RSA CONFERENCE 2009

Securing RSA against Fault Analysis by Double Addition Chain Exponentiation

Matthieu Rivain

Oberthur Technologies & Univ. of Luxembourg

04/24/09 | Session ID: CRYP-403

Session Classification:

Agenda

RSA and Fault Analysis

A New Self-Secure Exponentiation

A New Secure RSA-CRT

Complexity Analysis

Oberthur Technologies

RSA CONFERENCE 2009

RSA and Fault Analysis

Preliminaries

- RSA signature : $s = m^d \bmod N$
  - m : message
  - d : private exponent
  - $N = p.q$: public modulus

- RSA with CRT (4 times faster):
  - $s_p = m^{d_p} \bmod p$ where $d_p = d \bmod (p-1)$ $(s_p = s \bmod p)$
  - $s_q = m^{d_q} \bmod q$ where $d_q = d \bmod (q-1)$ $(s_q = s \bmod q)$
  - $s = CRT_{p,q}(s_p, s_q)$

Bellcore Fault Attack

- A fault corrupts the computation of s_p :
  - $f(s_p) \neq m^{d_p} \bmod p$
  - $s_q = m^{d_q} \bmod q$
  - $f(s) = CRT_{p,q}(f(s_p), s_q)$
- The faulty signature satisfies
  - $f(s) \neq s \bmod p$ and $f(s) = s \bmod q$
  - $(f(s) - s)$ is a multiple of q but not of p
  - $\gcd(f(s) - s, N) = q$
- N is factorized with a single faulty signature
- Other fault attacks exist on RSA without CRT

Problem

Problem: *Perform an RSA computation that detects errors.*

Straightforward solutions:

- Perform the computation twice
  - double the execution time
- Verify the computed signature : $s^e \bmod N = m$?
  - e is not necessarily available
  - e may be large → double the execution time

Problem: *Perform an RSA computation that detects errors while e is not available or possibly large.*

State of the Art

- **Modulus extension:** redundancy included in modular operations
    - s_{Nt} = m^d mod N¢t
    - m^d mod t = s_{Nt} mod t ?
  - Shamir's Trick [Eurocrypt'97 Rump Session]
  - [Vigilant CHES 2008]
- **Self-secure exponentiations** : redundancy included in the exponentiation algorithm
  - [Giraud IEEE-TC 2006]
    - (s' = m^{d-1} mod N, s) ← MontgomeryLadder(m, d, N)
    - s'¢m mod N = s ?
  - [Boscher *et al.* WISTP 2007]

A New Self-Secure Exponentiation

Basic Principle

- **Definition:** A *double exponentiation* computes the pair of powers (m^a, m^b) from an element m and a pair of exponents (a, b).

- **Basic principle:**
  - use a double exponentiation algorithm to compute

    $$s = m^d \bmod N \quad \text{and} \quad c = m^{\varphi(N)-d} \bmod N$$

    where $\varphi(N)$ is the Euler's totient of N

  - check: s¢c mod N = 1 ?

- If no error occurs then s¢c mod N = $m^{\varphi(N)} \bmod N = 1$
- Otherwise the check fails (with high probability)

- Problem: design a **double exponentiation algorithm**

Double Addition Chains

Definition: An *addition chain* for a is a sequence $x_0, x_1, \cdots, x_n$ s.t. :

- $x_0 = 1$ and $x_n = a$
- for every k there exist $i, j < k$ s.t. $x_k = x_i + x_j$

- An addition chain for a provides a way to compute m^a for every m:
  - Let $m_0 = m$
  - And $m_k = m_i$ ¢ m_j where $x_k = x_i + x_j$
  - By induction $m_k = m^{x_k}$ and $m_n = m^a$

Definition: A *double addition chain* for (a,b) is an addition chain for b s.t. $x_{n-1} = a$.

- provides a way to compute (m^a, m^b) for every m
- provides a **double exponentiation**

Our Goal

Goal: construct a double addition chain

- suitable for implementations constrained in memory
  - Nb. of registers for the exponentiation

    = nb. of intermediate x_i's to store
- as short as possible
  - Nb. of multiplications in the exponentiation

    = nb. of additions in the chain

Our Goal (2)

- Keep 3 temporary results: a_i, b_i and 1
  - i.e. m^{a_i}, m^{b_i} and m for the exponentiation
  - s.t. $(a_0, b_0) = (0,1)$ and $(a_n, b_n) = (a,b)$ for some n
- Construct a chain ω s.t.
  - $a_{i+1} = a_i + b_i$ if $\omega_i = 0$
  - $a_{i+1} = 2 ¢ a_i$ if $\omega_i = 1$
  - $a_{i+1} = a_i + 1$ if $\omega_i = 2$
  - $b_{i+1} = a_i + b_i$ if $\omega_i = 3$
  - etc …
- Restrict the nb. of possibilities for the ω_i's to optimize the storage of ω

Our Heuristic

Principle:

- Start from the pair (a,b)
- Construct the inverse chain by applying the inverse operations
- i.e. construct a sequence (α_i, β_i) s.t.
  - $(\alpha_0, \beta_0) = (a,b)$
  - $(\alpha_n, \beta_n) = (0, 1)$ for some n
  - $\alpha_{i+1}, \beta_{i+1}$ 2 $\{\alpha_i - \beta_i, \beta_i/2, \alpha_i - 1, \ldots\}$

Our Heuristic (2)

We assume $a \leq b$ and conserve $\alpha_i \leq \beta_i$ for every i

We iterate:

- if β_i is at least twice α_i then
  - if β_i is odd then $\beta_{i+1} = (\beta_i - 1)/2$ $\omega \leftarrow (01 \,||\, \omega)$
  - if β_i is even then $\beta_{i+1} = \beta_i / 2$ $\omega \leftarrow (00 \,||\, \omega)$
- if β_i is lower than twice α_i then
  - $\alpha_{i+1} = \beta_i - \alpha_i$ and $\beta_{i+1} = \alpha_i$ $\omega \leftarrow (1 \,||\, \omega)$

Example

- $(\alpha_0, \beta_0) = (a,b) = (9, 20)$
- $(\alpha_1, \beta_1) = (9, 20/2) = (9, 10)$ $\omega = 00$
- $(\alpha_2, \beta_2) = (10 - 9, 9) = (1, 9)$ $\omega = 100$
- $(\alpha_3, \beta_3) = (1, (9 - 1)/2) = (1, 4)$ $\omega = 01100$
- $(\alpha_4, \beta_4) = (1, 4/2) = (1, 2)$ $\omega = 0001100$
- $(\alpha_5, \beta_5) = (1, 2/2) = (1, 1)$ $\omega = 000001100$
- $(\alpha_6, \beta_6) = (1 - 1, 1) = (0, 1)$ $\omega = 1000001100$

Example (2)

- $(a_0, b_0) = (0,1)$
- $(a_1, b_1) = (0+1, 1) = (1, 1)$ $\omega = 1\ 000001100$
- $(a_2, b_2) = (1, 2¢1) = (1, 2)$ $\omega = 1\ 00\ 0001100$
- $(a_3, b_3) = (1, 2¢2) = (1, 4)$ $\omega = 100\ 00\ 01100$
- $(a_4, b_4) = (1, 2¢4+1) = (1, 9)$ $\omega = 10000\ 01\ 100$
- $(a_5, b_5) = (9, 1+9) = (9, 10)$ $\omega = 1000001\ 1\ 00$
- $(a_6, b_6) = (9, 2¢10) = (9, 20)$ $\omega = 10000011\ 00$

- Double Addition Chain:
  - if $\omega = (00 \,||\, \omega')$ then $b_i = 2¢b_i$
  - if $\omega = (01 \,||\, \omega')$ then $b_i = 2¢b_i+1$
  - if $\omega = (1 \,||\, \omega')$ then $a_i = b_i$; $b_i = a_i+b_i$

Double Exponentiation

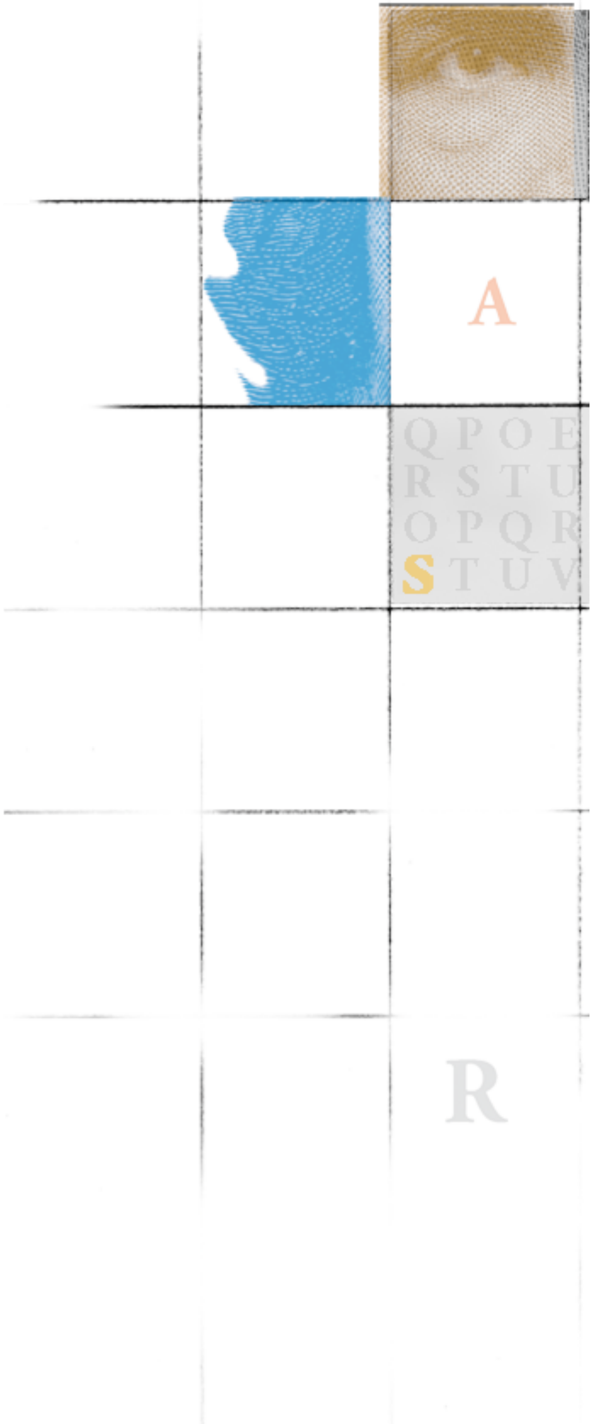
- $R_0 \leftarrow 0$; $R_1 \leftarrow m$; $R_2 \leftarrow m$
- $i \leftarrow 0$; $\gamma \leftarrow 1$ // γ : boolean s.t. $R_\gamma = m^{b_i}$ and $R_{1-\gamma} = m^{a_i}$
- **while** $i < \text{length}(\omega)$ **do**
  - **if** $(\omega_i = 0)$ **then**
    - $R_\gamma \leftarrow (R_\gamma)^2 \bmod N$
    - **if** $(\omega_{i+1} = 1)$ **then** $R_\gamma \leftarrow R_\gamma$¢$R_2 \bmod N$
    - $i \leftarrow i+2$
  - **else**
    - $R_\gamma \leftarrow R_\gamma$¢ $R_{1-\gamma} \bmod N$
    - $\gamma \leftarrow 1-\gamma$
    - $i \leftarrow i+1$
- **return** $(R_{1-\gamma}, R_\gamma)$

Self-Secure Exponentiation

- $\omega \leftarrow$ ChainCompute(d, 2 φ(N) – d)
- (s,c) $\leftarrow$ DoubleExp(m, ω, N)
- **if** (s¢c mod N $\neq$ 1) **then return** “error”
- **else return** s

- NB: we use 2 φ(N) – d in order to fit the constraint $a \leq b$
- The chain computation may be performed off-line
  - It is unique for (d,N)

A New Secure RSA-CRT



Secure RSA-CRT

- $\omega_{\mathbf{p}} \leftarrow$ ChainCompute(d_p, 2 (p-1) – d_p)
- (s_p, $c_{\mathbf{p}}$) $\leftarrow$ DoubleExp(m mod p, $\omega_{\mathbf{p}}$, p)
- $\omega_{\mathbf{q}} \leftarrow$ ChainCompute(d_q, 2 (q-1) – d_q)
- (s_q, $c_{\mathbf{q}}$) $\leftarrow$ DoubleExp(m mod q, $\omega_{\mathbf{q}}$, q)
- s $\leftarrow$ $CRT_{p,q}$(s_p, s_q)
- **if** (s¢c_p mod p $\neq$ 1 or s¢c_q mod q $\neq$ 1) **then return** “error”
- **else return** s

- Implementation security requirements:
  - The **exponents integrity** must be checked (e.g. with CRC) at the beginning of the chain computation (if done dynamically)
  - The **message integrity** must be checked (e.g. with CRC) at the beginning of each double exponentiation

Complexity Analysis

Time Complexity

- Mainly depends on the number of modular multiplications
- Multiplications-per-bit ratio : θ

	$l = 512$	$l = 640$	$l = 768$	$l = 896$	$l = 1024$
$\mathrm{E}[\theta]$	1.65	1.66	1.66	1.66	1.66
$\sigma(\theta)$	0.020	0.017	0.017	0.016	0.014

- Comparisons
  - For (insecure) *square-and-multiply* : E(θ) = 1.5

    → overhead of **10%**
  - For previous self-secure exponentiations : E(θ) = 2

    → gain of **18%**

Memory Complexity

- Three registers for the exponentiation (3 l bits of memory)
- Chain length : n*

	$l = 512$	$l = 640$	$l = 768$	$l = 896$	$l = 1024$
$\mathrm{E}[n^*]$	$2.03\ l$	$2.03\ l$	$2.03\ l$	$2.03\ l$	$2.03\ l$
$\sigma(n^*)$	$0.015\ l$	$0.013\ l$	$0.011\ l$	$0.010\ l$	$0.010\ l$

- The chain can be stored in a (2.2 ¢ l)-bit buffer
  - P [n* > 2.2 ¢ l] < 2^{-80}
- Total memory consumption:
  - 5.2 l bits with dynamic chain computation
  - 3 l bits with pre-computed chain

Comparison

- Extended modulus countermeasures
  - (+) works with every exponentiation algorithm
    - e.g. sliding window exponentiations (faster)
  - (-) larger modulus → slower modular multiplications

- Previous self-secure exponentiations
  - (+) no pre-computation
  - (-) more modular multiplications

Comparison (2)

Theoretical time & memory complexities

for an RSA 1024 with CRT

Countermeasure	Time ($10^6 \cdot t_0$)	Memory (Kb)
Vigilant [CHES 2008] ($q = 1$)	$\{511, 484\}$	$\{2.4,\ 2.3\}$
Vigilant [CHES 2008] ($q = 2$)	$\{468, 444\}$	$\{2.6,\ 2.5\}$
Vigilant [CHES 2008] ($q = 3$)	$\{440, 417\}$	$\{3.7,\ 3.6\}$
Giraud [IEEE-TC 2006]	537	3.5
Our scheme	443	2.5 (+1.1)

- Vigilant Scheme
  - → *q*-ary sliding widow exponentiation
  - → {64,80}-bit modulus extension

Conclusion

- New principle to check consistency of RSA computations based on a double exponentiation
- Heuristic to construct a double addition chain
  - → double exponentiation algorithm using 3 registers and 1.65 l multiplications
- New self-secure exponentiation and RSA-CRT
- Security and complexity analyses
- Updated paper version on the IACR ePrint

RSA CONFERENCE 2009

The end!

Questions ?