Square Always Exponentiation

Christophe Clavier[1] Benoit Feix[1,2] Georges Gagnerot[1,2]
Mylène Roussellet[2] **Vincent Verneuil**[2,3]

[1]XLIM-Université de Limoges, France

[2]INSIDE Secure, Aix-en-Provence, France

[3]Univ. Bordeaux, IMB, France

Indocrypt 2011 - December 12, 2011

Outline

Outline

Outline

Motivation

- Exponentiation is the core operation of RSA, DSA, Diffie-Hellman protocols.
- Embedded in constrained devices (smart cards, etc.) with low resources.
- Targeted by side-channel attacks in this sensitive context.

Context

Let consider the computation of m^d mod n with $d = (d_{k-1} d_{k-2} \dots d_0)_2$.

M the cost of a modular multiplication.
S the cost of a modular squaring.

Two cases : fast squaring ($S/M = .8$) or not ($S/M = 1$).

Outline

Basic Exponentiation

Square-and-Multiply Algorithms

Left-to-right

Right-to-left

Basic Exponentiation

Square-and-Multiply Algorithms

Left-to-right

$$m^d = m^{d_0} \times \left(m^{d_1} \times \left(\ldots \left(m^{d_{k-1}} \right)^2 \ldots \right)^2 \right)^2$$

Right-to-left

$$m^d = m^{d_{k-1} 2^{k-1}} \times m^{d_{k-2} 2^{k-2}} \times \ldots \times m^{d_0}$$

Basic Exponentiation

Square-and-Multiply Algorithms

Left-to-right

$$m^d = m^{d_0} \times \left(m^{d_1} \times \left(\ldots \left(m^{d_{k-1}} \right)^2 \ldots \right)^2 \right)^2$$

Input: $m, n, d \in \mathbb{N}$
Output: $m^d \bmod n$

$a \leftarrow 1$
for $i = k-1$ **to** 0 **do**
  $a \leftarrow a^2 \bmod n$
  if $d_i = 1$ **then**
    $a \leftarrow a \times m \bmod n$
return a

Right-to-left

$$m^d = m^{d_{k-1}2^{k-1}} \times m^{d_{k-2}2^{k-2}} \times \ldots \times m^{d_0}$$

Input: $m, n, d \in \mathbb{N}$
Output: $m^d \bmod n$

$a \leftarrow 1$; $b \leftarrow m$
for $i = 0$ **to** $k-1$ **do**
  if $d_i = 1$ **then**
    $a \leftarrow a \times b \bmod n$
  $b \leftarrow b^2 \bmod n$
return a

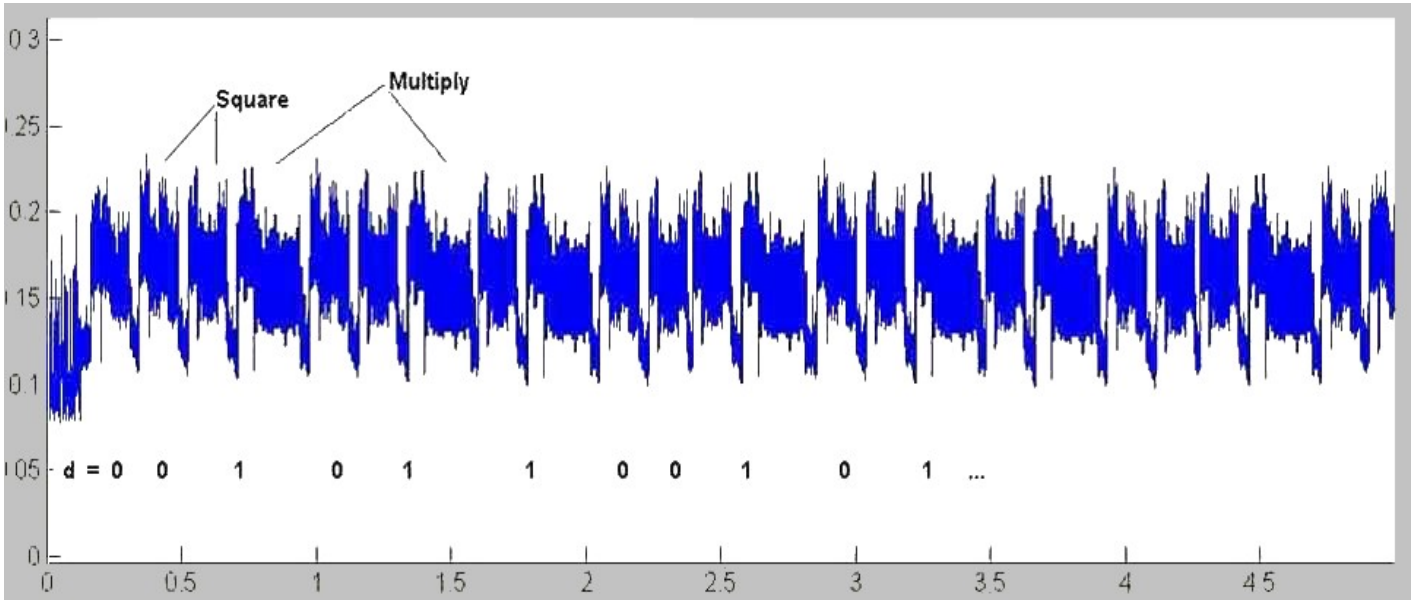
Side-Channel Threats

When a computation involving a secret occurs on an embedded devices, side-channels (power, EM) may be spotted to search for leakages.

Kocher introduced in 1999 the *simple* and *differential* side-channel analysis.

Simple Side-Channel Analysis on Exponentiation (SPA)

Side-channel leakage: power, EM, etc.



The whole exponent may be recovered using a single curve.

Regular Exponentiation

Montgomery ladder

Square & multiply: S, M, S, S, M, S, M, S, S, M. . .

Regular Exponentiation

Montgomery ladder

Square & multiply: S, M, S, S, M, S, M, S, S, M...
Square & multiply always: S, M, S, M, S, M, S, M, S, M, S, M...

Regular Exponentiation

Montgomery ladder

Square & multiply: S, M, S, S, M, S, M, S, S, M...
Square & multiply always: S, M, S, M, S, M, S, M, S, M, S, M...

Regular Exponentiation

Montgomery ladder

Square & multiply:	S, M, S, S, M, S, M, S, S, M. . .
Square & multiply always:	S, M, S, M, S, M, S, M, S, M, S, M. . .
Montgomery ladder:	S, M, S, M, S, M, S, M, S, M, S, M. . .

Regular Exponentiation

Montgomery ladder

Square & multiply: S, M, S, S, M, S, M, S, S, M...
Square & multiply always: S, M, S, M, S, M, S, M, S, M, S, M...
Montgomery ladder: S, M, S, M, S, M, S, M, S, M, S, M...

Input: $m, n, d \in \mathbb{N}$
Output: $m^d \bmod n$

1: $R_0 \leftarrow 1$
2: $R_1 \leftarrow m$
3: **for** $i = k - 1$ **to** 0 **do**
4: $\quad R_{1-d_i} \leftarrow R_0 \times R_1 \bmod n$
5: $\quad R_{d_i} \leftarrow {R_{d_i}}^2 \bmod n$
6: **return** R_0

Regular Exponentiation

Atomic Exponentiation “Multiply Always”

Square & multiply: S, M, S, S, M, S, M, S, S, M. . .

Regular Exponentiation

Atomic Exponentiation “Multiply Always”

Square & multiply: S, M, S, S, M, S, M, S, S, M...
Multiply always: M, M, M, M, M, M, M, M, M, M...

Regular Exponentiation

Atomic Exponentiation "Multiply Always"

Square & multiply: S, M, S, S, M, S, M, S, S, M...
Multiply always: M, M, M, M, M, M, M, M, M, M...

Input: $m, n, d \in \mathbb{N}$
Output: $m^d \bmod n$

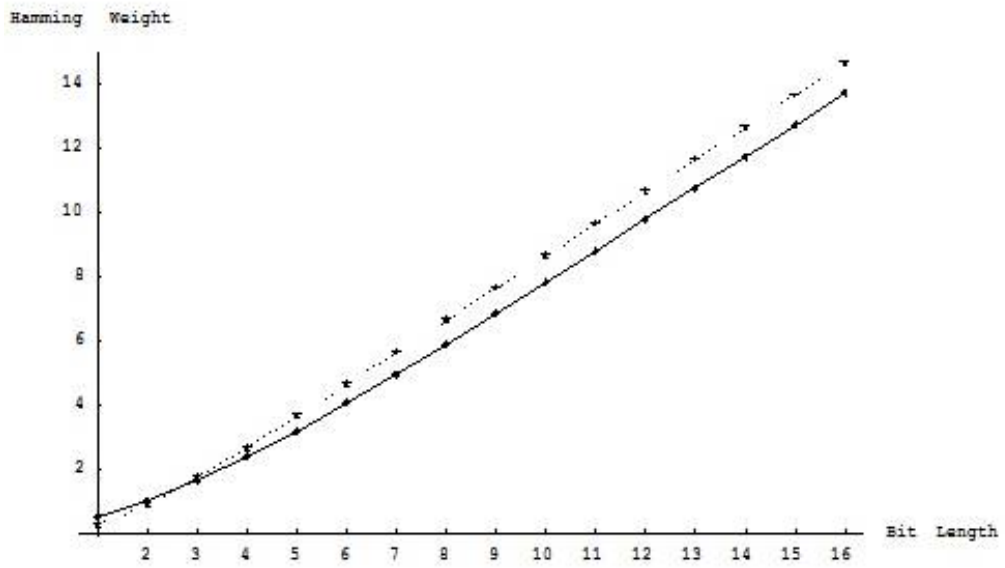
1: $R_0 \leftarrow 1$
2: $R_1 \leftarrow m$
3: $i \leftarrow k - 1$
4: $t \leftarrow 0$
5: **while** $i \geq 0$ **do**
6: $\quad R_0 \leftarrow R_0 \times R_t \bmod n$
7: $\quad t \leftarrow t \oplus d_i$
8: $\quad i \leftarrow i - 1 + t$
9: **return** R_0

[$\oplus$ is bitwise XOR]

Squaring-Multiplication Discrimination Attack

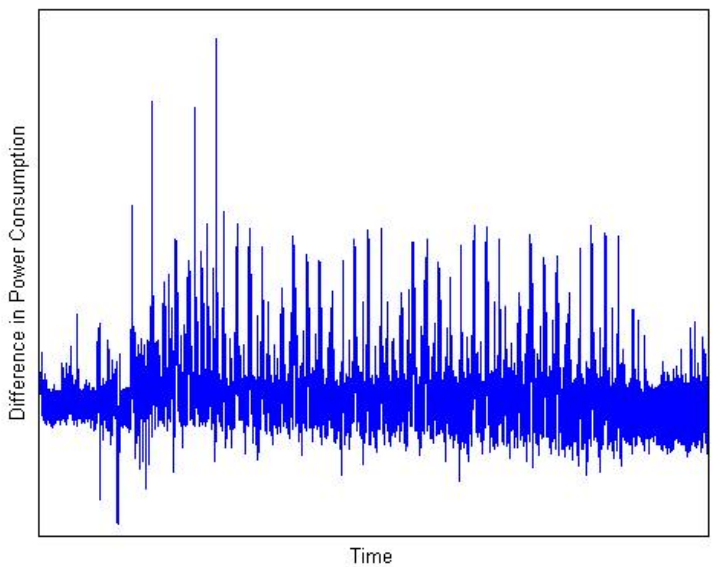
In [*Distinguishing Multiplications from Squaring Operations*, SAC 2008], Amiel et al. observed that $E_{x,y}(\mathrm{HW}(x \times y))$ has a different value whether:

- $x = y$ uniformly distributed in $[0, 2^k - 1]$,
- x and y independent and uniformly distributed in $[0, 2^k - 1]$.



Squaring-Multiplication Discrimination Attack

Attack: subtract two (averaged) power traces of consecutive atomic multiplications.



Countermeasure: exponent blinding $d^* \leftarrow d + r\psi(n)$.

Cost Summary

Algorithm	Cost / bit	$S/M=1$	$S/M=.8$	# reg
Square & multiply [1,2,3]	$.5M+1S$	$1.5M$	$1.3M$	2
Multiply always [2,3]	$1.5M$	$1.5M$	$1.5M$	2
Montgomery ladder	$1M+1S$	$2M$	$1.8M$	2

[1] algorithm unprotected towards the SPA
[2] algorithm sensitive to S−M discrimination
[3] possible sliding window optimization

Outline

Our Contribution

- Atomic exponentiation algorithms immune to the S−M discrimination
- Better efficiency than ladder algorithms
- Study of algorithms for parallelized (co)processors and space/time trade-offs

Outline

Outline

Replacing Multiplications by Squarings

$$x \times y = \frac{(x+y)^2 - x^2 - y^2}{2} \tag{1}$$

$$x \times y = \left(\frac{x+y}{2}\right)^2 - \left(\frac{x-y}{2}\right)^2 \tag{2}$$

Replacing Multiplications by Squarings

$$x \times y = \frac{(x+y)^2 - x^2 - y^2}{2} \tag{1}$$

$$x \times y = \left(\frac{x+y}{2}\right)^2 - \left(\frac{x-y}{2}\right)^2 \tag{2}$$

Outline

Left-to-Right Algorithm Using (1)

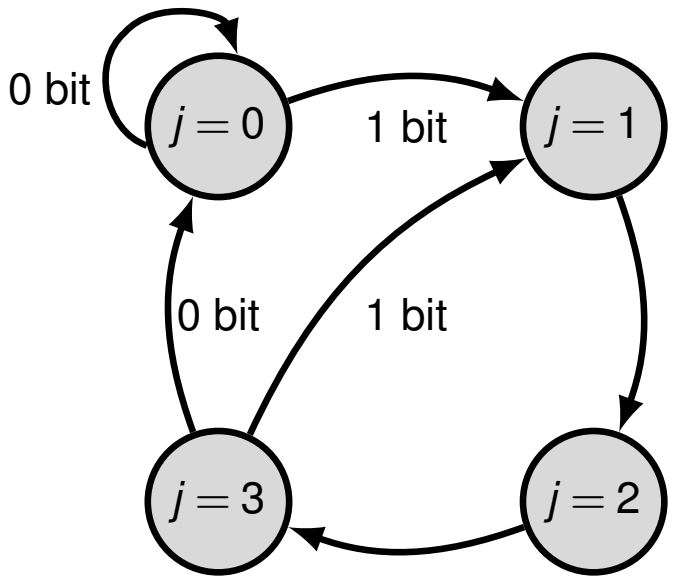
Input: $m, n, d \in \mathbb{N}$
Output: $m^d \bmod n$

$a \leftarrow 1$
for $i = k - 1$ **to** 0 **do**
  $a \leftarrow a^2 \bmod n$
  if $d_i = 1$ **then**
    $a \leftarrow \frac{(a+m)^2 - a^2}{2} - \frac{m^2}{2} \bmod n$
return a

Atomic Left-to-Right Algorithm

Input: $m, n, d \in \mathbb{N}$
Output: $m^d \bmod n$

1: $R_0 \leftarrow 1$; $R_1 \leftarrow m$; $R_2 \leftarrow 1$
2: $R_3 \leftarrow m^2/2 \bmod n$
3: $j \leftarrow 0$; $i \leftarrow k-1$
4: **while** $i \geq 0$ **do**
5: $\quad R_{M_{j,0}} \leftarrow R_{M_{j,1}} + R_{M_{j,2}} \bmod n$
6: $\quad R_{M_{j,3}} \leftarrow {R_{M_{j,3}}}^2 \bmod n$
7: $\quad R_{M_{j,4}} \leftarrow R_{M_{j,5}}/2 \bmod n$
8: $\quad R_{M_{j,6}} \leftarrow R_{M_{j,7}} - R_{M_{j,8}} \bmod n$
9: $\quad j \leftarrow d_i(1 + (j \bmod 3))$
10: $\quad i \leftarrow i - M_{j,9}$
11: **return** R_0



$$M = \begin{pmatrix} 1 & 1 & 1 & 0 & 2 & 1 & 1 & 1 & 2 & 1 \\ 2 & 0 & 1 & 2 & 2 & 2 & 2 & 2 & 3 & 0 \\ 1 & 1 & 3 & 0 & 0 & 0 & 0 & 2 & 0 & 0 \\ 3 & 3 & 3 & 0 & 3 & 3 & 1 & 1 & 3 & 1 \end{pmatrix}$$

Atomic Left-to-Right Algorithm

Atomic Patterns

$j = 0$ ($d_i = 0$ or 1)

$R_1 \leftarrow R_1 + R_1 \bmod n$	$\star$
$R_0 \leftarrow {R_0}^2 \bmod n$	
$R_2 \leftarrow R_1/2 \bmod n$	$\star$
$R_1 \leftarrow R_1 - R_2 \bmod n$	$\star$
$j \leftarrow d_i$	[$\star$ if $d_i = 0$]
$i \leftarrow i - (1 - d_i)$	[$\star$ if $d_i = 1$]

$j = 1$ ($d_i = 1$)

$R_2 \leftarrow R_0 + R_1 \bmod n$	
$R_2 \leftarrow {R_2}^2 \bmod n$	
$R_2 \leftarrow R_2/2 \bmod n$	
$R_2 \leftarrow R_2 - R_3 \bmod n$	
$j \leftarrow 2$	
$i \leftarrow i$	$\star$

$j = 2$ ($d_i = 1$)

$R_1 \leftarrow R_1 + R_3 \bmod n$	$\star$
$R_0 \leftarrow {R_0}^2 \bmod n$	
$R_0 \leftarrow R_0/2 \bmod n$	
$R_0 \leftarrow R_2 - R_0 \bmod n$	
$j \leftarrow 3$	
$i \leftarrow i - 1$	

$j = 3$ ($d_i = 0$ or 1)

$R_3 \leftarrow R_3 + R_3 \bmod n$	$\star$
$R_0 \leftarrow {R_0}^2 \bmod n$	
$R_3 \leftarrow R_3/2 \bmod n$	$\star$
$R_1 \leftarrow R_1 - R_3 \bmod n$	$\star$
$j \leftarrow d_i$	
$i \leftarrow i - (1 - d_i)$	[$\star$ if $d_i = 1$]

Right-to-Left Algorithm Using (2)

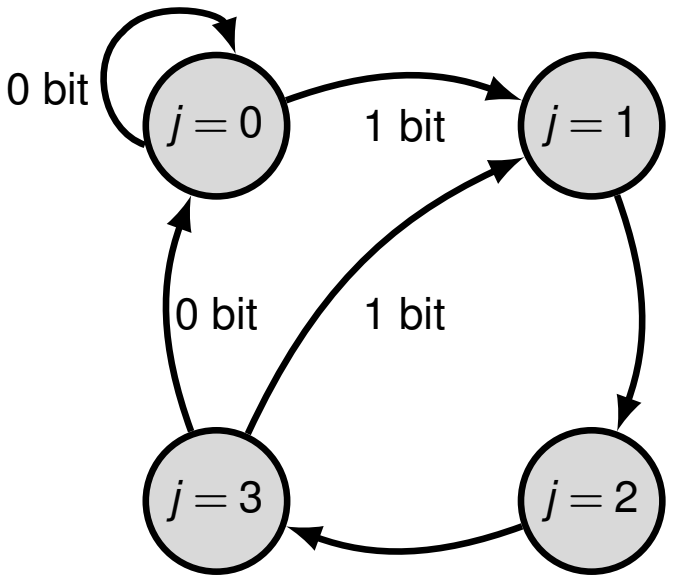
Input: $m, n, d \in \mathbb{N}$
Output: $m^d \bmod n$

$a \leftarrow 1$; $b \leftarrow m$
for $i = 0$ **to** $k - 1$ **do**
  if $d_i = 1$ **then**
    $a \leftarrow \left(\frac{a+b}{2}\right)^2 - \left(\frac{a-b}{2}\right)^2 \bmod n$
  $b \leftarrow b^2 \bmod n$
return a

Atomic Right-to-Left Algorithm

Input: $m, n, d \in \mathbb{N}$
Output: $m^d \bmod n$

1: $R_0 \leftarrow m$; $R_1 \leftarrow 1$; $R_2 \leftarrow 1$
2: $i \leftarrow 0$; $j \leftarrow 0$
3: **while** $i \leq k-1$ **do**
4: $\quad j \leftarrow d_i(1+(j \bmod 3))$
5: $\quad R_{M_{j,0}} \leftarrow R_{M_{j,1}} + R_0 \bmod n$
6: $\quad R_{M_{j,2}} \leftarrow R_{M_{j,3}}/2 \bmod n$
7: $\quad R_{M_{j,4}} \leftarrow R_{M_{j,5}} - R_{M_{j,6}} \bmod n$
8: $\quad R_{M_{j,3}} \leftarrow {R_{M_{j,3}}}^2 \bmod n$
9: $\quad i \leftarrow i + M_{j,7}$
10: **return** R_1



$$M = \begin{pmatrix} 0 & 0 & 2 & 0 & 0 & 0 & 2 & 1 \\ 2 & 1 & 2 & 2 & 1 & 0 & 1 & 0 \\ 0 & 2 & 1 & 1 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 1 & 2 & 1 & 1 \end{pmatrix}$$

Atomic Right-to-Left Algorithm

Atomic Patterns

$j = 0$ ($d_i = 0$)

$j \leftarrow 0$	[$\star$ if j was 0]
$R_0 \leftarrow R_0 + R_0 \bmod n$	$\star$
$R_2 \leftarrow R_0/2 \bmod n$	$\star$
$R_0 \leftarrow R_0 - R_2 \bmod n$	$\star$
$R_0 \leftarrow {R_0}^2 \bmod n$	
$i \leftarrow i + 1$	

$j = 1$ ($d_i = 1$)

$j \leftarrow 1$	
$R_2 \leftarrow R_1 + R_0 \bmod n$	
$R_2 \leftarrow R_2/2 \bmod n$	
$R_1 \leftarrow R_0 - R_1 \bmod n$	
$R_2 \leftarrow {R_2}^2 \bmod n$	
$i \leftarrow i$	$\star$

$j = 2$ ($d_i = 1$)

$j \leftarrow 2$	
$R_0 \leftarrow R_2 + R_0 \bmod n$	$\star$
$R_1 \leftarrow R_1/2 \bmod n$	
$R_0 \leftarrow R_0 - R_2 \bmod n$	$\star$
$R_1 \leftarrow {R_1}^2 \bmod n$	
$i \leftarrow i$	$\star$

$j = 3$ ($d_i = 1$)

$j \leftarrow 3$	
$R_0 \leftarrow R_0 + R_0 \bmod n$	$\star$
$R_0 \leftarrow R_0/2 \bmod n$	$\star$
$R_1 \leftarrow R_2 - R_1 \bmod n$	
$R_0 \leftarrow {R_0}^2 \bmod n$	
$i \leftarrow i + 1$	

Cost Comparison

Algorithm	Cost / bit	$S/M=1$	$S/M=.8$	# reg
Square & multiply [1,2,3]	$.5M+1S$	$1.5M$	$1.3M$	2
Multiply always [2,3]	$1.5M$	$1.5M$	$1.5M$	2
Montgomery ladder	$1M+1S$	$2M$	$1.8M$	2
L.-to-r. square always[3]	$2S$	$2M$	$\mathbf{1.6M}$	4
R.-to-l. square always[3]	$2S$	$2M$	$\mathbf{1.6M}$	3

$\rightarrow$ **11 % speed-up** over Montgomery ladder

[1] algorithm unprotected towards the SPA
[2] algorithm sensitive to S−M discrimination
[3] possible sliding window optimization

Implementation

AT90SC chip @ 30MHz with AdvX arithmetic coprocessor:

Algorithm	Key len. (b)	Code (B)	RAM (B)	Timing (ms)
Mont. ladder	512	360	128	30
	1024	360	256	200
	2048	360	512	1840
Square Always	512	510	192	28
	1024	510	384	190
	2048	510	768	1740

$\rightarrow$ **5 %** practical speed-up obtained in practice

Security Considerations

- Immune to any S−M discrimination
- Compatible with classical DPA/FA countermeasures
- Right-to-left is more robust than left-to-right
- Despite DPA countermeasures prevent safe-errors, we provide algorithms immune to C safe-errors

Outline

Outline

Motivation

- Trendy topic
- Parallelized Montgomery ladder : $1M$ / bit
- Squarings are independent in eq. (1) and (2)

What can we do if **two** parallel squarings are available ?

Basic Parallelization

	Core 1	Core 2
0 bit	S	S
1 bit	S	S
	S	S

Many wasted squaring slots :-(

Scanning Direction

Left-to-right:

$$m^d = m^{d_0} \times \left(m^{d_1} \times \left(\ldots \left(m^{d_{k-1}} \right)^2 \ldots \right)^2 \right)^2$$

Right-to-left:

$$m^d = m^{d_{k-1}2^{k-1}} \times m^{d_{k-2}2^{k-2}} \times \ldots \times m^{d_0}$$

$\rightarrow$ Right-to-left is more flexible

Better Parallelization

Strategy: use the wasted 1 bit squaring slot to compute 1 squaring in advance.

if$(d_i = 1)$
$a \leftarrow ((a+b)/2)^2 - ((a-b)/2)^2 \bmod n$
$b \leftarrow b^2 \bmod n$

Core 1	Core 2
S	S
S	S

Remark: requires 1 additional memory register to store the precomputed squaring.

Better Parallelization

Strategy: use the wasted 1 bit squaring slot to compute 1 squaring in advance.

if$(d_i = 1)$
$\quad a \leftarrow ((a+b)/2)^2 - ((a-b)/2)^2 \bmod n$
$b \leftarrow b^2 \bmod n$
if$(d_{i+1} = 1)$
$\quad a \leftarrow ((a+b)/2)^2 - ((a-b)/2)^2 \bmod n$
$b \leftarrow b^2 \bmod n$

Core 1	Core 2
S	S
S	S

Remark: requires 1 additional memory register to store the precomputed squaring.

Better Parallelization

Strategy: use the wasted 1 bit squaring slot to compute 1 squaring in advance.

if$(d_i = 1)$
$a \leftarrow ((a+b)/2)^2 - ((a-b)/2)^2 \bmod n$
$b \leftarrow b^2 \bmod n$
if$(d_{i+1} = 1)$
$a \leftarrow ((a+b)/2)^2 - ((a-b)/2)^2 \bmod n$
$b \leftarrow b^2 \bmod n$

Core 1	Core 2
S	S
S	S

Remark: requires 1 additional memory register to store the precomputed squaring.

Outline

Right-to-Left Parallelized Algorithm

Input: $m, n \in \mathbb{N}$, $m < n$, $d = (d_{k-1} \dots d_0)_2$, require 5 k-bit registers a, b, R_0, R_1, R_2
Output: $m^d \bmod n$

1: $a \leftarrow 1$; $b \leftarrow m$; $extra \leftarrow 0$
2: **for** $i = 0$ **to** $k-1$ **do**
3: **if** $d_i = 1$ **then**
4: **if** $extra = 0$ **then**
5: $R_0 \leftarrow (a-b)^2 \bmod n$ || $R_1 \leftarrow b^2 \bmod n$
6: $a \leftarrow (a+b)^2 \bmod n$ || $R_2 \leftarrow {R_1}^2 \bmod n$
7: $a \leftarrow (a-R_0)/4 \bmod n$; $b \leftarrow R_1$; $R_1 \leftarrow R_2$
8: $extra \leftarrow 1$
9: **else**
10: $R_0 \leftarrow (a-b)^2 \bmod n$ || $a \leftarrow (a+b)^2 \bmod n$
11: $a \leftarrow (a-R_0)/4 \bmod n$; $b \leftarrow R_1$
12: $extra \leftarrow 0$
13: **else**
14: **if** $extra = 0$ **then**
15: $b \leftarrow b^2 \bmod n$ || <nothing>
16: **else**
17: $b \leftarrow R_1$
18: $extra \leftarrow 0$
19: **return** a

Atomic Parallelized Algorithm

Input: $m, n \in \mathbb{N}$, $m < n$, $d = (d_{k-1} d_{k-2} \dots d_0)_2$, require 7 k-bit registers R_0 to R_6
Output: $m^d \bmod n$

1: $R_0 \leftarrow 1$; $R_1 \leftarrow m$; $v \leftarrow (0,0,0)$; $u \leftarrow 1$
2: **while** $v_0 \leq k-1$ **do**
3: $\quad j \leftarrow d_{v_0}(v_1 + u + 1)$
4: $\quad R_5 \leftarrow (R_0 - R_1)/2 \bmod n$
5: $\quad R_6 \leftarrow (R_0 + R_1)/2 \bmod n$
6: $\quad R_{M_{j,0}} \leftarrow {R_{M_{j,1}}}^2 \bmod n \quad || \quad R_{M_{j,2}} \leftarrow {R_{M_{j,3}}}^2 \bmod n$
7: $\quad R_{M_{j,4}} \leftarrow R_0 - R_2 \bmod n$
8: $\quad R_{M_{j,5}} \leftarrow R_3$
9: $\quad R_{M_{j,6}} \leftarrow R_4$
10: $\quad v_1 \leftarrow M_{j,7}$
11: $\quad u \leftarrow M_{j,8}$
12: $\quad t \leftarrow 1 - v_1(1 - d_{v_0+1})$
13: $\quad R_{N_{t,0}} \leftarrow R_3$
14: $\quad v_{N_{t,1}} \leftarrow 0$
15: $\quad v_{N_{t,2}} \leftarrow v_{N_{t,2}} + 1$
16: $\quad v_0 \leftarrow v_0 + u$
17: **return** R_0

Atomic Parallelized Algorithm

Matrices

$$M = \begin{pmatrix} 1 & 1 & 5 & 6 & 5 & 5 & 5 & 0 & 1 \\ 0 & 6 & 4 & 3 & 0 & 1 & 3 & 1 & 1 \\ 2 & 5 & 3 & 1 & 5 & 5 & 5 & 0 & 0 \\ 2 & 5 & 0 & 6 & 0 & 1 & 5 & 0 & 1 \end{pmatrix}$$

$$N = \begin{pmatrix} 1 & 1 & 0 \\ 5 & 2 & 2 \end{pmatrix}$$

Even Better Parallelization

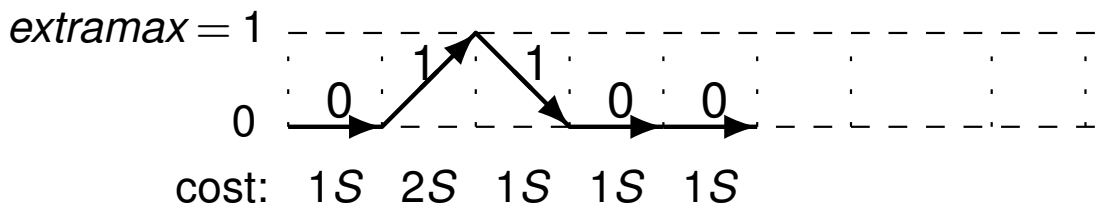
Strategy: use the wasted 1 bit squaring slots to compute several squarings in advance.

The variable *extra*, $0 \leq extra \leq extramax$, stores the number of precomputed squarings.

Remark: requires *extramax* additional memory registers to store the precomputed squarings.

Even Better Parallelization

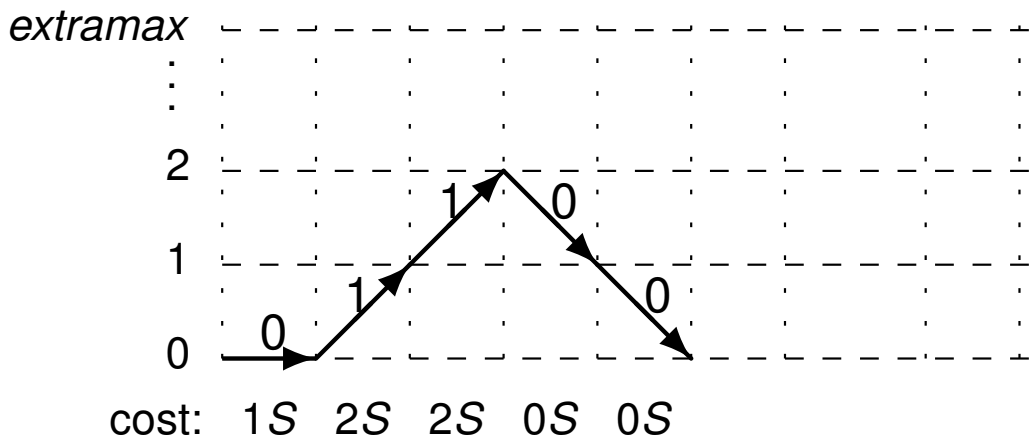
Example : $d = (\ldots 00110)_2$, $extramax = 1$



Cost: $6S$

Even Better Parallelization

Example : $d = (\ldots 01100)_2$, $extramax \geq 2$



Cost: $5S$

Generalized Parallel Square Always

Input: $m, n \in \mathbb{N}$, $m < n$, $d = (d_{k-1} d_{k-2} \dots d_0)_2$, $extramax \in \mathbb{N}^*$, require $extramax+4$ k-bit registers $a, R_0, R_1, \dots R_{extramax+2}$
Output: $m^d \bmod n$

1: $a \leftarrow 1$; $R_1 \leftarrow m$; $extra \leftarrow 0$
2: **for** $i = 0$ **to** $k-1$ **do**
3: **if** $d_i = 1$ **then**
4: **if** $extra < extramax$ **then**
5: $R_0 \leftarrow (a - R_1)^2 \bmod n$ || $R_{extra+2} \leftarrow {R_{extra+1}}^2 \bmod n$
6: $a \leftarrow (a + R_1)^2 \bmod n$ || $R_{extra+3} \leftarrow {R_{extra+2}}^2 \bmod n$
7: $a \leftarrow (a - R_0)/4 \bmod n$
8: $(R_1, R_2, \dots R_{extramax+1}) \leftarrow (R_2, R_3, \dots R_{extramax+2})$
9: $extra \leftarrow extra+1$
10: **else**
11: $R_0 \leftarrow (a - R_1)^2 \bmod n$ || $a \leftarrow (a + R_1)^2 \bmod n$
12: $a \leftarrow (a - R_0)/4 \bmod n$
13: $(R_1, R_2, \dots R_{extramax+1}) \leftarrow (R_2, R_3, \dots R_{extramax+2})$
14: $extra \leftarrow extra-1$
15: **else**
16: **if** $extra = 0$ **then**
17: $R_1 \leftarrow {R_1}^2 \bmod n$
18: **else**
19: $(R_1, R_2, \dots R_{extramax+1}) \leftarrow (R_2, R_3, \dots R_{extramax+2})$
20: $extra \leftarrow extra-1$
21: **return** a

Cost Comparison

We demonstrate that the cost of the parallelized algorithms tends to:

$$\left(1+\frac{1}{4\,extramax+2}\right)S$$

Algorithm	General cost	$S/M=1$	$S/M=0.8$
Parallel Montgomery ladder	$1M$	$1M$	$1M$
Parallel Sq. Al. $extramax=1$	$7S/6$	$1.17M$	$\mathbf{0.93M}$
Parallel Sq. Al. $extramax=2$	$11S/10$	$1.10M$	$\mathbf{0.88M}$
Parallel Sq. Al. $extramax=3$	$15S/14$	$1.07M$	$\mathbf{0.86M}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
Parallel Sq. Al. $extramax\to\infty$	$1S$	$1M$	$\mathbf{0.8M}$

Outline

Conclusion

- Square Always is an alternative countermeasure to S – M discrimination
- It provides better efficiency than the Montgomery ladder
- Parallelization brings best exponentiation performances to our knowledge
- Two parallel squaring blocks yields faster exponentiation than two parallel multiplication blocks !

Thank you for your attention !

Turning Squarings into Multiplications

r, r_1, r_2 being random integers $< n$

$$\begin{aligned} x \times x &= (x + r) \times x - x \times r \\ &= (x + r) \times (x - r) + r^2 \\ &= (x + r_1) \times (x + r_2) - x \times (r_1 + r_2) - r_1 \times r_2 \end{aligned}$$