire, si l'on utilise des algorithmes *boîte noire* comme ce sera le cas de ceux présentés au chapitre 3, est l'opération de multiplication de la matrice creuse par un vecteur ou par un bloc de vecteurs. Nous avons besoin d'une architecture qui exploite différents niveaux du parallélisme (voir section 3.3). L'architecture doit satisfaire les critères suivants :

1. La taille importante de la matrice et des vecteurs fait que nous avons besoin de pouvoir stocker et traiter d'une manière efficace des données dont les tailles peuvent atteindre quelques dizaines de gigaoctets.
2. La matrice est creuse, avec des zones très creuses. Ceci fait que les accès sur le ou les vecteurs d'entrée risquent d'être irréguliers. C'est pourquoi nous avons besoin de mémoires et de mécanismes de cache qui permettent de limiter les pénalités dues à l'irrégularité des accès.
3. Les opérations arithmétiques sont effectuées sur un grand corps fini. Il est important que l'architecture logicielle permette d'écrire des briques de calcul arithmétique spécifiques et efficaces. De plus, cette arithmétique peut être accélérée en utilisant une représentation qui exploite la vectorisation (voir chapitre 6).
4. Si nous envisageons de distribuer l'algèbre linéaire sur un cluster de machines, les calculs ne peuvent pas être complètement indépendants et sont donc amenés à partager des données de taille importante. C'est pourquoi nous avons besoin d'un réseau et de mécanismes de communication efficaces en terme de latence et de débit pour minimiser les coûts de communication. Nous pourrons aussi mettre en place un parallélisme de tâches, et dans ce cas, nous aurons besoin d'un modèle approprié.

À partir de cette description, nous concluons que les circuits configurables ne sont pas appropriés à cause de leur mémoire limitée (le critère 1 est non satisfait). Par contre, les CPU munis de plusieurs cœurs et d'unités vectorielles, ainsi que des accélérateurs de type GPU ou processeur *many-cœurs* se positionnent comme des architectures appropriées (les critères 1, 2 et 3 sont satisfaits ; le critère 1 peut être contraignant pour les GPU à partir de très grandes tailles).

Dans le cadre de ce travail, nous allons étudier l'utilisation des CPU multi-cœurs et vectoriels et des GPU pour le problème d'algèbre linéaire.

2.2 CPU multi-cœurs et vectoriels

Un CPU multi-cœurs est un processeur qui contient deux cœurs ou plus gravés sur la même puce. Le premier processeur multi-cœur est apparu au milieu des années 80 avec le CPU bi-cœur de Rockwell International. Dès le début des années 2000, différents constructeurs tels que IBM, AMD, Intel ont commencé à commercialiser des processeurs multi-cœurs. L'idée d'introduire plusieurs unités de calcul dans un même processeur permet d'implémenter physiquement le *multi-processing* et répond à la problématique du besoin continu d'augmenter la fréquence du CPU.

Un processeur vectoriel est un processeur pourvu d'instructions vectorielles, c'est-à-dire d'instructions appropriées au traitement parallèle de données d'un bloc de taillé fixée, qu'on appelle vecteur. Par le passé, les architectures vectorielles ont été développées pour les supercalculateurs, par exemple pour les machines Cray et ILLIAC. De nos jours, la vectorisation est exploitée dans les processeurs incorporant des instructions SIMD, par exemple dans tous les processeurs Intel à partir du Pentium III, dans les processeurs POWER de IBM et également dans le processeur CELL de la PlayStation 3.