Algorithmes Parallèles de Multiplication Scalaire Optimisée sur Courbes Elliptiques Binaires

Jean-Marc Robert

Approches Parallèles de Multiplication Scalaire sur Courbe Elliptique Binaire

Jean-Marc ROBERT

Team DALI/LIRMM, Université de Perpignan, France

le 3 mars 2014

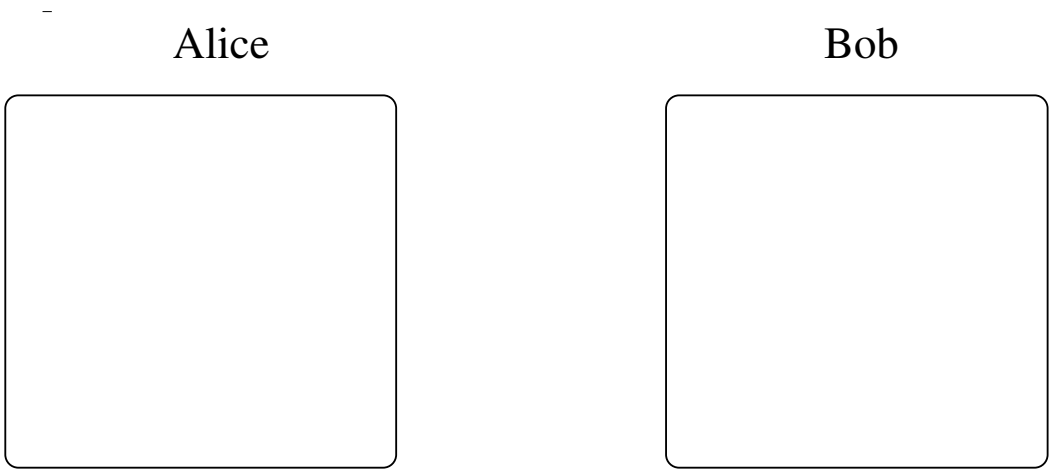
Thèse effectuée dans le cadre du projet PAVOIS ANR 12 BS02 002 02

Table des matières

Table des matières

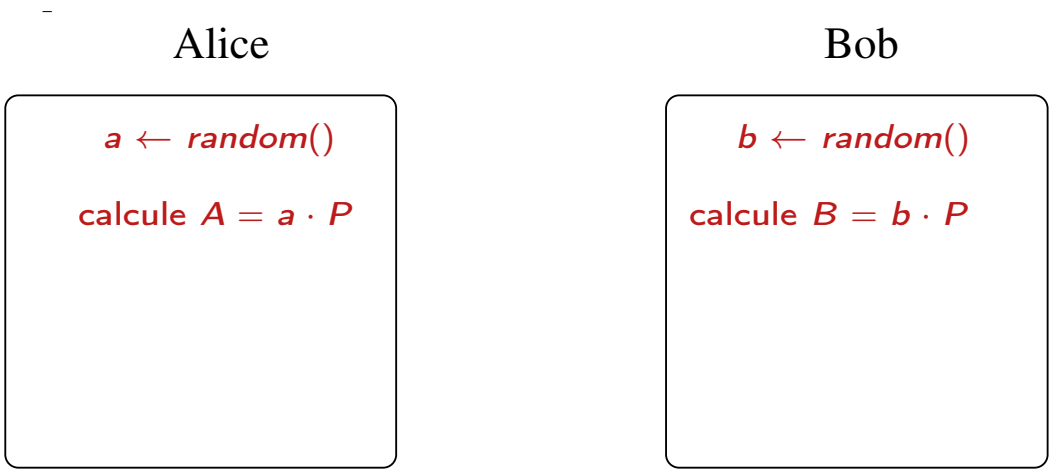
Échange de clé de Diffie-Hellmann

Alice et Bob s'accordent sur un groupe mathématique $(G, +, \mathcal{O})$ et un générateur de ce groupe P.



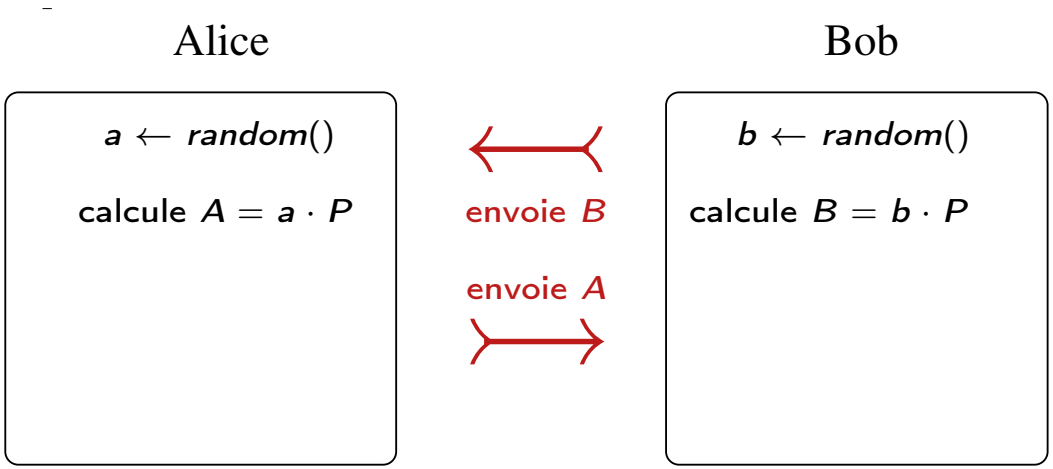
Échange de clé de Diffie-Hellmann

Alice et Bob s'accordent sur un groupe mathématique $(G, +, \mathcal{O})$ et un générateur de ce groupe P.



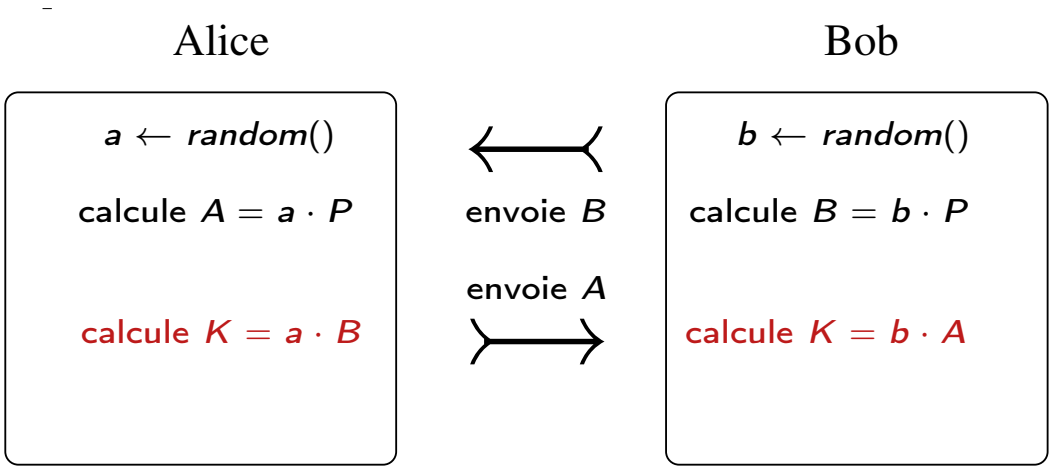
Échange de clé de Diffie-Hellmann

Alice et Bob s'accordent sur un groupe mathématique $(G, +, \mathcal{O})$ et un générateur de ce groupe P.



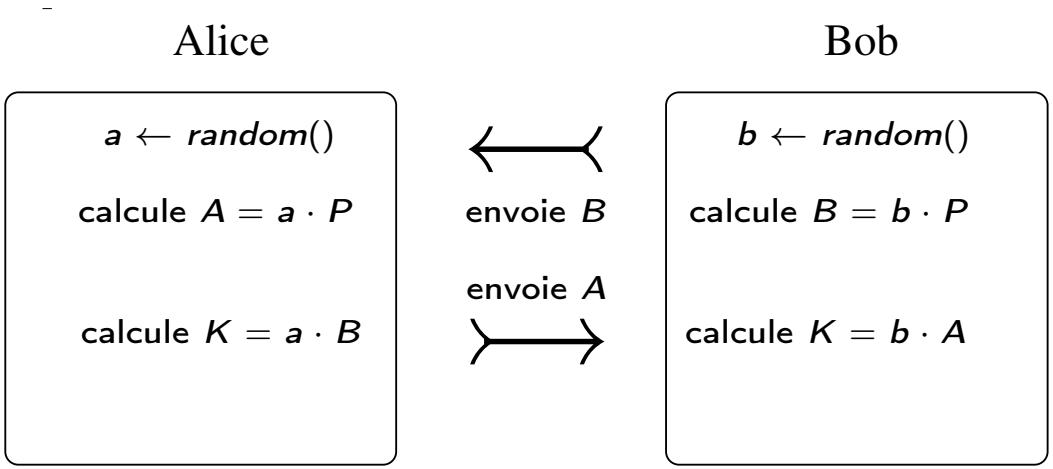
Échange de clé de Diffie-Hellmann

Alice et Bob s'accordent sur un groupe mathématique $(G, +, \mathcal{O})$ et un générateur de ce groupe P.



Échange de clé de Diffie-Hellmann

Alice et Bob s'accordent sur un groupe mathématique $(G, +, \mathcal{O})$ et un générateur de ce groupe P.

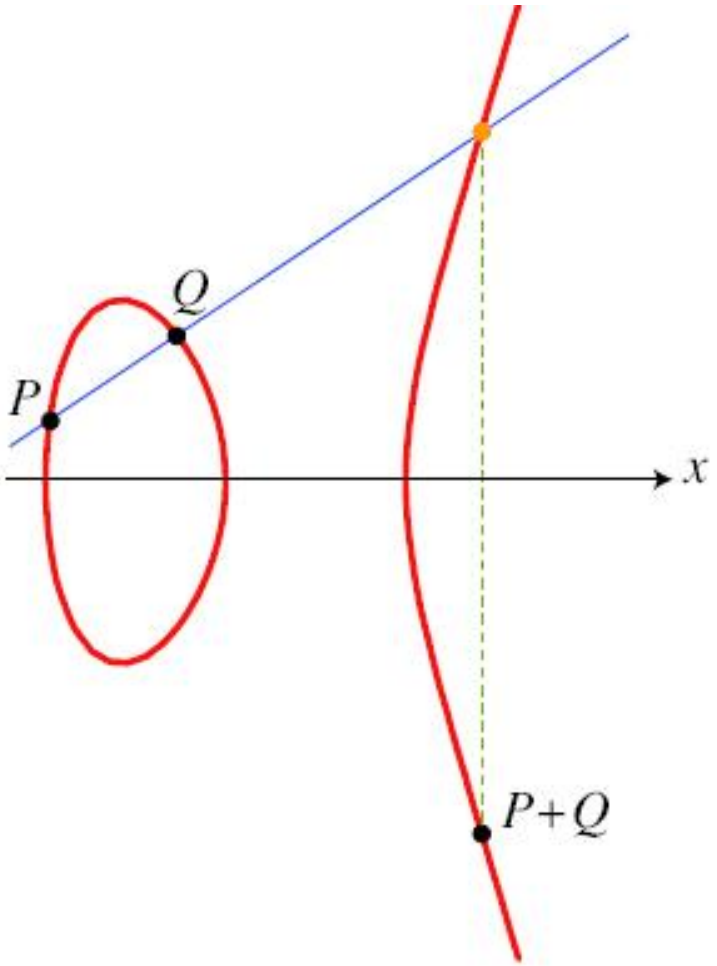


clé secrète partagée $K = a \cdot b \cdot P$

→La multiplication scalaire est la principale opération effectuée $a \cdot P$.

Notre groupe : Courbe Elliptique

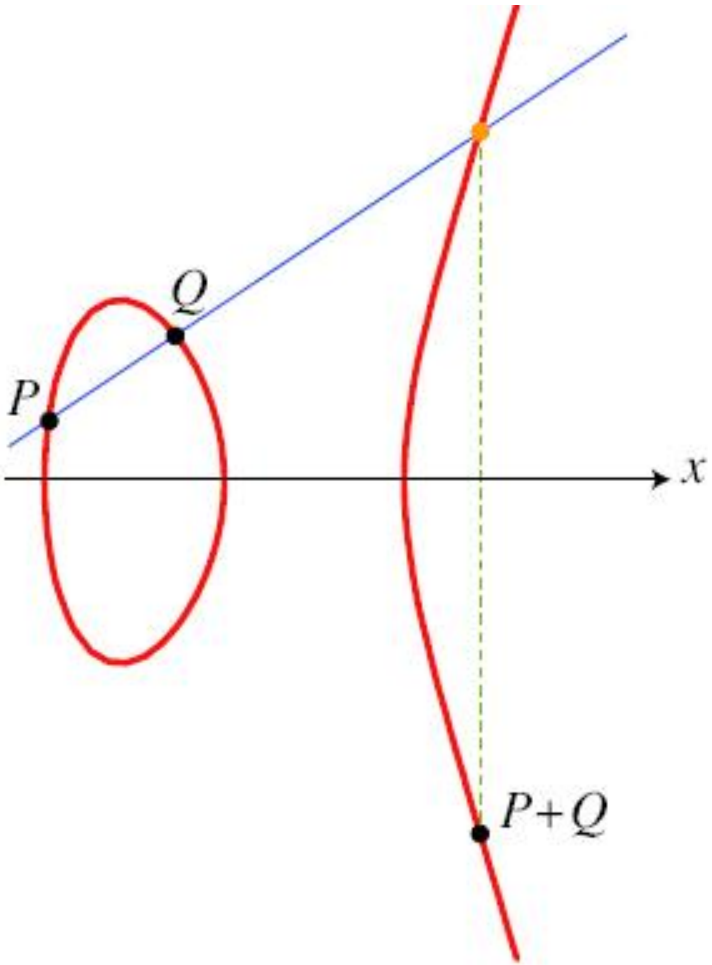
Exemples sur $\mathbb{R}$:



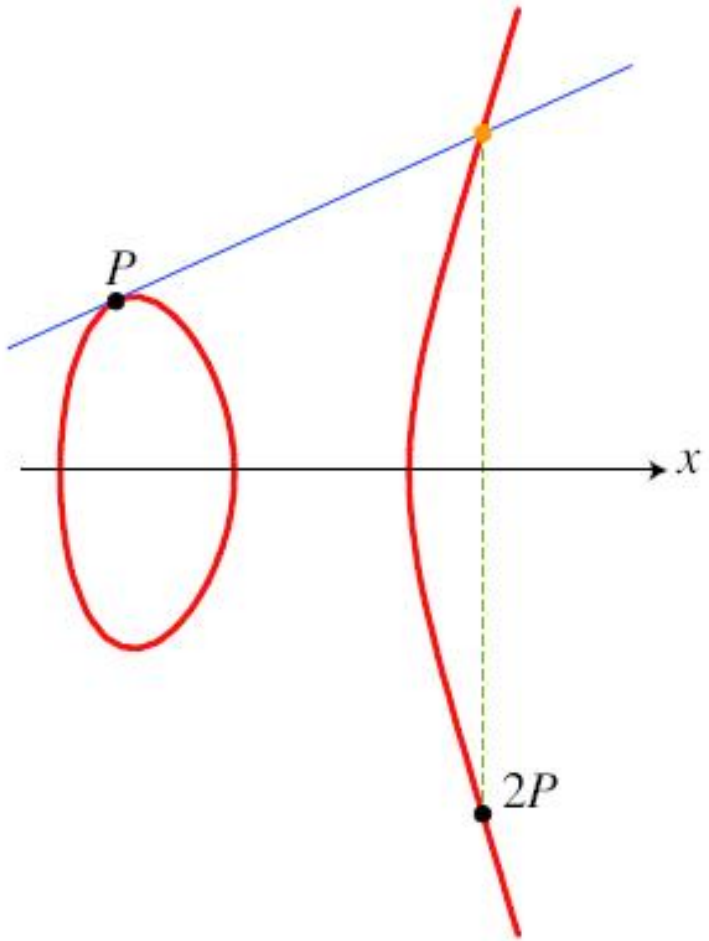
Addition de points

Notre groupe : Courbe Elliptique

Exemples sur $\mathbb{R}$:



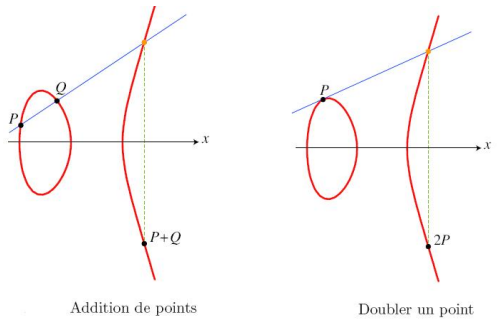
Addition de points



Doubler un point

Notre groupe : Courbe Elliptique

Notre courbe est sur $\mathbb{F}_{2^m}$ (et non $\mathbb{R}$) :



$$E : Y^2 + XY = X^3 + aX^2 + b, \quad a, b \in \mathbb{F}_{2^m}.$$

- Les coordonnées des points appartiennent à $\mathbb{F}_{2^m} = \mathbb{F}_2[x]/(f(x) \cdot \mathbb{F}_2[x])$
- Soit $A = \sum_{i=0}^{m-1} a_i \cdot x^i$ et $B = \sum_{i=0}^{m-1} b_i \cdot x^i, \quad a_i, b_i \in \{0, 1\}$

$$\text{alors : } A + B = \sum_{i=0}^{m-1} (a_i + b_i) \cdot x^i,$$

$$\text{et : } A \times B = A \cdot B \mod f.$$

Table des matières

Opérations entre points de courbe elliptique

Les formules pour le doublement et l'addition sont :

$$\begin{cases} x_3 = \lambda^2 + \lambda + x_1 + x_2 + a \\ y_3 = (x_1 + x_3)\lambda + x_3 + y_1 \end{cases} \quad \text{avec} \quad \begin{cases} \lambda = \frac{y_1+y_2}{x_1+x_2} \text{ if } P_1 \neq P_2 \\ \\ \lambda = \frac{y_1}{x_1} + x_1 \text{ if } P_1 = P_2 \end{cases}$$

Opérations entre points de courbe elliptique

Les formules pour le doublement et l'addition sont :

$$\begin{cases} x_3 = \lambda^2 + \lambda + x_1 + x_2 + a \\ y_3 = (x_1 + x_3)\lambda + x_3 + y_1 \end{cases} \quad \text{avec} \quad \begin{cases} \lambda = \frac{y_1+y_2}{x_1+x_2} \text{ if } P_1 \neq P_2 \\ \\ \lambda = \frac{y_1}{x_1} + x_1 \text{ if } P_1 = P_2 \end{cases}$$

- Un doublement nécessite 1 inversion, 2 multiplications, 1 élévation au carré , et 8 additions de polynômes ;
- Une addition nécessite 1 inversion, 2 multiplications, 1 élévation au carré , et 9 additions de polynômes ;

Points de Courbes Elliptiques : Coordonnées Projectives

- Lopez-Dahab ont montré l'intérêt des coordonnées "*projectives*" . Un point en coordonnées affines est transformé comme suit :

$$P = (x, y) \text{ devient } (X : Y : Z) \text{ avec } \begin{cases} x = \frac{X}{Z} \\ y = \frac{Y}{Z^2} \end{cases}$$

Points de Courbes Elliptiques : Coordonnées Projectives

- Lopez-Dahab ont montré l'intérêt des coordonnées "*projectives*" . Un point en coordonnées affines est transformé comme suit :

$$P = (x, y) \text{ devient } (X : Y : Z) \text{ avec } \begin{cases} x = \frac{X}{Z} \\ y = \frac{Y}{Z^2} \end{cases}$$

- Maintenant, on effectue le doublement de la façon suivante :

$$2.(X : Y : Z) = (X_1 : Y_1 : Z_1) \text{ avec } \begin{cases} X_1 = X^4 + b \cdot Z^4 \\ Y_1 = bZ^4 \cdot Z_1 + X_1 \cdot (aZ_1 + Y^2 + bZ^4) \\ Z_1 = X^2 \cdot Z^2 \end{cases}$$

Points de Courbes Elliptiques : Coordonnées Projectives

- Lopez-Dahab ont montré l'intérêt des coordonnées "*projectives*" . Un point en coordonnées affines est transformé comme suit :

$$P = (x, y) \text{ devient } (X : Y : Z) \text{ avec } \begin{cases} x = \frac{X}{Z} \\ y = \frac{Y}{Z^2} \end{cases}$$

- Maintenant, on effectue le doublement de la façon suivante :

$$2.(X : Y : Z) = (X_1 : Y_1 : Z_1) \text{ avec } \begin{cases} X_1 = X^4 + b \cdot Z^4 \\ Y_1 = bZ^4 \cdot Z_1 + X_1 \cdot (aZ_1 + Y^2 + bZ^4) \\ Z_1 = X^2 \cdot Z^2 \end{cases}$$

Operation	Affine	Lopez-Dahab
Doublement	2M + 1S + 1I	4M + 4S
Addition	2M + 1S + 1I	13M + 4S
Addition (coordonnées mélangées)	-	9M + 5S (Projectives + Affine)

Table des matières

L'Algorithme de Produit Scalaire : *Double-and-add*

- Soit $k = (k_{t-1}, ..., k_1, k_0)_2 \in \mathbb{N}, P \in E(\mathbb{F}_{2^m})$

$$\begin{aligned} k \cdot P &= (\sum_{i=0}^{t-1} 2^i k_i) \cdot P \\ &= ((\ldots((k_{t-1} \cdot 2 + k_{t-2}) \cdot 2 + k_{t-3}) \cdot 2 \ldots + k_1) \cdot 2 + k_0) \cdot P \\ &= ((\ldots((k_{t-1} \cdot P \cdot 2 + k_{t-2} \cdot P) \cdot 2 + k_{t-3} \cdot P) \cdot 2 \ldots + k_1 \cdot P) \cdot 2 + k_0 \cdot P) \end{aligned}$$

L'Algorithme de Produit Scalaire : *Double-and-add*

- Soit $k = (k_{t-1}, ..., k_1, k_0)_2 \in \mathbb{N}, P \in E(\mathbb{F}_{2^m})$

$$\begin{aligned} k \cdot P &= (\sum_{i=0}^{t-1} 2^i k_i) \cdot P \\ &= ((\ldots((k_{t-1} \cdot 2 + k_{t-2}) \cdot 2 + k_{t-3}) \cdot 2 \ldots + k_1) \cdot 2 + k_0) \cdot P \\ &= ((\ldots((k_{t-1} \cdot P \cdot 2 + k_{t-2} \cdot P) \cdot 2 + k_{t-3} \cdot P) \cdot 2 \ldots + k_1 \cdot P) \cdot 2 + k_0 \cdot P) \end{aligned}$$

- On traduit cela dans l'algorithme suivant :

Left-to-Right double-and-add Elliptic Curve Scalar Multiplication (ECSM)

Require: $k = (k_{t-1}, \ldots, k_1, k_0), P \in E(\mathbb{F}_{2^m})$
Ensure: $Q = k \cdot P$
1: $Q \leftarrow \mathcal{O}$
2: **for** i from $t-1$ downto 0 **do**
3: $\quad Q \leftarrow 2 \cdot Q$
4: $\quad$ **if** $k_i = 1$ **then**
5: $\quad\quad Q \leftarrow Q + P$
6: $\quad$ **end if**
7: **end for**
8: **return** (Q)

L'Algorithme de Produit Scalaire : *Double-and-add*

- Soit $k = (k_{t-1}, ..., k_1, k_0)_2 \in \mathbb{N}, P \in E(\mathbb{F}_{2^m})$

$$\begin{aligned} k \cdot P &= (\sum_{i=0}^{t-1} 2^i k_i) \cdot P \\ &= ((\ldots((k_{t-1} \cdot 2 + k_{t-2}) \cdot 2 + k_{t-3}) \cdot 2 \ldots + k_1) \cdot 2 + k_0) \cdot P \\ &= ((\ldots((k_{t-1} \cdot P \cdot 2 + k_{t-2} \cdot P) \cdot 2 + k_{t-3} \cdot P) \cdot 2 \ldots + k_1 \cdot P) \cdot 2 + k_0 \cdot P) \end{aligned}$$

- On traduit cela dans l'algorithme suivant :

Left-to-Right double-and-add
Elliptic Curve Scalar Multiplication (ECSM)

Require: $k = (k_{t-1}, \ldots, k_1, k_0), P \in E(\mathbb{F}_{2^m})$
Ensure: $Q = k \cdot P$
1: $Q \leftarrow \mathcal{O}$
2: **for** i from $t-1$ downto 0 **do**
3: $\quad Q \leftarrow 2 \cdot Q$
4: $\quad$ **if** $k_i = 1$ **then**
5: $\quad\quad Q \leftarrow Q + P$
6: $\quad$ **end if**
7: **end for**
8: **return** (Q)

"mixed coordinates addition" $\Rightarrow$

Amélioration de *Double-and-add* : *NAF* et *W-NAF*.

- *NAF* remplace les séquences de 1 consecutifs : soit $k \in \mathbb{N}$ tel que $k = 2^i - 1$, alors

$$(k)_2 = \underbrace{111...1}_{i \text{ fois}} \text{ et on peut écrire : } (k)_{NAF} = \underbrace{100...00 - 1}_{i+1 \text{ chiffres}}.$$

$\Rightarrow$ Nombre moyen de chiffres non nuls : de $n/2$ à $n/3$.

Amélioration de *Double-and-add* : *NAF* et *W-NAF*.

- *NAF* remplace les séquences de 1 consecutifs : soit $k \in \mathbb{N}$ tel que $k = 2^i - 1$, alors

$$(k)_2 = \underbrace{111...1}_{i \text{ fois}} \text{ et on peut écrire : } (k)_{NAF} = \underbrace{100...00 - 1}_{i+1 \text{ chiffres}}.$$

  $\Rightarrow$ Nombre moyen de chiffres non nuls : de $n/2$ à $n/3$.

- *W-NAF* réduit davantage encore le nombre moyen de chiffres non nuls à $n/(w+1)$ en utilisant :

$$\{-2^{w-1} + 1, ..., -5, -3, -1, 0, 1, 3, 5, ..., 2^{w-1} - 1\}.$$

- Bilan sur la complexité du produit scalaire sur $E(\mathbb{F}_{2^m})$:

	nb. de doublements	nb. d'additions
Double-and-add	n	$n/2$
NAF Double-and-add	n	$n/3$
W-NAF Double-and-add	n	$n/(w+1) + 2^{w-2}$

Table des matières

Comparaison *Left-to-right/Right-to-left*

Left-to-Right double-and-add Elliptic Curve Scalar Multiplication (ECSM)

Require: $k = (k_{t-1}, \ldots, k_1, k_0), P \in E(\mathbb{F}_{2^m})$
Ensure: $Q = k \cdot P$
1: $Q \leftarrow \mathcal{O}$
2: **for** i from $t-1$ downto 0 **do**
3: $\quad Q \leftarrow 2 \cdot Q$
4: $\quad$ **if** $k_i = 1$ **then**
5: $\quad\quad Q \leftarrow Q + P$
6: $\quad$ **end if**
7: **end for**
8: **return** (Q)

Right-to-Left double-and-add Elliptic Curve Scalar Multiplication (ECSM)

Require: $k = (k_{t-1}, \ldots, k_1, k_0), P \in E(\mathbb{F}_{2^m})$
Ensure: $Q = k \cdot P$
1: $Q \leftarrow \mathcal{O}$
2: **for** i from 0 to $t-1$ **do**
3: $\quad$ **if** $k_i = 1$ **then**
4: $\quad\quad Q \leftarrow Q + P$
5: $\quad$ **end if**
6: $\quad P \leftarrow 2 \cdot P$
7: **end for**
8: **return** (Q)

Comparaison *Left-to-right/Right-to-left*

Left-to-Right double-and-add Elliptic Curve Scalar Multiplication (ECSM)

Require: $k = (k_{t-1}, \ldots, k_1, k_0), P \in E(\mathbb{F}_{2^m})$
Ensure: $Q = k \cdot P$
1: $Q \leftarrow \mathcal{O}$
2: **for** i from $t-1$ downto 0 **do**
3: $\quad Q \leftarrow 2 \cdot Q$
4: $\quad$ **if** $k_i = 1$ **then**
5: $\quad\quad Q \leftarrow Q + P$
6: $\quad$ **end if**
7: **end for**
8: **return** (Q)

⇑
"mixed coordinates addition"

Right-to-Left double-and-add Elliptic Curve Scalar Multiplication (ECSM)

Require: $k = (k_{t-1}, \ldots, k_1, k_0), P \in E(\mathbb{F}_{2^m})$
Ensure: $Q = k \cdot P$
1: $Q \leftarrow \mathcal{O}$
2: **for** i from 0 to $t-1$ **do**
3: $\quad$ **if** $k_i = 1$ **then**
4: $\quad\quad Q \leftarrow Q + P$
5: $\quad$ **end if**
6: $\quad P \leftarrow 2 \cdot P$
7: **end for**
8: **return** (Q)

⇑
"general addition"

Comparaison *Left-to-right/Right-to-left*

Left-to-Right double-and-add Elliptic Curve Scalar Multiplication (ECSM)

Require: $k = (k_{t-1}, \ldots, k_1, k_0), P \in E(\mathbb{F}_{2^m})$
Ensure: $Q = k \cdot P$
1: $Q \leftarrow \mathcal{O}$
2: **for** i from $t-1$ downto 0 **do**
3: $\quad Q \leftarrow 2 \cdot Q$
4: $\quad$ **if** $k_i = 1$ **then**
5: $\quad\quad Q \leftarrow Q + P$
6: $\quad$ **end if**
7: **end for**
8: **return** (Q)

Right-to-Left double-and-add Elliptic Curve Scalar Multiplication (ECSM)

Require: $k = (k_{t-1}, \ldots, k_1, k_0), P \in E(\mathbb{F}_{2^m})$
Ensure: $Q = k \cdot P$
1: $Q \leftarrow \mathcal{O}$
2: **for** i from 0 to $t-1$ **do**
3: $\quad$ **if** $k_i = 1$ **then**
4: $\quad\quad Q \leftarrow Q + P$
5: $\quad$ **end if**
6: $\quad P \leftarrow 2 \cdot P$
7: **end for**
8: **return** (Q)

⇑ "mixed coordinates addition"

⇑ "general addition"

- L'algorithme *Left-to-right* est plus performant ("mixed coordinates addition")
- L'algorithme *Right-to-left* permet de paralléliser !

Double-and-add parallèle

Require: scalar k, $P \in \mathbb{F}_{2^m}$.
Ensure: kP.
(Barrier)

Compute Doublings
1: $D[0] \leftarrow P$
2: **for** $i = 1$ to 21 **do**
3: //*Doubling* LD projective
$D[i] \leftarrow D[i-1] \times 2$
4: **end for**
5: signal to thread addition
6: **for** $i = 22$ to $M - 1$ **do**
7: //*Doubling* LD projective
$D[i] \leftarrow D[i-1] \times 2$
8: **end for**

(Barrier)
16: **return** Q

Compute Additions
9: $Q \leftarrow \mathcal{O}$
10: Wait for signal from thread Doubling
11: **for** $i = 0$ to $M - 1$ **do**
12: **if** $k_i = 1$ **then**
13: //Full LD projective addition
$Q \leftarrow Q + D[i]$
14: **end if**
15: **end for**

Rappel sur le *Halving*

Si Q est un point de la courbe d'ordre impair, il appartient à un sous groupe au sein duquel le halving de $Q(u, v)$ est l'unique point $P(x, y)$ tel que $P = Q/2$.

$$
\begin{aligned}
(u, v) &= 2 \cdot (x, y) \\
\lambda &= x + y/x, && (1)\\
u &= \lambda^2 + \lambda + a, && (2)\\
v &= x^2 + u(\lambda + 1). && (3)
\end{aligned}
$$

Rappel sur le *Halving*

Si Q est un point de la courbe d'ordre impair, il appartient à un sous groupe au sein duquel le halving de $Q(u, v)$ est l'unique point $P(x, y)$ tel que $P = Q/2$.

$$\begin{aligned} (u, v) &= 2 \cdot (x, y) \\ \lambda &= x + y/x, && (1) \\ u &= \lambda^2 + \lambda + a, && (2) \\ v &= x^2 + u(\lambda + 1). && (3) \end{aligned}$$

Pour effectuer le calcul de P :

- on résoud l'équation (2): $\lambda^2 + \lambda = u + a \rightarrow \lambda$ (*Quadratic Solver*);
- on résoud l'équation (3): $x^2 = v + u(\lambda + 1) \rightarrow x$ (*Square Root*);
- on calcule y (1): $y = \lambda x + x^2$.

Rappel sur le *Halving*

Si Q est un point de la courbe d'ordre impair, il appartient à un sous groupe au sein duquel le halving de $Q(u, v)$ est l'unique point $P(x, y)$ tel que $P = Q/2$.

$$\begin{aligned} (u, v) &= 2 \cdot (x, y) \\ \lambda &= x + y/x, && (1) \\ u &= \lambda^2 + \lambda + a, && (2) \\ v &= x^2 + u(\lambda + 1). && (3) \end{aligned}$$

Pour effectuer le calcul de P :

- on résoud l'équation (2): $\lambda^2 + \lambda = u + a \rightarrow \lambda$ (*Quadratic Solver*);
- on résoud l'équation (3): $x^2 = v + u(\lambda + 1) \rightarrow x$ (*Square Root*);
- on calcule y (1): $y = \lambda x + x^2$.

$\rightarrow$ un *halving* nécessite : 1 *Quadratic Solver*, 1 *Square Root*, 1 M et 1 S.

$\rightarrow$ c'est plus efficace qu'un *doubling*.

Halve-and-add parallèle

Require: scalar k, $P \in \mathbb{F}_{2^m}$ of odd order r.
Ensure: kP.

1: Recode: $k' = k \cdot 2^m \mod r$.
2: Compute $NAFk'$, NAF representation of k' (Barrier)

Compute Halvings
3: $H[M-1] \leftarrow P$
4: $H[M-2] \leftarrow P/2$
5: **for** $i = M-2$ downto $M-11$ **do**
6: //*Halving*
$H[i] \leftarrow H[i+1]/2$
7: **end for**

8: signal to thread addition

9: **for** $i = M-12$ downto 0 **do**
10: $H[i] \leftarrow H[i+1]/2$ //
11: **end for**

(Barrier)
22: **return** Q

Compute Additions
12: Wait for signal from thread halving
13: $Q \leftarrow \mathcal{O}$
14: **for** $i = M-1$ downto 0 **do**
15: **if** $NAFk'_i = 1$ **then**
16: $Q \leftarrow Q + H[i]$ //mixed addition projective-affine
17: **end if**
18: **if** $NAFk'_i = -1$ **then**
19: $Q \leftarrow Q - H[i]$ //mixed addition projective-affine
20: **end if**
21: **end for**

Table des matières

Algorithme parallèle

Le scalaire k est recodé en $k' = k \cdot 2^{\ell-1} \mod \#<P>$:

$$k = k'2^{-(\ell-1)} = (\sum_{i=0}^{m-1} k'_i 2^i)2^{-(\ell-1)} = \underbrace{\sum_{i=0}^{\ell-1} k'_i 2^{i-(\ell-1)}}_{\leq 0 \text{ powers of } 2} + \underbrace{\sum_{i=\ell}^{m-1} k'_i 2^{i-(\ell-1)}}_{>0 \text{ powers of } 2}$$

Algorithme parallèle

On en déduit l'algorithme suivant :

Recode k

k is split in two subkeys (> 0 powers of 2, and ≤ 0 powers of 2).

Algorithme parallèle

On en déduit l'algorithme suivant :

Recode k

k is split in two subkeys (> 0 powers of 2, and ≤ 0 powers of 2).

↙ ↘

Double-and-add

Compute $\sum_{i=\ell}^{m-1} k'_i 2^{i-(\ell-1)} \cdot P$.

Halve-and-add

Compute $\sum_{i=0}^{\ell-1} k'_i 2^{i-(\ell-1)} \cdot P$.

Algorithme parallèle

On en déduit l'algorithme suivant :

Recode k

k is split in two subkeys (> 0 powers of 2, and ≤ 0 powers of 2).

Double-and-add

Compute $\sum_{i=\ell}^{m-1} k'_i 2^{i-(\ell-1)} \cdot P$.

Halve-and-add

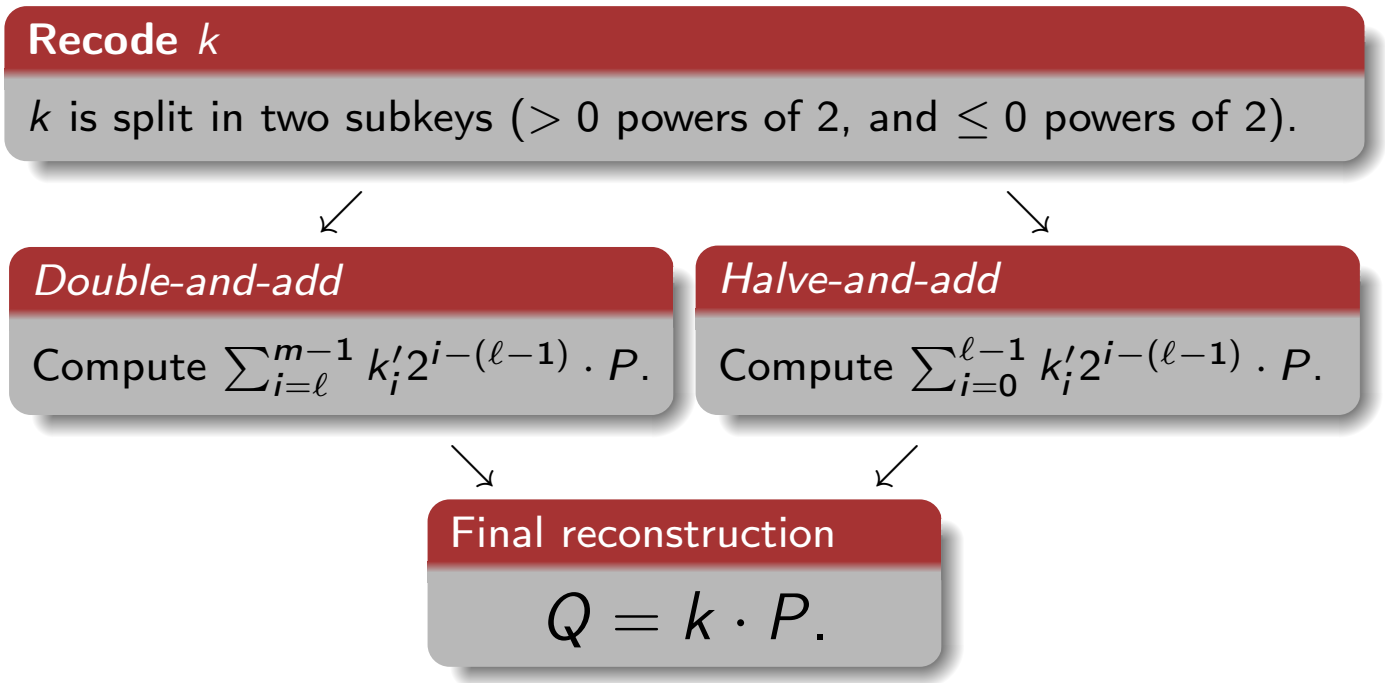
Compute $\sum_{i=0}^{\ell-1} k'_i 2^{i-(\ell-1)} \cdot P$.

Final reconstruction

$$Q = k \cdot P.$$

Algorithme parallèle

On en déduit l'algorithme suivant :



Cet algorithme lance donc 2 *threads* (référence) !

Algorithme parallèle

On en déduit l'algorithme suivant :

Recode k

k is split in two subkeys (> 0 powers of 2, and ≤ 0 powers of 2).

Double-and-add // (2 threads)

Compute $\sum_{i=\ell}^{m-1} k'_i 2^{i-(\ell-1)} \cdot P$.

thread 1

Compute doublings of P

thread 2

Compute additions

Halve-and-add // (2 threads)

Compute $\sum_{i=0}^{\ell-1} k'_i 2^{i-(\ell-1)} \cdot P$.

thread 3

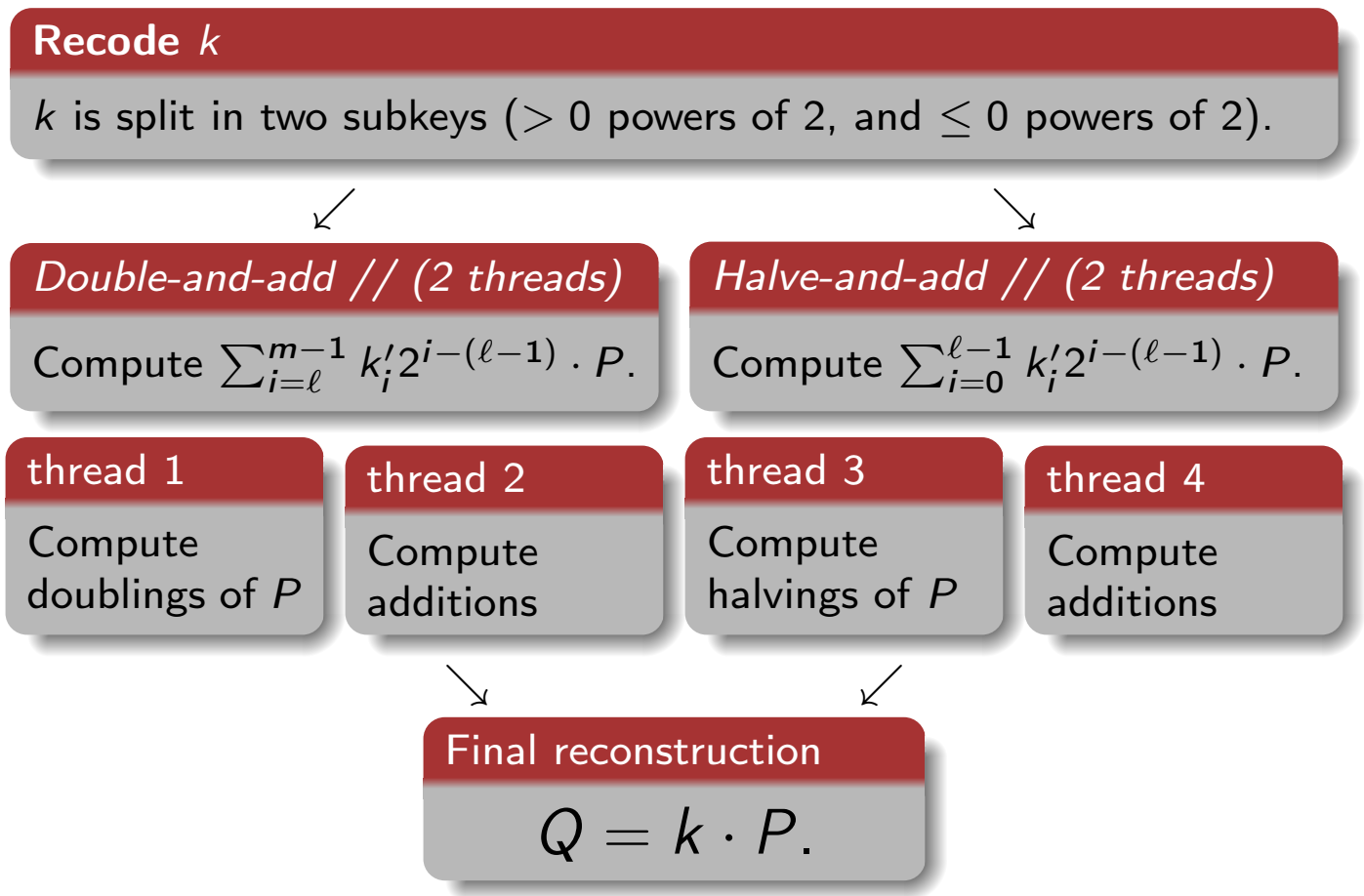
Compute halvings of P

thread 4

Compute additions

Algorithme parallèle

On en déduit l'algorithme suivant :



Cet algorithme lance donc 4 *threads* !

Table des matières

Synthèse sur ces algorithmes

Stratégies d'implémentation :

- langage C ;
- utilisation du jeux d'instructions SSE2/AVX (processeur Intel core i7, multiplication `PCLMULQDQ`, 128 bits) ;
- compilateur gcc 4.6.3, option `-O3` ;
- bibliothèque `PThread` ;
- optimisations de l'état de l'art (opérations combinées, *lazy reduction*...) ;
- courbes elliptiques sur $\mathbb{F}_{2^m}$: NIST B233 et B409.

Synthèse sur ces algorithmes

Résultats (performances) :

Algorithmes parallèles						
courbe NIST	Double-and-add		Halve-and-add		Parallel	
	R-L 2 threads	WNAF L-R seq.	R-L 2 threads	WNAF L-R seq.	R-L 4 threads	WNAF L-R 2 threads
B233	130696	159000	98872	135000	104492	98000
	(-18,6 %)		(-26,7 %)		(+6,6 %)	
B409	615328	706000	407772	539000	326436	347000
	(-12,8 %)		(-24,3 %)		(-5,9 %)	

Nombre de cycles (minimum sur 2000 exécutions, implémentations en C, Optiplex 990, Intel core i7, TurboMode et hyperthreading désactivés)

Synthèse sur ces algorithmes

Algorithmes parallèles						
courbe NIST	Double-and-add		Halve-and-add		Parallel	
	R-L 2 threads	WNAF L-R seq.	R-L 2 threads	WNAF L-R seq.	R-L 4 threads	WNAF L-R 2 threads
B233	130696	159000	98872	135000	104492	98000
	(-18,6 %)		(-26,7 %)		(+6,6 %)	
B409	615328	706000	407772	539000	326436	347000
	(-12,8 %)		(-24,3 %)		(-5,9 %)	

Nombre de cycles (minimum sur 2000 exécutions, implémentations en C, Optiplex 990, Intel core i7, TurboMode et hyperthre

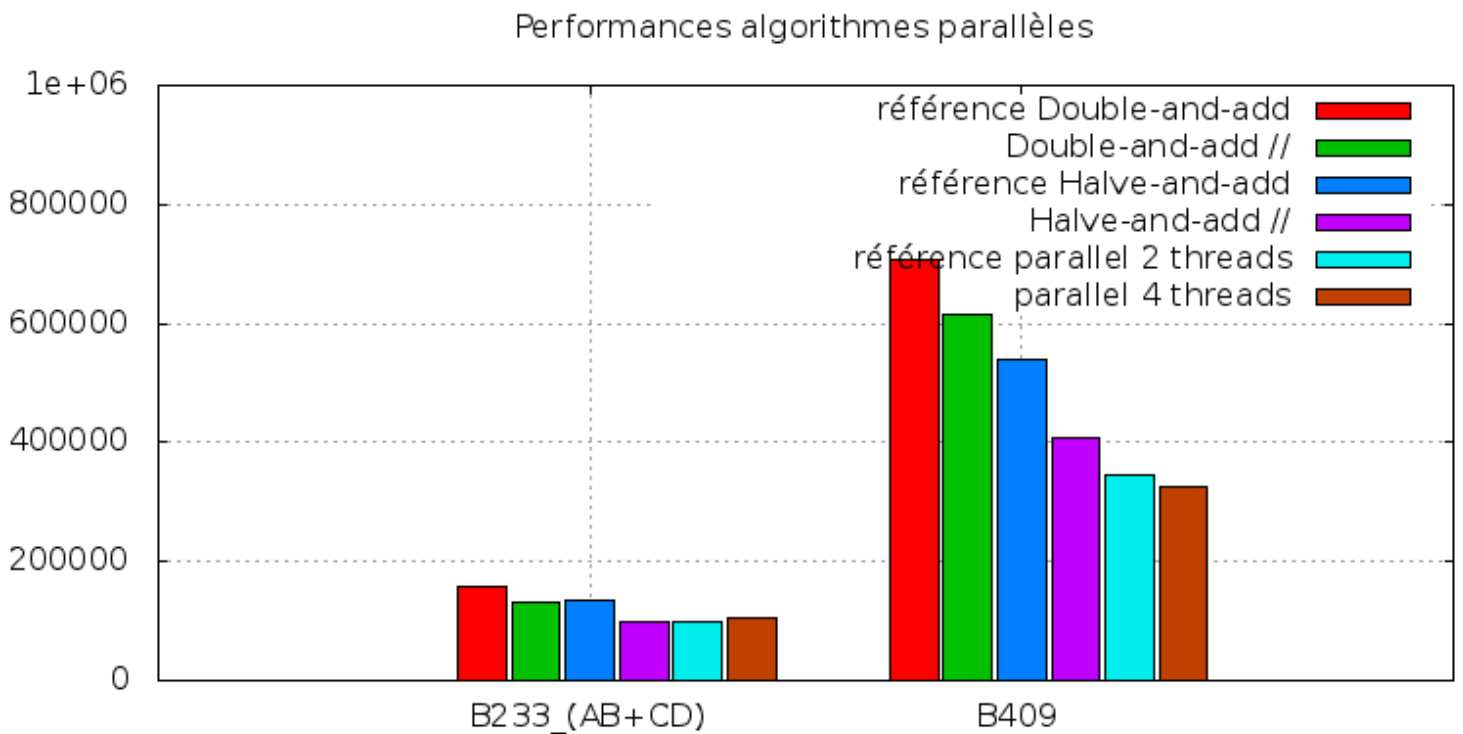
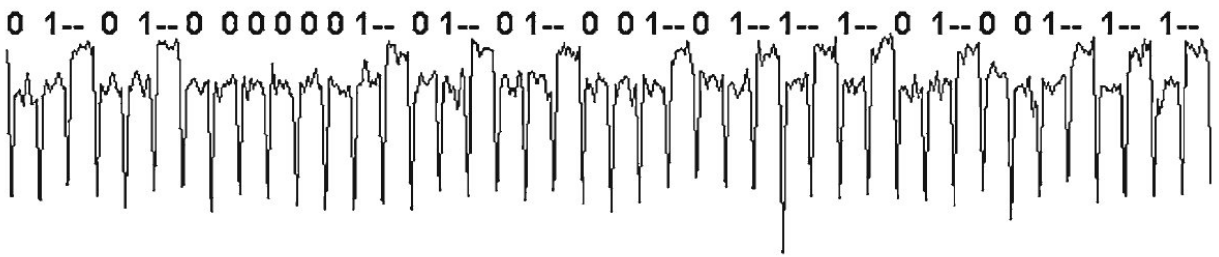


Table des matières

ECSM vulnérable : cas Double-And-Add

Left-to-Right double-and-add
Elliptic Curve Scalar Multiplication (ECSM)

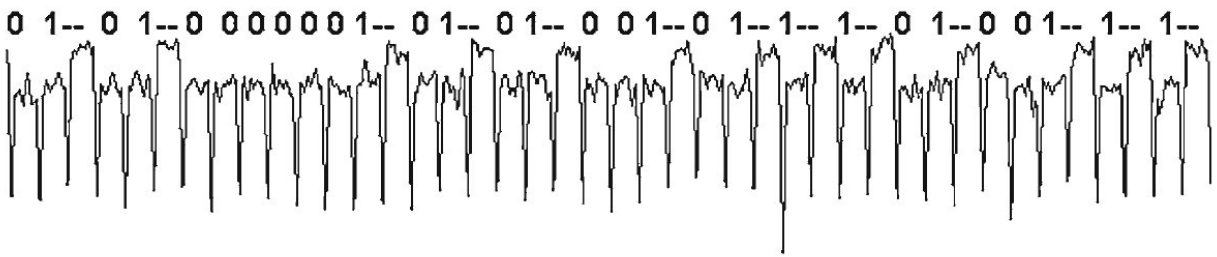
Require: $k = (k_{t-1}, \ldots, k_1, k_0), P \in E(\mathbb{F}_{2^m})$
Ensure: $Q = k \cdot P$
1: $Q \leftarrow \mathcal{O}$
2: **for** i from $t-1$ downto 0 **do**
3: $\quad Q \leftarrow 2 \cdot Q$
4: $\quad$ **if** $k_i = 1$ **then**
5: $\quad\quad Q \leftarrow Q + P$
6: $\quad$ **end if**
7: **end for**
8: **return** (Q)



ECSM vulnérable : cas Double-And-Add

Left-to-Right double-and-add
Elliptic Curve Scalar Multiplication (ECSM)

Require: $k = (k_{t-1}, \ldots, k_1, k_0), P \in E(\mathbb{F}_{2^m})$
Ensure: $Q = k \cdot P$
1: $Q \leftarrow \mathcal{O}$
2: **for** i from $t-1$ downto 0 **do**
3: $\quad Q \leftarrow 2 \cdot Q$
4: $\quad$ **if** $k_i = 1$ **then**
5: $\quad\quad Q \leftarrow Q + P$
6: $\quad$ **end if**
7: **end for**
8: **return** (Q)



$\rightarrow$ Vulnérable : pas de régularité des opérations

$\rightarrow$ Simple Power Analysis

Table des matières

Une contre mesure à l'attaque précédente : Montgomery

Montgomery

Require: $k = (k_{t-1}, \ldots, k_1, k_0)$ with $k_{t-1} = 1, P \in E(\mathbb{F}_q)$
Ensure: $Q = k \cdot P$

1: $Q_0 \leftarrow P, Q_1 \leftarrow 2P$
2: **for** i from $t-2$ downto 0 **do**
3: **if** $(k_i = 0)$ **then**
4: $Q_1 \leftarrow Q_0 + Q_1, Q_0 \leftarrow 2 \cdot Q_0$
5: **else**
6: $Q_0 \leftarrow Q_0 + Q_1, Q_1 \leftarrow 2 \cdot Q_1$
7: **end if**
8: **end for**
9: **return** (Q_0)

Basic Montgomery's ladder ECSM

Comment cet algorithme fonctionne-t-il ?

Exemple : $k = 45_{10} = 0x2D = [1,0,1,1,0,1]_2$

On initialise : $Q_0 = P; Q_1 = 2P$.

1: **if** $(k_i = 0)$ **then**
2: $\quad Q_1 \leftarrow Q_0 + Q_1, Q_0 \leftarrow 2 \cdot Q_0$
3: **else**
4: $\quad Q_0 \leftarrow Q_0 + Q_1, Q_1 \leftarrow 2 \cdot Q_1$
5: **end if**

i	init	4	3	2	1	0
k_i	1	0	1	1	0	1
Q_0 Q_1						

Comment cet algorithme fonctionne-t-il ?

Exemple : $k = 45_{10} = 0x2D = [1,0,1,1,0,1]_2$

On initialise : $Q_0 = P; Q_1 = 2P$.

1: **if** $(k_i = 0)$ **then**
2: $\quad Q_1 \leftarrow Q_0 + Q_1, Q_0 \leftarrow 2 \cdot Q_0$
3: **else**
4: $\quad Q_0 \leftarrow Q_0 + Q_1, Q_1 \leftarrow 2 \cdot Q_1$
5: **end if**

i	init	4	3	2	1	0
k_i	1	0	1	1	0	1
Q_0 Q_1	P					

Comment cet algorithme fonctionne-t-il ?

Exemple : $k = 45_{10} = 0x2D = [1,0,1,1,0,1]_2$

On initialise : $Q_0 = P; Q_1 = 2P.$

1: **if** $(k_i = 0)$ **then**
2: $\quad Q_1 \leftarrow Q_0 + Q_1, Q_0 \leftarrow 2 \cdot Q_0$
3: **else**
4: $\quad Q_0 \leftarrow Q_0 + Q_1, Q_1 \leftarrow 2 \cdot Q_1$
5: **end if**

i	init	4	3	2	1	0
k_i	1	0	1	1	0	1
Q_0	P					
Q_1	2P					

Comment cet algorithme fonctionne-t-il ?

Exemple : $k = 45_{10} = 0x2D = [1,0,1,1,0,1]_2$

On initialise : $Q_0 = P; Q_1 = 2P$.

1: **if** $(k_i = 0)$ **then**
2: $\quad Q_1 \leftarrow Q_0 + Q_1, Q_0 \leftarrow 2 \cdot Q_0$
3: **else**
4: $\quad Q_0 \leftarrow Q_0 + Q_1, Q_1 \leftarrow 2 \cdot Q_1$
5: **end if**

i	init	4	3	2	1	0
k_i	1	0	1	1	0	1
Q_0	P					
Q_1	2P	3P				

Comment cet algorithme fonctionne-t-il ?

Exemple : $k = 45_{10} = 0x2D = [1,0,1,1,0,1]_2$

On initialise : $Q_0 = P; Q_1 = 2P$.

1: **if** $(k_i = 0)$ **then**
2: $\quad Q_1 \leftarrow Q_0 + Q_1, Q_0 \leftarrow 2 \cdot Q_0$
3: **else**
4: $\quad Q_0 \leftarrow Q_0 + Q_1, Q_1 \leftarrow 2 \cdot Q_1$
5: **end if**

$\Longleftarrow$

Double :

$Q_0 \leftarrow 2 \cdot Q_0$

($Q_1 \leftarrow Q_0 + Q_1 = 2 \cdot Q_0 + P$
d'où $Q_1 - Q_0 = P$)

i	init	4	3	2	1	0
k_i	1	0	1	1	0	1
Q_0	P	2P				
Q_1	2P	3P				

Comment cet algorithme fonctionne-t-il ?

Exemple : $k = 45_{10} = 0x2D = [1,0,1,1,0,1]_2$

On initialise : $Q_0 = P; Q_1 = 2P$.

1: **if** $(k_i = 0)$ **then**
2: $\quad Q_1 \leftarrow Q_0 + Q_1, Q_0 \leftarrow 2 \cdot Q_0$
3: **else**
4: $\quad Q_0 \leftarrow Q_0 + Q_1, Q_1 \leftarrow 2 \cdot Q_1$
5: **end if**

$\Longleftarrow$

Double-and-add :

$Q_0 \leftarrow Q_0 + Q_1 = 2 \cdot Q_0 + P$

($Q_1 \leftarrow 2 \cdot (Q_0 + P)$
d'où $Q_1 - Q_0 = P$)

i	init	4	3	2	1	0
k_i	1	0	1	1	0	1
Q_0	P	2P	5P			
Q_1	2P	3P				

Comment cet algorithme fonctionne-t-il ?

Exemple : $k = 45_{10} = 0x2D = [1,0,1,1,0,1]_2$

On initialise : $Q_0 = P; Q_1 = 2P$.

1: **if** $(k_i = 0)$ **then**
2: $\quad Q_1 \leftarrow Q_0 + Q_1, Q_0 \leftarrow 2 \cdot Q_0$
3: **else**
4: $\quad Q_0 \leftarrow Q_0 + Q_1, Q_1 \leftarrow 2 \cdot Q_1$
5: **end if**

i	init	4	3	2	1	0
k_i	1	0	1	1	0	1
Q_0	P	2P	5P			
Q_1	2P	3P	6P			

Comment cet algorithme fonctionne-t-il ?

Exemple : $k = 45_{10} = 0x2D = [1,0,1,1,0,1]_2$

On initialise : $Q_0 = P; Q_1 = 2P$.

1: **if** $(k_i = 0)$ **then**
2: $\quad Q_1 \leftarrow Q_0 + Q_1, Q_0 \leftarrow 2 \cdot Q_0$
3: **else**
4: $\quad Q_0 \leftarrow Q_0 + Q_1, Q_1 \leftarrow 2 \cdot Q_1$
5: **end if**

$\Longleftarrow$

Double-and-add :

$Q_0 \leftarrow Q_0 + Q_1 = 2 \cdot Q_0 + P$

($Q_1 \leftarrow 2 \cdot (Q_0 + P)$
d'où $Q_1 - Q_0 = P$)

i	init	4	3	2	1	0
k_i	1	0	1	1	0	1
Q_0	P	2P	5P	11P		
Q_1	2P	3P	6P			

Comment cet algorithme fonctionne-t-il ?

Exemple : $k = 45_{10} = 0x2D = [1,0,1,1,0,1]_2$

On initialise : $Q_0 = P; Q_1 = 2P$.

1: **if** $(k_i = 0)$ **then**
2: $\quad Q_1 \leftarrow Q_0 + Q_1, Q_0 \leftarrow 2 \cdot Q_0$
3: **else**
4: $\quad Q_0 \leftarrow Q_0 + Q_1, Q_1 \leftarrow 2 \cdot Q_1$
5: **end if**

i	init	4	3	2	1	0
k_i	1	0	1	1	0	1
Q_0	P	2P	5P	11P		
Q_1	2P	3P	6P	12P		

Comment cet algorithme fonctionne-t-il ?

Exemple : $k = 45_{10} = 0x2D = [1,0,1,1,0,1]_2$

On initialise : $Q_0 = P; Q_1 = 2P$.

1: **if** $(k_i = 0)$ **then**
2: $\quad Q_1 \leftarrow Q_0 + Q_1, Q_0 \leftarrow 2 \cdot Q_0$
3: **else**
4: $\quad Q_0 \leftarrow Q_0 + Q_1, Q_1 \leftarrow 2 \cdot Q_1$
5: **end if**

i	init	4	3	2	1	0
k_i	1	0	1	1	0	1
Q_0	P	2P	5P	11P		
Q_1	2P	3P	6P	12P	23P	

Comment cet algorithme fonctionne-t-il ?

Exemple : $k = 45_{10} = 0x2D = [1,0,1,1,0,1]_2$

On initialise : $Q_0 = P; Q_1 = 2P$.

1: **if** $(k_i = 0)$ **then**
2: $\quad Q_1 \leftarrow Q_0 + Q_1, Q_0 \leftarrow 2 \cdot Q_0$
3: **else**
4: $\quad Q_0 \leftarrow Q_0 + Q_1, Q_1 \leftarrow 2 \cdot Q_1$
5: **end if**

$\Longleftarrow$

Double :

$Q_0 \leftarrow 2 \cdot Q_0$

($Q_1 \leftarrow Q_0 + Q_1 = 2 \cdot Q_0 + P$
d'où $Q_1 - Q_0 = P$)

i	init	4	3	2	1	0
k_i	1	0	1	1	0	1
Q_0	P	2P	5P	11P	22P	
Q_1	2P	3P	6P	12P	23P	

Comment cet algorithme fonctionne-t-il ?

Exemple : $k = 45_{10} = 0x2D = [1,0,1,1,0,1]_2$

On initialise : $Q_0 = P; Q_1 = 2P$.

1: **if** $(k_i = 0)$ **then**
2: $\quad Q_1 \leftarrow Q_0 + Q_1, Q_0 \leftarrow 2 \cdot Q_0$
3: **else**
4: $\quad Q_0 \leftarrow Q_0 + Q_1, Q_1 \leftarrow 2 \cdot Q_1$
5: **end if**

$\Longleftarrow$

Double-and-add :

$Q_0 \leftarrow Q_0 + Q_1 = 2 \cdot Q_0 + P$

$(Q_1 \leftarrow 2 \cdot (Q_0 + P)$
d'où $Q_1 - Q_0 = P)$

i	init	4	3	2	1	0
k_i	1	0	1	1	0	1
Q_0	P	2P	5P	11P	22P	45P
Q_1	2P	3P	6P	12P	23P	

Comment cet algorithme fonctionne-t-il ?

Exemple : $k = 45_{10} = 0x2D = [1,0,1,1,0,1]_2$

On initialise : $Q_0 = P; Q_1 = 2P$.

1: **if** $(k_i = 0)$ **then**
2: $\quad Q_1 \leftarrow Q_0 + Q_1, Q_0 \leftarrow 2 \cdot Q_0$
3: **else**
4: $\quad Q_0 \leftarrow Q_0 + Q_1, Q_1 \leftarrow 2 \cdot Q_1$
5: **end if**

i	init	4	3	2	1	0
k_i	1	0	1	1	0	1
Q_0	P	2P	5P	11P	22P	45P
Q_1	2P	3P	6P	12P	23P	46P

Comment cet algorithme fonctionne-t-il ?

Exemple : $k = 45_{10} = 0x2D = [1, 0, 1, 1, 0, 1]_2$

On initialise : $Q_0 = P; Q_1 = 2P$.

1: **if** $(k_i = 0)$ **then**
2: $\quad Q_1 \leftarrow Q_0 + Q_1, Q_0 \leftarrow 2 \cdot Q_0$
3: **else**
4: $\quad Q_0 \leftarrow Q_0 + Q_1, Q_1 \leftarrow 2 \cdot Q_1$
5: **end if**

i	init	4	3	2	1	0
k_i	1	0	1	1	0	1
Q_0	P	2P	5P	11P	22P	45P
Q_1	2P	3P	6P	12P	23P	46P

Valeur retournée par l'algorithme :

$$\rightarrow Q_0 = 45P$$

Résultats de performances

Résultats des implémentations, multiplication scalaire de point de courbe elliptique.

Comparaison		
	Algorithmes Montgomery's Binary Ladder	WNAF Halve-and-add
B233	157307	135000
B409	734256	539000

Nombre de cycles (minimum sur 2000 exécutions, implémentations en C, Optiplex 990, Intel core i7, TurboMode et hyperthreading désactivés)

Table des matières

Construction avec *Halving*

Comme pour l'algorithme de Montgomery classique (avec *doublings*), on recherche :

Construction avec *Halving*

Comme pour l'algorithme de Montgomery classique (avec *doublings*), on recherche :

- la régularité des opérations ;

Construction avec *Halving*

Comme pour l'algorithme de Montgomery classique (avec *doublings*), on recherche :

- la régularité des opérations ;
- la différence constante entre les deux points Q_0 et Q_1 calculés au long de la multiplication scalaire.

Construction avec *Halving*

Comme pour l'algorithme de Montgomery classique (avec *doublings*), on recherche :

- la régularité des opérations ;
- la différence constante entre les deux points Q_0 et Q_1 calculés au long de la multiplication scalaire.

Pour un algorithme *Montgomery-Halving*, on est contraint à un recodage du scalaire k comme suit :

$$k \cdot P = (k' 2^{-m} \mod \#<P>) \cdot P.$$

Algorithme Montgomery *Halving*

Montgomery *Halving*

Require: $k = \sum_{i=0}^{t-1} k_i \cdot 2^i = \sum_{i=0}^{m} \frac{k'_i}{2^i} \bmod n, k'_i \in \{0,1\} =$ $(k'_{t-1}, \ldots, k'_1, k'_0)$ with $k'_{t-1} = 1, P \in E(\mathbb{F}_q)$
Ensure: $Q = k \cdot P$
1: $Q_0 \leftarrow P, Q_1 \leftarrow -P$
2: **for** i from $t-2$ downto 0 **do**
3: **if** $(k'_i = 0)$ **then**
4: $T \leftarrow Q_0/2, Q_0 \leftarrow T, Q_1 \leftarrow Q_1 - T$
5: **else**
6: $T \leftarrow Q_1/2, Q_1 \leftarrow T, Q_0 \leftarrow Q_0 - T$
7: **end if**
8: **end for**
9: **return** (Q_0)

Les deux critères recherchés sont maintenant bien remplis :

- la régularité recherchée est bien obtenue, de $1H + 1A$ à chaque itération ;
- maintenant, on a $Q_{1,j} - Q_{0,j} = -2 \cdot P$ à chaque itération ;
- enfin, l'efficacité reste à vérifier.

Algorithme parallèle

Le scalaire k est recoded en $k' = k \cdot 2^{\ell-1} \mod \#<P>$, et on a dans ce cas :

$$k = k'2^{-(\ell-1)} = (\sum_{i=0}^{m-1} k'_i 2^i)2^{-(\ell-1)} = \underbrace{\sum_{i=0}^{\ell-1} k'_i 2^{i-(\ell-1)}}_{\leq 0 \text{ powers of } 2} + \underbrace{\sum_{i=\ell}^{m-1} k'_i 2^{i-(\ell-1)}}_{>0 \text{ powers of } 2}$$

Algorithme parallèle

Le scalaire k est recoded en $k' = k \cdot 2^{\ell-1} \mod \#<P>$, et on a dans ce cas :

$$k = k'2^{-(\ell-1)} = (\sum_{i=0}^{m-1} k'_i 2^i)2^{-(\ell-1)} = \underbrace{\sum_{i=0}^{\ell-1} k'_i 2^{i-(\ell-1)}}_{\leq 0 \text{ powers of } 2} + \underbrace{\sum_{i=\ell}^{m-1} k'_i 2^{i-(\ell-1)}}_{>0 \text{ powers of } 2}$$

Parallel Montgomery *Halving/Doubling*

Require: scalar k, $P \in \mathbb{F}_{2^m}$ of odd order r, constant ℓ.
Ensure: kP.

1: Recode: $k' = 2^{\ell-1}k \mod r$. (Barrier)

2: Compute
Montgomery Doubling ECSM
$Q_D \leftarrow [(k'_t, \ldots, k'_\ell)] \cdot P$

3: Compute
Montgomery Halving ECSM
$Q_H \leftarrow [(k'_{\ell-1}, \ldots, k'_0)/2^\ell] \cdot P$

(Barrier)

4: **return** $Q \leftarrow Q_D + Q_H$

Table des matières

Synthèse sur ces algorithmes

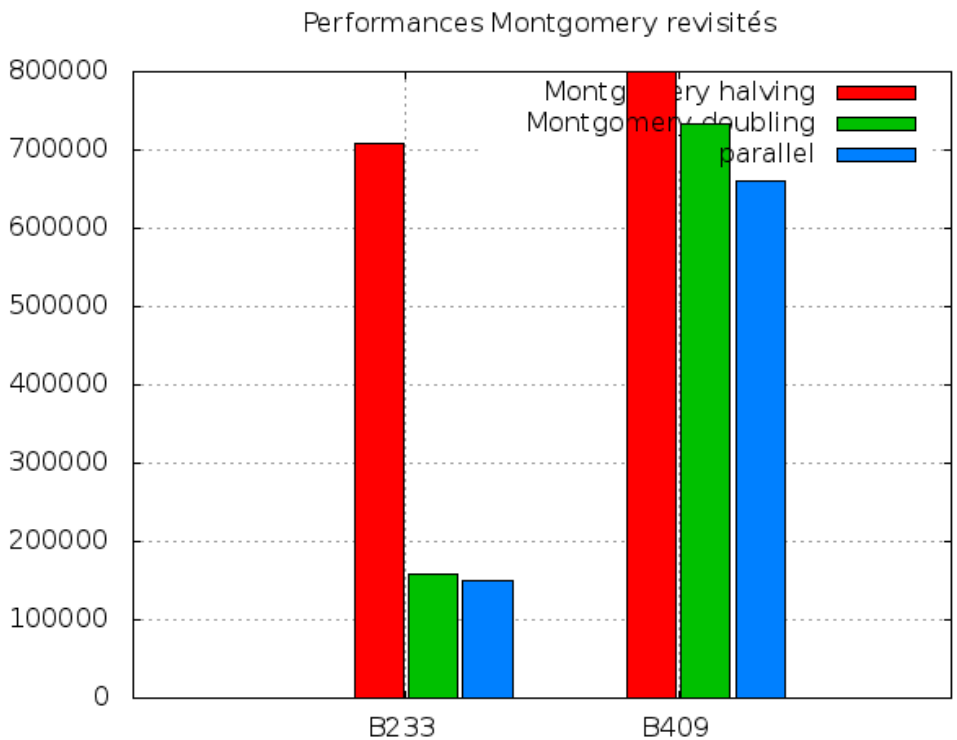
Algorithmes Montgomery's Binary Ladder				
B233	Doubling	Halving	Parallel	Gain (//-Doubling)
	157307	707385	149117	5,09 %
B409	Doubling	Halving	Parallel	Gain (//-Doubling)
	734256	4212043	659460	10.5%

Nombre de cycles (minimum sur 2000 exécutions, implémentations en C, Optiplex 990, Intel core i7, TurboMode et hyperthreading désactivés)

Synthèse sur ces algorithmes

Algorithmes Montgomery's Binary Ladder				
B233	Doubling	Halving	Parallel	Gain (//-Doubling)
	157307	707385	149117	5,09 %
B409	Doubling	Halving	Parallel	Gain (//-Doubling)
	734256	4212043	659460	10.5%

Nombre de cycles (minimum sur 2000 exécutions, implémentations en C, Optiplex 990, Intel core i7, TurboMode et hyperthreading désactivés)



Conclusion et Travaux en cours

Nous avons passé en revue les deux principales approches sur lesquelles nous travaillons, et conclu sur des résultats d'implémentation.
Travaux en cours :

Conclusion :

Conclusion et Travaux en cours

Nous avons passé en revue les deux principales approches sur lesquelles nous travaillons, et conclu sur des résultats d'implémentation.
Travaux en cours :

- débuts d'implémentation hardware (VHDL) en collaboration avec l'équipe CAIRN à l'IRISA (Lannion) ;

Conclusion :

Conclusion et Travaux en cours

Nous avons passé en revue les deux principales approches sur lesquelles nous travaillons, et conclu sur des résultats d'implémentation.
Travaux en cours :

- débuts d'implémentation hardware (VHDL) en collaboration avec l'équipe CAIRN à l'IRISA (Lannion) ;
- implémentation sur $\mathbb{F}_p$;

Conclusion :

Conclusion et Travaux en cours

Nous avons passé en revue les deux principales approches sur lesquelles nous travaillons, et conclu sur des résultats d'implémentation.
Travaux en cours :

- débuts d'implémentation hardware (VHDL) en collaboration avec l'équipe CAIRN à l'IRISA (Lannion) ;
- implémentation sur $\mathbb{F}_p$;
- gestion des *threads* ;

Conclusion :

Conclusion et Travaux en cours

Nous avons passé en revue les deux principales approches sur lesquelles nous travaillons, et conclu sur des résultats d'implémentation.
Travaux en cours :

- débuts d'implémentation hardware (VHDL) en collaboration avec l'équipe CAIRN à l'IRISA (Lannion) ;
- implémentation sur $\mathbb{F}_p$;
- gestion des *threads* ;

Conclusion :

- les performances de tous ces algorithmes restent perfectibles ;

Conclusion et Travaux en cours

Nous avons passé en revue les deux principales approches sur lesquelles nous travaillons, et conclu sur des résultats d'implémentation.
Travaux en cours :

- débuts d'implémentation hardware (VHDL) en collaboration avec l'équipe CAIRN à l'IRISA (Lannion) ;
- implémentation sur $\mathbb{F}_p$;
- gestion des *threads* ;

Conclusion :

- les performances de tous ces algorithmes restent perfectibles ;
- l'évaluation de la robustesse aux attaques reste théorique ;

Conclusion et Travaux en cours

Nous avons passé en revue les deux principales approches sur lesquelles nous travaillons, et conclu sur des résultats d'implémentation.
Travaux en cours :

- débuts d'implémentation hardware (VHDL) en collaboration avec l'équipe CAIRN à l'IRISA (Lannion) ;
- implémentation sur $\mathbb{F}_p$;
- gestion des *threads* ;

Conclusion :

- les performances de tous ces algorithmes restent perfectibles ;
- l'évaluation de la robustesse aux attaques reste théorique ;
- on n'en est qu'au début !

Je vous remercie de votre attention,

et suis à l'écoute de vos questions ?