AES-GCM plus rapide et résistant aux attaques temporelles

Emilia Käsper, Peter Schwabe

Eindhoven University of Technology

13/11/2009

Séminaire de cryptographie, Université de Rennes 1

Le Standard de chiffrement avancé (AES)

- Proposé par Rijmen et Daemen en 1998
- Taille des blocs : 128 bits (16 octets)
- Longueur de la clé : 128, 192 ou 256 bits
- Nombre de tours : 10, 12 ou 14

Le Standard de chiffrement avancé (AES)

- Proposé par Rijmen et Daemen en 1998
- Taille des blocs : 128 bits (16 octets)
- Longueur de la clé : 128, 192 ou 256 bits
- Nombre de tours : 10, 12 ou 14

Le Standard de chiffrement avancé (AES)

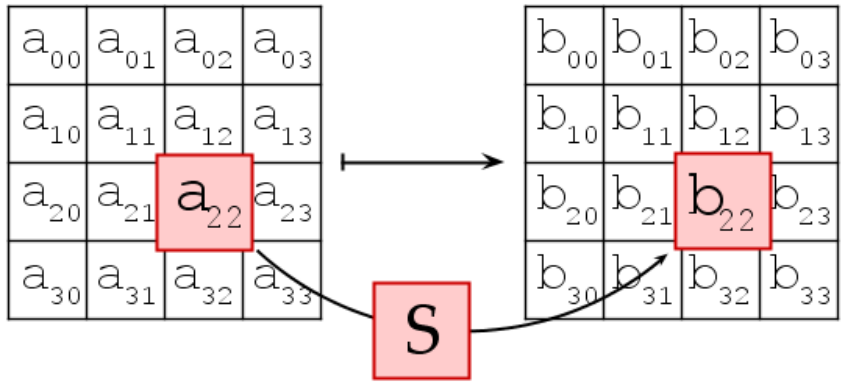
- Proposé par Rijmen et Daemen en 1998
- Taille des blocs : 128 bits (16 octets)
- Longueur de la clé : 128, 192 ou 256 bits
- Nombre de tours : 10, 12 ou 14
- 4 opérations par tour : `SubBytes`, `ShiftRows`, `MixColumns` et `AddRoundKey`
- Dernier tour consiste en `SubBytes`, `ShiftRows` et `AddRoundKey`

Le Standard de chiffrement avancé (AES)

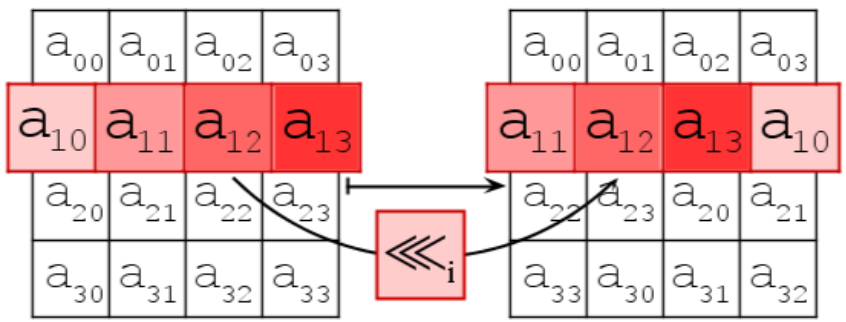
- Proposé par Rijmen et Daemen en 1998
- Taille des blocs : 128 bits (16 octets)
- Longueur de la clé : 128, 192 ou 256 bits
- Nombre de tours : 10, 12 ou 14
- 4 opérations par tour : `SubBytes`, `ShiftRows`, `MixColumns` et `AddRoundKey`
- Dernier tour consiste en `SubBytes`, `ShiftRows` et `AddRoundKey`
- Implantation d'AES dans OpenSSL : $\approx$ 18 cycles/octet sur un Core 2

Les opérations d'AES I

SubBytes

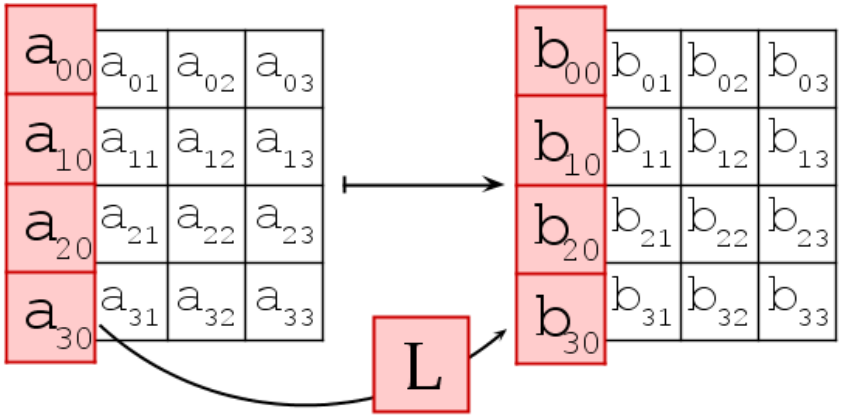


ShiftRows

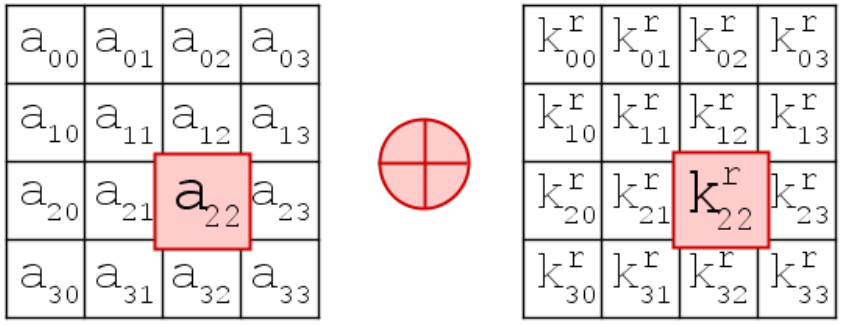


Les opérations d'AES II

MixColumns



AddRoundKey



Implantation « classique », le premier tour

Entrée et sortie

Entrée : 16 octets représentés par 4 entiers de 32 bits : y0, y1, y2, y3
Sortie : 16 octets représentés par 4 entiers de 32 bits : z0, z1, z2, z3

Tables et clés

```
uint32 rk[44], T0[256], T1[256], T2[256], T3[256];
```

Le premier tour d'AES

```
z0 = T0[ y0 >> 24        ] ^ T1[(y1 >> 16) & 0xff] \
^ T2[(y2 >>  8) & 0xff] ^ T3[ y3        & 0xff] ^ rk[4];
z1 = T0[ y1 >> 24        ] ^ T1[(y2 >> 16) & 0xff] \
^ T2[(y3 >>  8) & 0xff] ^ T3[ y0        & 0xff] ^ rk[5];
z2 = T0[ y2 >> 24        ] ^ T1[(y3 >> 16) & 0xff] \
^ T2[(y0 >>  8) & 0xff] ^ T3[ y1        & 0xff] ^ rk[6];
z3 = T0[ y3 >> 24        ] ^ T1[(y0 >> 16) & 0xff] \
^ T2[(y1 >>  8) & 0xff] ^ T3[ y2        & 0xff] ^ rk[7];
```

Résultats avec cette technique

Bernstein–S., Indocrypt 2008 :

- UltraSparc III : 12,06 cycles/octet
- PowerPC G4 : 14,57 cycles/octet
- Pentium 4 : 14,15 cycles/octet
- Core 2 (65 nm) : 10,57 cycles/octet
- Athlon 64 : 10,43 cycles/octet

(en mode d'opération CTR)

Résultats avec cette technique

Bernstein–S., Indocrypt 2008 :

- UltraSparc III : 12,06 cycles/octet
- PowerPC G4 : 14,57 cycles/octet
- Pentium 4 : 14,15 cycles/octet
- Core 2 (65 nm) : 10,57 cycles/octet
- Athlon 64 : 10,43 cycles/octet

(en mode d'opération CTR)

C'est déjà bien, non ?

- Par exemple le Core 2 peut exécuter une instruction de chargement par cycle
- Chaque tour comporte 16 instructions de chargement
- Barrière de 160 cycles/bloc ou 10 cycles/octet

Résultats avec cette technique

Bernstein–S., Indocrypt 2008 :

- UltraSparc III : 12,06 cycles/octet
- PowerPC G4 : 14,57 cycles/octet
- Pentium 4 : 14,15 cycles/octet
- Core 2 (65 nm) : 10,57 cycles/octet
- Athlon 64 : 10,43 cycles/octet

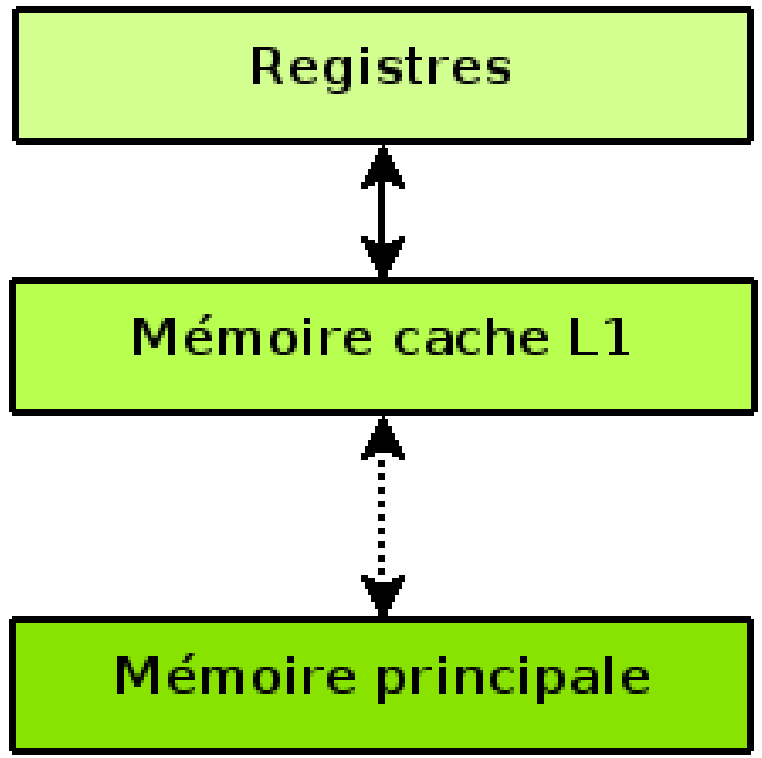
(en mode d'opération CTR)

C'est déjà bien, non ?

- Par exemple le Core 2 peut exécuter une instruction de chargement par cycle
- Chaque tour comporte 16 instructions de chargement
- Barrière de 160 cycles/bloc ou 10 cycles/octet
- Plus important : Les implantations utilisant des tables sont vulnérables aux attaques dites de « cache timing »

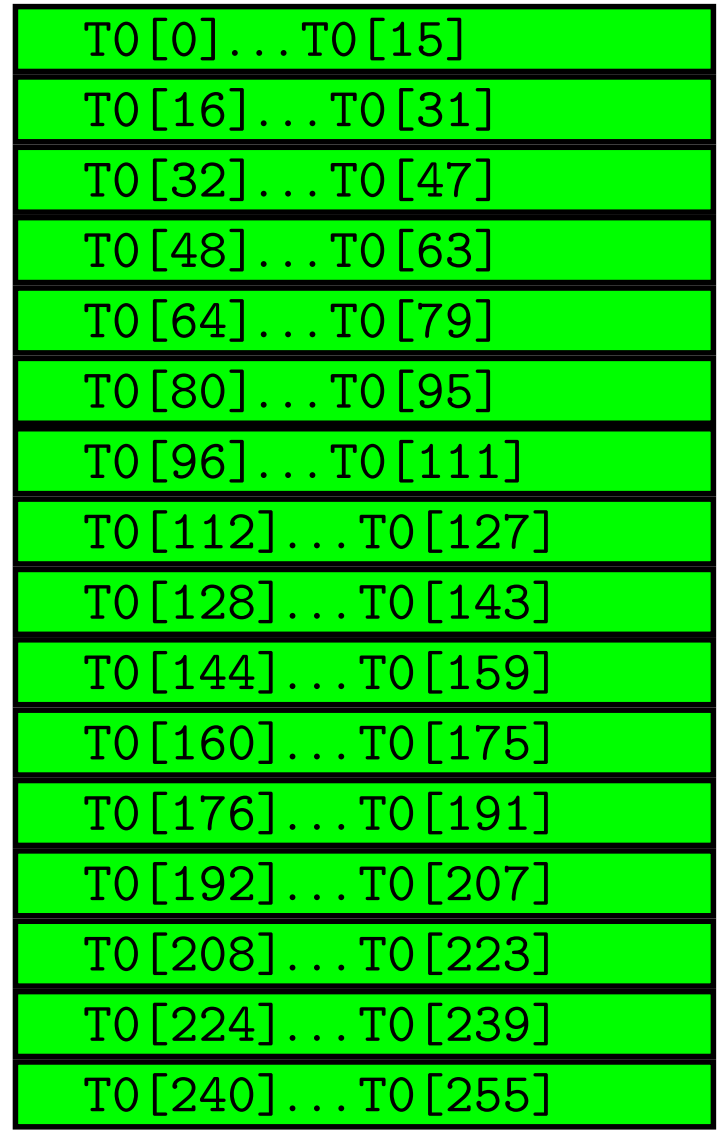
La memoire d'un ordinateur moderne

- Quand le processeur charge des données, elles sont aussi enregistrées en mémoire cache
- Elles restent en mémoire cache jusqu'à ce que d'autres données les en déplacent
- Il est beaucoup plus rapide de charger des données depuis la mémoire cache que depuis la mémoire principale
- Les tables d'AES tiennent (normalement) en mémoire cache du premier niveau (L1).



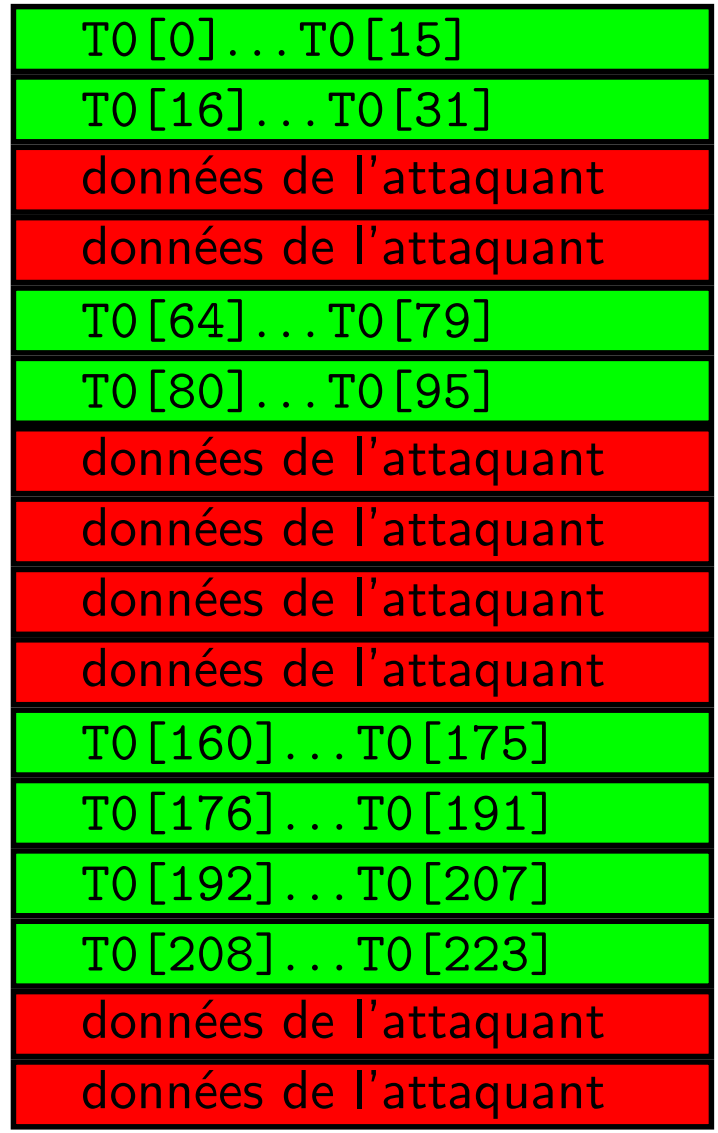
Idée des attaques de type « cache-timing »

- AES et le logiciel de l'attaquant s'exécutent simultanément sur un processeur
- Les tables sont dans la mémoire cache



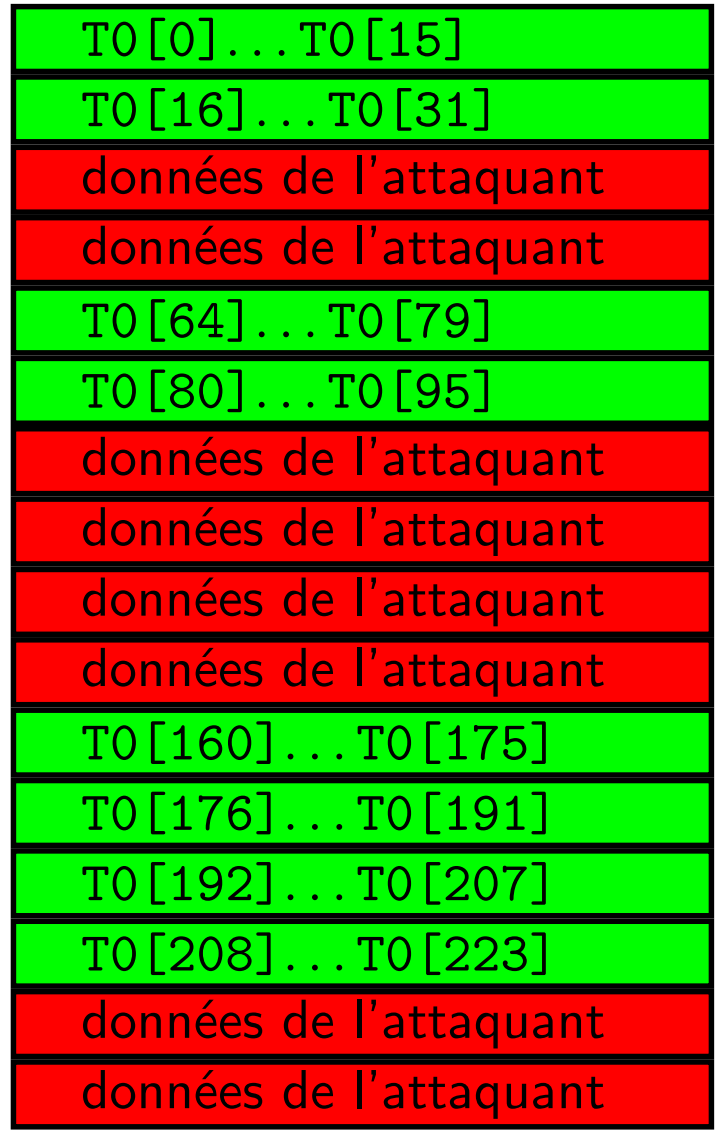
Idée des attaques de type « cache-timing »

- AES et le logiciel de l'attaquant s'exécutent simultanément sur un processeur
- Les tables sont dans la mémoire cache
- Le logiciel de l'attaquant déplace quelques lignes de la mémoire cache



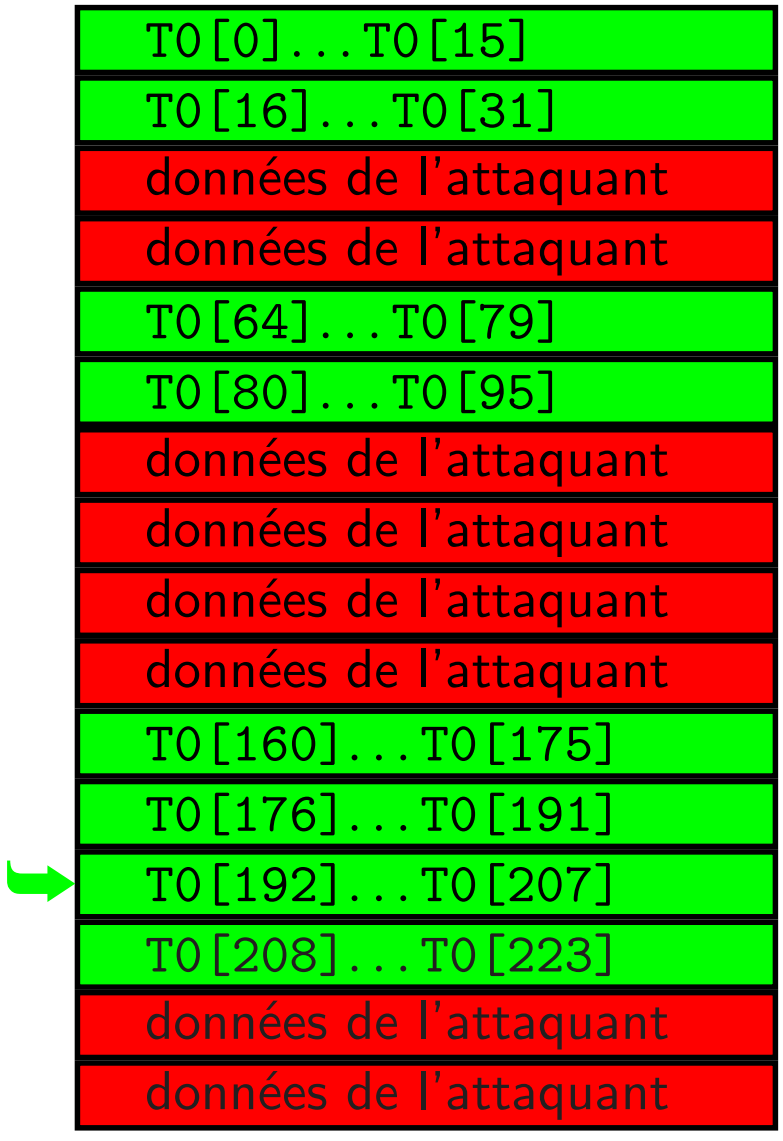
Idée des attaques de type « cache-timing »

- AES et le logiciel de l'attaquant s'exécutent simultanément sur un processeur
- Les tables sont dans la mémoire cache
- Le logiciel de l'attaquant déplace quelques lignes de la mémoire cache
- AES charge des données



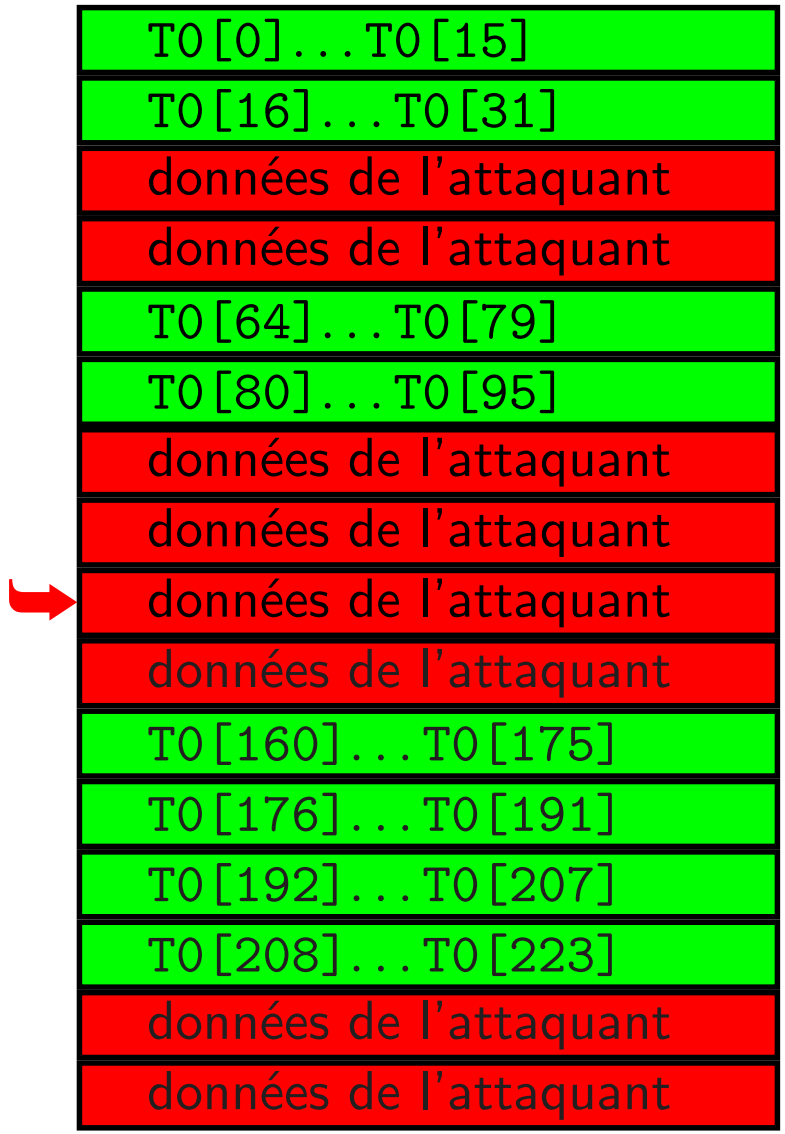
Idée des attaques de type « cache-timing »

- AES et le logiciel de l'attaquant s'exécutent simultanément sur un processeur
- Les tables sont dans la mémoire cache
- Le logiciel de l'attaquant déplace quelques lignes de la mémoire cache
- AES charge des données
  - Succès de cache : rapide



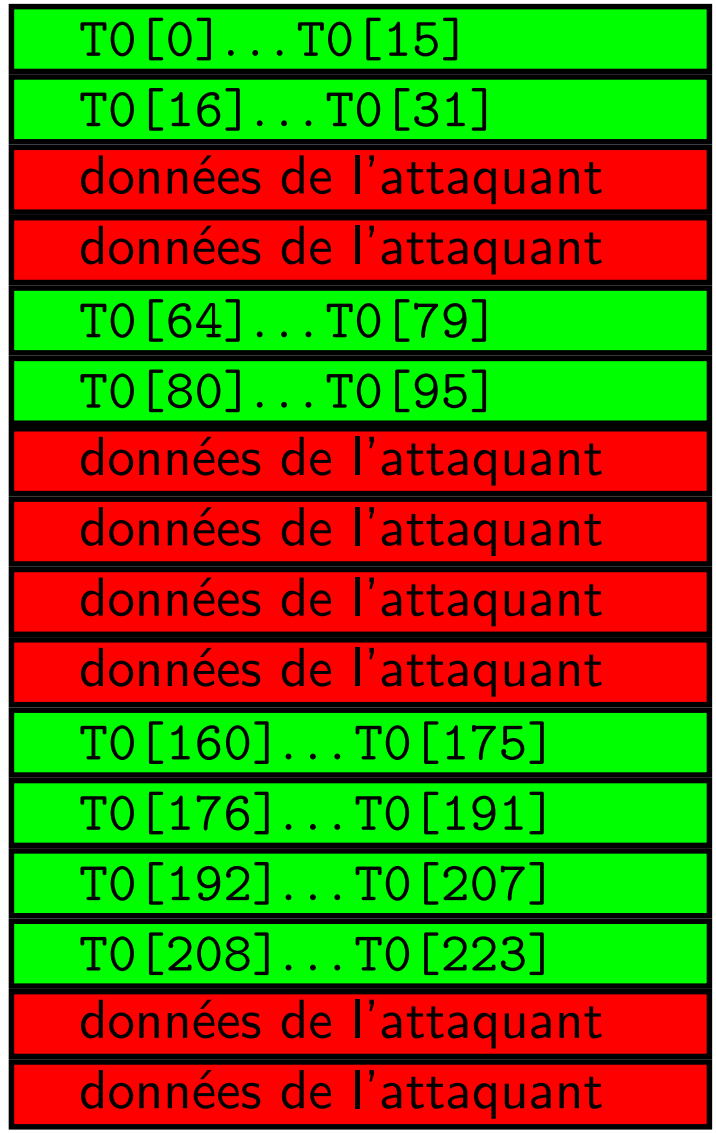
Idée des attaques de type « cache-timing »

- AES et le logiciel de l'attaquant s'exécutent simultanément sur un processeur
- Les tables sont dans la mémoire cache
- Le logiciel de l'attaquant déplace quelques lignes de la mémoire cache
- AES charge des données
  - Succès de cache : rapide
  - Défaut de cache : lent



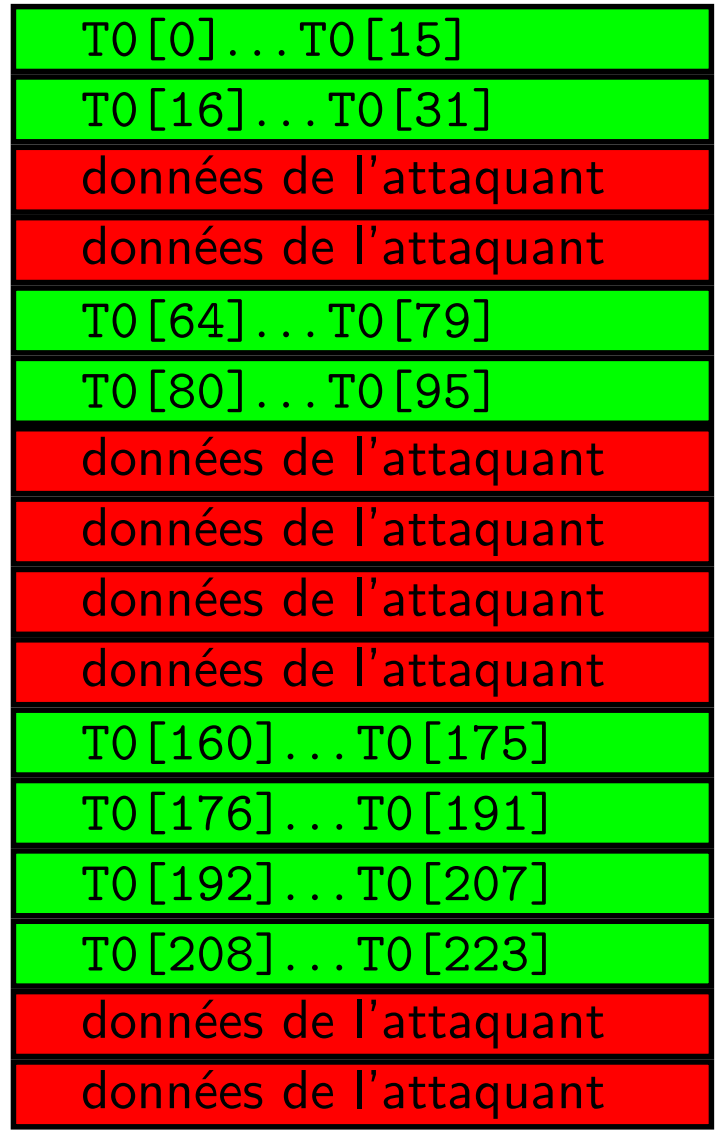
Idée des attaques de type « cache-timing »

- AES et le logiciel de l'attaquant s'exécutent simultanément sur un processeur
- Les tables sont dans la mémoire cache
- Le logiciel de l'attaquant déplace quelques lignes de la mémoire cache
- AES charge des données
  - Succès de cache : rapide
  - Défaut de cache : lent
- L'attaquant mesure le temps et en déduit des information secrètes du chiffrement



Idée des attaques de type « cache-timing »

- AES et le logiciel de l'attaquant s'exécutent simultanément sur un processeur
- Les tables sont dans la mémoire cache
- Le logiciel de l'attaquant déplace quelques lignes de la mémoire cache
- AES charge des données
  - Succès de cache : rapide
  - Défaut de cache : lent
- L'attaquant mesure le temps et en déduit des information secrètes du chiffrement
- Chaque implantation qui charge des données des positions secrètes est vulnerable !



Contre-mesures pour AES

- Ajouter du « souffle »contenant des instructions s'exécutant en temps variable
  - Dégrade la performance
  - L'attaque fonctionne toujours, mais elle requiert plus de mesures
- Remplacer les chargements par des calculs
  - Exploite la structure algébrique des tables
  - Faisable mais pas efficace
- Utiliser des instruction de rotation et de permutation
  - Approche présentée par Mike Hamburg à CHES 2009
  - Haute vitesse sur quelques processeurs (ppc64)
  - Instructions non disponibles sur tous les processeur
- Bitslicing

« Bitslicing »

Exemple : Multiplication de polynômes de degré 127 sur $\mathbb{F}_2$

- Implantation normale : f dans un registre de 128 bits, g dans un registre de 128 bits
- On a besoin d'une instruction de multiplication sans retenue
- Cette instruction n'existe pas sur la plupart des processeurs $\Rightarrow$ on utilise des tables
- 270 cycles par multiplication sur le Xeon 5460 [Hankerson, Karabina, Menezes, 2008]

« Bitslicing »

Exemple : Multiplication de polynômes de degré 127 sur $\mathbb{F}_2$

- Implantation normale : f dans un registre de 128 bits, g dans un registre de 128 bits
- On a besoin d'une instruction de multiplication sans retenue
- Cette instruction n'existe pas sur la plupart des processeurs $\Rightarrow$ on utilise des tables
- 270 cycles par multiplication sur le Xeon 5460 [Hankerson, Karabina, Menezes, 2008]
- Implantation bitsliced : Un bit (un coefficient) par registre
- On simule une implantation hardware avec des instruction ET et XOR.
- Faisable en 11486 opérations [Bernstein, 2009]

« Bitslicing »

Exemple : Multiplication de polynômes de degré 127 sur $\mathbb{F}_2$

- Implantation normale : f dans un registre de 128 bits, g dans un registre de 128 bits
- On a besoin d'une instruction de multiplication sans retenue
- Cette instruction n'existe pas sur la plupart des processeurs $\Rightarrow$ on utilise des tables
- 270 cycles par multiplication sur le Xeon 5460 [Hankerson, Karabina, Menezes, 2008]
- Implantation bitsliced : Un bit (un coefficient) par registre
- On simule une implantation hardware avec des instruction ET et XOR.
- Faisable en 11486 opérations [Bernstein, 2009]
- Ce n'est pas efficace pour une multiplication
- Ça devient efficace pour 128 multiplications parallèles (~ 90 opérations/multiplication)

Et pour AES ?

- L'idée est la même : traiter beaucoup de blocs en parallèle et simuler une implantation hardware
- La performance dépend de l'architecture (du processeur)

Et pour AES ?

- L'idée est la même : traiter beaucoup de blocs en parallèle et simuler une implantation hardware
- La performance dépend de l'architecture (du processeur)

Les processeurs 64 bits d'Intel (`amd64`)

- Core 2 (65 nm), Core 2 (45 nm) et Core i7
- 16 registres XMM de 128 bits
- Instructions SSE (suivies par SSE2, SSE3, SSSE3, SSE4)
- 3 unités arithmétiques ($\leq$ 3 instructions logiques par cycle)
- Instructions avec 2 opérandes :

$$\texttt{XOR a b} \quad \hat{=} \quad b = a \oplus b$$

- Instruction de permutation d'octets (`pshufb`, SSSE3) :
  - Core 2 (65 nm) : micro opération
  - Core 2 (45 nm) : unité de permutation
  - Core i7 : 2 unités de permutation

Bitslicing pour AES sur amd64

- Matsui, Nakajima (CHES 2007) : 9,2 cycles/octet sur un Intel Core 2 (65 nm)
  - Utilise les registres 128 bits
  - Doit traiter 128 blocs (2 ko) en parallèle
  - Nécessite une transformation des entrées et de la sortie
- Könighofer (CT-RSA 2008) : 19,8 cycles/octet sur un AMD Opteron 146
  - Utilise les registres 64 bits
  - Doit traiter 4 blocs en parallèle
  - Exploite le fait que les opérations sur les 16 octets d'un bloc sont identiques

Notre implantation d'AES

- Nous utilisons les registres 128 bits
- Nous traitons 8 blocs en parallèle

row 0																.....	row 3											
column 0				column 1				column2				column 3				.....	column 0				.....	column 3					
block 0	block 1	...	block 7	block 0	block 1	...	block 7	block 0	block 1	...	block 7	block 0	block 1	...	block 7	.....	block 0	block 1	...	block 7	.....	block 0	block 1	...	block 7		

- Les 8×16 octets des 8 blocs sont dans 8 registres XMM
- L'opération `SubBytes` est exécutée sur 128 octets parallèlement
- Les bits de la même position dans les 8 blocs sont groupés dans le même octet
- Toutes les instructions sont au niveau d'octets

L'implantation de SubBytes

- Part de l'implantation la plus compacte en hardware : 117 portes [Canright 2005, Boyar/Peralta 2009]
- Utilise des instructions équivalentes sur des registres de 128 bits
- **Problème nº 1 :** instructions avec 2 opérandes, la sortie écrase une entrée
- Il nous faut plusieurs instructions additionnelles pour copier des données dans d'autres registres
- **Problème nº 2 :** pas assez de registres pour résultats intermédiaires
- Nous calculons quelques résultats plusieurs fois (alternative : les mettre sur la pile)
- Au total : 162 instructions (15% de moins que les précédents résultats)

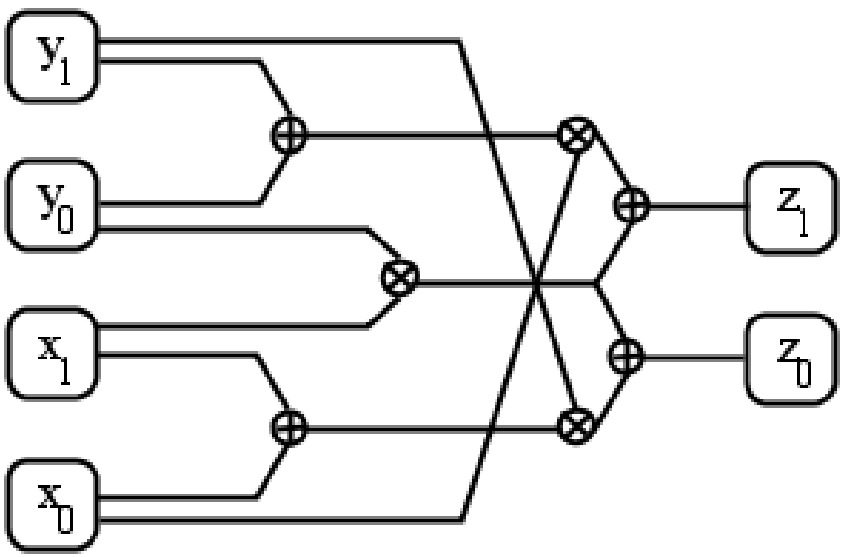
	XOR	AND/OR	MOV	TOTAL
Hardware	82	35	–	117
Software	93	35	35	163

Comparaison : hardware/software

Exemple : multiplication dans $\mathbb{F}_{2^2}$

$(x_1, x_0) \otimes (y_1, y_0) \rightarrow (z_1, z_0)$

$$z_1 = (y_0 + y_1)x_0 + x_1 y_0$$
$$z_0 = (x_0 + x_1)y_1 + x_1 y_0$$




```
movdqa   \x0, \z0
movdqa   \x1, \z1
movdqa   \y0, \t0
pxor     \y1, \t0
pand     \z0, \t0
pxor     \z1, \z0
pand     \y1, \z0
pand     \y0, \z1
pxor     \z1, \z0
pxor     \t0, \z1
```

Implantation des opérations linéaires

- Chaque octet d'un des registres XMM correspond à une position d'un bit dans chaque bloc d'AES
- `ShiftRows` est une permutation des octets
- Nous utilisons l'instruction `pshufb` (SSSE3)
- `MixColumns` est implanté avec `pshufb` et quelques XOR (43 instructions au total)
- `AddRoundKey` est implanté avec 8 instructions XOR

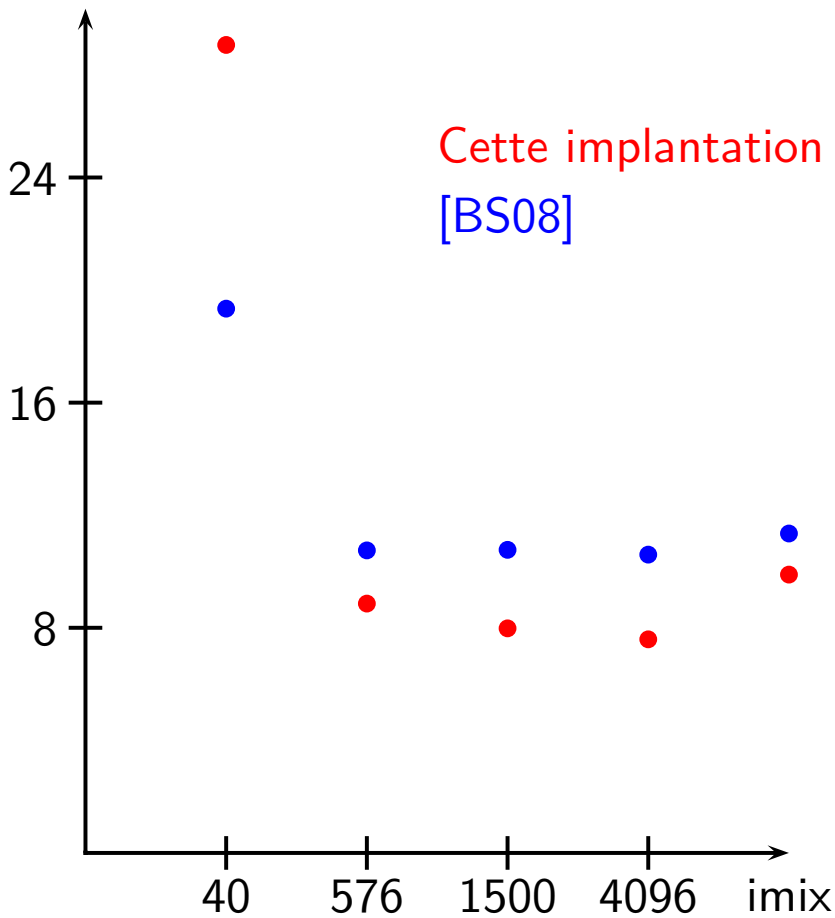
Avertissements

- Les clés de tour sont plus grandes : 8×128 bits par tour, l'expansion de la clé est plus lente
- L'instruction `pshufb` n'est pas disponible sur les processeurs AMD

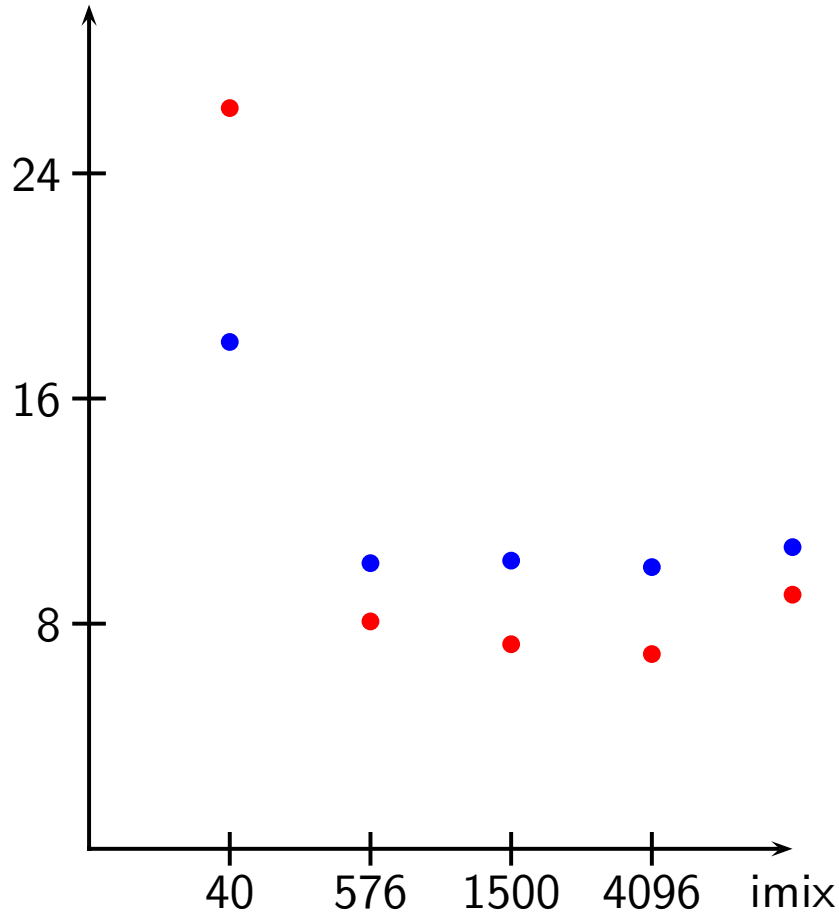
Résultats I (AES-CTR)

cycles/octet pour des messages de differentes longueurs

Core 2 Q9550



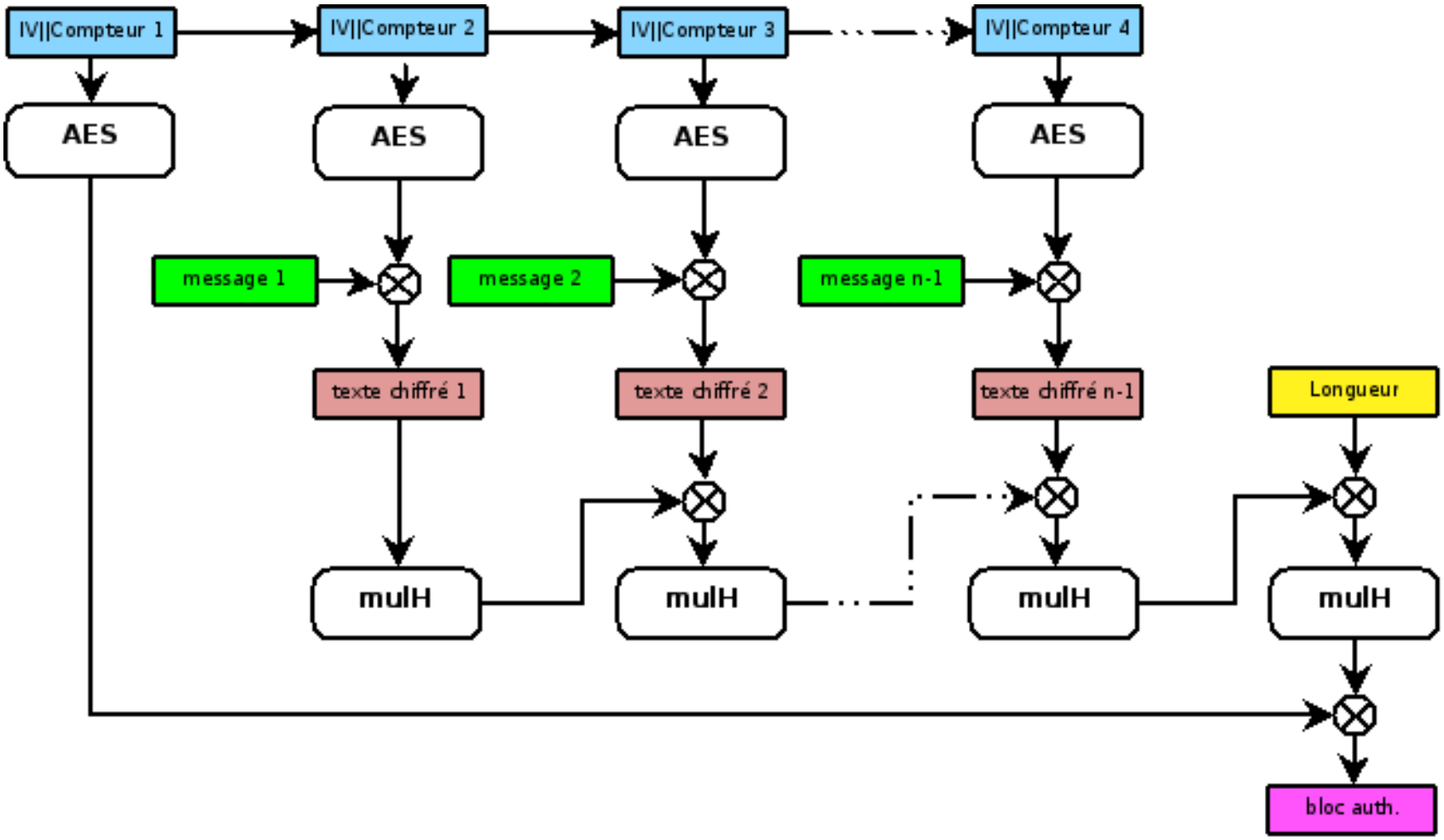
Core i7 920



Le mode d'opération GCM

- Mode d'opération de chiffrement et d'authentification
- Chiffrement : utilise AES-CTR
- Authentification : utilise des multiplications dans $\mathbb{F}_{2^{128}} = \mathbb{F}_2[x]/(x^{128} + x^7 + x^2 + x + 1)$ avec H
- H est obtenu comme $\mathrm{AES}_K(0)$
- Longueur du bloc authentificateur : 128 bits

Le mode d'opération GCM



Multiplications dans $\mathbb{F}_{2^{128}}$

Est-ce-qu'on peut utiliser bitslicing ?

- Le bitslicing est très efficace pour les multiplications dans $\mathbb{F}_{2^{128}}$
- Il faut en faire 128 en parallèle
- On peut transformer $(\ldots((c_1 H) + c_2)H)\cdots + c_{128})H$ en $c_1 H^{128} + c_2 H^{127} + \cdots + c_{128} H$
- Problèmes :
  - Requiert une transformation des données
  - Nécessite au moins 128 blocs
  - Efficace seulement pour une grande quantité de données

Multiplications dans $\mathbb{F}_{2^{128}}$

Est-ce-qu'on peut utiliser bitslicing ?

- Le bitslicing est très efficace pour les multiplications dans $\mathbb{F}_{2^{128}}$
- Il faut en faire 128 en parallèle
- On peut transformer $(\ldots((c_1 H) + c_2)H)\cdots + c_{128})H$ en $c_1 H^{128} + c_2 H^{127} + \cdots + c_{128} H$
- Problèmes :
  - Requiert une transformation des données
  - Nécessite au moins 128 blocs
  - Efficace seulement pour une grande quantité de données

Peut-on utiliser la même représentation que nous utilisons pour AES ?

- Probablement, mais ça ne serait pas efficace (je ne vois pas comment)

Multiplications dans $\mathbb{F}_{2^{128}}$

Est-ce-qu'on peut utiliser bitslicing ?

- Le bitslicing est très efficace pour les multiplications dans $\mathbb{F}_{2^{128}}$
- Il faut en faire 128 en parallèle
- On peut transformer $(\ldots((c_1 H) + c_2)H)\cdots + c_{128})H$ en $c_1 H^{128} + c_2 H^{127} + \cdots + c_{128} H$
- Problèmes :
  - Requiert une transformation des données
  - Nécessite au moins 128 blocs
  - Efficace seulement pour une grande quantité de données

Peut-on utiliser la même représentation que nous utilisons pour AES ?

- Probablement, mais ça ne serait pas efficace (je ne vois pas comment)

$\Longrightarrow$ Multiplications dans répresentation « normale »

Une implantation rapide

- Idée : précalculer 32 tables $T_0, \ldots, T_{31}$ avec 16 multiples de H par table
- T_0 contient $0, H, xH, \ldots, (x^3 + x^2 + x + 1)H$
- T_1 contient $0, x^4H, (x^5)H, \ldots, (x^7 + x^6 + x^5 + x^4)H$
- $\ldots$
- Au total : 8 ko de tables

Une implantation rapide

- Idée : précalculer 32 tables $T_0, \ldots, T_{31}$ avec 16 multiples de H par table
- T_0 contient $0, H, xH, \ldots, (x^3 + x^2 + x + 1)H$
- T_1 contient $0, x^4H, (x^5)H, \ldots, (x^7 + x^6 + x^5 + x^4)H$
- ...
- Au total : 8 ko de tables
- Soit $C = \sum_{i=0}^{31} C_i x^{4i}$
- CH est $\sum_{i=0}^{31} T_i[C_i]$
- Nous calculons CH avec 84 instructions arithmetiques et 32 instruction de chargement
- $\approx$ 3 cycles/octet pour la multiplication
- 10.68 cycles/octet pour AES-GCM sur un Core 2 Q9550

Attaques cache-timing contre GCM ?

- La première multiplication avec H est avec c_1
- L'attaquant le connait de toute façon
- La deuxième multiplication est avec $c_1 H$, ça contient le secret H

Attaques cache-timing contre GCM ?

- La première multiplication avec H est avec c_1
- L'attaquant le connait de toute façon
- La deuxième multiplication est avec $c_1 H$, ça contient le secret H
- Taille d'une ligne de cache : 64 octets (4 multiples de H)
- Si on connait la ligne de cache, on apprend 2 bits de $c_1 H$
- Au total (32 chargements) : 64 bits de $c_1 H$

$\Longrightarrow$ En principe la multiplication rapide est vulnérable

Une implantation protégée aux attaques temporelles

- Précalcule 127 multiples de H : $H, xH, x^2H, \ldots, x^{127}H$
- Calcule CH comme

  $CH \leftarrow 0$
  for $i \leftarrow 0$ to 127 **do**
  if bit(i) of C is 1 **then**
  $CH \leftarrow CH \oplus X^i H$
  end if
  end for
- Remplace l'instruction conditionelle avec des instructions arithmetiques
- Requiert 6 instructions logiques par bit

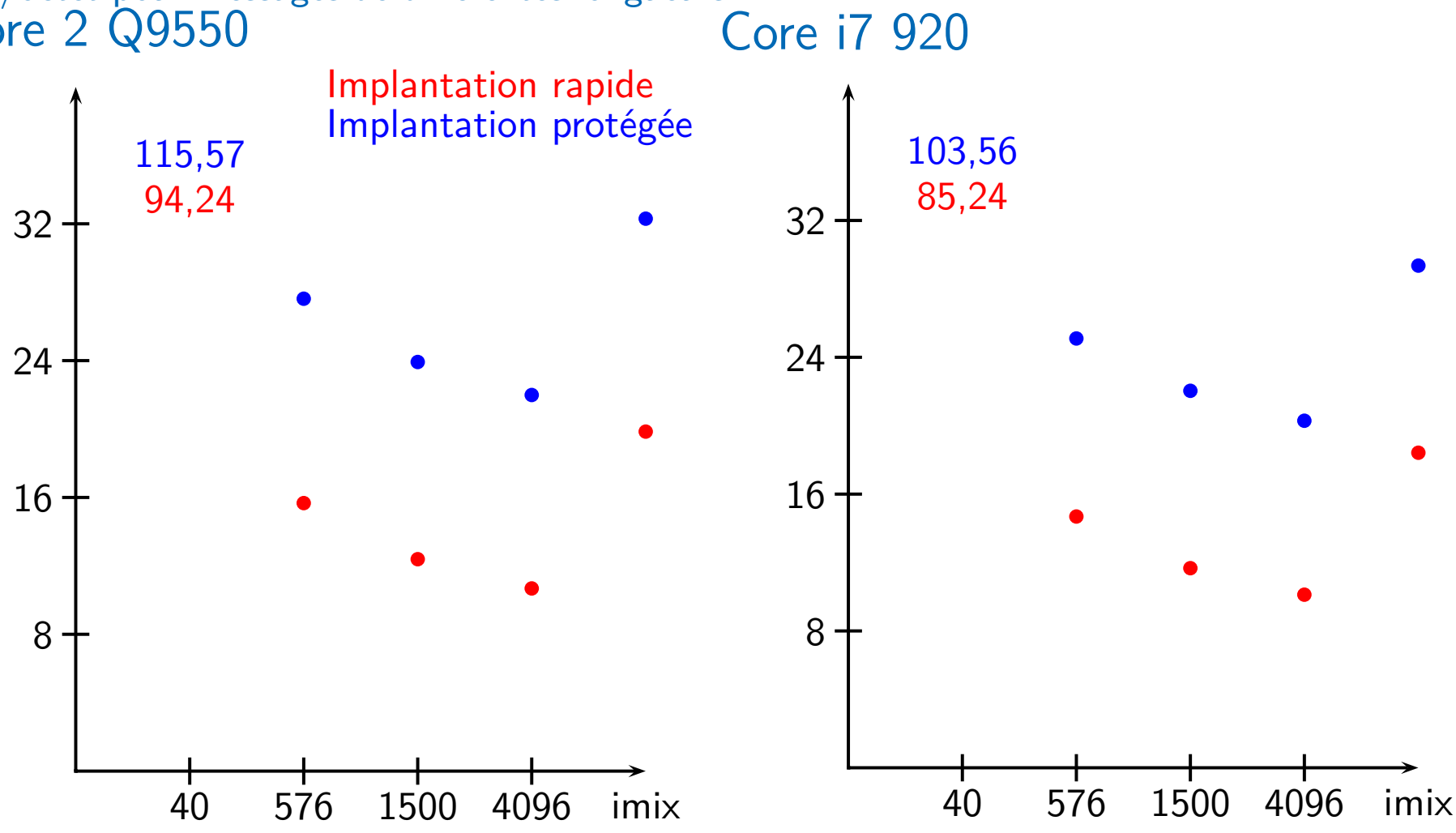
Une implantation protégée aux attaques temporelles

- Précalcule 127 multiples de H : $H, xH, x^2H, \ldots, x^{127}H$
- Calcule CH comme

  $CH \leftarrow 0$
  for $i \leftarrow 0$ to 127 **do**
  if bit(i) of C is 1 **then**
  $CH \leftarrow CH \oplus X^i H$
  end if
  end for
- Remplace l'instruction conditionelle avec des instructions arithmetiques
- Requiert 6 instructions logiques par bit
- $\approx$ 14 cycles/octet pour la multiplication
- 21.99 cycles/octet pour AES-GCM sur un Core 2 Q9550

Résultats II (AES-GCM)

cycles/octet pour messages de différentes longueurs



Gladman : 19.8 cycles/octet (tables de 64 ko), 22.3 cycles/octet (tables de 8 ko) sur `amd64`

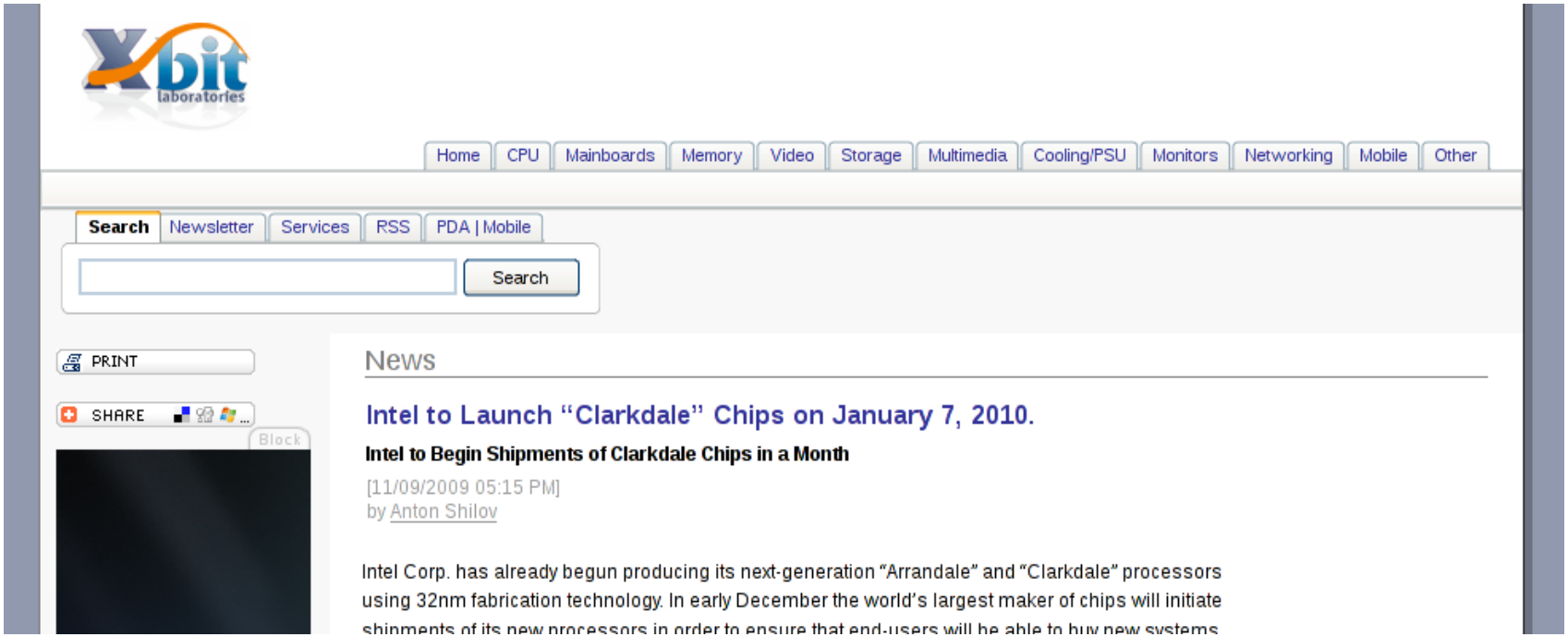
Conclusions

- Implantation d'AES-CTR :
  - 7.59 cycles/octet sur Core 2 Q9550
  - 6.92 cycles/octet sur Core i7 920
- Implantation d'AES-GCM :
  - 10.68 cycles/octet sur Core 2 Q9550
  - 10.12 cycles/octet sur Core i7 920
- Première implantation d'AES-GCM protégée aux attaques temporelles

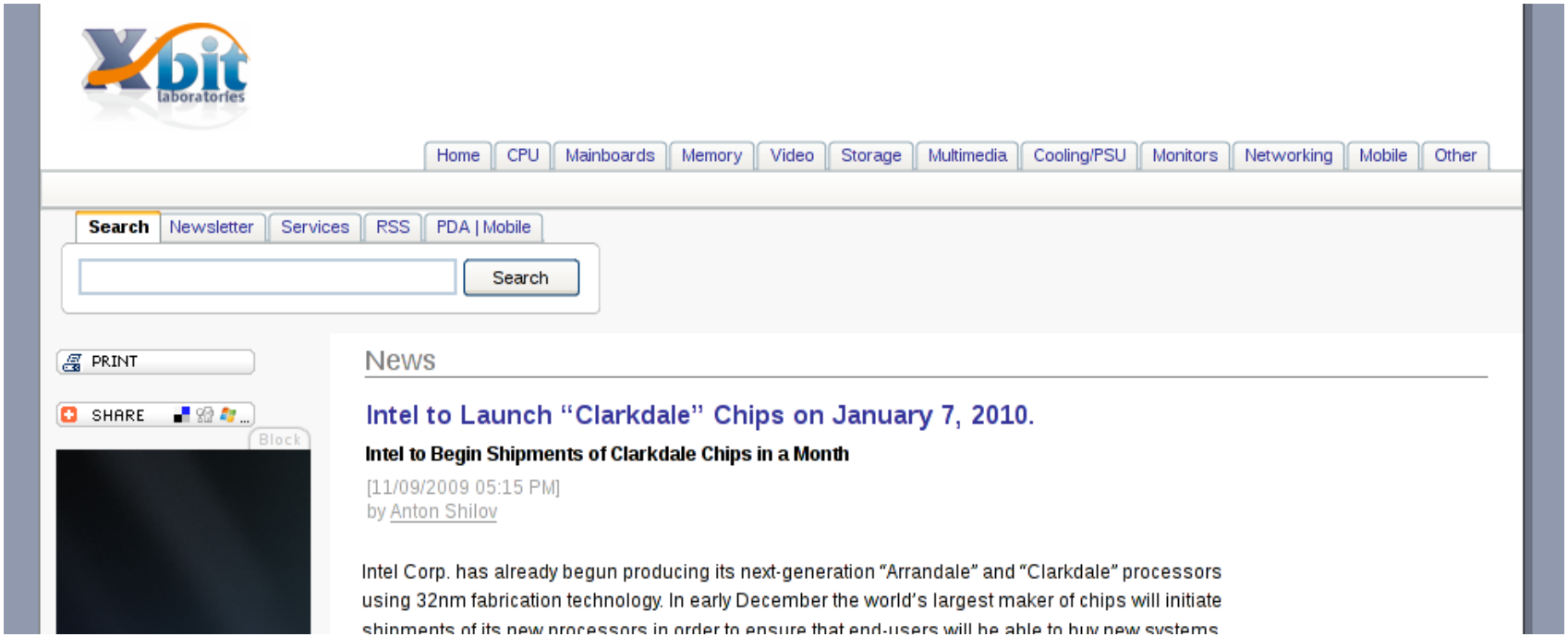
Conclusions

- Implantation d'AES-CTR :
  - 7.59 cycles/octet sur Core 2 Q9550
  - 6.92 cycles/octet sur Core i7 920
- Implantation d'AES-GCM :
  - 10.68 cycles/octet sur Core 2 Q9550
  - 10.12 cycles/octet sur Core i7 920
- Première implantation d'AES-GCM protégée aux attaques temporelles
- Les techniques (bitslicing) sont interessantes pour beaucoup d'autres chiffres
- Ça devient même plus efficace avec `AVX` (registres de 256 bits)
- `AVX` aura aussi instructions avec 3 opérandes

Et les instructions d'AES ?



Et les instructions d'AES ?



- Les processeurs « Clarkdale »ne sont pas (encore) disponibles
- Beaucoup de monde possède des Core 2 ou Core i7
- Pas tout le monde achètera les nouveaux processeurs

Logiciels et Papier

Logiciels

http://homes.esat.kuleuven.be/~ekasper/#software (assembler)
http://cryptojedi.org/crypto/#aesbs (qhasm)
Les logiciels sont dans le domaine public

Papier

http://eprint.iacr.org/2009/129