Quatre millions d'échanges de clés par seconde

Carlos Aguilar-Melchor, Serge Guelton, Adrien Guinet
et Tancrède Lepoint
carlos.aguilar@enseeiht.fr
{sguelton,aguinet}@quarkslab.com
tancrede.lepoint@cryptoexperts.com

ENSEEITH, Quarkslab and CryptoExperts

Résumé Cet article présente une bibliothèque cryptographique : NFLlib. Nous décrivons les performances qu'elle permet d'atteindre au niveau des échanges de clés, ainsi que les optimisations qui ont mené à ces résultats.

La cryptographie à clé publique est réputée pour le surcoût important engendré tant au niveau de l'implémentation (calculs sur les grands nombres, manipulation de points dans une courbe elliptique, etc.) qu'au niveau calculatoire. Un des principaux usages de la cryptographie à clé publique est l'échange de clés lors de la mise en place d'une communication sécurisée (par exemple pour une connexion TLS). Cette opération, gourmande en temps de calcul, permet au mieux (en utilisant ECDH avec un Core i7-4770) de gérer de l'ordre de vingt mille connexions entrantes par seconde à 128 bits de sécurité ou six mille connexions à 256 bits de sécurité.

La cryptographie basée sur les réseaux est entrée en scène en force depuis quelques années, en particulier grâce aux fonctionnalités avancées, de type chiffrement complètement homomorphe, qu'elle permet d'atteindre. Notre bibliothèque, NFLlib, permet d'implémenter simplement des algorithmes cryptographiques basés sur les réseaux euclidiens, le tout avec des performances supérieures à l'état de l'art connu. Pour illustrer cela, nous implémentons, en une dizaine de lignes de code, un système de chiffrement standard avec une estimation conservatrice du niveau de sécurité. Cette implémentation permet de gérer quatre millions d'échanges de clés de type TLS par seconde à 128 bits de sécurité et deux millions à 256 bits de sécurité.

Si nous partons sur un proxy web utilisé pour du partage de charge et pouvant gérer les connexions TLS, il est possible de traiter jusqu'à 70 000 requêtes par seconde [1] sur un unique serveur. L'utilisation de ECDH demandera trois Core i7-4770 à 100% rien que pour les échanges de clés, alors que notre bibliothèque permet d'utiliser uniquement 2% d'un unique CPU libérant ainsi les processeurs pour traiter les requêtes.

1. http ://nginx.com/products/technical-specs/

1 Introduction

Depuis le début des années 90, une branche de la complexité est entrée avec force dans le monde de la cryptographie : la complexité des problèmes algorithmiques dans les réseaux euclidiens. Cette branche a donné naissance à ce qu'on appelle de nos jours la cryptographie basée sur les réseaux (*lattice-based cryptography* en anglais).

Ce domaine est de plus en plus présent dans les conférences en cryptographie pour plusieurs raisons :

- **Force des preuves de sécurité :** les protocoles cryptographiques basés sur les réseaux ont des réductions de sécurité de type pire-cas/cas-moyen. En pratique cela veut souvent dire que la sécurité de ces protocoles est basée sur la difficulté de problèmes bien connus des réseaux dans le cas général et pas dans des cas spécifiques (par exemple pour une sous-famille particulière de réseaux).
- **Polyvalence :** en plus des fonctionnalités classiques (chiffrements INP-CPA et IND-CCA2, signature, etc.) les réseaux euclidiens ont permis d'obtenir de nouvelles fonctionnalités (chiffrement complètement homomorphe [9], applications multilinéaires [8], etc.) qui restent de nos jours inatteignables par la cryptographie classique.
- **Performances asymptotiques :** Les réseaux euclidiens permettent de réaliser les calculs nécessaires aux protocoles en un temps qui est quasi linéaire en la taille des éléments utilisés et la sécurité augmente exponentiellement en cette taille. En pratique pour passer de 80 à 160 bits de sécurité les coûts sont doublés. Ceci est à comparer à la cryptographie asymétrique classique avec des coûts qui sont souvent cubiques en la taille et une sécurité qui augmente beaucoup plus lentement (*e.g.* sous-exponentiellement en la taille pour RSA).

Dans cet article nous nous intéressons aux performances pratiques de la cryptographie basée sur les réseaux.

Un comportement asymptotique, aussi excellent soit-il, n'est pas garant d'une supériorité lors d'une utilisation pratique. Un bon exemple est la multiplication basée sur la transformée de Fourrier qui a un coût quasi linéaire en la taille n d'un nombre (proportionnel à $n \log n$) mais dont le coût de base est important de telle façon que cet algorithme n'est intéressant que quand les nombres sont très grands (milliers ou dizaines de milliers de bits).

Dans cet article nous présentons NFLlib, une bibliothèque permettant de développer facilement des algorithmes de cryptographie basée sur

les réseaux euclidiens, le tout avec d'excellents résultats au niveau des performances. Cette bibliothèque permet de faire abstraction des objets sous-jacents dans les réseaux (des polynômes) et de juste décrire les opérations à réaliser (par exemple générer un élément aléatoire, multiplier celui-ci par un autre élément, etc.). Nous présentons également une étude de performances comparative dans le cadre des échanges de clés.

1.1 Travaux proches

Une version primitive de NFLlib est présente dans le projet XPIRe [1] (comme un sous-module). XPIRe est une bibliothèque C++ disponible [2] sur GitHub, permettant de réaliser du téléchargement privé (*i.e.* de télécharger un élément d'une base de données sans que le serveur contenant la base sache quel élément a été téléchargé) à haute vitesse. La comparaison à ce module initial est donnée en section 1.2.

Il existe différentes bibliothèques génériques permettant de définir les polynômes utilisés dans les réseaux et de réaliser les opérations de base nécessaires aux algorithmes cryptographiques. Parmi ces bibliothèques, celles permettant d'obtenir les plus hautes performances sont NTL [18] et FLINT [12]. Nous comparons dans cet article certaines fonctions de base pour montrer que les performances obtenues dans NFLlib sont très supérieures. Ceci est relativement normal puisque NTL et FLINT sont des bibliothèques génériques permettant de réaliser des calculs dans des contextes variés alors que notre bibliothèque est fortement optimisée pour un type de polynômes précis et uniquement pour une utilisation en cryptographie.

Dans le domaine de la cryptographie basée sur les réseaux, la bibliothèque HElib [11] propose une implémentation d'un système du chiffrement complètement homomorphe décrit dans [4]. Le pari de cette bibliothèque est de partir de NTL, de l'utiliser de manière optimisée et en profitant des nombreux algorithmes implémentés. La sélection des paramètres étant automatisée par HElib, et le schéma utilisé beaucoup plus compliqué qu'un simple schéma de chiffrement classique (ce dernier étant complètement homomorphe, il rend possible d'opérer publiquement sur des données chiffrées), la comparaison avec cette bibliothèque n'est pas significatif. Nous nous comparerons néanmoins, en section 3.2, avec une implémentation basée sur NTL qui permet d'obtenir une estimation du gain en performances que pourrait apporter la bibliothèque NFLlib pour HElib.

2. `http://github.com/XPIRe-team/XPIRe`

Enfin, un article, disponible dans les archives de l'IACR [3], présente un protocole permettant de faire un échange de clés basé sur les réseaux euclidiens, avec des propriétés de sécurité telles que la *perfect forward secrecy* ou la résistance aux analyses temporelles. Les auteurs développent un module intégré dans `OpenSSL` qui leur permet de réaliser des échanges de clés de type TLS en atteignant le nombre de requêtes par seconde similaire aux protocoles classiques. Dans cet article nous présentons une preuve de concept et non un module complètement intégré dans `OpenSSL`, et nous n'étudions pas la sécurité de l'implémentation elle-même. En revanche, nous pouvons réaliser environ mille fois plus d'opérations par seconde que dans cet article. Au vu de cet écart, il nous semble raisonnable de penser que les performances qu'ils présentent pourraient significativement s'améliorer par l'utilisation de NFLlib.

1.2 Contributions

La version de NFLlib que nous présentons dans cet article est beaucoup plus évoluée que celle incluse dans XPIRe. Les principales modifications que nous introduisons dans cette bibliothèque par rapport au sous-module d'XPIRe sont :

- Utilisation intensive des instructions SIMD, en explicitant les alignements pour les variables, et en créant du code avec des intrinsèques quand la sectorisation automatique fournie par le compilateur peut être améliorée ou n'est pas applicable ;
- Utilisation intensive de la méta-programmation pour déplacer à la compilation une partie des calculs ;
- Possibilité d'utiliser des mots de base de 64 bits, 32 bits ou 16 bits (avant seulement 64 bits étaient possible).

Ces modifications permettent d'augmenter grandement la polyvalence de la bibliothèque (en permettant de traiter efficacement les tout petits et les très grands polynômes). Nous comparons dans cet article les performances avec et sans plusieurs des modifications introduites pour montrer l'évolution positive que ces modifications apportent.

La deuxième contribution de cette bibliothèque est sa facilité d'utilisation à travers la surcharge d'opérateurs, sans que cela n'ait d'impact significatif sur les performances, grâce à plusieurs techniques de méta-programmation.

La troisième contribution est propre à l'article en soit. Nous montrons que la cryptographie basée sur les réseaux permet d'atteindre des performances très supérieures à la cryptographie classique pour l'échange de clés. Ceci est déjà vrai pour un niveau de sécurité basique (128 bits

de sécurité) et la différence de performance explose pour les niveaux de sécurité les plus élevés. Nous montrons ainsi qu'il est possible de passer d'une gestion de quelques milliers de connexions de type TLS par seconde dans un serveur classique à des millions.

La suite de l'article est structurée de la façon suivante : la section 2 introduit les bases théoriques sur lesquelles repose notre approche et décrit l'algorithme de chiffrement que nous utiliserons pour faire les échanges de clés ; la section 3 décrit les performances atteintes et les compare avec l'existant ; enfin, la section 4 détaille les choix de conceptions et les optimisations qui ont permis d'atteindre des performances élevées en conservant une API simple.

2 Bases

2.1 Objets manipulés

D'un point de vue purement mathématique, les réseaux euclidiens sont les « sous-groupes discrets de $\mathbb{R}^n$ ». Ici, nous nous intéressons à une famille importante des réseaux, appelée les réseaux d'idéaux. À nouveau, d'un point de vue purement mathématique, il s'agit des « réseaux en bijection avec les anneaux d'entiers du corps des nombres ». En pratique, quand nous implémentons les algorithmes de la cryptographie basée sur les réseaux, nous n'avons pas du tout besoin d'appréhender ces concepts mathématiques et nous manipulons tout simplement des polynômes sur les entiers.

Ainsi, les objets que nous manipulons sont des polynômes du type :

$$17 \cdot X^{1023} + 12 \cdot X^{1022} + \cdots + 21 \cdot X + 3.$$

Le contexte de travail est fixé par un degré maximal $n-1$ pour les polynômes, où n est une puissance de deux, et un module q pour les coefficients. Mathématiquement, on dit qu'on travaille dans R/qR pour $R = \mathbb{Z}[X]/\langle X^n+1\rangle$. En pratique, cela veut dire que chacun des coefficients est réduit à chaque opération modulo q, et que quand on a un polynôme avec des termes avec un degré supérieur ou égal à n (par ex. suite à une multiplication de deux polynômes), on ramène le degré à $n-1$ en remplaçant chaque terme au dessus du degré maximal par $a \cdot X^{n+i} \mapsto -a \cdot X^i$. Typiquement, n est une puissance de deux supérieure ou égale à 256 et le module utilisé pour les coefficients est un nombre allant d'une dizaine de bits (pour les échanges de clés ou les signatures numériques [6]) à plusieurs milliers de bits (pour le chiffrement complètement homomorphe [4]).

Algorithmiquement, un polynôme est stocké comme un tableau P de taille n où $P[i]$ contient le coefficient de X^i dans le polynôme.

2.2 Représentations

L'addition de polynômes se fait en additionnant les coefficients modulo q. La multiplication de deux polynômes est un peu plus compliquée puisqu'il faut multiplier tous les termes deux à deux, ce qui a un coût quadratique en n, puis réduire les coefficients avec un degré supérieur ou égal à n en les translatant et inversant le signe (avec l'opération $a \cdot X^{n+i} \mapsto -a \cdot X^i$).

Dans NFLlib deux techniques, changeant la représentation, permettent de réduire les coûts calculatoires : la représentation des polynômes par NTT (Number-Theoretic Transform, un cas particulier de FFT) et la représentation des coefficients par CRT (*Chinese Remainder Theorem*, Théorème des restes chinois). Une troisième technique permet d'accélérer les réductions modulaires, dans le cas où on travaille avec des entiers de 64 bits, en précalculant des quotients de certaines valeurs par les modules (pour 32 et 16 bits GCC le fait automatiquement). Cette technique est décrite dans [1], et le lecteur est renvoyé sur ce papier pour les détails. Quand une multiplication modulaire fait appel à de tels précalculs on parlera de multiplication modulaire de Shoup, ou de multiplication faisant appel à une représentation de Shoup.

Représentation NTT Comme la FFT, la NTT transforme un vecteur de coefficients représentant un polynôme en un autre vecteur qui représente l'évaluation du polynôme en des points particuliers. Nous appelons ce vecteur de valeurs la représentation NTT du polynôme. La principale différence entre la FFT et la NTT est la façon dont les points d'évaluation sont choisis, mais nous n'entrerons pas dans ces détails.

Quand les polynômes sont représentés par de tels vecteurs de valeurs, la multiplication des polynômes se fait coordonnée par coordonnée, et donc en une complexité linéaire en n. En effet, si à un des points d'évaluation, un polynôme f vaut a et un polynôme g vaut b, alors le produit des deux polynômes vaudra $a \times b$ (modulo q) et ceci est vrai pour chaque coordonnée des vecteurs décrivant les évaluations.

Transformer un polynôme en forme NTT ou faire la transformation inverse (permettant de passer d'un vecteur de valeurs à un vecteur de coefficients du polynôme) a un coût proportionnel à $n \log n$.

Le lecteur attentif remarquera que, dans les réseaux, nous avons besoin de multiplier les polynômes *et de les réduire modulo* $X^n + 1$ (*i.e.*

réaliser la transposition $a \cdot X^{n+i} \mapsto -a \cdot X^i$). En pratique, cela se fait automatiquement si un prétraitement est réalisé sur les vecteurs avant d'appliquer la NTT puis un autre avant d'appliquer la NTT inverse. Ces traitements correspondent à une multiplication par une valeur particulière pour chaque coordonnée.

Représentation utilisant le théorème des restes chinois Comme présenté dans la section 2.1, les polynômes que nous devons manipuler sont définis par le degré maximum $n-1$ et par le module q utilisé pour les coefficients. En utilisant une représentation de type NTT, nous avons la garantie que toutes les opérations sur les polynômes ont un coût linéaire en n (additions, soustractions, multiplications), ou quasi-linéaire (application de la NTT ou de la NTT inverse).

En revanche, sans autre technique, le coût des opérations peut augmenter rapidement en $\log q$, le nombre de bits de q. Les additions et soustractions ont un coût linéaire en $\log q$, mais les multiplications et les divisions (nécessaires pour la réduction modulaire) ont un coût quadratique en $\log q$. Ainsi, si pour un module de 64 bits, une multiplication s'appuiera sur les multiplicateurs hardware et ne prendra (en pipeline) qu'un cycle d'horloge, pour un module de 6400 bits cette opération prendra des dizaines de milliers de cycles.

Pour éviter ce problème, le module que nous utilisons est le produit d'une série de nombres premiers, tous stockés sur des entiers de même taille. Dans un premier temps, l'utilisateur doit choisir une taille d'entier non signé sur laquelle travailler (16, 32 ou 64 bits). Ceci impose que le module soit le produit d'une série de nombres premiers, chacun de taille égale à la taille des entiers choisis moins deux bits (*i.e.* 14, 30 ou 62 bits). Enfin, l'utilisateur doit choisir combien de nombres premiers il veut utiliser pour atteindre une taille de module qui lui convienne. Ainsi, par exemple, un utilisateur peut choisir d'utiliser des entiers de 64 bits et 100 premiers, ce qui donnera un module de 6200 bits, ou des entiers de 16 bits et un unique premier ce qui donnera un module de 14 bits.

Cette approche permet de réaliser les calculs efficacement en utilisant le théorème des restes chinois. Celui-ci dit que, si $q = p_1 \times \cdots \times p_t$, il est possible de transformer les nombres modulo q en des séries de t nombres modulo chacun des petits nombres premiers (cette transformation est appelée CRT), de réaliser les calculs sur les petits nombres, et de transformer les résultats finaux en un nombre modulo q (CRT inverse). En pratique, au lieu de réaliser les calculs sur un polynôme en travaillant modulo q sur les coefficients il est possible de travailler sur t polynômes avec

des coefficients modulo p_1 pour le premier, modulo p_2 pour le deuxième, etc.

Grâce à cette technique, si q fait 6200 bits, on travaillera avec 100 polynômes pour faire les opérations (addition, soustraction, multiplication et division) et le coût ne sera que 100 fois plus grand que si on travaillait avec un module de 62 bits. Les coûts augmentent donc linéairement en $\log q$ pour la multiplication et la division et pas en $\log^2 q$ comme ce serait le cas si on n'utilisait pas le théorème des restes chinois.

Le passage d'un nombre modulo q vers la représentation CRT a lui aussi un coût linéaire en $\log q$ (cela consiste tout simplement à prendre le reste du nombre modulo chacun des premiers indépendamment). L'opération inverse, en revanche, à un coût en quadratique en $\log q$. De façon générale, quand on crée un polynôme on le stocke directement sous forme NTT et CRT, et on n'applique les opérations de NTT inverse et CRT inverse qu'à la fin des opérations, quand cela est nécessaire.

2.3 Les échanges de clés

Les standards définissant comment faire un échange de clés sont, comme à l'habitude en cryptographie, nombreux. D'un côté, il y a bien sûr les standards du NIST et de l'ANSI : NIST SP 800-56A (pour la cryptographie basée sur le logarithme discret), NIST SP 800-56B (pour la cryptographie basée sur la factorisation), et ANSI X9.24 (considéré obsolète). De l'autre, il y a les RFC de l'IETF correspondant aux principaux protocoles utilisés de nos jours sur Internet : RFC 2409 et RFC 5996 pour IKEv2, et RFC 5246 pour TLS v1.2.

Dans ce document, nous utilisons l'équivalent de l'échange de clés par RSA dans TLS, ou par RSASVE dans NIST SP 800 56B, en remplaçant RSA par notre fonction de chiffrement. Le client choisit un message complètement aléatoire et le chiffre avec la clé publique du serveur, et le serveur déchiffre cette valeur qui sera ensuite utilisée pour dériver (par une fonction de hachage) un secret commun.

Pour illustrer le type de calculs à réaliser dans le système de chiffrement utilisé, nous décrivons ici ses sous-fonctions. Ce système est décrit dans [15] et nous orientons le lecteur vers ce papier pour les démonstrations de sécurité et de justesse du déchiffrement. Nous utilisons de façon opaque une fonction décrite dans [15], GaussianPoly, qui génère un polynôme avec des coefficients suivant une distribution de type gaussien pour un écart type adapté au paramètre de sécurité. UniformPoly retourne un polynôme avec des coefficients choisis uniformément entre 0 et $q - 1$.

Dans un premier temps, voici la fonction pour la génération de clés. [3]

Génération de clés : $\mathsf{KeyGen}(\kappa)$
Entrées :
- Un paramètre de sécurité k
Sortie : Une paire de clés (pk, sk)
— Si $k = 128$ prendre $n = 256$ et $q = 2^{14} - 2^{10} + 1$
— Si $k = 256$ prendre $n = 512$ et $q = 2^{14} - 2^{10} + 1$
— Sinon ERREUR
— $sk = \text{GaussianPoly}(n, q, k)$
— $pka = \text{UniformPoly}(n, q, k)$
— $pkb = pka \cdot sk + 2 \cdot \text{GaussianNoise}(n, q, k)$
— $pk = (pka, pkb)$

Pour simplifier le code et donner une idée des paramètres utilisés, nous avons explicité les sorties pour $k = 128$ et $k = 256$ et rendu les autres paramétrages impossibles. Une fois la paire de clés générée, il est possible de chiffrer/déchiffrer avec les fonctions suivantes.

Chiffrement : $\mathsf{Enc}(pk, m)$
Entrées :
- Une clé publique pk obtenue par KeyGen
- Un message m (polynôme à coefficients binaires) à chiffrer
Sortie : Un chiffré α
— $u = \text{GaussianPoly}(n, q, k)$
— $e_1 = 2 \cdot \text{GaussianPoly}(n, q, k)$
— $e_2 = 2 \cdot \text{GaussianPoly}(n, q, k)$
— $\alpha = (pka \cdot u + e_1, pkb \cdot u + e_2 + m)$
Déchiffrement : $\mathsf{Dec}(sk, \alpha)$
Entrées :
- Une clé secrète sk issue de KeyGen
- Un chiffré $\alpha = (\alpha_1, \alpha_2)$ issu de Enc
Sortie : Un message en clair m
— $m = \alpha_2 - \alpha_1 \cdot sk$
— Pour chaque coefficient $m[i]$ de m :
— $m[i] = (m[i] > q/2)?\ m[i]\%2\ :\ 1 - m[i]\%2$

---

3. Il convient de noter que nous n'avons proposé que des paramètres utilisant des modules tenant sur un registre pour optimiser la vitesse d'exécution. Utiliser une représentation sous forme CRT sera utile pour implémenter du chiffrement homomorphe [9,4].

L'objectif de ce schéma étant de transporter une clé, on choisit les plus petites valeurs de $\log q$ et n permettant d'atteindre la sécurité demandée. Ainsi les opérations entre coefficients seront très peu coûteuses, mais pour les optimiser il faudra faire attention à la vectorisation du code. Au niveau des opérations entre polynômes, les multiplications de type $pka \cdot u$, $pkb \cdot u$, et $\alpha_1 \cdot sk$ seraient très coûteuses (millions d'opérations) sans améliorations. Transformer les opérandes par la NTT permettra de réduire fortement les coûts. La NTT inverse ne sera faite qu'avant la dernière étape du déchiffrement où un travail par coefficient est nécessaire. Ainsi :

— la génération de clés applique la NTT à sk, pka et pkb ;
— le chiffrement applique la NTT à m, u, e_1 et e_2 ;
— le déchiffrement applique la NTT inverse à m après le premier pas.

Ces opérations permettent de remplacer le coût des multiplications (millions d'opérations) par le coût des NTTs (milliers d'opérations) et multiplications par coordonnées (milliers d'opérations). Nous appelons notre implémentation de ce système NFLLWE.

2.4 Sécurité

Il est important de remarquer que le schéma cryptographique que nous proposons est un schéma standard [15] et qu'en aucun cas les résultats de performance annoncés ne viennent de choix de paramètres audacieux ou d'un choix d'une sous-famille exotique de réseaux.

Dans les réseaux il y a plusieurs problèmes standards réputés infaisables en un temps raisonnable. Au niveau des fonctions de type chiffrement les deux grands problèmes sont Learning With Errors (LWE) et Ring Learning With Errors (RLWE). Ces problèmes s'appuient sur d'autres problèmes plus profonds venant de la complexité (trouver un ou des vecteurs courts dans des réseaux) mais nous n'entrerons pas dans les détails, le lecteur voulant approfondir ces questions peut lire l'excellent article de Regev sur le sujet [16].

Le système de chiffrement que nous présentons a une sécurité basée sur la difficulté du problème RLWE. Le choix des paramètres se fait par rapport aux meilleures attaques connues. Il est important de remarquer que les algorithmes permettant d'attaquer ce genre de systèmes (notamment LLL [13], BKZ [17] et BKZ-2.0 [5]), ont aussi un grand intérêt dans d'autres domaines que la cryptographie, et sont étudiés depuis plusieurs dizaines d'années.

Les paramètres que nous utilisons en section 3 sont obtenus en utilisant l'algorithme de choix de paramètres de [14], connu pour être assez conservateur. Notons que si ces choix ne se montraient pas suffisamment

conservateurs, il suffirait de doubler la sécurité, ce qui ne donnerait qu'une augmentation d'un facteur deux des coûts.

3 Performances atteintes

Dans cette section nous présentons les performances que notre bibliothèque permet d'atteindre. Tout d'abord, nous comparons NFLLWE aux primitives classiques de la cryptographie et nous montrons qu'il existe un écart très important de performances (facteur supérieur à 200). Dans un deuxième temps, nous comparons NFLLWE avec l'implémentation qu'on obtient en utilisant d'autres bibliothèques comme NTL et FLINT. Nous montrons ainsi que le gain en performance découle de l'utilisation des réseaux (facteur 10 par rapport aux bibliothèques alternatives) mais surtout des optimisations de notre bibliothèque (facteur 20 par rapport aux implémentations NTL ou FLINT)

3.1 Comparaison aux coûts calculatoires classiques

La cryptographie classique, basée sur la factorisation ou sur le logarithme discret, a deux problèmes par rapport à la cryptographie basée sur les réseaux : une coût pour les niveaux de sécurité basiques relativement élevé et une mauvaise mise à l'échelle quand la sécurité augmente.

Au niveau de l'évolution des coûts, le tableau 1 montre les paramétrages pour 128 bits de sécurité et pour 256 bits de sécurité pour des échanges de clés basés sur : RSA, DH (Diffie-Hellman sur des entiers), ECDH (Diffie-Hellman sur les courbes elliptiques), et NFLLWE notre implémentation basée sur RLWE.

Protocole	Paramètres pour 128 bits	Paramètres pour 256 bits	Surcoût calculatoire
RSA	3072	15360	×125
DH	256/3072	512/15360	×50
ECDH	256	521	×8
NFLLWE	14/256	14/512	×2

Table 1. Paramètres et coûts relatifs pour divers protocoles. Pour RSA, le paramètre est la taille du module (et des chiffrés), pour DH les paramètres sont les tailles de l'exposant et du module, pour ECDH l'exposant et le module ont la même taille, pour NFLLWE, les paramètres sont la taille du module et le degré du polynôme quotient.

Comme c'est bien connu, RSA est le moins bon élève pour la mise à l'échelle au niveau de la sécurité. Ceci à d'ailleurs été le principal argument qui a propulsé les courbes elliptiques au devant de la scène. DH et surtout ECDH font mieux mais on voit qu'il n'y a pas un rapport linéaire (mais au moins cubique) entre l'augmentation au niveau de la sécurité et l'augmentation des coûts théoriques.

Ceci a une importance significative au niveau de la crédibilité de la résistance à de nouvelles attaques. Si un jour un nouvel algorithme ou une nouvelle technologie (par exemple l'ordinateur quantique) permet de diviser par dix la sécurité d'un système basé sur les réseaux, il suffira d'augmenter les paramètres linéairement et les performances seront divisées par dix. Ceci nous laisserait encore 400 000 échanges de clés par seconde, 20 fois plus qu'avec ECDH. Si ECDH subissait la même attaque les coûts augmenteraient d'un facteur 1000 et pour RSA ça serait encore pire.

Protocole	80bits	128 bits	256 bits
RSA	7.95 Kops/s	0.31 Kops/s	N/D
ECDH	7.01 Kops/s	5.93 Kops/s	1.61 Kops/s
NFLLWE	N/D	1020 Kops/s	508 Kops/s

Table 2. Nombre d'échanges de clés par seconde pour un serveur avec un Core i7-4770 en utilisant un seul cœur (pour RSA et ECDH, nous avons utilisé le test intégré à openssl 1.0.1f.). En utilisant les quatre cœurs toutes les performances sont globalement multipliées par quatre. Il n'y a pas d'implémentation standard de RSA15360 et notre bibliothèque ne permet pas un paramétrage à 80 bits de sécurité pour cette application (d'où les entrées non disponibles, marquées N/D).

Le tableau 2 montre les performances des différents protocoles pour 80, 128 et 256 bits de sécurité. Dans RSA et NFLLWE l'opération que doit réaliser le serveur est un déchiffrement, dans ECDH c'est l'opération de base de Diffie-Hellman, une exponentiation modulaire. NFLLWE permet de traiter plus de clients ou d'utiliser moins de temps processeur pour un même nombre de clients. L'écart est d'au moins un facteur 200, ce qui permet de gagner sur tous les plans. Par exemple il est possible de traiter 10 fois plus de clients avec 10 fois moins de temps processeur et en doublant la sécurité par rapport à ECDH à 128 bits de sécurité.

3.2 Comparaison à NTL et FLINT

Comme mentionné en introduction, les bibliothèques NTL [18] et FLINT [12] permettent de manipuler aisément les polynômes utilisés en

cryptographie basée sur les réseaux. Ces bibliothèques sont génériques : elles peuvent donc être utilisées pour manipuler des polynômes de degrés arbitraires, modulo des entiers arbitraires. Inversement, on rappelle que la bibliothèque NFLlib se restreint volontairement à des polynômes de degré $n-1$ (où n est une puissance de 2), modulo des entiers produits de plus petits entiers (pour manipuler les coefficients sous forme CRT), c'est-à-dire des paramètres utilisés en cryptographie basée sur les réseaux (voir par exemple [6]). Cette restriction permet, du fait des nombreuses optimisations décrites dans cet article, des performances significativement supérieures à NTL et FLINT pour les mêmes opérations. Nous présentons les gains obtenus dans la suite de la section. Tous les *benchmarks* ont été obtenus sur un processeur Intel Xeon E5-2666 v3. [4]

Paramètres. Dans les trois bibliothèques, nous avons utilisé les mêmes quatre ensembles de paramètres :

1. $n = 256$ et un module q de 14 bits,
2. $n = 512$ et un module q de 30 bits,
3. $n = 1024$ et un module q de 62 bits, et
4. $n = 1024$ et un module q de 6200 bits (produit de 100 modules de 62 bits).

Pour les tests sous NFLlib, nous avons utilisé respectivement des mots de base de 16, 32, 64 et 64 bits, pour profiter des implémentations SIMD optimisées (dans les deux premiers cas, voir section 4.7). Pour NTL, nous avons utilisé la classe `zz_pX` pour les paramètres (1) et (2) puisque le module fait moins de 50 bits, et la classe `ZZ_pX` pour les paramètres (3) et (4). Finalement, nous avons travaillé avec le type `fmpz_mod_poly_t` pour FLINT.

Génération de polynômes aléatoires. Un élément clé en cryptographie basée sur les réseaux consiste à générer des polynômes aléatoires, c'est-à-dire dont les coefficients sont indépendamment et uniformément échantillonnés modulo q. Nous avons utilisé les fonctions de génération de nombres aléatoires proposées dans chaque bibliothèque, à savoir la fonction `random` de NTL, le générateur donné par le type `flint_rand_t` dans FLINT, et le générateur par défaut [5] de NFLlib. Les résultats sont illustrés par le tableau 3.

4. Instance `c4.8xlarge` d'Amazon Web Services EC2.

5. Ce générateur est basé sur Salsa20 [2], et a été développé par Gim Güneysu, Tobias Oder, Thomas Pöppelmann et Peter Schwabe dans [10].

Bibliothèque	NTL	FLINT	NFLlib (seq.)
$(1) = (256, 14)$	8.7μs	4.6μs	0.6μs
$(2) = (512, 30)$	22.1μs	9.2μs	2.5μs
$(3) = (1024, 62)$	169μs	18.1μs	9.3μs
$(4) = (1024, 6200)$	8149μs	1010μs	992μs

Table 3. Temps moyen de génération d'un polynôme de degré $n-1$ uniformément échantillonné modulo q pour plusieurs bibliothèques.

NFLlib permet un gain significatif comparé à NTL, mais le générateur d'aléa de FLINT (qui est essentiellement celui de GMP) est aussi rapide que celui de NFLlib quand une grande quantité d'aléa est générée. Néanmoins, il utilise l'algorithme Mersenne Twister. Ce dernier est très efficace, mais malheureusement **insuffisant pour une utilisation en cryptographie** : connaître une grande quantité d'aléa permet de prédire avec certitude les nombres aléatoires qui seront générés dans le futur. A contrario, l'aléa dans NTL et NFLlib provient de générateurs de nombres pseudo-aléatoires qui conviennent pour une utilisation en cryptographie.

Addition. Comme expliqué précédemment, que les polynômes soient en représentations NTT ou représentés par leurs coefficients, additionner deux polynômes revient à additionner des vecteurs de n valeurs modulo q. Pour NTL nous avons utilisé la fonction `add`, pour FLINT la fonction `fmpz_mod_poly_add`, et pour NFLlib l'opérateur +. Les résultats sont donnés au tableau 4. NFLlib permet un gain d'un facteur entre 4 et 20 en vitesse comparé à NTL et FLINT.

Bibliothèque	NTL	FLINT	NFLlib (seq.)
$(1) = (256, 14)$	1.4μs	3.6μs	0.1μs
$(2) = (512, 30)$	2.6μs	6.8μs	0.3μs
$(3) = (1024, 62)$	37.6μs	13.3μs	1.3μs
$(4) = (1024, 6200)$	588μs	521μs	168μs

Table 4. Temps moyen pour additionner deux polynômes de degré $n-1$ modulo q pour plusieurs bibliothèques.

Calcul de la représentation NTT et multiplication. La représentation sous forme NTT est un élément clé de notre optimisation qui permet d'accélérer la multiplication de deux polynômes. Calculer la représentation

NTT est possible dans NTL grâce aux fonctions `TofftRep` et `ToFFTRep` (pour `zz_pX` et `ZZ_pX` respectivement) mais n'a malheureusement pas l'air d'être une fonction disponible dans FLINT. Les résultats sont donnés dans les tableaux 5 et 6. L'algorithme de conversion vers ou depuis une représentation NTT est 10 à 30 fois plus rapide dans NFLlib que dans NTL.

Bibliothèque	NTL	FLINT	NFLlib (seq.)	NFLlib (SSE)	NFLlib (AVX2)
$(1) = (256, 14)$	6.7μs	–	1.6μs	0.4μs	0.4μs
$(2) = (512, 30)$	14.0μs	–	3.4μs	2.2μs	1.7μs
$(3) = (1024, 62)$	46.2μs	–	11.1μs	–	–
$(4) = (1024, 6200)$	31568μs	–	1126μs	–	–

Table 5. Temps moyen pour effectuer la transformation NTT pour un polynômes de degré $n - 1$ modulo q pour plusieurs bibliothèques.

Bibliothèque	NTL	FLINT	NFLlib (seq.)	NFLlib (SSE)	NFLlib (AVX2)
$(1) = (256, 14)$	11.4μs	–	2.3μs	0.9μs	0.9μs
$(2) = (512, 30)$	24.9μs	–	3.6μs	2.4μs	1.9μs
$(3) = (1024, 62)$	188μs	–	12.1μs	–	–
$(4) = (1024, 6200)$	38131μs	–	1218μs	–	–

Table 6. Temps moyen pour effectuer la transformation NTT inverse pour un polynômes de degré $n - 1$ modulo q pour plusieurs bibliothèques.

Multiplication avec et sans représentation de Shoup. Nous avons ensuite simulé une multiplication sous représentation NTT dans NTL et FLINT (les fonctions n'existant pas par défaut) pour se comparer à l'opérateur $*$ dans NFLlib. Cette opération consiste en la multiplication coefficient par coefficient de deux tableaux de nombres modulo q. Nous avons aussi *benchmarcké* la multiplication quand une représentation de Shoup des nombres est connue. Les résultats sont donnés dans le tableaux 7. La multiplication « classique » est déjà 7 à 10 fois plus rapide dans NFLlib que dans NTL ou FLINT. Mais quand une représentation de Shoup d'un des opérandes est connue, un facteur 10 supplémentaire peut être obtenu. Cette technique n'est actuellement pas disponible dans NTL et FLINT : nos résultats illustrent donc le gain potentiel qu'une telle technique pourrait apporter à ces bibliothèques.

Bibliothèque	NTL	FLINT	NFLlib (seq.)	NFLlib (seq, w/ Shoup)
$(1) = (256, 14)$	1.0μs	4.8μs	0.1μs	0.15μs
$(2) = (512, 30)$	1.7μs	9.7μs	0.8μs	0.6μs
$(3) = (1024, 62)$	52.6μs	33.5μs	28.0μs	2.3μs
$(4) = (1024, 6200)$	15259μs	15839μs	2801μs	231μs

Table 7. Temps moyen pour multiplier deux polynômes de degré $n-1$ modulo q pour plusieurs bibliothèques. Seule NFLlib permet d'effectuer cette pultiplication en utilisant la forme de Shoup.

Implémentation d'un schéma de chiffrement à clé publique. Tous les *benchmarks* précédents illustrent étape par étape les gains en performance que NFLlib permet d'obtenir par rapport à NTL et FLINT. Cependant, ceci ne reflète pas totalement le comportement de chaque bibliothèque lorsqu'elle est utilisée **en pratique** pour implémenter *e.g.* un schéma cryptographique. Par exemple, les *benchmarks* des multiplications proviennent d'implémentations faites-main qui peuvent ne pas correspondre aux algorithmes sous-jacents utilisés dans les bibliothèques. De même la conversion vers une représentation NTT n'est pas disponible dans FLINT, ce qui fait pressentir un comportement en pratique assez coûteux.

Pour comparer l'efficacité relative des trois bibliothèques sur un cas pratique, nous avons choisi d'implémenter le schéma de chiffrement basé sur les réseaux tel que décrit dans la section 2.3 . Les paramètres sont choisis afin d'assurer 128 ou 256 bits de sécurité selon [14]. Cependant pour atteindre un niveau suffisant de sécurité, nous devons modifier les paramètres (4), et travaillons donc à la place avec les paramètres suivants :

(4') $n = 16384$ et q est un module de 620 bits (produit de 10 modules de 62 bits).

Nous avons implémenté ce schéma en utilisant les trois bibliothèques. Les polynômes des clés publiques et secrètes étant constants, nous avons cherché à optimiser les multiplications par ces derniers. Pour NFLlib, nous avons calculé leur représentation de Shoup. Pour NTL, nous avons utilisé les fonctions `zz_pXMultiplier` et `zz_pXModulus` (et leurs équivalents `ZZ_`) qui permettent respectivement d'optimiser les multiplications par un polynôme constant d'une manière similaire à notre représentation de Shoup), et d'optimiser les opérations de réduction modulo $X^n + 1$. Pour FLINT, nous calculé l'inverse de $X^n + 1$ modulo X^{n+1} pour optimiser la multiplication modulaire en utilisant la fonction `fmpz_mod_poly_mulmod_preinv` au lieu de `fmpz_mod_poly_mulmod`. Finalement, nous obtenons les résultats donnés dans les tableaux 8 et 9. Utiliser NFLlib permet de gagner un à

deux ordres de magnitudes par rapport à l'utilisation des bibliothèques génériques que sont NTL et FLINT.

Bibliothèque	NTL	FLINT	NFLlib (seq.)	NFLlib (SSE)	NFLlib (AVX2)
$(1) = (256, 14)$	156.9μs	134.8μs	10.9μs	7.1μs	6.7μs
$(2) = (512, 30)$	321.9μs	296.3μs	18.2μs	15.8μs	14.1μs
$(3) = (1024, 62)$	1599.8μs	673.5μs	69.7μs	–	–
$(4') = (16384, 620)$	233068μs	221863μs	10723μs	–	–

Table 8. Temps moyen pour effectuer le chiffrement à clef publique représentée par un polynôme de degré $n - 1$ modulo q pour plusieurs bibliothèques.

Bibliothèque	NTL	FLINT	NFLlib (seq.)	NFLlib (SSE)	NFLlib (AVX2)
$(1) = (256, 14)$	54.3μs	52.1μs	2.7μs	1.20μs	1.12μs
$(2) = (512, 30)$	111.8μs	130.4μs	4.3μs	3.58μs	3.1μs
$(3) = (1024, 62)$	570.8μs	301.9μs	16.6μs	–	–
$(4') = (16384, 620)$	112358μs	111625μs	4528μs	–	–

Table 9. Temps moyen pour effectuer le déchiffrement à clef publique représentée par un polynôme de degré $n - 1$ modulo q pour plusieurs bibliothèques.

4 Analyse des contributions

L'amélioration des performances peut se faire à plusieurs niveaux :

— algorithmiquement, à savoir réduire la complexité algorithmique des opérations utilisées. Ce sont les améliorations qui apporteront le plus de gain.

— mathématiquement, en utilisant des astuces prouvées pour améliorer certains calculs

— au niveau de l'implémentation, à savoir tenter d'améliorer le code décrivant les algorithmes en utilisant par exemple les fonctionnalités spécifiques de certains CPU.

Les deux premières parties ont été décrites section 2. Les sections suivantes décrivent les améliorations réalisées au niveau de l'implémentation.

4.1 Structure

La bibliothèque NFLlib fournit principalement une classe `poly` fournissant les abstractions nécessaires pour effectuer les opérations décrites

dans la section 2. Cette section présente les différents composants de l'architecture décrite dans la figure 1.

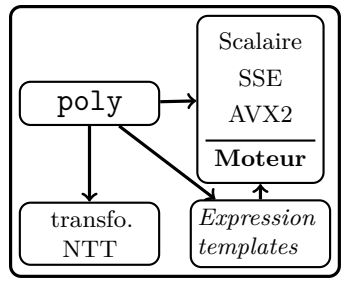


Figure 1. Architecture de la bibliothèque NFLlib.

4.2 Utilisation des spécialisations pour le SIMD

Chaque opération de calcul sur les polynômes jugée « critique » a été identifiée grâce à une analyse dynamique permettant de détecter les « points chauds » du binaire produit, comme présenté section 4.5. Il apparaît que plusieurs fonctions peuvent bénéficier d'une vectorisation manuelle, basée sur les jeux d'instructions SSE et AVX2 si ces derniers sont disponibles sur la machine cible.

Ainsi, si une version de fonction spécialisée pour les jeux d'instructions SSE et/ou AVX2 s'avère nécessaire afin de gagner en performance, alors cette opération est « extraite » afin de pouvoir être spécialisée. Pour éviter les doublons par architectures, le mécanisme de vectorisation générique suivant a été adopté :

— déclaration d'une structure contenant le corps de boucle,
— cette structure est spécialisable suivant le « moteur SIMD » utilisé et le type d'entier utilisé (16, 32 ou 64 bits),
— spécialisation de cette structure suivant les implémentations manuelles écrites.

La liste des moteurs SIMD disponibles pour l'instant est :

— *serial* : code original,
— *sse* : implémentation utilisant des registres 128 bits et intrinsèques SSE,
— *avx2* : implémentation utilisant des registres 256 bits et intrinsèques AVX2.

4.3 Code paramétrique

Une version naïve de la classe des polynômes encapsulerait un pointeur vers une zone mémoire allouée dynamiquement. En procédant ainsi, on remarque que l'opération `inv_ntt` est particulièrement coûteuse. Elle est composée d'un appel à la fonction `ntt` entourée de deux permutations, comme illustré au listing 1.

```
void inv_ntt(value_type * x, value_type p, size_t degree)
{
    value_type *y = new value_type[degree];
    for (size_t i = 0; i < degree; i += 1)
        y[perm[i]] = x[i];
    ntt(y, inv_wtab, inv_winvtab, p);
    for (size_t i = 0; i < degree; i += 1)
      x[perm[i]] = y[i];
    delete [] y;
}
```

Listing 1. Implémentation de référence de `inv_ntt`.

Chaque itération exécute donc des accès à trois tableaux différents dont un accès avec une indirection.

En se basant sur l'observation que la permutation est calculée à l'initialisation de la bibliothèque et ne dépend que du degré du polynôme, `degree`, nous avons transformé l'argument `degree` en un paramètre template, ce qui permet de réécrire la fonction précédente sous la forme donnée au listing 2

```
template<size_t degree>
void inv_ntt(value_type * x, value_type p) {
    value_type y[degree];
    do_perm<degree>(y, x);
    ntt(y, inv_wtab, inv_winvtab, p);
    do_perm<degree>(x, y);
}
```

Listing 2. version paramétrique de `inv_ntt`.

En utilisant plusieurs techniques de méta-programmation connues, le tableau `perm` est remplacé par un calcul **à la compilation**, au prix d'un déroulage complet de la boucle d'origine. Ainsi, chaque itération ne fait plus que deux accès de tableaux à des indices constants.

Un effet de bord important de la paramétrisation est que l'allocation dynamique du tableau temporaire `y` est remplacée par une allocation statique, phénomène également observé pour le tableau utilisé pour représenter le polynôme, ce qui nous permet d'éviter quelques appels et surtout d'avoir une meilleur localité mémoire. On pousse la logique en rendant le

polynôme également paramétré par le type d'entier utilisé pour stocker les coefficient et la taille de la série de nombre premiers utilisés, ce qui permet de supprimer toute forme d'allocation dynamique à la création du polynôme.

Rendre ces paramètres constants fait également apparaître de nouvelles possibilité d'optimisation pour le compilateur (*e.g. constant folding*, *full loop unrolling*, *partial evaluation*...).

Au final, le gain en temps d'exécution observé pour cette optimisation est de l'ordre de 15% et la classe polynôme devient une classe template paramétrée par un type et deux valeurs, au prix d'un temps de compilation supérieur.

4.4 Agrégation des opérateurs

Un problème classique en C++ est la création d'objets temporaires lors du calcul d'une expression complexe. Par exemple, en supposant que les opérateur idoines aient été surchargés, l'évaluation de l'expression polynomiale $p0 + p1 \cdot p2$ conduira à la création de deux objets temporaires, l'un pour la multiplication, l'autre pour l'addition, comme représenté dans le listing qui suit :

```
poly<T, degree, nmoduli> tmp0, tmp1;
for(size_t cm = 0; cm < nmoduli; ++cm)
    for(size_t d = 0; d < degree; ++d)
        tmp0(cm, d) = p1(cm, d) * p2(cm, d);
for(size_t cm = 0; cm < nmoduli; ++cm)
    for(size_t d = 0; d < degree; ++d)
        tmp1(cm, d) = p0(cm, d) * tmp(cm, d);
```

Cette forme est loin d'être optimale : les boucles pourraient être fusionnées, et le polynôme intermédiaire pourrait être supprimé pour aboutir à :

```
poly<T, degree, nmoduli> tmp0, tmp1;
for(size_t cm = 0; cm < nmoduli; ++cm)
    for(size_t d = 0; d < degree; ++d)
        tmp1(cm, d) = p0(cm, d) + p1(cm, d) * p2(cm, d);
```

On notera par ailleurs que cette forme est plus intéressante du point de vue des instructions vectorielles, puisqu'elle donne un ratio de deux opérations pour un *store* et trois *load*, là où la précédente donnait un ratio de deux opérations pour deux *store* et quatre *load*[6].

6. On notera que ce ratio est intéressant puisque les opérations de multiplications et d'additions sur nos polynômes sont plus coûteuses que des opérations scalaires.

L'approche qui consiste à fournir une fonction optimisée pour chaque combinaison d'opérateurs ne passe bien évidement pas à l'échelle. Les *expression templates* [19] proposent une solution élégante à ce problème en retardant l'évaluation d'une expression à l'assignation. L'idée est de construire l'arbre d'évaluation à travers le système de type et de déclencher l'évaluation de l'expression au moment de l'assignation en se servant du type de retour de l'expression pour reconstruire la boucle fusionnée. Ainsi, dans notre exemple, le type de retour sera de la forme donnée au listing 3.

```
expr<add, poly, expr<mul, poly, poly>>
```

Listing 3. Exemple de type représentant une *expression template*

Cette construction est compatible avec la vectorisation manuelle, pour peu qu'une forme vectorielle de chacun des opérateurs soit disponible, ce qui a été fait à chaque fois que possible. Dans la cas contraire, un mécanisme de trait permet de détecter à la compilation, et sans intervention de l'utilisateur, si la forme vectorielle est disponible pour toute l'expression ou si la forme séquentielle doit être utilisée. En ce sens, notre approche est très semblable à celle utilisée par `boost.simd` [7].

Un autre avantage des *expression templates* est qu'elles permettent la réécriture de l'arbre avant son évaluation. Dans notre cas, nous disposons d'une version accélérée du *multiply-add* qui évite un coûteux calcul de modulo. Toujours à la compilation, en utilisant la spécialisation partielle des *templates*, on peut donc remplacer l'expression donnée au listing 3 par `expr<fma, poly, poly, poly>`, sans surcoût à l'exécution.

4.5 État des lieux

Afin d'optimiser les performances de la bibliothèque NFLlib, un cas test représentatif des opérations de chiffrement et déchiffrement asymétrique a été étudié grâce à `valgrind/callgrind`[7]. Cela nous a permis d'identifier les « points chauds » de ces deux opérations, et de s'attarder premièrement sur ceux-ci. Après optimisations du code associé, une nouvelle analyse est lancée afin de découvrir éventuellement de nouveaux points chauds. Ce processus est répété jusqu'à ce qu'il devienne difficile d'optimiser les parties coûteuses.

Le profilage montre que le calcul du chiffrement est réparti entre le calcul de la transformation NTT ($\sim 64\%$), des opérations arithmétiques sur les polynômes considérés (`modular addition`, `modular multiply shoup`

7. `http://valgrind.org/docs/manual/cl-manual.html`

et la fusion de ces deux opérations) ($\sim$ 20%) et le calcul d'aléa ($\sim$ 16%) . Le calcul d'aléa étant considéré comme déjà optimisé, les efforts sont ainsi à concentrer sur la fonction `ntt`.

En ce qui concerne le déchiffrement, la fonction la plus importante en terme de temps de calcul est `inv_ntt` ($\sim$ 80%), qui appelle elle-même `ntt` ($\sim$ 96% du temps d'exécution de `inv_ntt`). Il faudra ainsi se focaliser sur cette fonction après avoir optimisé la fonction `ntt`. Le reste du temps de calcul est réparti entre les opérations arithmétiques polynomiales citées ci-dessus.

Les sections suivantes décrivent les différentes étapes de transformation qui ont été appliquées afin d'optimiser ces fonctions et d'autres parties du code.

4.6 Réécriture de la boucle NTT

La boucle NTT a tout d'abord été réécrite afin d'expliciter plus facilement les accès mémoires. La version originale est basée sur l'implémentation de David Harvey. [8]

La partie du code prenant le plus de temps CPU, modifiée pour utiliser des entiers 32 bits, est présentée listing 4 .

```
for (size_t M = 1; N > 4; N /= 2, M *= 2) {
  uint32_t* x0 = x;
  uint32_t* x1 = x + N/2;
  for (size_t r = 0; r < M; r++, x0 += N, x1 += N) {
    ptrdiff_t i = N/2 - 2;
    do {
      {
        uint32_t u0 = x0[i+1];
        uint32_t u1 = x1[i+1];

        uint32_t t0 = u0 + u1;
        t0 -= ((t0 >= 2*p) ? (2*p) : 0);
        uint32_t t1 = u0 - u1 + 2*p;
        uint32_t q = ((uint64_t) t1 * winvtab[i+1]) >> 32;
        uint32_t t2 = t1 * wtab[i+1] - q * p;

        x0[i+1] = t0;
        x1[i+1] = t2;
      }
      {
        uint32_t u0 = x0[i];
        uint32_t u1 = x1[i];

        uint32_t t0 = u0 + u1;
        t0 -= ((t0 >= 2*p) ? (2*p) : 0);
        uint32_t t1 = u0 - u1 + 2*p;
```

8. http://web.maths.unsw.edu.au/~davidharvey/papers/fastntt/

```
        uint32_t q = ((uint64_t) t1 * winvtab[i]) >> 32;
        uint32_t t2 = t1 * wtab[i] - q * p;

        x0[i] = t0;
        x1[i] = t2;
      }
      i -= 2;
    }
    while (i >= 0);
  }
  wtab += N/2; winvtab += N/2;
}
```

Listing 4. Code original de la boucle principale NTT.

La boucle `while` est manuellement déroulée d'un pas de deux. Les compilateurs effectuent cette opération automatiquement et auraient même tendance à être perturbés par cette modification, nous commençons donc par réécrire cette boucle en une boucle `for` canonique, en enlevant les pré-calculs d'indices qui seront optimisés par le compilateur (passe `-fmove-loop-invariants`), pour aboutir au listing 5.

```
for (size_t M = 1 ; N > 4; N /= 2, M *= 2) {
  for (size_t r = 0; r < M; r++) {
    for (size_t i = 0; i < N/2; i++) {
      uint32_t u0 = x[N*r + i];
      uint32_t u1 = x[N*r + i + N/2];

      uint32_t t0 = u0 + u1;
      t0 -= ((t0 >= 2*p) ? (2*p) : 0);
      uint32_t t1 = u0 - u1 + 2*p;
      uint32_t q = ((uint64_t) t1 * winvtab[i]) >> 32;
      uint32_t t2 = t1 * wtab[i] - q * p;

      x[N*r + i]       = t0;
      x[N*r + i + N/2] = t2;
    }
    wtab += N/2; winvtab += N/2;
  }
}
```

Listing 5. Code simplifié de la boucle principale NTT.

Le compilateur est capable d'analyser la dernière boucle et de la vectoriser, impliquant ainsi un gain de performance.

Nous avons comparé les performances obtenues en compilant chacune des versions de la boucle avec différents drapeaux, simulant différentes architectures cibles :

— `-O3 -fno-tree-vectorize` : code portable, sans vectorisation
— `-O3 -march=westmere` : vectorisation, ciblant le jeu d'instructions SSE4

— `-O3 -march=core-avx2` : vectorisation en utilisation les instructions AVX2

Le tableau 10 résume les résultats obtenus. On observe que nos changements impliquent au moins une augmentation de performances dans tous les cas, et ce dû principalement à la vectorisation automatique rendue possible de la dernière boucle de la fonction `ntt`.

NTT (référence)			
Type	Sans vectorisation	SSE4	AVX2
64 bits	16.7μs	14.5μs	13.5μs
32 bits	6.9μs	4.3us	4.3μs
16 bits	7.4μs	5.0us	5.0μs
NTT (nouvelle boucle)			
Type	Sans vectorisation	SSE4	AVX2
64 bits	14.0μs	13.7μs	13.4μs
32 bits	5.7μs	3.6μs	3.4μs
16 bits	7.2μs	3.4μs	3.4μs

Table 10. Comparaison des performances du noyau NTT avec et sans vectorisation pour différentes architectures.

4.7 Vectorisation manuelle

Plusieurs boucles ont été vectorisées en utilisant les jeux d'instructions SSE et AVX2 (suivant disponibilité sur la machine cible). Les compilateurs modernes sont capables d'une telle transformation sur certains algorithmes. Une première étape a ainsi été d'inspecter le code généré par le compilateur utilisé (GCC 4.9.2), et de regarder s'il n'était pas possible de l'améliorer. Nous allons dans cette section décrire comment certaines opérations ont été optimisées grâce au jeu d'instructions SSE par rapport aux versions produites par le compilateur utilisé (GCC 4.9.2 sauf mention contraire). Nous n'avons pas décrit en détail l'optimisation de toutes les diverses opérations contenues dans la bibliothèque (et l'utilisation du jeu d'instructions AVX2), car les mêmes techniques ont globalement été utilisées.

Multiply high Une des opérations présente à plusieurs endroits dans la bibliothèque est `multiply high`. Pour deux nombres a et b entiers non signés de N bits, elle consiste à calculer $(a \times b) >> N$. Autrement dit, cette opération consiste à multiplier deux nombres de N bits et récupérer

la partie supérieure comprise entre les bits N et $2 \cdot N$. De manière plus détaillée, cette opération peut être décrite de la façon suivante :

```
tmp[0:2*N[ = a[0:N[ * b[0:N[
res[0:N[   = tmp[N:2*N[
```

L'équivalent en langage C avec des entiers 32 bits non signés est le suivant :

```
uint32_t mulhigh(uint32_t a, uint32_t b)
{
  return ((uint64_t)a*b)>>32;
}
```

Pour la suite, nous nous plaçons dans le cas d'entiers **non signés** 32 bits. Nous discuterons ensuite des versions avec des entiers non signés 64 et 16 bits.

L'instruction `mul` présente dans l'architecture Intel x86 32 bits multiplie deux entiers de 32 bits et renvoie le résultat sur 64 bits, réparti entre deux registres (`eax` pour la partie basse, et `edx` pour la partie haute). Il suffit ainsi de récupérer la valeur nécessaire dans le bon registre (`edx` en l'occurrence). Un compilateur comme GCC reconnaît cette opération et fait exactement cela. En effet, la version désassemblée de la fonction ci-dessus ressemble à ceci :

```
; uint32_t mulhi(const uint32_t a, const uint32_t b)
a= dword ptr  4
b= dword ptr  8
mov     eax, [esp+b]
mul     [esp+a]
mov     eax, edx
retn
```

Par contre, avec le jeu d'instructions Intel x86 64 bits, GCC utilise l'opération native 64 bit et fait un *shift* logique à droite de 32 bits :

```
; uint32_t mulhi(uint32_t a, uint32_t b)
; rdi = a                   ; uint32_t
; rsi = b                   ; uint32_t
mov     eax, edi
mov     esi, esi
imul    rax, rsi ; rsi = b
shr     rax, 20h ; rax >>= 32
retn
```

Il s'avère que sur un processeur Intel Core i7-3520M avec un système d'exploitation 64-bit, la version 64-bit est meilleure d'environ 10%.

Ce qui nous intéresse est l'utilisation de cette opération au sein d'une boucle, à savoir un code équivalent à celui-ci :

```
for (size_t i = 0; i < n; i++) {
  r[i] = mulhigh(a[i], b[i]);
}
```

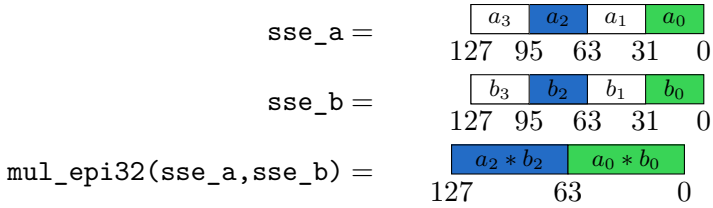
GCC arrive à vectoriser la boucle ci-dessus en utilisant le jeu d'instruction SSE ou AVX2 (et ce de la même façon sur x86 32 ou 64 bits). En prenant pour exemple la version SSE, le corps de boucle est le suivant :

```
; r11 = &a[0]
; r10 = &b[0]
; rax = &r[0]
; rcx = index de boucle
vmovdqa     xmm1, xmmword ptr [r11+rcx]
vmovdqu     xmm3, xmmword ptr [r10+rcx]
vpunpckldq xmm2, xmm1, xmm1
vpunpckhdq xmm1, xmm1, xmm1
vpunpckldq xmm4, xmm3, xmm3
vpunpckhdq xmm3, xmm3, xmm3
vpmuludq   xmm2, xmm2, xmm4
vpmuludq   xmm1, xmm1, xmm3
vpsrlq      xmm2, xmm2, 20h
vpsrlq      xmm1, xmm1, 20h
vshufps     xmm1, xmm2, xmm1, 88h
vmovups     xmmword ptr [rax+rcx], xmm1
add         rcx, 16
```

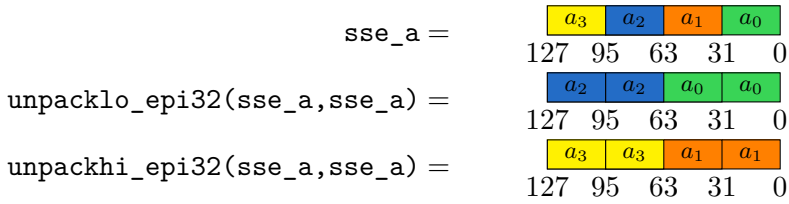
En réécrivant cela de manière symbolique, ce code est équivalant à :

```
sse_a    = sse_load(&a[rcx]);
sse_b    = sse_load(&b[rcx]);

alow     = unpacklo_epi32(sse_a, sse_a);
blow     = unpacklo_epi32(sse_b, sse_b);
mullowt  = mul_epu32(alow,  blow);
mullow   = srli_epi64(mullowt, 32)

ahigh    = unpackhi_epi32(sse_a, sse_a);
bhigh    = unpackhi_epi32(sse_b, sse_b);
mulhight = mul_epu32(ahigh, bhigh);
mulhigh  = srli_epi64(mulhight, 32);

sse_r   = shuffle_ps(mullow, mulhigh, 0x88);
sse_store(&r[rcx], sse_r);
rcx += 16
```

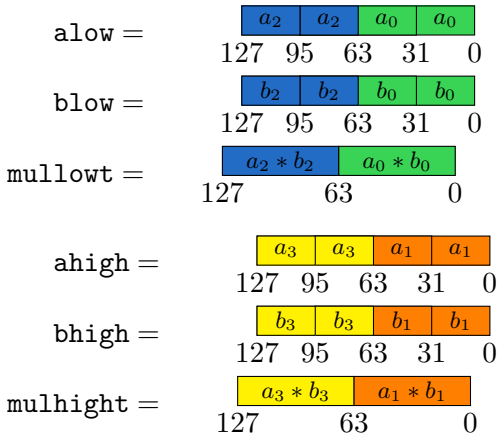
L'idée est donc d'utiliser l'instruction `mul_epu32`, qui va multiplier seulement deux entiers des quatre disponibles dans un registre SSE et stocker le résultat sur 64 bits, comme décrit sur le schéma ci-dessous :



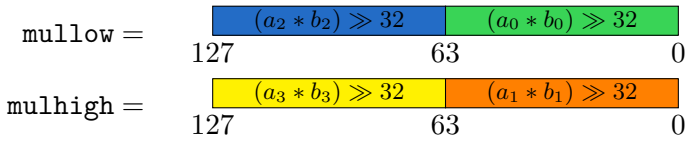
L'opération totale doit ainsi être faite grâce à 2 `mul_epu32`. Il faut ainsi ensuite réordonner les vecteurs `sse_a` et `sse_b`. Les instructions `unpacklo_epi32` et `unpackhi_epi32` sont utilisées à ces fins. Ces deux fonctions sont décrites ci-dessous :



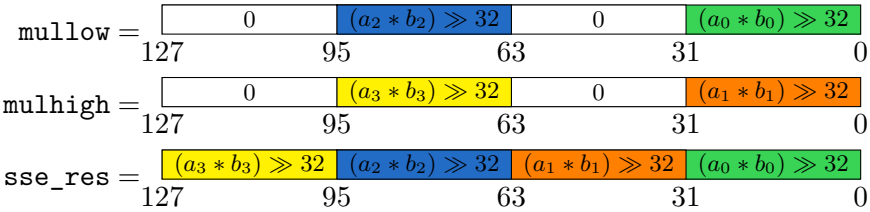
Ainsi, la combinaison de `unpack*_epi32` et `mul_epu32` va nous donner, pour `mullowt` et `mulhight` :



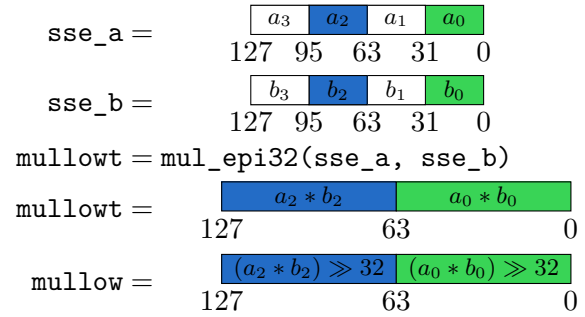
L'instruction `srli_epi64` pemet d'appliquer un *shift* logique sur les entiers 64 bits qui viennent d'être produits. Cela donne :



Afin de reconstruire le vecteur final, GCC utilise l'instruction `shuffle_ps` avec le coefficient 0x88, ce qui est équivalent à :

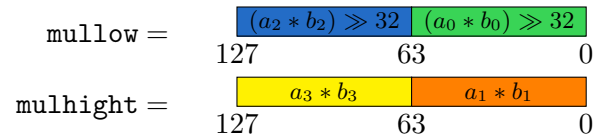


La question est de savoir si nous pouvons optimiser ce calcul. Il est en effet présent dans un certain nombres d'opérations de la bibliothèque. Si nous regardons un peu plus en détail, nous voyons que les deux premières opérations `unpacklo_epi32` sont inutiles. En effet, utiliser l'instruction `mul_epi32` juste après les deux opérations `sse_load` donne déjà la valeur de `mullowt`. Nous pouvons ensuite appliquer le décalage de 32 bits.

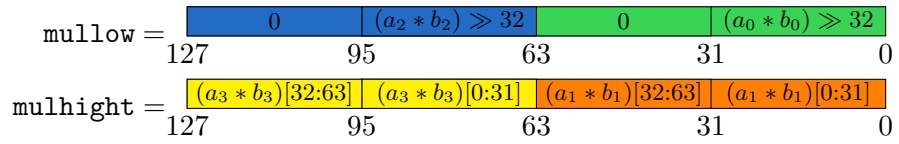


Pour calculer `mulhigh`, les opérations `unpackhi_epi32` restent par contre nécessaire. En effet, aucune instruction SSE utilisant des entiers sur 32 bits ne fait de multiplication permettant de faire

. Par contre, la dernière opération `srli_epi64` peut être omise. En effet, nous avons pour l'instant :



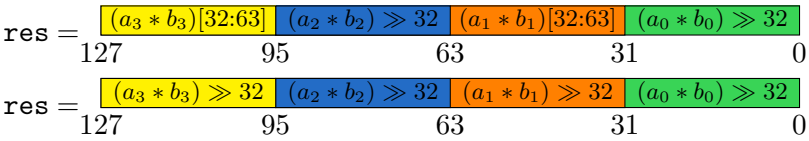
En considérant ces vecteurs comme quatre entiers de 32 bits, nous avons :



Les valeurs que nous recherchons sont ainsi déjà présentes mais de manière « entrelacées » dans les vecteurs `mullow` et `mulhight`. L'instruction `blend_ps` permet de mélanger ces deux vecteurs.

Ainsi, en utilisant une valeur de `0b1010` pour `imm8`, les deux vecteurs précédents peuvent être mélangés directement afin d'obtenir le résultat attendu :

`res` = `blend_ps(mullow, mulhight, 0b1010)`



L'algorithme final est ainsi le suivant :

```
sse_a    = sse_load(&a[rcx]);
sse_b    = sse_load(&b[rcx]);

mullowt  = mul_epu32(sse_a,  sse_b);
mullow   = srli_epi64(mullowt, 32)

ahigh    = unpackhi_epi32(sse_a, sse_a);
bhigh    = unpackhi_epi32(sse_b, sse_b);
mulhight = mul_epu32(ahigh, bhigh);

sse_r   = blend_ps(mullow, mulhigh, 0b1010);
sse_store(&r[rcx], sse_r);
rcx += 16
```

Nous avons donc gagné trois instructions (deux `unpack_epi32` et un `srli_epi32`). En ce qui concerne les entiers sur 16 bits, GCC va vectoriser la boucle d'origine de la même façon qu'il le fait pour 32 bits. Cela est dommage car il existe une instruction SSE qui fait directement ce que nous recherchons, à savoir `_mm_mulhi_epi16`. C'est cette instruction que nous allons utiliser dans ce cas-là. Pour les entiers sur 64 bits, il n'existe aucune instruction SSE/AVX/AVX2 permettant d'effectuer des multiplications sur ce type d'entiers. Le code séquentiel est ainsi utilisé.

Le même principe a été appliqué avec l'utilisation du jeu d'instructions AVX2.

L'opération `multiply high` est utilisée lors de plusieurs noyaux de calcul, à savoir :

— le calcul de la NTT
— l'opération `modular multiply shoup`
— l'opération `modular add and multiply shoup`

Modular multiply shoup Nous allons maintenant nous intéresser à l'opération `modular multiply shoup`, disponible dans le listing 4.7.

```
template<class T>
struct mulmod_shoup<T, simd::serial> {
```

```
  using simd_mode = simd::serial;

  T operator()(T x, T y, T yprime, T p) const
  {
    // Utilisation du type d'entier de taille superieur (64 bits
        pour 32 bits par exemple)
    using greater_value_type = typename params<T>::
        greater_value_type;
    greater_value_type res;

    // Calcul du "multiply high"
    T q = ((greater_value_type) x * yprime) >> params<T>::
        kModulusRepresentationBitsize;

    // Finalisation (calculs sur 64 bits)
    res = x * y - q * p;
    return res - ((res>=p) ? p : 0);
  }
};
```

Lorsque des entiers de 16 bits sont utilisés, GCC 4.9 n'arrive pas à vectoriser cette opération, principalement à cause du passage par `greater_value_type`. Or obtenir une version vectorielle ce code est possible, et une écriture manuelle est ainsi nécessaire.

Nous avons vu ci-dessus comment faire l'opération `multiply high`, et pouvons ainsi déjà réutiliser cette opération. Nous avons ainsi pour l'instant :

```
sse_q = mulhi_epi16(sse_x, sse_yprime)
```

Pour effectuer les opérations $x \times y$ et $q \times p$, nous devons d'abord convertir nos vecteurs d'entiers de 16 bits en vecteur d'entiers de 32 bits. Dans le cas où seul le jeu d'instruction SSE est disponible, le résultat de cette conversion sera stocké dans un registre 128 bits. Il faudra ainsi faire l'opération en deux temps, en traitant d'abord les quatre premiers entiers de 16 bits puis réitérer avec les quatre derniers.

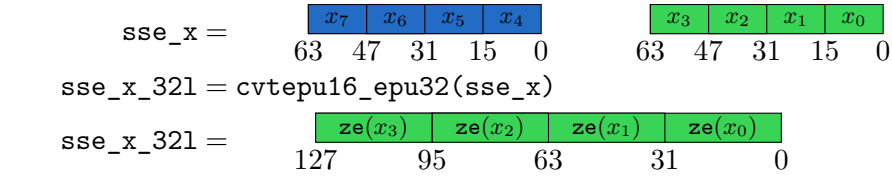
Nous avons premièrement besoin d'un vecteur de quatre entiers de 32 bits contenant tous la valeur de p. Cela se fait simplement avec l'instruction `set1_epi32` :

`sse_p` = `set1_epi32(p)`

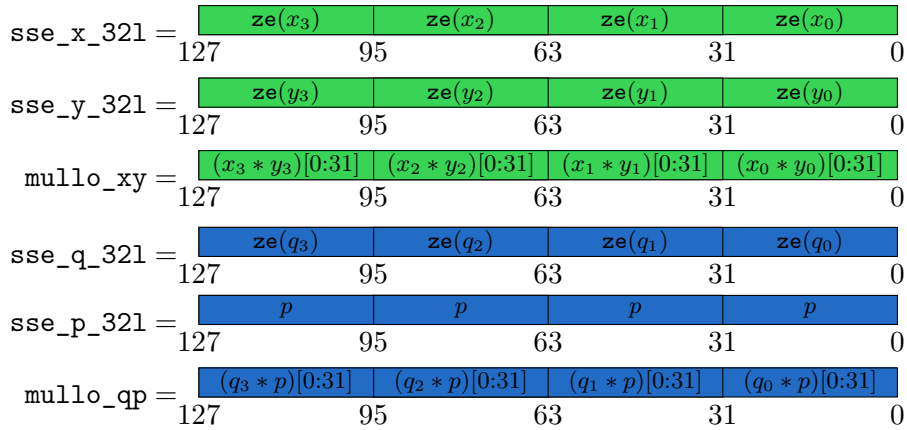
`sse_p` =

p	p	p	p
127	95	63	31	0

L'instruction `cvtepu16_epu32` nous permet de convertir les quatre premiers entiers de 16 bits en quatre entiers de 32 bits (avec `ze` = `zero_extend`) :



En faisant la même opération avec `sse_y`, `sse_q` et `sse_p`, nous pouvons ensuite utiliser l'instruction `mullo_epi32`, qui multiple deux entiers 32 bits et ne renvoie que la partie basse, créant ainsi un vecteur de quatre entiers de 32 bits avec le résultat attendu :



Il ne reste plus qu'à soustraire ces deux résultats intermédiaire avec l'opération `sub_epi32`.

`sse_res` = `sub_epi32(mullow_xy,mullow_qp)`

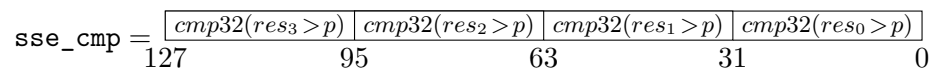
Il faut maintenant effectuer la dernière opération, à savoir :

```
res -= ((res>=p) ? p : 0);
```

Concrètement, il faut comparer `sse_res` à `sse_p`, et suivant ce résultat, soustraire ou non l'entier `p` à l'entier correspondant dans `sse_res`.

L'opération `cmp_epi32` va donner ce résultat (avec `cmp32(b) = b ? 0xFFFFFF : 0`) :

`sse_cmp` = `cmp_epi32(sse_res, sse_p)`



Nous avons cependant deux problèmes ici :

— la comparaison est stricte, il faut donc comparer à $p - 1$ (surmontable :))

— la comparaison considère que les entiers sont signés (or nous travaillons avec des entiers non signés), et sera ainsi invalide si $res_i > 2^{31}$ ou $p > 2^{31}$ (ou exclusif) (un peu plus problématique)

Une technique classique pour résoudre le second problème est de simplement soustraire 2^{31} (soit `0x80000000`) aux deux entiers à comparer. En effet, cela a pour effet de « rétablir » l'ordre de nos deux entiers non signés dans l'espace des entiers signés en projetant 0 sur -2^{31}.

Une fois que la comparaison est faite et valide, il suffit de prendre le vecteur résultat et de faire un ET bit à bit avec `sse_p` :

`sse_cmp` =

$cmp32(res_3 \geq p)$	$cmp32(res_2 \geq p)$	$cmp32(res_1 \geq p)$	$cmp32(res_0 \geq p)$
127	95	63	31 0

`sse_p` =

p	p	p	p
127	95	63	31 0

`sse_cmp = and_si128(sse_cmp, sse_p)`

`sse_cmp` =

$res_3 \geq p?p:0$	$res_2 \geq p?p:0$	$res_1 \geq p?p:0$	$res_0 \geq p?p:0$
127	95	63	31 0

`sse_res = sub_epi32(sse_res, sse_cmp)`

L'algorithme final pour traiter les quatre premiers entiers de 16 bits est donc le suivant :

```
sse_x_32l = cvtepu16_32(sse_x)
sse_y_32l = cvtepu16_32(sse_y)
sse_q_32l = cvtepu16_32(sse_q)
sse_p     = set1_epi32(p)
sse_80    = set1_epi32(0x80000000)
sse_pc    = set1_epi32(p-0x80000000-1)

mullo_xy  = mullo_epi32(sse_x_32l, sse_y_32l)
mullo_qp  = mullo_epi32(sse_q_32l, sse_p)
sse_res   = sub_epi32(mullo_xy, mullo_qp)

sse_cmp   = cmpgt_epi32(sub_epi32(sse_res, sse_80), sse_pc)
sse_res   = sub_epi32(sse_res, and_si128(sse_cmp, sse_p))
```

Pour traiter les quatre derniers entiers de 16 bits, une permutation est effectuée sur `sse_x`, `sse_y` et `sse_q` afin de placer ces quatre derniers entiers en premiers, et réitérer le même processus.

Nous arrivons ainsi avec ces deux vecteurs :

`sse_reslow` =

res_3	res_2	res_1	res_0
127	95	63	31 0

`sse_reshigh` =

res_7	res_6	res_5	res_4
127	95	63	31 0

Il reste donc à tronquer chaque entier de 32 bits en 16 bits et créer le résultat final. L'instruction `packus_epi32` fait exactement cela :

`sse_res = packus_epi32(sse_reslow, sse_reshigh)`

`sse_res` =

res_7	res_6	res_5	res_4		res_3	res_2	res_1	res_0
63	47	31	15	0	63	47	31	15 0

Nous avons ainsi le résultat escompté !

En ce qui concerne l'opération `modular multiply shoup` avec des entiers 32 bits, GCC arrive ici à vectoriser la boucle. Elle est néanmoins spécialisée dans la bibliothèque afin de tenir compte des optimisations effectuées sur l'opération `multiply high`. Nous n'avons pas connaissance d'une technique permettant de faire ces mêmes calculs vectoriels avec des entiers de 64 bits, principalement à cause du manque de multiplication vectorielles avec ce type d'entiers dans les jeux d'instructions SSE et AVX2. GCC ne vectorise pas cette opération.

Gains de performances obtenus *Benchmarker* individuellement chaque opération n'est pas évident, car la limite du bus mémoire est vite atteinte sur ce type d'opérations. Par exemple, si nous mesurons le temps d'exécution des versions optimisées par GCC et écrites manuellement de l'opération `multiply high`, nous obtenons ceci (sur un processeur Intel Core i7-3520M avec un OS 64 bit) :

```
gcc-32: 41.6198 ms / IN BW: 18331.2 MB/s / OUT BW: 9165.59 MB/s
man-32: 41.423 ms  / IN BW: 18418.2 MB/s / OUT BW: 9209.12 MB/s
gcc-16: 20.7753 ms / IN BW: 18361.7 MB/s / OUT BW: 9180.84 MB/s
man-16: 20.5924 ms / IN BW: 18524.7 MB/s / OUT BW: 9262.37 MB/s
```

`gcc-*` est associé au code vectorisé généré par gcc. `man-*` correspond au code vectorisé écrit manuellement.

Le calcul a été fait sur des vecteurs de 10.000.000 entiers, lancé 100 fois en moyennant les temps d'exécution et en ne prenant pas en compte les temps aux extrémités (pour ne pas fausser les résultats par des cas dégénérés).

Nous pouvons voir que les versions 16 bits vont « deux fois plus vite » que les versions 32 bits. Cela n'est pas dû au fait que les instructions vectorielles travaillant sur des entiers 16 bits vont deux fois plus vite leurs homologues sur 32 bits, mais bien que la limite dans ce cas n'est pas le CPU, mais le bus mémoire. En effet, le vecteur de 10.000.000 entiers de 16 bits est bel et bien deux fois plus petit que celui composé d'entiers de 32 bits. Cela se confirme avec le calcul de la bande passante moyenne de traitement du calcul, qui reste constante entre les versions 16 et 32 bits. Ainsi, le gain de la vectorisation manuelle n'est ici pas apparent.

La différence de performance obtenue grâce au travail précédent commence à se voir lorsque les différentes opérations optimisées sont enchaînées, comme la transformation NTT (voir section 4.6). Ainsi, en compilant le code de la bibliothèque en laissant GCC vectoriser les boucles

de calcul, nous obtenons comme performances (temps par opération) ($G = temps\_gcc/temps\_manuel$) [9] :

NTT (vectorisée par GCC, résultats de la section 4.6)		
Type	SSE	AVX2
32 bits	3.6μs	3.4μs
16 bits	3.4μs	3.4μs
NTT (vectorisation manuelle)		
Type	SSE	AVX2
32 bits	2.4μs (G = 1.5)	1.9μs (G = 2.8)
16 bits	1.2μs (G = 2.8)	0.9μs (G = 3.7)

Le cas lors d'un chiffrement ou déchiffrement asymétrique a aussi été *benchmarké*, soit en laissant GCC vectoriser les différentes boucles, soit en utilisant les versions vectorielles décrites ci-dessus. Nous obtenons les résultats suivant (avec ($G = temps\_gcc/temps\_manuel$) :

Chiffrement (vectorisée par GCC)			
Type	Sans vectorisation	SSE	AVX2
32 bits	27.8μs	26.1μs	17.3μs
16 bits	23.3μs	22.1μs	21.3μs
Chiffrement (vectorisation manuelle)			
Type	Sans vectorisation	SSE	AVX2
32 bits	27.8μs	22.8μs (G = 1.1)	14.0μs (G = 1.2)
16 bits	23.3μs	15.4μs (G = 1.4)	13.6μs (G = 1.6)
Déchiffrement (vectorisée par GCC)			
Type	Sans vectorisation	SSE	AVX2
32 bits	5.1μs	5.2μs	4.7μs
16 bits	6.2μs	6.2μs	5.9μs
Déchiffrement (vectorisation manuelle)			
Type	Sans vectorisation	SSE	AVX2
32 bits	5.1μs	3.6μs (G = 1.4)	3.0μs (G = 1.6)
16 bits	6.2μs	2.7μs (G = 2.3)	2.2μs (G = 2.7)

Nouvel état des lieux avec `valgrind` Après toutes les optimisations décrites ci-dessus, une nouvelle étude avec `valgrind/callgrind` a été effectuée, pour les mêmes opérations de chiffrement et de déchiffrement définies à la section 4.5.

9. On notera que l'accélération sur 64 bits n'a ici pas de sens car aucune opération n'a pu être réécrite.

En ce qui concerne le chiffrement, la fonction de génération d'aléa occupe maintenant $\sim 41\%$ du temps d'exécution de la fonction `encrypt`, contre $\sim 16\%$ au départ. Cette fonction ayant été la seule à ne pas avoir été optimisée, il est normal qu'elle devienne la partie dominante du chiffrement. Il pourrait ainsi maintenant être intéressant de voir si des optimisations sont possibles au sein de la génération d'aléa.

Pour la fonction de déchiffrement, la fonction prépondérante reste les fonctions `inv_ntt` et `ntt`, même si moins de temps CPU y est relativement consacré. Il pourrait ainsi être intéressant d'essayer d'améliorer encore cette fonction.

Conclusion

NFLlib permet d'obtenir des performances très supérieures à l'existant sur des preuves de concept. Pour passer à un prototypage plus près d'un usage réel, il reste encore de nombreux jalons. Parmi eux, la résistance aux attaques par canaux cachés est probablement le plus important. Une telle propriété aura sans le moindre doute un coût significatif et il ne sera pas trivial de le limiter.

La bibliothèque NFLlib est en développement mais vise, à terme, à être utilisée dans des applications grand public. Pour le moment, elle reste un outil de recherche permettant d'expérimenter simplement sur des algorithmes cryptographiques actuels et nouveaux. Nous espérons que le niveau de performance des versions ultérieures sera à la hauteur de ces premiers résultats.

Références

1. Carlos Aguilar-Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. XPIRe : Private Information Retrieval for Everyone. 2014.
2. Daniel J. Bernstein. The salsa20 family of stream ciphers. In Matthew J. B. Robshaw and Olivier Billet, editors, *New Stream Cipher Designs - The eSTREAM Finalists*, volume 4986 of *LNCS*, pages 84–97. Springer, 2008.
3. Joppe W. Bos, Craig Costello, Michael Naehrig, and Douglas Stebila. Post-quantum key exchange for the TLS protocol from the ring learning with errors problem. August 2014.
4. Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *TOCT*, 6(3) :13, 2014.
5. Yuanmi Chen and Phong Q. Nguyen. BKZ 2.0 : Better lattice security estimates. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*, volume 7073 of *LNCS*, pages 1–20. Springer, 2011.

6. Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal Gaussians. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, volume 8042 of *LNCS*, pages 40–56. Springer, 2013.
7. Pierre Estérie, Mathias Gaunard, Joel Falcou, Jean-Thierry Lapresté, and Brigitte Rozoy. Boost. simd : generic programming for portable simdization. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 431–432. ACM, 2012.
8. Sanjam Garg, Craig Gentry, and Shai Halevi. Candidate multilinear maps from ideal lattices. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, volume 7881 of *LNCS*, pages 1–17. Springer, 2013.
9. Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, pages 169–178. ACM, 2009.
10. Tim Güneysu, Tobias Oder, Thomas Pöppelmann, and Peter Schwabe. Software speed records for lattice-based signatures. In Philippe Gaborit, editor, *Post-Quantum Cryptography - 5th International Workshop, PQCrypto 2013, Limoges, France, June 4-7, 2013. Proceedings*, volume 7932 of *LNCS*, pages 67–82. Springer, 2013.
11. Shai Halevi and Victor Shoup. Algorithms in helib. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*, volume 8616 of *LNCS*, pages 554–571. Springer, 2014.
12. William Hart et al. *Fast Library for Number Theory (Version 2.4.4)*, 2015. `http://www.flintlib.org`.
13. Arjen K. Lenstra, Hendrik W. Lenstra Jr., and László Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261(4) :515–534, 1982.
14. Richard Lindner and Chris Peikert. Better key sizes (and attacks) for lwe-based encryption. In Aggelos Kiayias, editor, *Topics in Cryptology - CT-RSA 2011 - The Cryptographers' Track at the RSA Conference 2011, San Francisco, CA, USA, February 14-18, 2011. Proceedings*, volume 6558 of *LNCS*, pages 319–339. Springer, 2011.
15. Michael Naehrig, Kristin E. Lauter, and Vinod Vaikuntanathan. Can homomorphic encryption be practical ? In Christian Cachin and Thomas Ristenpart, editors, *Proceedings of the 3rd ACM Cloud Computing Security Workshop, CCSW 2011, Chicago, IL, USA, October 21, 2011*, pages 113–124. ACM, 2011.
16. Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *J. ACM*, 56(6), 2009.
17. Claus-Peter Schnorr and M. Euchner. Lattice basis reduction : Improved practical algorithms and solving subset sum problems. *Math. Program.*, 66 :181–199, 1994.
18. Victor Shoup. *Number Theory Library (Version 8.1)*, 2015. `http://www.shoup.net/ntl`.
19. Todd Veldhuizen. Expression templates. *C++ Report*, 7 :26–31, 1995.

A Appendices : Code

```
// Pour le choix de ces parametres voir http://eprint.iacr.org
    /2010/613
NOISE_PARAM=8.35
NOISE_AMPLIIFER=2

// On utilise n=256 et log q = 14 pour k = 128
if( k == 128) using poly_t = poly<uint16_t, 256, 1>;
// Et  n=512 et log q = 14 pour k = 256
else if( k == 256) using poly_t = poly<uint16_t, 512, 1>;
// Pas d'autre parametre de securite n'est possible pour simplifier
else exit(1);

// Generation de la cle secrete suivant une gaussienne
sk = nfl::non_uniform(NOISE_PARAM);

// Generation de la cle publique (pka,pkb)
pka = nfl::uniform();
pkb = nfl::non_uniform(NOISE_PARAM, NOISE_AMPLIFIER);
pkb = pkb + pka * sk;
```

Listing 6. Génération d'un couple clé publique/privée

```
// Entrees :
// pk = (pka,pkb) represente la cle publique
// m = message a chiffrer (polynome)

// Generation de bruits gaussiens
u = nfl::non_uniform(NOISE_PARAM);
e1 = nfl::non_uniform(NOISE_PARAM,NOISE_AMPLIFIER);
e2 = nfl::non_uniform(NOISE_PARAM,NOISE_AMPLIFIER);

resa = pka * u + e1;
resb = pkb * u + e2 + m;

// Sortie
// alpha = (resa, resb)
```

Listing 7. Chiffrement grâce à une clé publique

```
// Entrees :
// sk  represente la cle privee
// alpha = (a1, a2) chiffre obtenu par la fonction de chiffrement

res = a2 - a1 * sk;
for(coefficient v: res)
{
  v = (v<modulus/2) ? v%2 : 1-v%2;
}

// Sortie: res, message dechiffre
```

Listing 8. Déchiffrement grâce à une clé privée