Fault Detection Mechanisms for Smatcards Performing Modular Exponentiation

Shay Gueron[1,2]

[1] Discretix Technologies, Netanya, Israel
[2] Department of Mathematics, University of Haifa, Haifa, 31905, Israel
shay@math.haifa.ac.il

- **Task**: Foiling fault attacks on smartcards implementing modular exponentiation.

- **Problem**: Detecting error only at the end of the computations, may leak information.

  $(A^D)^E \bmod N = A$

- **Problem**: Detecting error only at the end of the computations – affects performance.

- **Proposal**:
- Check each step of the exponentiation (multiplication/squaring) independently.

- Immediate abortion of the procedure, as soon as any fault is detected, without completing the computation of the erroneous result.

- Test can be performed on-the-fly, to avoid any performance penalty.

- Proposed approach can be easily generalized.

- **Childhood methods: easiest way to understand the method**

- **Did we make a mistake?**

```
  1234
  4578
  8769
  4878
--------
 19357
```

- **Did we make a mistake here?**

```
    457
     36
  ------
   2742
  1361
--------
 16352
```

- **Observing that an error occurred is simple Does not tell you where the mistake occurred *(no need to correct or find where)*.**

• Easy way: Arithmetic Stamp in “Smartcad Basis”

Definition: the arithmetic stamp of a positive integer X, denoted X' or Stamp (X), is

$X' = \text{Stamp}(X) = (X-1) \bmod F + 1$

where F is some chosen modulus.

Use $F = 2^{32} - 1$

A natural choice for a system that uses a 32 bits bus.

Computing X' = Stamp (X): $X = [X_{L-1}, .., X_1, X_0]$ positive integer consisting of L words X_i

Algorithm Computing the Arithmetic Stamp
Input: $X = [X_{L-1}, .., X_1, X_0]$ **Output:** $X' = \text{Stamp}(X) = (X-1) \bmod F + 1$ **Computations:** 1. $S=X_0$ 2. for i from 1 to L-1 do $S = S+X_i$ End 3. S = lsb(S)+ msb(S) 4. S = lsb(S)+ msb(S) Output S

Example 1: X = 79228159673465750010344767471.
In binary representation, X has with L=3 words (of length t=32 bits).
X=[$X_2X_1X_0$]=
11111111111111111111111101100101
11111111111111111111111111001001
11111111111111111111111111101111
Computing X'=Stamp (X) within L+2 = 4 clock cycles is carried out as follows:

S= X_0=11111111111111111111111101100101

S = X_1+X_0 =
111111111111111111111111110111000
(S has 33 bits)

S = X_2+S=
1011111111111111111111111100011101
(S has 34 bits)

S = lsb(S) + msb(S)=
11111111111111111111111100011111
(S has 32 bits)

S = lsb(S) + msb(S) =
11111111111111111111111100011111
(S has 32 bits)

Output S = Stamp (X) = X'.

S = 4294967071in decimal representation, and it can be easily checked that indeed S = X-1 mod F + 1 (=X')

Using the Arithmetic Stamp to Check Load/Unload Integrity:

Add one "check–word" to the data structure

The Modular Exponentiation Algorithm

NRMM(A, B, N, r): Nonreduced Montgomery Multiplication of order r

Input:
A, B (r bits long integers)
N (n bits long odd integer)
$r \geq n$
Output: NRMM (A, B, N, r) = $(AB + YN) / 2^r$ and $Y = -ABN^{-1} \bmod 2^r$
Computations:

```
S=0
For i from 0 to r-1 do
    $S = S + A_i B$
    $Y_i = S_0$
    $S = S + Y_i N$
    $S = S/2$
End for
Return S, Y
```

Modular exponentiation with NRMM

Input:
X (x bits long integer)
A (n bits long integer, $A < N$)
N (n bits long odd integer)
$s = n+2$
Pre-computed value $H = 2^{2n} \bmod N$
Output:
$A^X \bmod N$
Computations:

```
1. B = NRMM (A, H, N, s)
2. T = B
3. For i from x-2 to 0 do
    T = NRMM (T, T, N, s)
    if $X_i=1$ then T = NRMM (T, B, N, s)
end for
4. T = NRMM (T, 1, N, s)
Return T
```

The Fault Detection Strategy: Test all intermediate results

Use the Arithmetic Stamp For Checking NRMM

External Checking of the NRMM Result, with No Additional Dedicated Hardware

$T = NRMM(A, B, N, s) = (AB + Y N)/2^s$

$Y = -ABN^{-1} \bmod 2^s$.

To verify the result (T), check that its arithmetic stamp (T') equals to the arithmetic stamp of $((A B + Y N)/2^s)$.

Use the following property (see Appendix):

$$((A B + Y N)/2^s)' = ((A' B' + Y' N')' (2^{-s})')' \qquad (*)$$

F is constant in our system, $Z = (2^{-s})' = 2^{-s} \bmod F$ is also fixed.

Pre-compute and store Z (requiring one word of storage).

N fixed (if RSA key is not changed): pre-compute and store N'.

Before exponentiation, also pre-compute B'

Verifying NRMM operation
Pre-Computations: Z, , B', N' Input: T = NRMM (T, B, N, s), Y. 1. Compute Y', and T'. 2. Compute Q = ((T'B' + Y'N') Z)' 3. Compare Q to T'. If the comparison fails- abort.

Example 2:

(data is written in hexadecimal base)

N = 8000082D80216E1B, A = 80002407,
B = 8000082D8020EE1B

We have n = 64 (the bit length of N), s= n+2 = 66.
Suppose that the CCP computed
T = NRMM (A, B, N, s) = 1D8921075EC05D7A, and
Y = EC48F921E8425BF9.

The result (T) needs to be verified.

Pre-Computation:
    N' = 217649, Z = 40000000
    A' = 80002407, B' = 20F649
Verification procedure
  (step 2 is broken to several sub-steps)
1. T' = 7C497E81, Y' = D48B551B.
2.a. Q1 = (A' B')' = 23997B28,
2.b. Q2 = (Y' N')' = CD8C7EDD,
2.c. Q3 = (Q1 + Q2)' =   F125FA05,
2.d. Q4 = (Q3 Z)' = 7C497E81.
3.Compare Q4 to T'.
(Here, Q4=T'= 7C497E81, and the result is verified)

Internal On-the-fly checking of the NRMM Result, with Additional Dedicated Hardware

Repeat all operations (using a small register)

NRMM (A, B, N, s) Computation	On-the-fly NRMM verification
	Pre-computation: B', N'
$S=0$	$T' = 0$
for i from 0 to s-1 do	for i from 0 to s-1 do
$S = S+A_i B$	$T' = T'+A_i B'$
$Y_i= S_0$	
$S = S+ Y_i N$	$T' = T'+Y_iN'$
$S = S/2$	$T' = (T' + T_0 F)/2$
end	end
Output S	Output T'
	Verification: $S'=T'$

On-the-fly computetion:

Check is ready (aolmost) together with the result.

- **Summay:**

- Easy method.
- High success probability.
- Several ways to implement (SW and HW).
- Can be generalized to other arithmetic operations.