Incorporating Error Detection in an RSA Architecture

L. Breveglieri\textsuperscript{1}, I. Koren\textsuperscript{2}, P. Maistri\textsuperscript{1}, M. Ravasio\textsuperscript{3}

\textsuperscript{1}Department of Electronics and Information Technology
Politecnico Di Milano, Milano, Italy

\textsuperscript{2}Department of Electrical and Computer Engineering
Univ. of Massachusetts, Amherst, MA

\textsuperscript{3}STMicroelectronics, Agrate Brianza, Milano, Italy
Outline

- Introduction
  - Motivation and objectives

- RSA
  - The Algorithm
  - The Basic Architecture

- Online detection
  - Code generation
  - Prediction
  - Verification

- Results
  - Overheads and error coverage

- Concluding remarks
Motivation

- Boneh et al. have shown successful attacks against RSA
  - CRT-based implementations can be broken with a single faulty signature
  - Other implementations can also be attacked, although more fault injections are necessary

- Countermeasures
  - Random multiplicative masking on modulus allows cross checking in CRT-RSA (Shamir)
  - Additional modular code is propagated through the encryption process (Walter @ CHES 2000)
Objectives

- Implementation of an RSA architecture
  - The reference model was presented by Mazzeo et al. at DATE ‘03
- Extension of the architecture with error detection capabilities
- Evaluation of the actual overheads (area and latency)
- Validation of the estimated coverage for transient faults
The RSA Cryptosystem

- Message is encrypted/decrypted by modular exponentiation
  \[ m^e \mod N \quad m = (m^e)^d \mod N \]

- Parameters:
  - \( N \) is the public modulus (\( N=p\times q \))
  - \( p \) and \( q \) are large primes
  - \( m \) is the message
  - \( e \) is the public exponent
  - \( d \) is the private exponent, satisfying \( d\times e = 1 \mod (p-1)(q-1) \)

- A faulty signature allows to factor the modulus \( N \) easily (and thus recover the private key)
The Basic RSA Architecture

It works in the **Montgomery domain**

Operations computed using a **32-bit** Processing Element (PE)
Online Detection

- A residue code can detect errors in the computation

- Three components are needed:
  - Code generation unit
  - Prediction rules, to keep the check bits consistent with data
  - Checkpoint, to validate the predicted check bits

- Residues fit modular arithmetic very well
  - The result check bits of any operation can be easily obtained from the check bits of the operands

- The use of the residue code must be transparent to the user
  - Interaction with device stays the same
The code/word size ratio is 1:8
- Carry-Save layer minimizes carry propagation delay
- Carry-Select Adder computes final check bits with no further modular reduction
Code Prediction

- Fetch operands
- Compute
- Write

Time

First word

Second word

... Last (most significant) word

Code word

Memory words

Correction is sometimes required
Code Verification

- Read requests are verified before revealing memory content
  - Starting address must be the least significant word
- Memory word are internally read, and actual code is computed
  - The number of reads depends on the operand size (768 up to 2048 bits)
- Last word read gives the predicted code, which can be verified against the actual code
- If verification is positive, then the read process is started again from the initial address, and data is sent to output
Code Verification

1. Read request
2. Read next word
3. Update Check bits
4. Check if most significant word?
   - Yes, retrieve predicted Check bits
     - Does prediction match?
       - Yes, output word
         - No, read next word
     - No, start read again
   - No, ready

Ready

FDTC 2005
Incorporating Error Detection in an RSA Architecture
Latency Overheads

- **Operand loading:** one idle cycle per operand is required to let the device initialize the code
  - **Last written word must be the most significant one**

- **Computation:** only one clock cycle per PE operation
  - **This holds for any operand size**

- **Result verification:** reading takes twice longer

- **Operation times for the basic and the error detecting designs:**

<table>
<thead>
<tr>
<th>Operand Size</th>
<th>Exponentiation Time (clock cycles, [M])</th>
<th>Basic</th>
<th>Error Detecting</th>
<th>Overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>768</td>
<td></td>
<td>30.18</td>
<td>31.07</td>
<td>+2.94</td>
</tr>
<tr>
<td>1024</td>
<td></td>
<td>66.68</td>
<td>68.27</td>
<td>+2.38</td>
</tr>
<tr>
<td>1536</td>
<td></td>
<td>207.09</td>
<td>210.66</td>
<td>+1.72</td>
</tr>
<tr>
<td>2048</td>
<td></td>
<td>469.59</td>
<td>475.94</td>
<td>+1.35</td>
</tr>
</tbody>
</table>
## Area Overheads

<table>
<thead>
<tr>
<th>Unit</th>
<th>Core Unit ($\mu m^2$)</th>
<th>Memory ($\mu m^2$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Version</td>
<td>140,000</td>
<td>585,000</td>
</tr>
<tr>
<td>Code Generator</td>
<td>18,900</td>
<td>-</td>
</tr>
<tr>
<td>Code Generator + Validation Unit</td>
<td>47,000</td>
<td>-</td>
</tr>
<tr>
<td>Error detecting version</td>
<td>193,000</td>
<td>615,000</td>
</tr>
</tbody>
</table>

Overall area increase:

+11 %
Error Coverage

 Errors are injected into the word read from memory

 Coverage Test: Transient random fault
  Random 16-bit value injected once at random times
  
  100% detection rate out of hundreds of tests

 Undetected faults: Transient double faults
  Random 16-bit value injected once in two different 16-bit blocks of the same word at a random time
  
  0% detection rate (as expected)
Further Improvements

- Larger code
  - Overhead increase is negligible but coverage improves
  - Upper bound is PE size (32 bits)

- Embedding the code into the most significant word
  - Code size is slightly less than PE size
  - No latency overhead in code prediction
  - No memory overhead
  - Code correction term is still needed

- Optimized unit for code correction
  - Actually a simpler adder since the second operand is known a priori (± the base of the code, i.e., +65535 or -65535)

- Optimized code verification unit
  - Second-phase reading involves address computation
Conclusions

- Error detecting code can be a viable solution even for an area-constrained architecture
- Latency overhead is negligible
  - It can be considerably reduced after optimization
- Area overhead is reasonable
  - Most additional area is required by the code validation unit
- Error coverage depends on the code size and is customizable
- Designed for regular RSA
  - It can work for CRT-RSA also