Quatre millions d'échanges de clés par seconde

Carlos Aguilar, Serge Guelton, Adrien Guinet, Tancrède Lepoint

IRT, IRIT, ENSEEIHT, INP Toulouse, Université de Toulouse
carlos.aguilar@enseeiht.fr
Quarkslab
{sguelton,aguinet}@quarkslab.com
CryptoExperts
{tancrede.lepoint}@cryptoexperts.com

June 4, 2015

Plan

Plan

Messages à passer

Ceci est une apologie

Lattice-based encryption rocks

Vitesse

Entre $\times 10^2$ et $\times 10^6$
Coût quasi-linéaire en la sécurité: $k \log k$ (DH k^3, RSA k^7)

Fait amusant : chiffrement à clé publique plus rapide qu'AES

Simplicité

Conceptuelle : chiffré = message + OTP, OTP = alea$\times$pubkey + bruit
Algorithmique : `for (i=0;i<1024;i++) res[i] = (op[i] * op2[i]) % p;`

Défi : one-liner pour une exponentiation RSA ou un produit scalaire-point (ECDH)

Standardisation

On ne devrait pas attendre le NIST pour faire un standard...
Difficulté : Lattice-based signature ****

Messages à passer

Ceci est une apologie

Lattice-based encryption rocks

Vitesse

Entre $\times 10^2$ et $\times 10^6$
Coût quasi-linéaire en la sécurité: $k \log k$ (DH k^3, RSA k^7)

Fait amusant : chiffrement à clé publique plus rapide qu'AES

Simplicité

Conceptuelle : chiffré = message + OTP, OTP = alea$\times$pubkey + bruit
Algorithmique : `for (i=0;i<1024;i++) res[i] = (op[i] * op2[i]) % p;`

Défi : one-liner pour une exponentiation RSA ou un produit scalaire-point (ECDH)

Standardisation

On ne devrait pas attendre le NIST pour faire un standard...
Difficulté : Lattice-based signature sucks

Messages à passer

Ceci est une apologie

Lattice-based encryption rocks

Vitesse

Entre $\times 10^2$ et $\times 10^6$
Coût quasi-linéaire en la sécurité: $k \log k$ (DH k^3, RSA k^7)

Fait amusant : chiffrement à clé publique plus rapide qu'AES

Simplicité

Conceptuelle : chiffré = message + OTP, OTP = alea$\times$pubkey + bruit
Algorithmique : `for (i=0;i<1024;i++) res[i] = (op[i] * op2[i]) % p;`

Défi : one-liner pour une exponentiation RSA ou un produit scalaire-point (ECDH)

Standardisation

On ne devrait pas attendre le NIST pour faire un standard...
Difficulté : Lattice-based signature ****

Qui veut faire des échanges de clés (KX) ?

RFC TLS1.0

```
Client                                               Server

ClientHello                  -------->
                                                ServerHello
                                               Certificate*
                                         ServerKeyExchange*
                                        CertificateRequest*
                             <--------      ServerHelloDone
Certificate*
ClientKeyExchange
CertificateVerify*
[ChangeCipherSpec]
Finished                     -------->
                                          [ChangeCipherSpec]
                             <--------             Finished
Application Data             <------->     Application Data
```

En pratique pour tout protocole sécurisé par un tunnel

IPSec, OpenVPN, SSH, WPA2 Enterprise, STARTTLS...

Ok c'est utilisé, mais est-ce coûteux ?

Recommandations standard

PFS : *Perfect Forward Secrecy* (confidentialité persistante)

- Si on compromet un serveur, on pourra obtenir les clés de session futures
- *Pas les passées,* même si on a enregistré les communications
- [Itkis 2004] Deux protocoles KX $\Rightarrow$ Protocole KX avec PFS
- Le surcoût de la PFS est au plus un facteur deux

128 bits de sécurité

Cas d'étude : serveurs à forte charge HTTPS

Utilisation d'un reverse proxy (e.g. Nginx, 70K connexions/s) faisant de l'équilibrage de charge et réalisant l'échange TLS

Résultat

Avec openssl 1.0.1f et Diffie-Hellman avec P-256 (NIST)
Pour 70K connexions/s il faut 6 Corei7-4770 à 100% (60 CPUs pour RSA!)

NFLProlib : 3.5% d'utilisation d'un unique processeur (/200 par rapport à DH)

Protocole

NIST SP800-56B

```
Client                                                    Serveur

                                  <--------      Clé publique certifiée pks
Aléa r chiffré par pks            -------->
Dérivation d'un secret de r                   Dérivation d'un secret de r
```

Variante avec PFS [Itkis 2004]

```
Client                                                    Serveur

                                                Clé publique certifiée pks
                                  <--------      Clé publique éphémère pke
Aléa r chiffré par pks
Aléa r' chiffré par pke           -------->
Dérivation d'un secret de r et r'           Dérivation d'un secret de r et r'
```

Facteur limitant

Opérations à réaliser par le serveur

Protocole	80 bits	128 bits	256 bits
RSA	7.7 Kops/s	0.34 Kops/s	N/A
ECDH	7.6 Kops/s	5.8 Kops/s	1.9 Kops/s
NFLProlib RLWE	N/A	4080 Kops/s	2032 Kops/s

Contributions

Algorithmique et implémentation

Nous n'avons pas inventé un nouvel algo de chiffrement

Nous avons optimisé les opérations de base dans les réseaux : NFLProlib
Nous avons étudié et optimisé la librairie en particulier pour les échanges de clés

Travaux proches

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" http://eprint.iacr.org/2014/599.pdf

Approche de type DH complètement intégrée dans openssl
Perfs de l'ordre de ECDH

Disclaimer

Usages immédiats et standardisation

Si vous mettez ça sur votre serveur web comme seul algo ça va pas passer...
Machin+ peut l'utiliser pour ses décodeurs s'il le souhaite

Les comparaisons sont simples

Nous isolons la partie la plus coûteuse des échanges de clés pour le serveur
Nous voyons combien de fois par seconde un ordi peut les faire

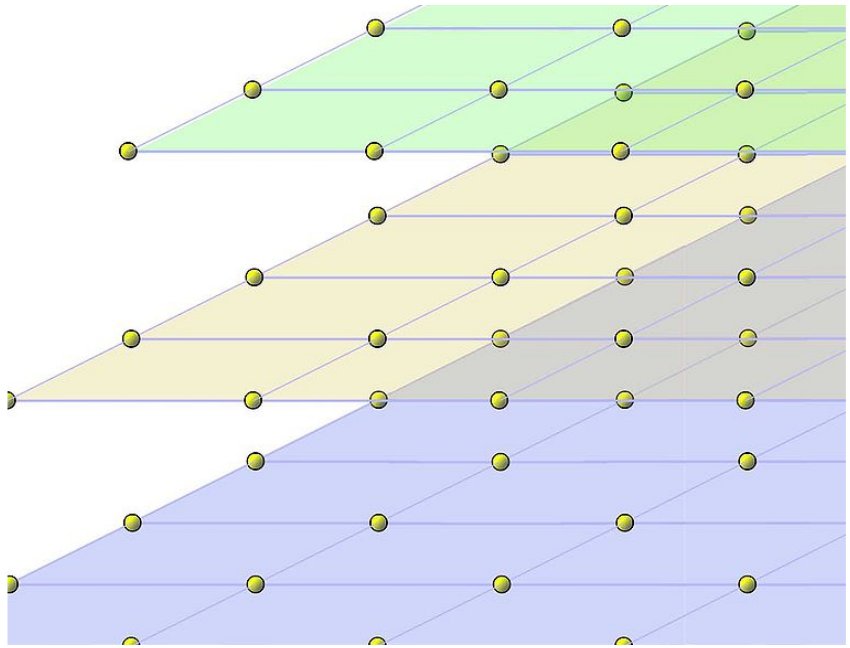
Un produit final sera moins performant

Intégré dans une code/protocole plus complexe (openssl, TLS)
Avec une meilleure résistance aux attaques par canaux cachés
$\Rightarrow$ Perte d'un facteur 1.2, 2, 3, ..., 10 ? Restera très intéressant

Plan

Le ras de marée des réseaux

$$\begin{pmatrix} 1 & 1 & 3 \\ 0 & 2 & 1 \\ 0 & 0 & 2 \end{pmatrix} \times \begin{pmatrix} z_1 \\ z_2 \\ z_3 \end{pmatrix}$$

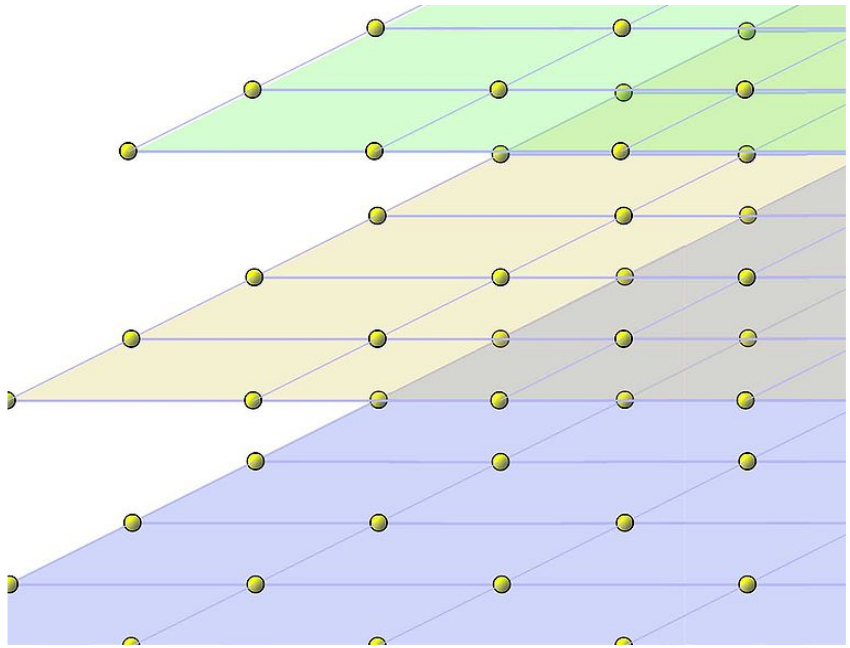


La cryptographie basée sur les réseaux euclidiens

Ajtai 1996 : Première preuve de sécurité pire-cas/cas-moyen
Faibles coûts (quadratiques depuis 1996, quasi-linéaires depuis 2008)

Le ras de marée des réseaux

$$\begin{pmatrix} 1 & 1 & 3 \\ 0 & 2 & 1 \\ 0 & 0 & 2 \end{pmatrix} \times \begin{pmatrix} z_1 \\ z_2 \\ z_3 \end{pmatrix}$$

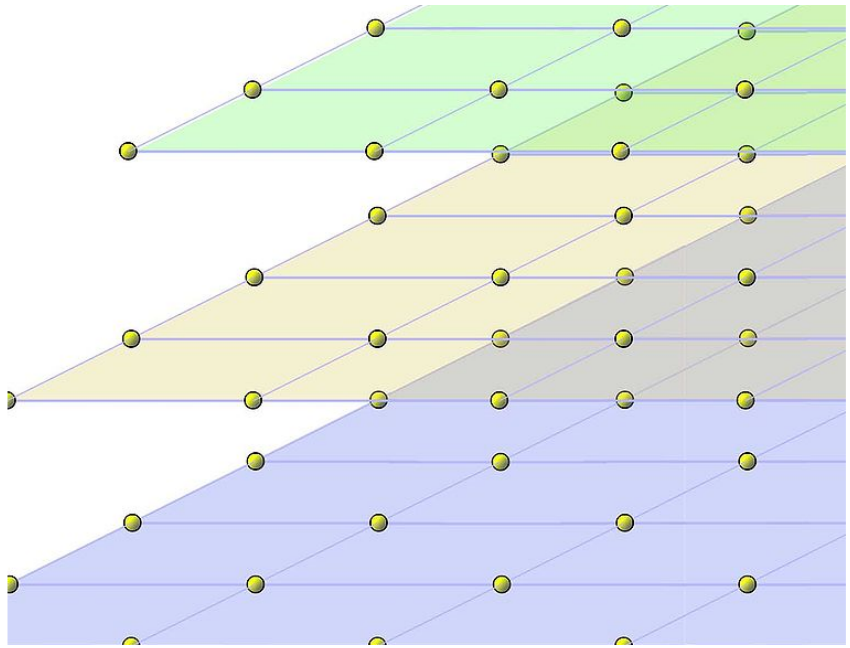


Tailles habituelles

Vecteurs d'entre 100 et 1000 coordonnées (transport de clés)
Scalaires modulo un entier de petite taille (e.g. 16, ou 32 bits)

Le ras de marée des réseaux

$$\begin{pmatrix} 1 & 1 & 3 \\ 0 & 2 & 1 \\ 0 & 0 & 2 \end{pmatrix} \times \begin{pmatrix} z_1 \\ z_2 \\ z_3 \end{pmatrix}$$



Cas typique

Chiffré : Vecteur proche d'un point au hasard dans le réseau
Clair : Correction nécessaire pour retomber sur le réseau (plus petit vecteur)

Éléments manipulés

Des maths ! Fuyez !

Polynômes de degré $n-1$ (n termes) avec des coefficients entiers modulo p
Somme classique : se fait coefficient à coefficient modulo p
Produit : produit polynômial puis remplacement de X^n par -1

Petit exemple, pour (n=3, p=5) :

- Polynômes avec degré inférieur à 3 et coefficients tous plus petits que 5, par exemple : $r = 4 + 2X + 2X^2 \leftrightarrow (4, 2, 2)$, $r = 2 + X + 3X^2 \leftrightarrow (2, 1, 3)$
- Somme classique : $r + s = (1, 3, 0)$
- Multiplication : $r * s = (4 + 2X + 2X^2) * (2 + X + 3X^2)$

$$\begin{aligned} &= 4(2 + X + 3X^2) + 2(2X + X^2 + 3X^3) + 2(2X^2 + X^3 + 3X^4) \\ &= 4(2 + X + 3X^2) + 2(2X + X^2 - 3) + 2(2X^2 - 1 - 3X) \\ &= 0 - 2X + 18X^2 = 0 - 2X + 3X^2 \end{aligned}$$

Coûts

Addition : n sommes modulaires
Multiplications : n^2 multiplications modulaires

NTT: Number-Theoretic Transform

Qu'est-ce ?

C'est juste une FFT sur les entiers : 10 lignes de code

Idée

Un polynôme de degré n est caractérise par ses n coefficients
Il l'est aussi par son évaluation sur n points différents $(f(x_1), \ldots, f(x_n))$

NTT : passage d'une représentation à l'autre ($n \log n$ opérations, typiquement $10n$)

Operations

$f = (f_1, \ldots, f_n)$, $g = (g_1, \ldots, g_n)$, vecteurs de valeurs de deux polynômes
$f + g$ et fg obtenus par des opérations coordonnée à coordonnée

Opérations équivalentes

Multiplication RSA

ECDH : Somme de deux points sur une courbe elliptique
Réseaux : Somme de deux polynômes
n additions natives et soustractions conditionnelles

Exponentiation RSA

ECDH : Multiplication d'un point par un scalaire sur une courbe elliptique (*double and add*)
Réseaux : Produit de deux polynômes
n multiplications natives et calculs d'un reste
Possibilité de faire deux multiplications + deux soustractions conditionnelles

Opération serveur (déchiffrement) dans le KX

```
for(int i=0; i<256;i++) clair[i] = (échiffr2[i] - échiffr1[i] * secret[i])%p;
clair.inv_ntt();
for(int i=0; i<256;i++) clair[i] = (clair[i]<p/2) ? clair[i]%2 : 1-clair[i]%2;
```

Plan

Les instructions vectorielles (SIMD)

Définition

SIMD = Single Instruction Multiple Data

- Permet d'effectuer en une instruction la même opération sur plusieurs entiers/flottants.
- Instructions `SSE` avec registres de 128 bits (`XMM*`)
- Instructions `AVX(2)` avec registres de 256 bits (`YMM*`)

Les instructions vectorielles (SIMD)

Définition

SIMD = Single Instruction Multiple Data

- Permet d'effectuer en une instruction la même opération sur plusieurs entiers/flottants.
- Instructions `SSE` avec registres de 128 bits (`XMM*`)
- Instructions `AVX(2)` avec registres de 256 bits (`YMM*`)

Exemple : addition de 4 entiers de 32 bits (registre de 128 bits)

Les instructions vectorielles (SIMD)

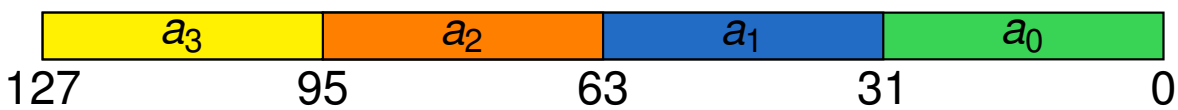
Définition

SIMD = Single Instruction Multiple Data

- Permet d'effectuer en une instruction la même opération sur plusieurs entiers/flottants.
- Instructions `SSE` avec registres de 128 bits (`XMM*`)
- Instructions `AVX(2)` avec registres de 256 bits (`YMM*`)

Exemple : addition de 4 entiers de 32 bits (registre de 128 bits)

XMM_a =



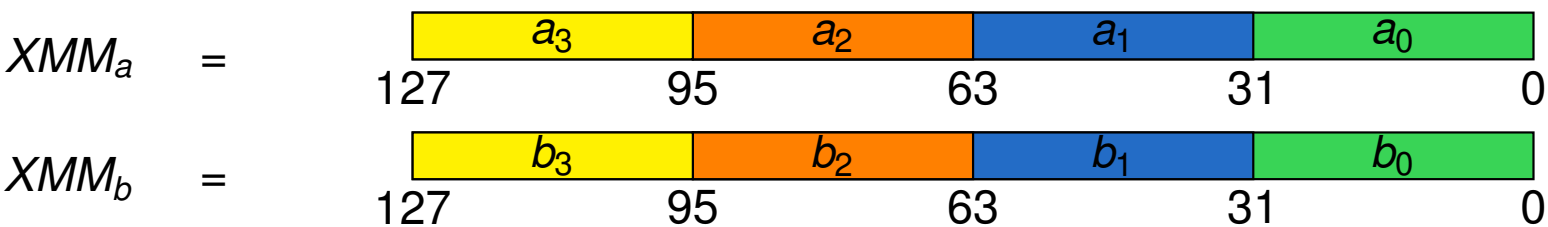
Les instructions vectorielles (SIMD)

Définition

SIMD = Single Instruction Multiple Data

- Permet d'effectuer en une instruction la même opération sur plusieurs entiers/flottants.
- Instructions `SSE` avec registres de 128 bits (`XMM*`)
- Instructions `AVX(2)` avec registres de 256 bits (`YMM*`)

Exemple : addition de 4 entiers de 32 bits (registre de 128 bits)



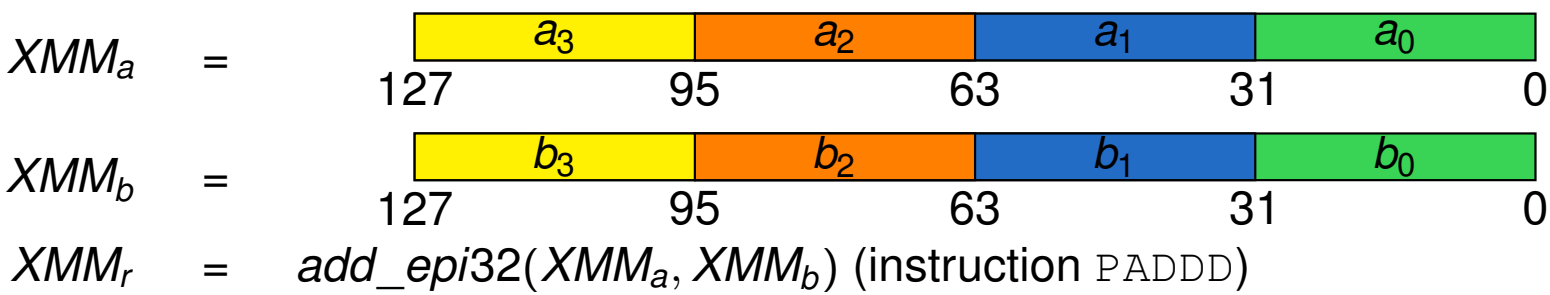
Les instructions vectorielles (SIMD)

Définition

SIMD = Single Instruction Multiple Data

- Permet d'effectuer en une instruction la même opération sur plusieurs entiers/flottants.
- Instructions `SSE` avec registres de 128 bits (`XMM*`)
- Instructions `AVX(2)` avec registres de 256 bits (`YMM*`)

Exemple : addition de 4 entiers de 32 bits (registre de 128 bits)



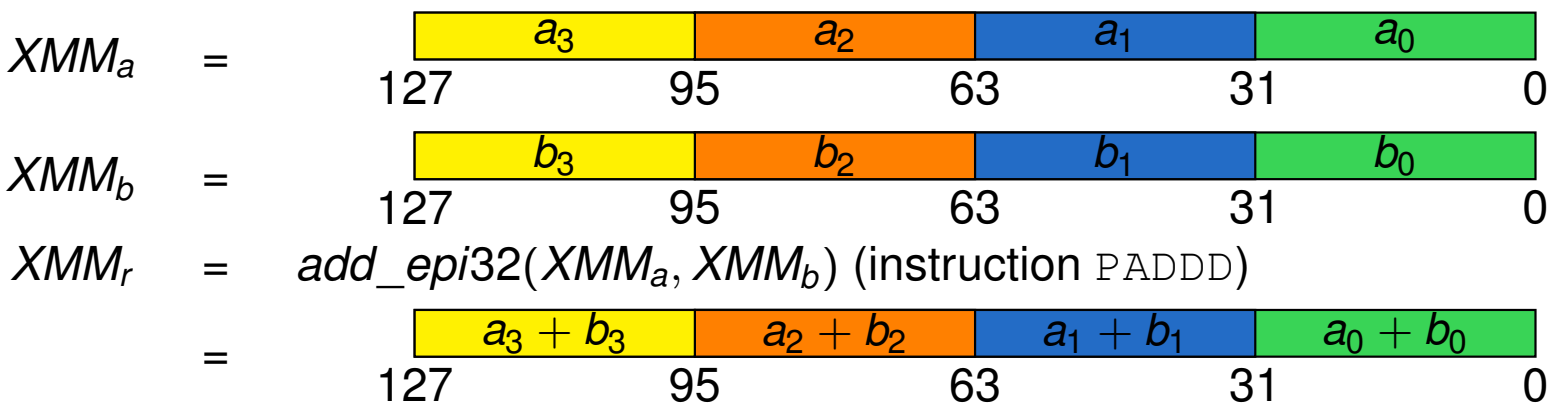
Les instructions vectorielles (SIMD)

Définition

SIMD = Single Instruction Multiple Data

- Permet d'effectuer en une instruction la même opération sur plusieurs entiers/flottants.
- Instructions `SSE` avec registres de 128 bits (`XMM*`)
- Instructions `AVX(2)` avec registres de 256 bits (`YMM*`)

Exemple : addition de 4 entiers de 32 bits (registre de 128 bits)



Module LaTeX pour faire des vecteurs (c) Serge Guelton. Vous pouvez lui payer du cidre si vous êtes intéressés.

État des lieux

Code étudié

- Étude des performances d'un algorithme de chiffrement et déchiffrement asymétrique utilisant `nfllib`

⇒ diminuer les temps de calcul
⇒ augmenter le nombre d'échanges possibles par seconde

Cible

- CPU Intel avec instructions `SSE4` et `AVX2` si disponibles

Méthodologie

1. « *Profiler* » le code existant pour trouver les « points chauds »
2. Tenter d'optimiser ces points chauds
3. Recommencer

État des lieux

Hot spots

Utilisation de `valgrind` avec le module `callgrind`.

Répartition du temps CPU pour le **chiffrement** :

- **64%** dans la transformation **NTT**
- **20%** dans les **opérations arithmétiques** sur les polynômes utilisés
- **16%** dans le **calcul d'aléa**

Pour le **déchiffrement** $\rightarrow$ **80%** du temps CPU dans la transformation **inverse NTT** (composée par 95% de transformation NTT)

État des lieux

Hot spots

Utilisation de `valgrind` avec le module `callgrind`.

Répartition du temps CPU pour le **chiffrement** :

- **64%** dans la transformation **NTT**
- **20%** dans les **opérations arithmétiques** sur les polynômes utilisés
- **16%** dans le **calcul d'aléa**

Pour le **déchiffrement** $\rightarrow$ **80%** du temps CPU dans la transformation **inverse NTT** (composée par 95% de transformation NTT)

Plan d'attaque

- **NTT** majoritaire dans l'utilisation du temps CPU $\rightarrow$ **première fonction** à optimiser
- Les calculs sur les polynômes viendront ensuite
- Le calcul d'aléa étant déjà considéré comme optimisé, il est mis de côté pour l'instant

Premature optimisation is premature

ou comment avoir de fausses bonnes idées

Déroulement de boucle

Transformer

```
for (int i = 0; i < n; i++) { work(i); }
```

en

```
int const bound = n / 2 * 2;
for (int i = 0; i < bound; i += 2) { work(i); work(i+1); }
for (int i = bound; i < n; ++i) work(i);
```

Le compilateur le fait déjà s'il juge que cela sera plus efficace.

Premature optimisation is premature

ou comment avoir de fausses bonnes idées

Déroulement de boucle

Constructions manuelles de boucles et d'indices

```
for (; N > 4; N /= 2, M *= 2) {
  uint64_t* x0 = x;
  uint64_t* x1 = x + N/2;
  for (size_t r = 0; r < M; r++, x0 += N, x1 += N) {
    ptrdiff_t i = N/2 - 2;
    do { work(i); work(i+1); i -= 2; } while (i >= 0);
  }
}
```

- Garder des **constructions à base de** `for` (et non *do* / *while*)
  → les compilos savent mieux analyser les boucles explicites

Premature optimisation is premature

ou comment avoir de fausses bonnes idées

Déroulement de boucle

Constructions manuelles de boucles et d'indices

- Garder des **constructions à base de** `for` (et non *do* / *while*)
  → les compilos savent mieux analyser les boucles explicites
- Garder les **indices de boucles** et les **accès mémoire explicites** → aide les analyses du compilateur. La passe `-fmove-loop-invariants` de `GCC` enlèvera les invariants de boucle.

```
size_t J = log2(N)-2;
for (size_t w = 0; w < J; w++) {
  const size_t M = 1 << w;
  const size_t N = poly::degree >> w;
  for (size_t r = 0; r < M; r++)
for (size_t i = 0; i < N/2; i++)
  work(N*r + N/2 + i);
}
```

Réécriture du code NTT

Code original

- Code original de David Harvey [a]
- Optimisé manuellement pour des entiers 64 bits : boucles déroulées, indices pré-calculés, etc...
- $\rightarrow$ non optimal pour des entiers 16 ou 32 bits

[a] http://web.maths.unsw.edu.au/ davidharvey/papers/fastntt/ntt.c

Réécriture du code NTT

Code original

- Code original de David Harvey [a]
- Optimisé manuellement pour des entiers 64 bits : boucles déroulées, indices pré-calculés, etc...
- $\rightarrow$ non optimal pour des entiers 16 ou 32 bits

[a] http://web.maths.unsw.edu.au/ davidharvey/papers/fastntt/ntt.c

Réécriture

- Réécriture de la boucle non déroulée sans pré-calcul d'indices
- Code C++ template [a] supportant des entiers de 16 à 64 bits

[a] ceci n'est pas un gros mot !

Compilation avec `gcc -ftree-vectorizer-verbose=1`

`-ftree-vectorizer-verbose=1` → `GCC` affiche les boucles qu'il a vectorisées.

Compilation avec `gcc -ftree-vectorizer-verbose=1`

`-ftree-vectorizer-verbose=1` → `GCC` affiche les boucles qu'il a vectorisées.

Avant

Aucune boucle n'est vectorisée au sein de la fonction NTT

Compilation avec `gcc -ftree-vectorizer-verbose=1`

`-ftree-vectorizer-verbose=1` → `GCC` affiche les boucles qu'il a vectorisées.

Avant

Aucune boucle n'est vectorisée au sein de la fonction NTT

Après

```
./nfl/algos.hpp:63: note: LOOP VECTORIZED.
```

→ `nfl/algos.hpp:63` correspond à la boucle réécrite précédemment

Performances obtenues

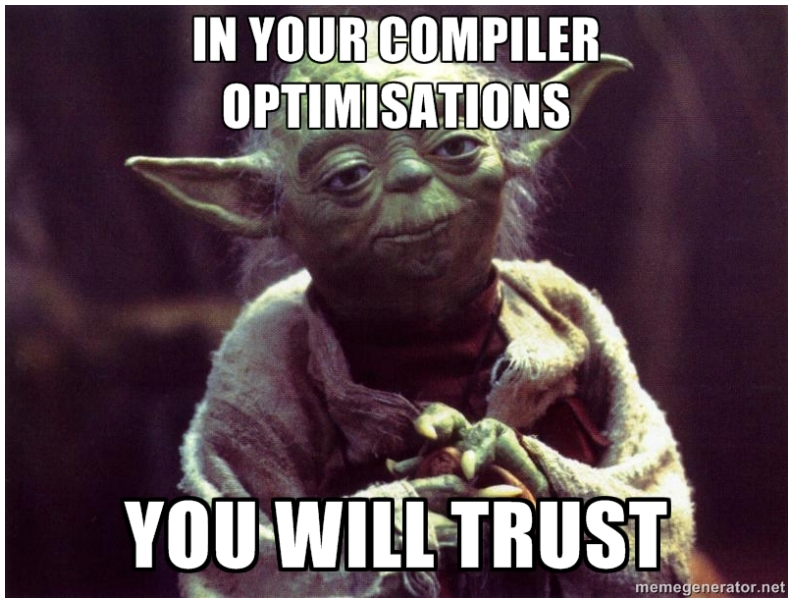
NTT (référence)			
Type	Sans vectorisation	SSE4	AVX2
64 bits	16.7μs	14.5μs	13.5μs
32 bits	6.9μs	4.3μs	4.3μs
16 bits	7.4μs	5.0μs	5.0μs

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

Ce qu'on apprend à l'école…

Our compilers *usually* are smarter than us

- **Ne pas empêcher les analyses des compilateurs modernes** avec des déroulements de boucles ou calculs d'indice de boucles inutiles
- Comprendre pourquoi certaines optimisations n'ont pas pû être faites (sortie de `-ftree-vectorizer-verbose` par exemple)
- Seulement ensuite tenter d'optimiser « à la main »



Going deeper...

On a aidé le compilateur pour qu'il fasse au mieux son travail, regardons ce que cela donne...

« *multiply high* »

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

- opération présente dans la boucle NTT
- vectorisée par GCC
- comment ?

« *multiply high* » sur 16 bits

Code de référence

```
void mulhigh_16(size_t n, uint16_t* res, uint16_t* a, uint16_t* b) {
  for (size_t i = 0; i < n; i++)
    res[i] = ((uint32_t)a*b)>>16;
}
```

Compilation

```
$ gcc -O3 -march=native -mtune=native -ftree-vectorizer-verbose=1
mulhi.c: note: LOOP VECTORIZED.
```

« *multiply high* » sur 16 bits

Code de référence

```
void mulhigh_16(size_t n, uint16_t* res, uint16_t* a, uint16_t* b) {
  for (size_t i = 0; i < n; i++)
    res[i] = ((uint32_t)a*b)>>16;
}
```

Sortie assembleur (boucle)

```
; Charge 8 entiers de 2 octets des vecteurs a et b
vmovdqu xmm0, xmmword ptr [a+r8]
vmovdqu xmm2, xmmword ptr [b+r8]

; Calcul par bloc de 8 le multiply high
vpmullw xmm1, xmm2, xmm0
vpmulhuw xmm0, xmm2, xmm0
vpunpcklwd xmm2, xmm1, xmm0
vpunpckhwd xmm0, xmm1, xmm0
vpsrld  xmm2, xmm2, 10h
vpsrld  xmm0, xmm0, 10h
vpshufb xmm2, xmm2, xmm4
vpshufb xmm0, xmm0, xmm3
vpor    xmm0, xmm2, xmm0

; Sauvegarde le resultat dans res
vmovdqu xmmword ptr [res+r8], xmm0
```

« *multiply high* » sur 16 bits

Possibilité de faire plus simple (et surtout rapide) ?

D'après le guide des *intrinsiques* Intel [a], en `SSE` :

```
__m128i _mm_mulhi_epi16 (__m128i a, __m128i b) [pmulhw]

Multiply the packed 16-bit integers in a and b, producing intermediate 32-bit
integers, and store the high 16 bits of the intermediate integers in dst.
```

[a] https://software.intel.com/sites/landingpage/IntrinsicsGuide/

« *multiply high* » sur 16 bits

Possibilité de faire plus simple (et surtout rapide) ?

D'après le guide des *intrinsiques* Intel [a], en `SSE` :

> **`__m128i _mm_mulhi_epi16 (__m128i a, __m128i b) [pmulhw]`**
>
> **Multiply the packed 16-bit integers in a and b, producing intermediate 32-bit integers, and store the high 16 bits of the intermediate integers in dst.**

→ remplacement des 9 instructions précédentes par celle-ci
(même variante en AVX2 sur des registres de 256 bits)

[a] `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`

« *multiply high* » sur 32 bits

Vectorisation manuelle

- Optimisation originale du compilateur correcte
- Gain néanmoins de quelques instructions de permutation
- $\rightarrow$ calcul en 6 instructions au lieu de 8
- Détaillée dans les actes

Multiplication modulaire

Opérations

```
uint16_t operator()(uint16_t x, uint16_t y, uint16_t yprime, uint16_t p) {
  uint32_t q = mulhi16(x, yprime);
  uint32_t res = x * y - q * p;
  return res - ((res>=p) ? p : 0);
}
```

Multiplication modulaire

Opérations

```
uint16_t operator()(uint16_t x, uint16_t y, uint16_t yprime, uint16_t p) {
  uint32_t q = mulhi16(x, yprime);
  uint32_t res = x * y - q * p;
  return res - ((res>=p) ? p : 0);
}
```

État actuel

- GCC ne vectorise pas cette fonction
- Principalement dû à la comparaison
- $\rightarrow$ vectorisation manuelle (détails dans les actes). Réutilisation de la fonction *multiply high*.

Vectorisation manuelle : gain de performances

Boucle NTT

Évaluation de performances

- Difficile d'évaluer les performances individuelles de chaque boucle car *memory-bound*
- Évaluation de la boucle NTT complète

Vectorisation manuelle : gain de performances

Boucle NTT

Évaluation de performances

- Difficile d'évaluer les performances individuelles de chaque boucle car *memory-bound*
- Évaluation de la boucle NTT complète

NTT (vectorisée par GCC)		
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**)

Table : $G = \frac{\text{temps\_gcc}}{\text{temps\_manuel}}$

Vectorisation manuelle : gain de performances

Chiffrement/déchiffrement asymétrique

$$G = \frac{\text{temps\_gcc}}{\text{temps\_manuel}}$$

Chiffrement (vectorisé par GCC)		
Type	SSE	AVX2
32 bits	26.1μs	17.3μs
16 bits	22.1μs	21.3μs
Chiffrement (vectorisation manuelle)		
Type	SSE	AVX2
32 bits	22.8μs (**G = 1.1**)	14.0μs (**G = 1.2**)
16 bits	15.4μs (**G = 1.4**)	13.6μs (**G = 1.6**)

Vectorisation manuelle : gain de performances

Chiffrement/déchiffrement asymétrique

$$G = \frac{\text{temps\_gcc}}{\text{temps\_manuel}}$$

Chiffrement (vectorisé par GCC)		
Type	SSE	AVX2
32 bits	26.1μs	17.3μs
16 bits	22.1μs	21.3μs
Chiffrement (vectorisation manuelle)		
Type	SSE	AVX2
32 bits	22.8μs (**G = 1.1**)	14.0μs (**G = 1.2**)
16 bits	15.4μs (**G = 1.4**)	13.6μs (**G = 1.6**)

Déchiffrement (vectorisé par GCC)		
Type	SSE	AVX2
32 bits	5.2μs	4.7μs
16 bits	6.2μs	5.9μs
Déchiffrement (vectorisation manuelle)		
Type	SSE	AVX2
32 bits	3.6μs (**G = 1.4**)	3.0μs (**G = 1.6**)
16 bits	2.7μs (**G = 2.3**)	2.2μs (**G = 2.7**)

État des lieux final

Après toutes les optimisations précédentes :

État des lieux du chiffrement/déchiffrement asymétrique

Répartition du temps CPU pour le chiffrement :

- **41%** dans le **calcul d'aléa** (contre 16% au départ)
- **35%** dans la transformation **NTT** (contre 64% au départ)
- **24%** dans les **opérations arithmétiques** sur les polynômes utilisés (contre 20% au départ)

État des lieux final

Après toutes les optimisations précédentes :

État des lieux du chiffrement/déchiffrement asymétrique

Répartition du temps CPU pour le chiffrement :

- **41%** dans le **calcul d'aléa** (contre 16% au départ)
- **35%** dans la transformation **NTT** (contre 64% au départ)
- **24%** dans les **opérations arithmétiques** sur les polynômes utilisés (contre 20% au départ)

Pour continuer

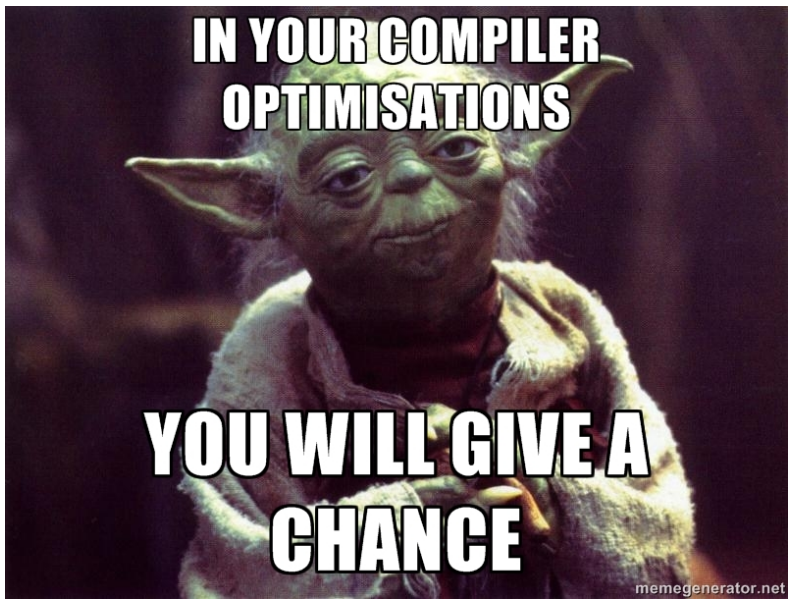
$\rightarrow$ La fonction de calcul d'aléa devient prépondérante. Une piste possible est d'étudier l'implémentation actuelle (écrite grâce à `qhasm` [a]).

[a] http://cr.yp.to/qhasm.html, http://cr.yp.to/qhasm/amd64/20061217-salsa20-xmm.s

Conclusion sur l'optimisation

Optimisations bas niveaux

- Faire en sorte que le compilateur soit « à l'aise »
- Vérifier les sorties produites, voir s'il n'est pas possible de faire mieux
- Toujours mesurer



Plan

Expression template

Le problème

Chaque opération arithmétique est effectuée sur des polynômes → opérations sur des vecteurs

Exemple de l'addition

```
poly p = a + b;
// est equivalent à
for (...) { p[i] = a[i]+b[i]; }
```

On aimerait pouvoir écrire...

```
poly p = a + b*c;
```

et que cela donne

```
for (...) { p[i] = a[i]+b[i]*c[i]; }
```

Mais, en C++, cela va donner...

```
for (...) { tmp[i] = b[i]*c[i]; }
for (...) { p[i] = a[i]+tmp[i]; }
```

Expression template

Le problème

Chaque opération arithmétique est effectuée sur des polynômes → opérations sur des vecteurs

Mais, en C++, cela va donner. . .

```
for (...) { tmp[i] = b[i]*c[i]; }
for (...) { p[i] = a[i]+tmp[i]; }
```

Loop fusion

Une phase de *loop fusion* pourrait combiner ces boucles

- malheureusement pas appliquée par `GCC` ou `clang`

Expression template (so 1990')

L'idée est de créer un arbre de type représentant l'expression et de l'évaluer de manière paresseuse

```
int expr(i) { return a[i]+b[i]*c[i]; } // cree à la compilation
for (...) { p[i] = expr(i); }
```

⇒ très bénéfique pour la vectorisation !

Plan

Conclusion

Diffusion

- Le gros de la librairie (90% du code) sur GitHub prochainement (+exemples)
- SIMD, expression-templates avancées propriétaires (Quarkslab)
- Exemple de performances KX : 1 Mops vs 4 Mops
- Disponible dès maintenant à la demande

TODOs

Standardisation nationale/européenne
Intégration dans TLS
Audit de la librairie (aléas, contrôle des entrées, side channel)

Questions

Merci, des questions ?