# Highly Regular Right-to-Left Algorithms for Scalar Multiplication

Marc Joye

Thomson Security Labs
marc.joye@thomson.net

CHES 2007, Vienna, Sept. 10–13

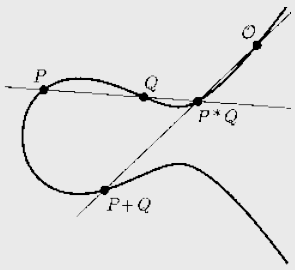

THOMSON
images & beyond

## Motivation (1/2)

- Exponentiation is central in public-key cryptography:
  - RSA cryptosystem

$\simeq (\mathbb{Z}/N\mathbb{Z})^*$

  - elliptic curve based cryptosystems




  - etc. . .

THOMSON
images & beyond

# Motivation (2/2)

- We are looking for right-to-left exponentiation algorithms such that:
  1. the algorithms are regular
  2. no dummy operations is involved
  3. the algorithms are generic
  4. a small number of temp. variables / code memory is required

THOMSON images & beyond

# Outline



THOMSON images & beyond

# Square-and-Multiply Algorithm

- Square-and-multiply algorithm

---

Input: $x, d = (d_{t-1}, \ldots, d_0)_2, N$
Output: $y = x^d \bmod N$

---

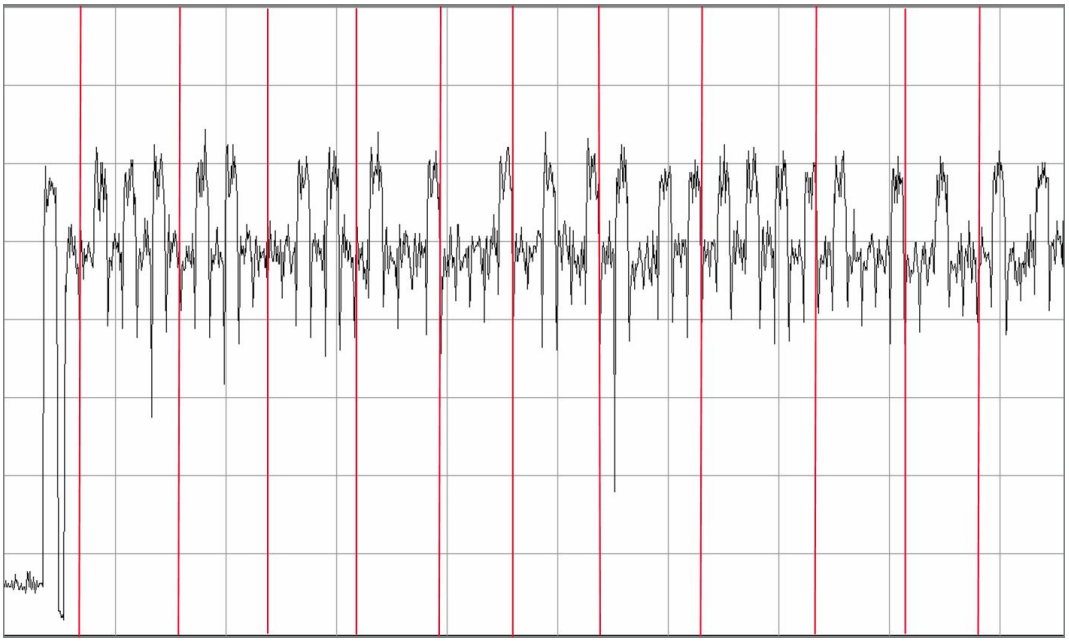
1. $R_0 \leftarrow 1$; $R_1 \leftarrow x$
2. For $i = t - 1$ downto 0 do
   - $R_0 \leftarrow {R_0}^2 \pmod N$
   - If $(d_i = 1)$ then $R_0 \leftarrow R_0 R_1 \pmod N$
3. Return $R_0$

---

  - left-to-right algorithm
  - 2 temporary variables ($R_0$, $R_1$)
- . . . subject to SPA-type attacks

THOMSON images & beyond

# SPA-type Attacks



Key: $d = 2\mathtt{E}\ \mathtt{C6}\ 91\ 5\mathtt{B}\ \mathtt{FE}\ 4\mathtt{A} \ldots$

THOMSON images & beyond

# Square-and-Multiply-Always Algorithm

- Square-and-multiply-*always* algorithm

---

Input: $x, d = (d_{t-1}, \ldots, d_0)_2, N$
Output: $y = x^d \bmod N$

---

1. $R_0 \leftarrow 1$; $R_1 \leftarrow 1$; $R_2 \leftarrow x$
2. For $i = t - 1$ downto 0 do
   - $R_0 \leftarrow {R_0}^2 \pmod N$
   - $b \leftarrow 1 - d_i$; $R_b \leftarrow R_b R_2 \pmod N$
3. Return $R_0$

---

  - when $b = 1$ (i.e., $d_i = 0$), there is a dummy multiplication
  - the power trace now appears as a regular succession of squares and multiplies
  - 3 temporary variables ($R_0$, $R_1$, $R_2$)
- . . . subject to safe-error attacks



# Safe-Error Attacks

- Timely induce a fault into the ALU during the multiply operation at iteration $i$
- Check the output
  - if the result is incorrect (invalid signature or error notification) then the multiplication was effective
    $\Rightarrow d_i = 1$
  - if the result is correct then the multiplication was dummy [safe error]
    $\Rightarrow d_i = 0$
- Re-iterate the attack for another value of $i$

**Lesson**

Protection against certain implementation attacks (e.g., SPA) may introduce new vulnerabilities

# Montgomery Powering Ladder

- Montgomery exponentiation algorithm

---

Input: $x, d = (d_{t-1}, \ldots, d_0)_2, \boldsymbol{NP}, d = (d_{t-1}, \ldots, d_0)_2$
Output: $y = x^d \bmod \boldsymbol{NQ} = d\boldsymbol{P}$

---

1. $R_0 \leftarrow 1$; $R_1 \leftarrow x\boldsymbol{R_0} \leftarrow \boldsymbol{O}$; $\boldsymbol{R_1} \leftarrow \boldsymbol{P}$
2. For $i = t - 1$ downto 0 do
   - $b \leftarrow 1 - d_i$; $R_b \leftarrow R_b\, R_{d_i} \pmod N \boldsymbol{R_b} \leftarrow \boldsymbol{R_b} + \boldsymbol{R_{d_i}}$
   - $R_{d_i} \leftarrow {R_{d_i}}^2 \pmod N \boldsymbol{R_{d_i}} \leftarrow 2\boldsymbol{R_{d_i}}$
3. Return $R_0\boldsymbol{R_0}$

---

  - left-to-right algorithm
  - behaves regularly without dummy operations
  - only 2 temporary variables $(R_0, R_1)(\boldsymbol{R_0}, \boldsymbol{R_1})$
- . . . subject to doubling attacks

THOMSON images & beyond

# Notation

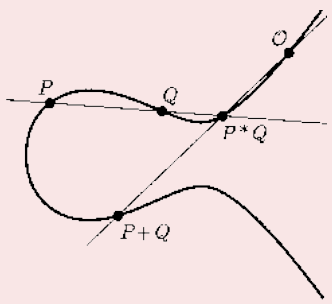
## Multiplicative notation

- $(\mathbb{Z}/N\mathbb{Z})^*$
- $x \cdot y \pmod N$
- $y = x^d \pmod N$

[exponentiation]

## Additive notation

- $E_{\mathbb{F}_p} : y^2 = x^3 + ax + b$
- $\boldsymbol{P} + \boldsymbol{Q}$
- $\boldsymbol{Q} = d\boldsymbol{P}$




[scalar multiplication]

THOMSON images & beyond

# The Trick

- Let $d = \sum_{i=0}^{t-1} d_i 2^i$ with $d_i \in \{0, 1\}$

$$\Longrightarrow \quad \boldsymbol{Q} := d\boldsymbol{P} = \sum_{i=0}^{t-1} (d_i 2^i)\boldsymbol{P} = \sum_{i=0}^{t-1} d_i \boldsymbol{B_i} \quad \text{with } \boldsymbol{B_i} = 2^i \boldsymbol{P}$$

- For $i \geqslant 0$, define

$$\boldsymbol{S_i} = \sum_{j=0}^{i} d_j \boldsymbol{B_j} \quad \text{and} \quad \boldsymbol{T_i} = \boldsymbol{B_{i+1}} - \boldsymbol{S_i}$$

- We get

$$\begin{aligned} \boldsymbol{S_i} &= \textstyle\sum_{j=0}^{i} d_j \boldsymbol{B_j} = d_i \boldsymbol{B_i} + \boldsymbol{S_{i-1}} = d_i(\boldsymbol{S_{i-1}} + \boldsymbol{T_{i-1}}) + \boldsymbol{S_{i-1}} \\ &= (1 + d_i)\boldsymbol{S_{i-1}} + d_i \boldsymbol{T_{i-1}} \\ \boldsymbol{T_i} &= \boldsymbol{B_{i+1}} - \boldsymbol{S_i} = 2\boldsymbol{B_i} - (d_i \boldsymbol{B_i} + \boldsymbol{S_{i-1}}) = (2 - d_i)\boldsymbol{B_i} - \boldsymbol{S_{i-1}} \\ &= (2 - d_i)\boldsymbol{T_{i-1}} + (1 - d_i)\boldsymbol{S_{i-1}} \end{aligned}$$

| | $d_i = 0$ | $d_i = 1$ |
|---|---|---|
| $\boldsymbol{R_0}$ | $\boldsymbol{S_i} = \boldsymbol{S_{i-1}}$ | $\boldsymbol{S_i} = 2\boldsymbol{S_{i-1}} + \boldsymbol{T_{i-1}}$ |
| $\boldsymbol{R_1}$ | $\boldsymbol{T_i} = 2\boldsymbol{T_{i-1}} + \boldsymbol{S_{i-1}}$ | $\boldsymbol{T_i} = \boldsymbol{T_{i-1}}$ |



# Double-Add Algorithm

- Double-add algorithm

Input: $\boldsymbol{P}, d = (d_{t-1}, \ldots, d_0)_2$
Output: $\boldsymbol{Q} = d\boldsymbol{P}$

1. $\boldsymbol{R_0} \leftarrow \boldsymbol{O}$; $\boldsymbol{R_1} \leftarrow \boldsymbol{P}$
2. For $i = 0$ to $t - 1$ do
   - $b \leftarrow 1 - d_i$; $\boldsymbol{R_b} \leftarrow 2\boldsymbol{R_b} + \boldsymbol{R_{d_i}}$
3. Return $\boldsymbol{R_0}$

- right-to-left algorithm
- behaves regularly without dummy operations
- only 2 temporary variables ($\boldsymbol{R_0}$, $\boldsymbol{R_1}$)

# Application

- Optimal extension fields (OEFs) [Bailey and Paar, 1998]
  - $\mathbb{F}_{p^m} \simeq \mathbb{F}_p[t]/(t^m - \omega)$
  - $p$ a pseudo-Mersenne prime of the form $2^n \pm c < \Omega$
  - offer the best performance for ECC even with affine coordinates
    - typically $I/M = 4$
- Cost of point operations (over fields of char. $\neq 2, 3$)
  - $\boldsymbol{P} + \boldsymbol{Q}$ : $1I + 1S + 2M$
  - $2\boldsymbol{P}$ : $1I + 2S + 2M$
  - $2\boldsymbol{P} + \boldsymbol{Q}$ : $2I + 2S + 3M$ or $1I + 2S + 9M$

**Performance**

Taking $S/M = 1$ and $I/M = 4$, we get a 13.3% speed-up improvement over OEFs

THOMSON images & beyond

# Add-Only Algorithm

- Add-only algorithm

Input: $\boldsymbol{P}, d = (d_{t-1}, \ldots, 1)_2$
Output: $\boldsymbol{Q} = d\boldsymbol{P}$

1. $\boldsymbol{R_0} \leftarrow \boldsymbol{O}\boldsymbol{R_0} \leftarrow \boldsymbol{P}$; $\boldsymbol{R_1} \leftarrow \boldsymbol{P}$; $\boldsymbol{R_2} \leftarrow \boldsymbol{P}$; $\boldsymbol{R_2} \leftarrow 2\boldsymbol{P}$
2. For $i = 0 i = 1$ to $t - 1$ do
   - $\boldsymbol{R_2} \leftarrow \boldsymbol{R_0} + \boldsymbol{R_1}$
   - $b \leftarrow 1 - d_i$;
     $\boldsymbol{R_b} \leftarrow 2\boldsymbol{R_b} + \boldsymbol{R_{d_i}}\boldsymbol{R_b} \leftarrow \boldsymbol{R_b} + (\boldsymbol{R_b} + \boldsymbol{R_{d_i}})\boldsymbol{R_b} \leftarrow \boldsymbol{R_b} + \boldsymbol{R_2}$
   - $b \leftarrow 1 - d_i$; $\boldsymbol{R_b} \leftarrow \boldsymbol{R_b} + \boldsymbol{R_2}$
   - $\boldsymbol{R_2} \leftarrow \boldsymbol{R_0} + \boldsymbol{R_1}$
3. $b \leftarrow d_0$; $\boldsymbol{R_b} \leftarrow \boldsymbol{R_b} - \boldsymbol{P}$
4. Return $\boldsymbol{R_0}$

  - right-to-left algorithm
  - behaves regularly without dummy operations
  - only involves additions (i.e., no doublings)

*Remark*: $2\boldsymbol{P}$ given as input $-$or$-$ $2\boldsymbol{P} = (\boldsymbol{P} + \boldsymbol{A}) + (\boldsymbol{P} - \boldsymbol{A})$

THOMSON images & beyond

# Application

- Only requires the *general* addition formula for adding points
  - no need to implement a routine for doubling points
  - results in code savings
- Fermat parameterization [Cohen, Exerc. 10.6.4, 1993]
  - point addition : $9M$
  - point doubling : $10M$

  (projective coord.)

**Performance**

Faster point addition leads to a 5% speed-up improvement

THOMSON images & beyond

# Add-Always Algorithm

- Add-always algorithm

Input: $\boldsymbol{P}, d = (d_{t-1}, \ldots, d_0)_2$
Output: $\boldsymbol{Q} = d\boldsymbol{P}$

1. $\boldsymbol{R_0} \leftarrow \boldsymbol{O}$; $\boldsymbol{R_1} \leftarrow \boldsymbol{P}$
2. For $i = 0$ to $t - 1$ do
   - $b \leftarrow 1 - d_i$; $\boldsymbol{R_b} \leftarrow 2\boldsymbol{R_b} + \boldsymbol{R_{d_i}}$
   - $\boldsymbol{R_b} \leftarrow \boldsymbol{R_b} + \boldsymbol{R_{d_i}}$
3. Return $\boldsymbol{R_0}$

  - right-to-left algorithm
  - behaves regularly without dummy operations
  - only 2 temporary variables ($\boldsymbol{R_0}$, $\boldsymbol{R_1}$)

Right-to-left analogue of Montgomery ladder

THOMSON images & beyond

# Application

- Montgomery ladder over $\mathbb{F}_{2^m}$ [López and Dahab, 1999]
  - evaluation of $\boldsymbol{Q} = d\boldsymbol{P}$ using $x$-coordinate only
  - non-supersingular elliptic curves over binary fields
  - only 6 multiplications per bit of $d$ are required
- Right-to-left analogue
  - evaluation of $\boldsymbol{Q} = d\boldsymbol{P}$ using $x$-coordinate only
  - Stam's parameterization
    $$y^2 + a_1\, xy = x^3 + a_2 x^2 + {a_1}^{-1}$$
  - 6 multiplications *plus* 1 multiplication by constant $a_1$ are required

**Open problem**

Find a right-to-left algorithm for evaluating $\boldsymbol{Q} = d\boldsymbol{P}$ that only requires (at most) 6 multiplications per bit of $d$

THOMSON images & beyond

# Summary

- Two types of scalar multiplication (exponentiation) algorithm
  - left-to-right –or– right-to-left
  - left-to-right algorithms are generally preferred because they require less memory
  - . . . but right-to-left algorithms offer better security guarantees

**This paper:**

- New right-to-left algorithms that have similar requirements
- And even, in some cases, lead to better performance
  - OEFs, code savings, Fermat parameterization, . . .

THOMSON images & beyond