# Optimisation des multiplications utilisées en cryptographie

François Courbier
courbier@enst.fr

25 juin 2014

Dans le but d'optimiser le calcul des multiplications utilisées en cryptographie, typiquement des multiplications de polynômes et d'éléments de corps finis, un récent travail de recherche mené par l'INRIA de Nancy[1] a débouché sur la mise en évidence d'un algorithme qui permet la recherche de formules de multiplication avec un minimum de multiplications de scalaires.
Le présent travail a consisté à implémenter cet algorithme de manière à étendre les résultats de l'INRIA (optimisation du temps de calcul et possibilité de travailler dans des corps finis de caractéristique quelconque - notamment 4-).

## Table des matières

# 1 Introduction

Un équipe de recherche de l'INRIA de Nancy a établi un algorithme qui permet de rechercher des formules pour les multiplications de polynômes et d'éléments de corps finis, qui minimisent le nombre de multiplications de scalaires. Ainsi, attendu que les multiplications de scalaires sont le facteur déterminant dans le temps de calcul d'une multiplication cryptographique, les formules permettent d'optimiser la vitesse de calcul de ce type d'opération.
Dans l'article de référence, l'algorithme en question est décrit in extenso, et des indications sur l'implémentation sont données. Dans les résultats publiés par l'INRIA(processeur de 2,2Ghz), on remarque que les corps de caractéristique strictement supérieure à 3 n'ont pas été implémentés.
Les objectifs du projet sont donc d'implémenter l'algorithme, en langage Python dans un premier temps (avec la librairie Numpy), tout en permettant de travailler avec des caractéristiques quelconques. Dans la phase d'optimisation du programme, une version en langage C a aussi été écrite, dans un souci de performance et de structure de données (en particulier pour les listes chaînées).

# 2 L'optimisation recherchée

## 2.1 Le levier d'optimisation

### 2.1.1 Les multiplications concernées

Soient $n, m, l, q$ des entiers non nuls, et $q \geq 2$ (notations utilisées jusqu'à la fin du présent article).
Les multiplications ciblées pas l'algorithme sont les multiplications de polynômes $K^n \times$

$K^m \to K^l$ (exemple de l'algorithme de chiffrement AES), et les multiplications dans les extensions de corps finis de type $F_{q^n}$ (exemple des courbes elliptiques).

### 2.1.2 Minimiser le nombre de multiplications de scalaires

Une multiplication cryptographique n'est autre qu'une application bilinéaire ($K^n \times K^m \to K^l$). Et pour chaque dimension de l'espace d'arrivée, la composante de l'application est une forme bilinéaire ($K^n \times K^m \to K$).
Pour l'ensemble des formes bilinéaires, on souhaite minimiser le nombre de multiplications de scalaires nécessaires, qui détermine la rapidité de la multiplication.

### 2.1.3 L'objectif de l'algorithme

Il s'agit d'exprimer les formes bilinéaires en fonction d'un certain nombre k de formes bilinéaires n'ayant qu'une multiplication de scalaires (pour $i \in [\![0, 2^n - 2]\!]$ et $j \in [\![0, 2^m - 2]\!]$) :

$$\begin{array}{rccccl} g_{ij} & : & K^n \times K^m & \to & K & \\ & & (A, B) & \mapsto & (\sum_{u=0}^{n-1} e_{ui} a_u)(\sum_{v=0}^{m-1} f_{vj} b_v) & \text{(les } e_{ui} \text{ et les } f_{vj} \text{ étant dans \{0,1\})} \end{array}$$

Si k est inférieur au nombre de multiplications de scalaires exécutées habituellement, alors l'optimisation est réussie.

## 2.2 Description de l'algorithme

### 2.2.1 Principe de l'algorithme

En entrée :
— $k$ : le nombre de multiplications de scalaires souhaité
— $G$ : les formes bilinéaires avec une seule multiplication de scalaires
— $T$ : les formes bilinéaires pour chaque dimension d'arrivée de la multiplication
Tous les vecteurs sont exprimés sur la base "canonique" des formes bilinéaires :

$$\begin{array}{rccccl} \gamma_{ij} & : & K^n \times K^m & \to & K & \\ & & (A, B) & \mapsto & a_i b_j & \text{(pour } i \in [\![0, n-1]\!] \text{ et } j \in [\![0, m-1]\!] \text{ )} \end{array}$$

En sortie : tous les sous-espaces $W$ de dimension $k$ qui sont générés par des éléments de $G$ et qui contiennent $T$.

Première idée : tester tous les espaces générés par $k$ éléments de $G$, en vérifiant que la dimension est bien $k$, et qu'ils contiennent $T$. Cela donne une complexité de $\binom{\#G}{k}$.

Idée de l'algorithme : partir de $T$, et rajouter récursivement un élément de $G$ qui n'est pas dans $T$, de manière à augmenter la dimension de l'espace $W$ obtenu.
Lorsque la dimension de $W$ est $k$, alors c'est un candidat potentiel, et on vérifie qu'une famille de $k$ éléments linéairement indépendants de $G$ génère cet espace. On améliore ainsi la complexité, qui est de $\binom{\#G}{k-\dim T}$.

### 2.2.2 Pseudo-code de l'algorithme

On travaille donc dans un certain espace vectoriel $V$.
$G$ est un ensemble de vecteurs de V.
$T$ est un sous-espace de $V$ (inclus dans vect($G$)).
$k$ est tel que $\dim T \leq k \leq \mathrm{rk}(G)$.

$S \leftarrow \emptyset$
**procedure** expand_subspace($W$)
  **if** $\dim W = k$ **and** $\mathrm{rk}(W \cap G) = k$ **then**
    $S \leftarrow S \cup W$
  **else if** $\dim W < k$ then
    for each $g \in G \setminus W$ do
      expand_subspace($W \bigoplus \mathrm{vect}(g)$)
**end procedure**
expand_subspace($T$)
**return** $S$

On remarque au passage que l'algorithme se concentre sur les espaces solutions, et non sur l'ensemble des formules possibles (toutes les familles possibles de $g \in G$ qui génèrent un espace $W$ solution). Dans le cadre du projet, les objectifs ont donc été de se référer aux espaces solutions (trouver l'ensemble des formules possibles à partir d'un espace solution étant considéré comme un problème facile).

## 2.3 Exemple avec la multiplication de polynômes $K_2[X] \times K_2[X] \to K_3[X]$

### 2.3.1 Construction de $T$

Dans cette section, et pour l'ensemble de l'article, on manipule indifféremment un espace et sa matrice (ie une famille de vecteurs qui génère l'espace).

$K$ est un corps.
Soient $A \in K_2[X]$ et $B \in K_2[X]$.
$A = a_0 + a_1 X$ avec $a_0$ et $a_1$ dans $K$
$B = b_0 + b_1 X$ avec $b_0$ et $b_1$ dans $K$
$A \times B = a_0 b_0 + (a_0 b_1 + a_1 b_0) X + a_1 b_1 X^2$

On en déduit $T$ dans la base des formes bilinéaires de type :

$$\gamma_{ij} \quad : \quad \begin{array}{ccc} K^2 \times K^2 & \to & K \\ (A, B) & \mapsto & a_i b_j \end{array} \quad (\text{pour } (i,j) \in [\![0,1]\!]^2).$$

$$T = \begin{array}{c} \begin{array}{cccc} {\scriptstyle a_0b_0} & {\scriptstyle a_0b_1} & {\scriptstyle a_1b_0} & {\scriptstyle a_1b_1} \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \begin{array}{c} {\scriptstyle a_0b_1} \\ {\scriptstyle a_0b_1 + a_1b_0} \\ {\scriptstyle a_1b_1} \end{array}$$

Ainsi, chaque vecteur de $T$ représente la forme bilinéaire sur une composante de $K_3[X]$, en l'espèce 1, X et $X^2$.

### 2.3.2 Construction de $G$

G est l'ensemble des formes bilinéaires qui comportent une seule multiplication de scalaires.
Ce sont donc les suivantes ($K^2 \to K$ avec les notations précédentes) :
$(A, B) \mapsto a_0b_0$
$(A, B) \mapsto a_0b_1$
$(A, B) \mapsto a_0(b_0 + b_1)$
$(A, B) \mapsto a_1b_0$
$(A, B) \mapsto a_1b_1$
$(A, B) \mapsto a_1(b_0 + b_1)$
$(A, B) \mapsto (a_0 + a_1)b_0$
$(A, B) \mapsto (a_0 + a_1)b_1$
$(A, B) \mapsto (a_0 + a_1)(b_0 + b_1)$.

Il y en a exactement $(2^n - 1)(2^m - 1) = 9$ avec ici $n = m = 2$.

La matrice $G$ correspondante est donc la suivante :

$$G = \begin{array}{c} \begin{matrix} a_0b_0 & a_0b_1 & a_1b_0 & a_1b_1 \end{matrix} \\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \end{array} \begin{matrix} a_0b_0 \\ a_0b_1 \\ a_0(b_0 + b_1) \\ a_1b_0 \\ a_1b_1 \\ a_1(b_0 + b_1) \\ (a_0 + a_1)b_0 \\ (a_0 + a_1)b_1 \\ (a_0 + a_1)(b_0 + b_1) \end{matrix}$$

### 2.3.3 Construction et test de W

$$W_0 = \begin{array}{c} \begin{matrix} a_0b_0 & a_0b_1 & a_1b_0 & a_1b_1 \end{matrix} \\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \begin{matrix} a_0b_0 \\ a_0b_1 + a_1b_0 \\ a_1b_1 \end{matrix}$$

La dimension de $W_0 = T$ est de 3.
Pour optimiser le nombre de multiplications on doit avoir $k < 4$ puisqu'il y a 4 multiplications dans la matrice $T$.
$k = 3$ est l'unique $k$ possible (car supérieur ou égal à $\dim T$).

On teste donc que $W_0$ est généré par des éléments de $G$ ($k$ éléments linéairement indépendants).

Ici, on trouve une solution évidente, puisqu'on remarque que :
$(a_0 + a_1)(b_0 + b_1) = a_0b_0 + (a_0b_1 + a_1b_0) + a_1b_1$
$\Leftrightarrow G[8] = W_0[0] + W_0[1] + W_0[2]$ avec aussi $W_0[0] = G[0]$ et $W_0[2] = G[4]$
$\Leftrightarrow W_0[0] = G[0]$, $W_0[1] = G[8] - G[0] - G[4]$ et $W_0[2] = G[4]$

G[0], G[4] et G[8] étant linéairement indépendants, on a trouvé l'unique espace solution, en l'espèce $W_0$.

# 3 L'implémentation existante

## 3.1 Structure récursive

Suivant le code algorithmique décrit, la fonction principale expand_subpace est récursive. La récursivité concerne l'extension de $W$ par un nouvel élément de $G$ non inclus dans $W$.
En réalité, il y a aussi une deuxième fonction récursive. Il s'agit de la fonction test, qui vérifie si un espace $W$ candidat (de dimension $k$) peut être généré par une famille de $k$ vecteurs indépendants de $G$ (cela correspond à $\mathrm{rk}(W \cap G) = k$).

## 3.2 La matrice $H_i$ pour chaque nœud

Pour chaque nœud de l'arbre de récursivité, une matrice $H_i$ est maintenue à jour : il s'agit des vecteurs $g$ de $G$ réduits par $W$.
$H_o := G$, $H_{i+1} := H_i$ réduit par $W$.
Ainsi les vecteurs de $H_i$ sont nuls s'ils sont dans $W$, et non nuls sinon.
Pour étendre l'espace $W$, il suffit d'y rajouter un vecteur de $G$ qui correspond à un vecteur non nul de $H_i$.
Il y a plusieurs avantages. On est sûr d'ajouter des vecteurs $g$ (de $G$) qui ne sont pas dans $W$ et on évite de tester tous les éléments de $G$. Et Lorsque l'on veut tester si un espace $W$ candidat de dimension $k$ est une solution (ie généré par $k$ éléments de $G$), on peut "piocher" directement les $g$ qui sont réduits à un vecteur nul dans $H_i$.

## 3.3 Échelonnement de $W$

$W$ est échelonnée au fur et à mesure des rajouts de $g$ de $G$.
Ainsi, la réduction des $g$ est facilitée (on réduit $H_i$ par le denier vecteur de $W$ à chaque tour), et la dimension de $W$ correspond à son nombre de lignes (avec $W_o := T$ réduite à ses générateurs).
La même méthode est utilisée pour calculer $W \cap G$ et vérifier que le rang est égal à $k$.

# 4 Les améliorations apportées

## 4.1 Les améliorations algorithmiques

### 4.1.1 Valider $\#G \cap W > k$ avant de tester un $W$ candidat

Lorsque l'on a construit une matrice $W$ candidate, c'est à dire une extension de $T$ avec des éléments de $G$ tel que $\dim W = k$, on se doit de vérifier que cet espace $W$ peut être généré par $k$ éléments de $G$ linéairement indépendants.
Mais s'il n'y a pas assez d'éléments de $G$ dans $W$ (cela correspondant au nombre de vecteurs nuls dans la matrice $H_i$), alors il est inutile de lancer ce test (qui est aussi une fonction récursive, donc très coûteuse).
Le test à vérifier est alors le suivant : $\#G \cap W > k$.
Les résultats obtenus sont assez probants et cela s'explique mathématiquement simplement.
Le $\#G \cap W$ peut être vu comme le #V (V l'espace total) multiplié par la densité de $G \cap W$ dans $V$.
Or la densité de $G \cap W$ dans $V$ n'est autre que le produit des densités de $G$ dans $V$ ($\frac{\#G}{q^{\dim V}}$) et $W$ dans $V$ ($\frac{q^k}{q^{\dim V}}$).
On en déduit donc la formule : $\#G \cap W = (\frac{\#G}{q^{\dim V}} \times \frac{q^k}{q^{\dim V}}) \times q^{\dim V} = \frac{\#G \cdot q^k}{q^{\dim V}}$.
Le test est donc d'autant plus utile pour $q^k$ et $\#G$ petits devant $q^{\dim V}$, et donc pour les grandes dimensions.

| Corps finis | k | Test de $\#G \cap W$ | # appels à Test (récursifs/ principaux) | Facteur de division du # appels à Test | # appels à Expand subspace (récursif/ principaux) | Facteur de division du # appels à Expand subspace | Temps de calcul (en secondes) |
|---|---|---|---|---|---|---|---|
| $F_{2^4}$ | 8 | non | 41819/2893 | 1 | 4821/1 | 1 | 11.34 |
| $F_{2^4}$ | 8 | oui | 4150/415 | 10,07 | 4821/1 | 1 | 4.07 |
| $F_{4^3}$ | 5 | non | 1244/210 | 1 | 232/1 | 1 | 0.39 |
| $F_{4^3}$ | 5 | oui | 1236/206 | 1 | 232/1 | 1 | 0.39 |

TABLE 1 – Comparaison avec et sans test "$\#G \cap W > k$" (programme Python et processeur 2Ghz)

### 4.1.2 Gestion des doublons dans la matrice $H_i$

L'implémentation de l'INRIA prévoit une matrice "de vie" $H_i$, tenue à jour pour chaque nœud, à commencer par $H_0 := G$ et qui est réduite par $W$ au fur et à mesure des appels récursifs.
Cette méthode permet d'éviter en particulier les doublons sur une même branche de l'arbre, en raison des réductions par les nouveaux vecteurs g rajoutés à $W$.
Cependant, la gestion des doublons pour un même niveau de l'arbre n'est pas prévue,

et une estimation heuristique permet de nous convaincre de sa nécessité, et conforte les résultats obtenus (lesquels sont à considérer principalement sur le nombre d'appels aux fonctions expand_subspace et test, la structure de données implémentée n'étant pas optimale au niveau du temps de calcul).
Pour chaque nœud, avec $\dim T \leq i \leq k$, la proportion de doublons est égale à $\frac{\#G}{2(q^{\dim E - i} - 1)}$, les $i$ colonnes pivot étant à zéro dans la matrice $H_i$.
Ainsi, la gestion des doublons est d'autant plus utile pour $(\dim E - k)$ petit.

| Corps finis | k | Gestion des doublons | # appels à Test (récursifs/ principaux) | Facteur de division du # appels à Test | # appels à Expand subspace (récursif/ principaux) | Facteur de division du # appels à Expand subspace | Temps de calcul (en secondes) |
|---|---|---|---|---|---|---|---|
| $F_{2^4}$ | 8 | non | 41819/2893 | 1 | 4821/1 | 1 | 11.34 |
| $F_{2^4}$ | 8 | oui | 5811/1554 | 7,19 | 2881/1 | 1,67 | 8.04 |
| $F_{4^3}$ | 5 | non | 1244/210 | 1 | 232/1 | 1 | 0.39 |
| $F_{4^3}$ | 5 | oui | 596/102 | 2,09 | 124/1 | 1,87 | 0.39 |

TABLE 2 – Comparaison avec et sans gestion des doublons dans la matrice $H_i$ (programme Python et processeur 2Ghz)

#### 4.1.3 Récursivité croissante

On ne parcourt l'arbre de récursivité que dans le sens croissant des indices des vecteurs $g$ de $G$.
Ainsi, les branches de l'arbre sont de la forme $g_{i_0}, g_{i_1}, ...g_{i_l}$, $l > 0$, avec la suite $(i_k)_{0 \leq k \leq l}$ croissante.
De cette manière, on ne reteste pas les mêmes extensions de $T$ (ex : $T \bigoplus \text{vect}(g_1) \bigoplus \text{vect}(g_2)$ et $T \bigoplus \text{vect}(g_2) \bigoplus \text{vect}(g_1)$).
Pour se convaincre que toutes les extensions possibles de $T$ sont décrites, si l'on choisit une liste quelconque d'éléments $g$ de $G$, en l'ordonnant dans le sens croissant on retombe sur une branche de l'arbre récursif.
En définitive, on divise donc le nombre de branches $(\binom{\#G}{k - \dim T})$ par $(k - \dim T)!$ .

### 4.2 Les améliorations d'implémentation

#### 4.2.1 Échelonnement et réduction

Une étude par profilage révèle que les fonctions d'échelonnement et de réduction (essentiellement cette dernière), sont les plus exécutées (en temps et en appels) par le programme.
En ce qui concerne l'échelonnement, pour chaque colonne pivot (ie colonne échelon), on met à zéro tous les éléments autres que le pivot, ce dernier étant normalisé à 1.
Ainsi, lorsque l'on réduit un vecteur de $H_i$ par $W$ échelonnée, on n'a plus qu'à calculer directement les colonnes non réduites à zéro.

Le nombre d'opérations est donc diminué, pour chaque nœud de l'arbre récursif, et pour chaque réduction, de $i \times \dim E$ à $i \times (\dim E - i)$, $\dim T \leq i \leq k$.

### 4.2.2 Les listes chaînées

Pour chaque nœud, on doit tester si les vecteurs de $H_i$ sont nuls ou pas.
Il est donc indispensable de tenir à côté une liste à jour, pour chaque nœud, des indices des $g$ dans $W$ (et sa liste complémentaire), afin de ne pas répéter inutilement ce test, à chaque appel et pour tous les vecteurs de $H_i$ (c'est à dire pour $\#G$ vecteurs).
Souhaitant travailler selon une récursivité croissante, les dites-listes doivent être ordonnées, et l'insertion doit se faire avec un minimum d'opérations. A ce titre, les listes chaînées (langage C) sont très appropriées, car le coût d'insertion est borné par $k$ au lieu de $2k$, ce qui est un gain substantiel au vu du nombre d'appels de la fonction d'insertion.

## 5 Conclusion

Avec l'implémentation réalisée, il est désormais possible de travailler avec des caractéristiques strictement supérieures à 3.
On peut distinguer par ailleurs deux types d'améliorations.
D'une part, les améliorations algorithmiques, avec en premier lieu, la vérification que le $\#W \cap G$ est supérieur ou égal à $k$, avant de tester l'espace candidat W (c'est à dire s'assurer qu'il y ait suffisamment d'éléments de $G$ dans $W$).
Ensuite, on a la gestion des doublons dans la matrice $H_i$, c'est à dire que l'on travaille avec les vecteurs représentants uniquement.
La dernière amélioration algorithmique est la récursivité croissante, qui réduit les explorations des branches de l'arbre récursif.
D'autre part, on trouve les améliorations d'implémentation. Les fonctions d'échelonnement et de réduction sont optimisées pour nécessiter un minimum de calcul.
Enfin, en langage C, l'utilisation des listes chaînées est très intéressante et plus appropriée que les tableaux pour tenir à jour les listes associées à la matrice $H_i$, au regard des coûts d'insertion minimes, et de l'immédiateté des passages entre éléments ordonnés.
Des améliorations sont encore possibles. On peut généraliser les listes chaînées dans le code, implémenter une structure de données de classe d'équivalence pour la gestion des doublons de la matrice $H_i$, paralléliser l'exécution du code sur une ou plusieurs machines, et enfin alimenter initialement les familles de générateurs de $G$ par les $g$ dans $W$ lors du test d'un espace $W$ candidat.

## 6 Remerciements

Je remercie mon encadrant, Mattieu Rambaud, ingénieur R&D au département INFRES de l'école Télécom ParisTech, qui m'a guidé avec bienveillance tout au long de mon projet. Bertrand Meyer, enseignant-chercheur à l'école Télécom ParisTech, nous a donné

par ailleurs de bonnes idées.

## Références

[1] Nicolas Estibals et Paul Zimmermann Razvan Barbulescu, Jérémie Detrey. Finding optimal formulae for bilinear maps. *International Workshop of the Arithmetics of Finite Fields 7369*, 2012. CARAMEL project-team, LORIA, Université de Lorraine / INRIA / CNRS.