Detecting Faults in Integer and Finite Field Arithmetic Operations for Cryptography

L. Breveglieri, I. Koren
and P. Maistri
Introduction

- Motivation and objectives
- Symmetric ciphers
  - Operations
- Error Detection
  - Error Detecting Codes
- Granularity of the code
- Frequency of checkpoints
- Results
- Conclusions and future research
Ciphers are developed to be resistant against linear and differential cryptanalysis
- Similar plain texts must lead to completely different ciphered outputs

Very few rounds are required to spread the difference over the whole block

A single bit flip can alter half the block (i.e., the block is randomly correlated to the correct output)
- Differences are fewer if error occurs at the end of the process (less rounds are computed afterwards)
Fault attacks are a very efficient technique to break a cipher
- Inject an error and collect information from the corrupted output

Most attacks directed against AES, RSA
- But also against DES and Elliptic Curves

Actual application is the critical point
- It requires physical access to the device and can be destructive; more difficult than power analysis

EDCs can help detecting a fault attack!
Objectives

- Identify the common components of block ciphers and model the behavior in response to errors
- Associate EDCs to data block and develop a code prediction rule for (possibly) each operation
- Evaluate the suitability of a code to the whole cipher (i.e., overhead and error coverage)
- Explore the way from Error Detection to Fault Tolerance
Symmetric Ciphers

- Designed to be fast and efficient
- Process block of data (8, 16 bytes)
- Different solutions exist
  - Each has its own properties (number of iterations, operations, ...): no major common characteristics
- Iterative structure simplifies design (even for detecting codes)
- Design based on confusion and diffusion principles
- We considered AES finalists together with Camellia, DES, IDEA and RC5
# Operations

<table>
<thead>
<tr>
<th>Ciphers</th>
<th>XOR</th>
<th>AND, OR</th>
<th>+, -</th>
<th>×</th>
<th>Sbox</th>
<th>Rot</th>
<th>Shift</th>
<th>Perm</th>
<th>× mod G(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Camellia</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>DES</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IDEA</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MARS</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>RC5</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RC6</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Rijndael</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Serpent</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Twofish</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

- **XOR**: Every cipher
- **AND, OR**: Camellia only
- **+**: often used
  - -: in encryption, only MARS
- **×**: slow and area-consuming
  - IDEA uses uncommon modulus
- **Rotations**: even data-dependent
- **Shift**: Serpent only
- **Permutation**: provides confusion
- **Polynomial ×**: Rijndael and TwoFish, over GF(2<sup>8</sup>)
- **S-Box**: non-linear
Operations (1/3)

- eXclusive OR: the only operation used by all the ciphers (e.g., key mixing)

- Bit-wise AND and OR: logical operations, used only by Camellia

- Shifts and rotations: even data-dependent, they pose a challenge to hardware designers
  - Shift used only by Serpent: original input has to be forwarded anyway since shift is not invertible

- Permutations: easiest way to achieve confusion (input regularities are dispersed)
Modular arithmetic operations

- Addition is often used
  - Subtraction is obviously used in decryption
  - Subtraction in encryption datapath is used only in MARS
- Multiplication is used only in RC6, MARS and IDEA
  - It is a “complex” operation, relatively slow and area-consuming
  - Idea uses modulus \((2^{16}+1)\), others use \((2^{32})\)

Polynomial multiplication over binary extension fields

- used in Rijndael and Twofish
Substitution Box:

- It is a replacement of bytes or words
- It is often the main non-linear component; only IDEA and RC5/RC6 do not use it explicitly
- It is usually implemented by means of a lookup table
  - it is usually byte-to-byte, in order to limit size
- Sometimes it can be computed on-the-fly (AES), but more often its specification is a table itself
- IDEA multiplication can be seen as a (very large) S-Box
  - (H. Raddum at Fast Software Encryption 2003)
Error Detection (1/2)

- First approach: duplication
  - Use two independent path and compare results
  - 100% hardware overhead, no additional latency

- Second approach: repeated computation
  - After the first computation, repeat the process and compare the results
  - It gives protection against temporary faults, not against permanent ones
  - No significant hardware overhead, but twice the latency

- These are generic solutions
Solutions specific to cryptographic device:
- US patent 5432848: DES tables are extended to include error codes

Exploit unused hardware (Karri et al.)
- Use decryption datapath to validate encrypted output
- It can be done at encryption level, round level or operation level
- No significant hardware overhead, if the device already supports decryption
- Latency is minimized checking at the operation level

Exploit idle units (Karri et al.)
- Use encryption functional units in idle state (RC6)
- Decryption datapath is not required
- Protection only against temporary faults
Error Detecting Codes

- High coverage with low-order errors
  - They often provide 100% coverage of single bit errors

- With high-order error, coverage depends on redundancy
  - Output code and data match randomly

- Hardware overhead smaller than duplication
  - They need a code generator, a comparator and propagation units implementing the prediction rules

- They work better when simple prediction rules are available for the whole encryption process
  - The code is generated at the beginning and it is validated at the end of the process
  - Checkpoint frequency can be increased for higher coverage
Parity Codes

- It can be computed at the byte or at the word level
  - It can be tuned from a single bit per word up to the desired redundancy level

- Parity of \( n \)-bit word is computed using \( n-1 \) XOR ports
  - Simple computation

- It intrinsically fails on even-order faults (i.e., an even number of errors)
  - It can detect all odd-order faults, when frequent checkpoints are scheduled
Residue Codes

- It is computed taking the modulo \((2^s-1)\)
  - \(s\) is the number of check bits
  - It can be computed through a weighted sum of the word bits

- Unlike parity, it does not allow using a single check bit
  - Minimum redundancy is 2 bits (residue base 3)
  - It is usually computed at the word level, to minimize overhead

- Coverage is similar for even-order and odd-order faults
Matching EDCs to Operations (1/3)

- Parity is more suited to logical operations; the prediction rules are...
  - eXclusive OR: ...the XOR of the input parities
  - Rotation: ...parity is unchanged, if the code is at the same level of the operation
  - Shifting: ...must consider bits leaving and entering the word
  - Polynomial multiplication: ...if defined over GF(2^n), it is easily predictable when one of the operands is known a priori
  - Data-dependent operations (RC5, RC6) are obviously more complex

- Addition and multiplication must consider all the carries that are required to compute the result
Residues are more suited to arithmetic operations; the prediction rules are...

- **Addition**: ...the sum of the input residues
  - but overflow needs correction!
- **Multiplication**: ...the product of the input residues
  - but most significant (and neglected) bits need a corrective term
- **Shifting**: ...it can be seen as a multiplication by a power of 2
- **eXclusive OR**: ...the sum of the input residues, but a correction term is needed

Prediction of the code after polynomial multiplication over GF(2^n) is expensive
Some operations are not suited to parity codes:

- Logical AND and OR: prediction is much more expensive than duplication
- Validate the code, protect by duplication and generate the code from scratch

Some operations are suited both to residue and parity:

- Substitution boxes: the output code is stored together with the result; the input code is used for implicit validation
- Address protection by concatenating check bits introduces a large overhead (1 additional bit doubles the table size)
  - Use custom address decoding unit to reduce the area overhead
<table>
<thead>
<tr>
<th>Operation</th>
<th>Parity Cost</th>
<th>Residue Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>XOR</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>AND, OR</td>
<td>More expensive than duplication</td>
<td></td>
</tr>
<tr>
<td>Integer +, -</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Integer Mul.</td>
<td>Expensive</td>
<td>Yes</td>
</tr>
<tr>
<td>Substitution Box</td>
<td>Yes</td>
<td>Expensive</td>
</tr>
<tr>
<td>Rotation</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Shift</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Permutation</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Polynomial Mul.</td>
<td>Yes</td>
<td>Expensive</td>
</tr>
</tbody>
</table>
EDC Granularity

- Symmetric ciphers operate on different word size (8, 16, 32 bits)
- Code granularity should not be larger than operand size
  - The code should be validated and regenerated with each operation! (e.g., substitution tables)
- Finer code adds further complexity and overhead
  - Detection rate improves
  - Prediction rule may become more complex
Choosing the Proper EDC

<table>
<thead>
<tr>
<th>Cipher</th>
<th>Suggested Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Camellia</td>
<td>Intractable by EDC</td>
</tr>
<tr>
<td>DES</td>
<td>Parity</td>
</tr>
<tr>
<td>IDEA</td>
<td>Residue, but expensive</td>
</tr>
<tr>
<td>MARS</td>
<td>Residue, but expensive</td>
</tr>
<tr>
<td>RC5</td>
<td>Parity or residue</td>
</tr>
<tr>
<td>RC6</td>
<td>Residue</td>
</tr>
<tr>
<td>Rijndael (AES)</td>
<td>Parity, per byte</td>
</tr>
<tr>
<td>Serpent</td>
<td>Parity, per byte</td>
</tr>
<tr>
<td>Twofish</td>
<td>Parity, per byte</td>
</tr>
</tbody>
</table>
Choosing the Proper EDC
(1/3)

- If operations do not allow affordable code prediction, prefer **duplication** of functional unit
  - (Camellia and logical operations AND and OR)
- If cipher uses multiplication, then use **residue** (RC6)
- If cipher uses polynomial multiplication, then use **parity** (AES, Twofish)
Some ciphers use operations suitable for different codes:

- MARS uses substitution boxes (parity) and integer multiplication (residue)
- RC5 uses addition, rotations, XORs

Using residue codes is the only (expensive) choice:

- MARS: use residue, but validate before S-Box
- RC5: both parity code and residue are affordable; the choice can be done according to the desired coverage/overhead ratio
Choosing the Proper EDC (3/3)

- IDEA uses multiplication
  - Use residue codes
  - The modulus is uncommon
    - The computation of the correction term (due to discarded bits) is complex and expensive
  - Use residues, but insert checkpoints before products

- DES is based on lookup tables
  - Expansion and S-Box work on small nibbles
    - Residue has excessive overhead
  - Round permutation does not alter word-level parity
    - Simple, but poor coverage
  - Use parity (per byte), but frequent checkpoints are required
There are three main levels:
- After whole encryption, at the end of some rounds, after inner operations
- With higher checkpoint frequency, the detection latency is lower
  - But critical path is longer, hence lower clock rate can be achieved
- Frequency affects also detection coverage
  - Error masking can be avoided by frequent checkpoints
  - Frequent checkpoints may increase the false positives
- Checkpoint must be scheduled *before* any error is completely masked by encryption process
Single-fault model:
- Any difference between the predicted and the actual code allows detecting the error.
- After fault injection, each round makes the error evolve by spreading and cancelling the differences.
- If the error is completely cancelled, the fault will not be detected.
  - If a premature cancellation may occur, a checkpoint MUST be scheduled.
- AES and RC5 models allow the single fault to reach the end of encryption.
  - Single fault is always detected.
- IDEA simulations have shown that error cancellation can occur even for single faults.
Coverage – Parity Code

RC5 - Undetected Faults - Parity Code

Undetection Percentage

# injected faults

- Word Parity - End Check
- Word Parity - Round Check
- Byte Parity - End Check
- Byte Parity - Round Check
Error detecting codes are a reasonable alternative to duplication:

- Reduced hardware overhead
- Parity and residue codes cover a wide range of cryptographic operations
- Many degrees of freedom allow to choose the desired coverage/cost tradeoff
  - Type of code
  - Granularity
  - Frequency of checkpoints
- Optimum detection rate
  - Often 100% of single errors are detected
  - Detection rate depends on the number of check bits, when multiple errors are injected
Future Research

- Develop a library of cryptographic functional units with support to error detection

- Evaluate accurate hardware and latency overhead, depending on code and checkpoint frequency

- Develop fault tolerant architectures
  - AES model allows for fault location at the byte level

- Exploit error detection as a countermeasure against fault attacks
  - Recompute the result and output only correct data
  - Stop the device/erase key memory when an attack is detected