Multi-Exponentiation Algorithm

Chien-Ning Chen

Email: chienning@ntu.edu.sg

Feb 15, 2012 Coding and Cryptography Research Group
Outline

• Review of multi-exponentiation algorithms

• Double/Multi-exponentiation based on binary GCD algorithm

• Side-channel analysis and countermeasures
Double-/Multi-Exponentiation

- Evaluating the product of several exponentiations
  - *Example:* double-exponentiation, $x^a y^b$, in many digital signature verification primitives
  - *Example:* multi-exponentiation, $g_1^{e_1} \cdots g_h^{e_h}$, in bilinear ring signature and batch verification of signatures

- Existing better algorithms than "multiplying the results of individual exponentiations"
  - Most are developed based on *Shamir’s simultaneous squaring algorithm* (1985)
    (Also proposed by Straus in 1964)
Square and Multiply Algorithm

INPUT: base number $g$, exponent $e = (e_{k-1} \cdots e_0)$

OUTPUT: $g^e$

01 $R = 1$ \hspace{1cm} \rightleftharpoons \text{Accumulator}
02 for $i = k - 1$ to 0 step $-1$ \hspace{1cm} \rightleftharpoons \text{From MSB to LSB}
03 \hspace{0.3cm} R = R^2 \hspace{1cm} \rightleftharpoons \text{Square}
04 \hspace{0.3cm} \text{if } e_i = 1 \text{ then } R = R \times g \hspace{1cm} \rightleftharpoons \text{Multiply (optional)}
05 \hspace{0.3cm} \text{return } R

- Left-to-right algorithm, complexity $= 1.5k$ multiplications
Shamir’s Simultaneous Squaring Multi-Exponentiation Algorithm

**INPUT:** base numbers $g_1 \sim g_h$, exponents $e_1 \sim e_h$ where $e_j = (e_{j,k-1} \cdots e_{j,0})$

**OUTPUT:** $g_1^{e_1} \times \cdots \times g_h^{e_h}$

01 $R = 1$
02 for $i = k - 1$ to 0 step -1
03 $R = R^2$ $\iff$ Simultaneous squaring
04 $R = R \times (g_1^{e_{1,i}} \times \cdots \times g_h^{e_{h,i}})$ $\iff$ How to compute it?
05 return $R$

- If $g_1 \times g_2$ is pre-computed $\quad$ Table $= \{g_1, g_2, g_1 \times g_2\}$
- Complexity of double-exponentiation $= 1.75k$
Interleaving / Simultaneous Multi-Exponentiation\(^1\)

- How to compute \( R \times (g_1^{e_1,i} \times \cdots \times g_h^{e_h,i}) \)
  - Interleaving: compute each of multiplications
    * \( hk/2 \) multiplications on average (\( h \) terms, \( k \)-bit exponents)
  - Simultaneous: prepare a table \( \{g_1^{e_1} \times \cdots \times g_h^{e_h}\} \)
    * \( k(1 - 2^{-h}) \approx k \) multiplications, where \( 2^{-h} \) is the prob. of \( e_1,i = \cdots = e_h,i = 0 \)
    * Table size = \( (2^h - h - 1) \)

- Exponent recoding can further improve performance
  - Table size grows faster than in single-exponentiation

\(^1\text{Named by Bodo Möller}\)
Interleaving Double-Exp. with Window Method

- Separately recode exponents by sliding/fractional window method

<table>
<thead>
<tr>
<th>Recoding Method</th>
<th>Avg. HW</th>
<th>Double-Exp. with $w = 2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w$-bit sliding window</td>
<td>$\frac{k}{w+1}$</td>
<td>$(1 + \frac{2}{w+1})k = 1.66k$</td>
</tr>
<tr>
<td>$w$-bit signed sliding window</td>
<td>$\frac{k}{w+2}$</td>
<td>$(1 + \frac{2}{w+2})k = 1.5k$</td>
</tr>
</tbody>
</table>

- Example: 2-bit signed sliding window, digit set $\{0, \pm 1, \pm 3\}$

  $b = 334 = 0 \ 1 \ 0 \ 1 \ 0 \ 0 \ 1 \ 1 \ 1 \ 0 \ \Rightarrow \ binary$

  $0 \ 1 \ 0 \ 1 \ 0 \ 0 \ 1 \ 0 \ 3 \ 0 \ \Rightarrow \ 2$-bit sliding window

  $0 \ 0 \ 3 \ 0 \ 0 \ 3 \ 0 \ 0 \ \overline{1} \ 0 \ \Rightarrow \ 2$-bit signed sliding window

- Prepare separate tables: $\{x, x^3, x^{-1}, x^{-3}\}, \{y, y^3, y^{-1}, y^{-3}\}$
Simultaneous Double-Exp. with Recodings

- Recode exponents by NAF or Joint Sparse Form

- Example: double-exp. $x^a y^b$

\[
\begin{align*}
a &= 403 = 0 1 1 0 0 1 0 0 1 1 \\
b &= 334 = 0 1 0 1 0 0 1 1 1 0 \\
\end{align*}
\]

- Prepare joint table: $\{x, x^{-1}, y, y^{-1}, xy, (xy)^{-1}, xy^{-1}, x^{-1}y\}$
Binary GCD Multi-Exponentiation Algorithm
Euclidean Double-Exponentiation Algorithm

- Find GCD of $a$ and $b$ when evaluating $x^a y^b$
  - $\gcd(a, b) = \gcd(b, a \mod b) = \gcd(b, r)$ where $a = bq + r$
  - $x^a y^b = x^{(bq+r)} y^b = x^r (x^{bq} y^b) = (yx^q)^b x^r = z^b x^r = \ldots$

- Evaluate the double-exponentiation $x^a y^b$ by
  1. Initialize: $(A_{\langle 0 \rangle}, B_{\langle 0 \rangle}, X_{\langle 0 \rangle}, Y_{\langle 0 \rangle}) = (a, b, x, y)$
  2. $(A_{\langle i+1 \rangle}, B_{\langle i+1 \rangle}, X_{\langle i+1 \rangle}, Y_{\langle i+1 \rangle}) =$
     $(B_{\langle i \rangle}, A_{\langle i \rangle} \mod B_{\langle i \rangle}, Y_{\langle i \rangle} \times X_{\langle i \rangle}^{\lfloor A_{\langle i \rangle}/B_{\langle i \rangle} \rfloor}, X_{\langle i \rangle})$
  3. Terminate: $x^a y^b = X_{\langle i \rangle}^{A_{\langle i \rangle}}$ when $B_{\langle i \rangle} = 0$

---

2In 1989, Bergeron et al. firstly employed Euclidean algorithm to construct continued fractions for evaluating double-exponentiation.
Binary GCD Algorithm

• Alternate method to find greatest common divisor, base on
  1. \( \gcd(a, b) = 2 \gcd(a/2, b/2) \), when both \( a \) and \( b \) are even
  2. \( \gcd(a, b) = \gcd(a/2, b) \), when \( a \) is even and \( b \) is odd
  3. \( \gcd(a, b) = \gcd(a - b, b) \), when \( a \geq b \)

• Recursively perform \( \gcd(a, b) = \begin{cases} 
\gcd((a - b)/2^k, b) \\
\gcd(a, (b - a)/2^k)
\end{cases} \)

• More efficient when handling long integers
  – No long-integer modular operation
  – Only subtraction and right shifting (divided by 2)
Binary GCD Double-Exponentiation Algorithm

- Compute GCD of exponents by binary GCD algorithm
  \[
  x^a y^b = (x^2)^{a/2} y^b \text{ (if } a \text{ is even)} \quad \text{or} \quad = x^{a-b}(xy)^b \text{ (if } a \geq b) \]

- \[
  \begin{align*}
  (A_{i+1}, B_{i+1}, X_{i+1}, Y_{i+1}) &= \begin{cases} 
  (A_i/2, B_i, X_i^2, Y_i) & \text{if } A_i \text{ is even} \\
  (A_i, B_i/2, X_i, Y_i^2) & \text{if } B_i \text{ is even} \\
  (A_i - B_i, B_i, X_i, X_i Y_i) & \text{if } A_i \geq B_i \\
  (A_i, B_i - A_i, X_i Y_i, Y_i) & \text{if } B_i > A_i
  \end{cases}
  \end{align*}
  \]

- Require about \(1.4 \log_2 a\) squarings and \(0.7 \log_2 a\) multiplications when evaluating \(x^a y^b\) when \(a \approx b\)

Complexity = \(2.1k\)
Analysis of bGCD Double-Exp. Alg.

- Evaluate performance\(^3\) by \(\log_2(A_{\langle i \rangle}B_{\langle i \rangle})\) (i.e., length of \(A_{\langle i \rangle}B_{\langle i \rangle}\))

- Halving: \((A_{\langle i+1 \rangle}, B_{\langle i+1 \rangle}, X_{\langle i+1 \rangle}, Y_{\langle i+1 \rangle}) = (A_{\langle i \rangle}/2, B_{\langle i \rangle}, X_{\langle i \rangle}^2, Y_{\langle i \rangle})\)
  - \(\log_2(A_{\langle i \rangle}) - \log_2(A_{\langle i \rangle}/2) = 1\), always reduce 1 bit

- Subtraction: \((\cdots) = (A_{\langle i \rangle} - B_{\langle i \rangle}, B_{\langle i \rangle}, X_{\langle i \rangle}, X_{\langle i \rangle}Y_{\langle i \rangle})\)
  - \(\log_2(A_{\langle i \rangle}) - \log_2(A_{\langle i \rangle} - B_{\langle i \rangle})\) depends on \(A_{\langle i \rangle}/B_{\langle i \rangle}\)
  - Reduce more bits when \(A_{\langle i \rangle} \approx B_{\langle i \rangle}\)
  - Reduce almost nothing when \(A_{\langle i \rangle} \gg B_{\langle i \rangle}\)

---

\(^3\)Referring to the analysis of Brent in 1976.
Improvement to bGCD Double-Exp. Alg.

- **Strategy 1**: Always perform subtraction when \( A_{i} \approx B_{i} \)
  - Subtraction has better performance than halving if \( A_{i} \approx B_{i} \)
  - Determine \( A_{i} \approx B_{i} \) by length, \( \lfloor \log_2 A_{i} \rfloor - \lfloor \log_2 B_{i} \rfloor \leq 1 \)

- **Strategy 2**: Append \( 1^1 \) to be a triple-exp. \( x^a y^b 1^1 \)
  - Solve the worst case, \( A_{i} \) is odd, \( B_{i} \) is even, \( A_{i} \gg B_{i} \)
  - \( A_{i+1+k} = (A_{i} - 1)/2^k \),
    until \( A_{i+1+k} \) is odd, or \( A_{i+1+k} \approx B_{i} \)
  - Require 1 additional variable,
    \[
    X_{i}^{A_{i}} Y_{i}^{B_{i}} Z_{i} = X_{i}^{(A_{i} - 1)} Y_{i}^{B_{i}} (Z_{i} \times X_{i})
    \]
### Comparison of Double-Exp. Alg.

- Performance comparison of 1024-bit double-exponentiation

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Avg. # of Operations</th>
<th>Avg. Compu.</th>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Square</td>
<td>Mul.</td>
<td>Sum</td>
</tr>
<tr>
<td>Euclidean</td>
<td>749.7</td>
<td>896.8</td>
<td>1646.5</td>
</tr>
<tr>
<td>Binary GCD</td>
<td>1445.3</td>
<td>723.3</td>
<td>2168.5</td>
</tr>
<tr>
<td>Strategy 1</td>
<td>724.8</td>
<td>1048.1</td>
<td>1772.9</td>
</tr>
<tr>
<td>Strategy 1&amp;2</td>
<td>503.5</td>
<td>1106.8</td>
<td>1610.3</td>
</tr>
<tr>
<td>Simult. binary</td>
<td>1024</td>
<td>768.0</td>
<td>1792.0</td>
</tr>
<tr>
<td>Simult. JSF</td>
<td>1024</td>
<td>512.0</td>
<td>1536.0</td>
</tr>
<tr>
<td>Inter. binary</td>
<td>1024</td>
<td>1024.0</td>
<td>2048.0</td>
</tr>
<tr>
<td>Inter. 2-uSW</td>
<td>1024</td>
<td>682.7</td>
<td>1706.7</td>
</tr>
<tr>
<td>Inter. 2-sSW</td>
<td>1024</td>
<td>512.0</td>
<td>1536.0</td>
</tr>
</tbody>
</table>

Feb 15, 2012  Coding and Cryptography Research Group 14
binary GCD **Multi-Exp. Algorithm**

- Follow the same strategies of binary GCD double-exponentiation to reduce the largest exponent as efficient as possible
- No pre-computation table, memory efficiency
- Scalable from single exp. $g^e1^1$ to multi-exp. Good performance for any bit length of exponents
Performance of High-Dimensional

- Performance comparison of 1024-bit multi-exponentiation

<table>
<thead>
<tr>
<th>Term</th>
<th>Algorithm</th>
<th>Avg. # of</th>
<th>Avg. Comp.</th>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Square</td>
<td>Mul.</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>bGCD</td>
<td>284.3</td>
<td>1503.5</td>
<td>1.746</td>
</tr>
<tr>
<td></td>
<td>Simult. binary</td>
<td>1024.0</td>
<td>4+ 896.0</td>
<td>0.004+1.875</td>
</tr>
<tr>
<td></td>
<td>Inter. binary</td>
<td>1024.0</td>
<td>1536.0</td>
<td>2.500</td>
</tr>
<tr>
<td>4</td>
<td>bGCD</td>
<td>173.7</td>
<td>1822.4</td>
<td>1.949</td>
</tr>
<tr>
<td></td>
<td>Simult. binary</td>
<td>1024.0</td>
<td>11+ 960.0</td>
<td>0.010+1.938</td>
</tr>
<tr>
<td></td>
<td>Inter. binary</td>
<td>1024.0</td>
<td>2048.0</td>
<td>3.000</td>
</tr>
<tr>
<td>10</td>
<td>bGCD</td>
<td>20.8</td>
<td>3301.1</td>
<td>3.244</td>
</tr>
<tr>
<td></td>
<td>Simult. binary</td>
<td>1024.0</td>
<td>1013+1023.0</td>
<td>0.989+1.999</td>
</tr>
<tr>
<td></td>
<td>Inter. binary</td>
<td>1024.0</td>
<td>5120.0</td>
<td>6.000</td>
</tr>
</tbody>
</table>
Lim-Lee Algorithm and BGMW Method

- Lim-Lee: simultaneous exponentiation with multiple smaller tables
  - Split $h$ base numbers into $l$ set, construct table on each set
  - Table size reduced from $O(2^h)$ to $O(l2^{h/l})$, $l$-fold multiplications

\[
\begin{align*}
03 & \quad R = R^2 \\
04 & \quad R = R \times (g_1^{e_{1,i}} \times \cdots \times g_w^{e_{w,i}}) \times (g_{w+1}^{e_{w+1}} \times \cdots \times g_{2w}^{e_h}) \times \cdots
\end{align*}
\]

- BGMW: $w$-bit fixed window with a special comp. sequence

  - Example: $R^8 \times (g_1^3 \times g_2 \times g_3^3 \times g_4^4) \rightarrow$
    
    $R^8 \times (g_4) \times (g_4g_1g_3) \times (g_4g_1g_3) \times (g_4g_1g_3g_2)$
Comparison with Lim-Lee and BGMW

Complexity = \# of mul. 
Len. × Dim.

- Lim-Lee 160-bit
- Lim-Lee 1024-bit
- BGMW 160-bit
- BGMW 1024-bit
- binaryGCD 160-bit
- binaryGCD 1024-bit

Dimension

Feb 15, 2012  Coding and Cryptography Research Group
Side-Channel Analysis and Countermeasures
Simple Power Analysis

• Attacker can distinguish **squaring** and **multiplication**
  – How much info can be retrieved from S and M sequence?

• Left-to-right binary square-and-multiply algorithm

\[
\begin{align*}
03 & \quad R = R^2 \quad \iff \text{Squaring always happens} \\
04 & \quad \text{if } e_i = 1 \text{ then } R = R \times g \quad \iff \text{Mul. indicates a nonzero bit}
\end{align*}
\]

– Fully recover private exponent when retrieving one sequence
– *Example*: \(... \text{S}_M \text{S} \text{S} \text{S}_M \text{S}_M \text{S}_M \) \... indicates \... 10011 \...
Immunity Against Simple Power Analysis

• bGCD multi-exp. alg. is natively with immunity against SPA, because both base numbers are updated

  – When squaring occurs, \( (X_{i+1}, Y_{i+1}) = \begin{cases} (X_i^2, Y_i) \\ (X_i, Y_i^2) \end{cases} \),
    can not distinguish which variable is squared

  – When multiplication occurs, \( (X_{i+1}, Y_{i+1}) = \begin{cases} (X_i Y_i, Y_i) \\ (X_i, X_i Y_i) \end{cases} \),
    can not distinguish which variable is overwritten

• More than \( 1.5k \) indistinguishable operations in \( k \)-bit double-exp.
Differential Power Analysis

- Statistical methods to test whether expected values appear
  - Power consumption depends on operand

- In left-to-right square-and-multiply algorithm, if attacker has retrieved MSBs of exponent, \( E_{i+1} = (e_{k-1} \cdots e_{i+1}) \)

1. Calculate \( R_{i+1} = g^{E_{i+1}} \)
2. \( E_i = (e_{k-1} \cdots e_{i+1} e_i) \), guess \( e_i = 0 \) or \( 1 \)
3. Either \( R_i = R_{i+1}^2 \) or \( R_i = R_{i+1}^2 \times g \)
4. Test whether \( (R_i)^2 \) or \( (R_i)^2 \) appears by DPA
Immunity Against Differential Power Analysis

- To prevent DPA by appending \( r^\phi \), where \( r \) is a random number and \( \phi \) is the order of group
  - Single exp. \( g^e \implies g^{e\phi1^1} \), Complexity = 1.5726k
  - Double exp. \( x^a y^b \implies x^a y^{b\phi1^1} \), Complexity = 1.7461k

- The intermediate values will be of the form: \( g^{\alpha r^\beta} \)
  - Cannot guess them because \( r \) is unknown \( \Rightarrow \) NO DPA
  - After computation, we have \( \left( g^{\alpha r^\beta}\right)^0 \left( g^{\alpha' r^{\beta'}}\right)^0 \left( g^e\right)^1 \)
    * Either \( g^{\alpha r^\beta} \) or \( g^{\alpha' r^{\beta'}} \) will be the next random number
Summary of bGCD Multi-exp. Alg.

- Comparable performance, scalable from single exp. to multi-exp.
- No pre-computation table, no inversion computation
- Side-channel immunity
- No explicit proof of complexity, only simulation
- All variables will be overwritten during computation
Thank You