Automated Fault Analysis of Block Cipher Implementations

Jakub Breier
Senior Cryptography Security Analyst
Underwriters Laboratories
Singapore
http://jbreier.com
Outline

- Fault Analysis in Cryptography
- Differential Fault Analysis (DFA) of Symmetric Block Ciphers
- Automation of DFA for Software Implementations
- Countermeasure Implementation
Fault Analysis in Cryptography
Physical Attacks in Cryptography

- Cryptography provides algorithms that enable secure communication in theory
- In real world, these algorithms have to be implemented on real devices:
  - software implementations - general-purpose devices
  - hardware implementations - dedicated secure hardware devices
- To evaluate security level of cryptographic implementations, it is necessary to include physical security assessment
First IC Disturbances – Cosmic Rays and Satellites

Fault Injection Techniques in Practice

Voltage Glitching

EM Pulse Injection

Laser Fault Injection

$ $ $
Why Fault Attacks?

- The best cryptanalysis of AES needs complexity of $2^{126.1}$
  - A. Bogdanov et al. Biclique cryptanalysis of the full AES, ASIACRYPT 2011.

- The best fault attack on AES needs just one faulty and one correct ciphertext pair
Differential Fault Analysis

of Symmetric Block Ciphers
Working Principle

- Attacker injects a fault in a chosen round of the algorithm to get the desired fault propagation at the end of an encryption.
- The secret key can then be determined by examining the differences between a correct and a faulty ciphertext.

E. Biham and A. Shamir: Differential fault analysis of secret key cryptosystems, CRYPTO’97.
Example – SIMON Block Cipher

Non-linear operation → exploitable by DFA

R. Beaulieu et. al. The SIMON and SPECK Families of Lightweight Block Ciphers, ePrint 2013/404.
Exploiting AND Operation by DFA

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c = a &amp; b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a’</th>
<th>b</th>
<th>c’ = a’ &amp; b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

• If the result does not change → ‘b’ is 0
• If the result changes → ‘b’ is 1
DFA - Discussion

- Different cipher families can be exploited by similar attack procedure, e.g.:
  - In SPN designs, Sbox is targeted
  - In ARX designs, modular addition is targeted
  - If a cipher uses MDS matrix, such as *MixColumns* in AES, this can be exploited for more efficient attack with lesser faults

- There is normally a trade-off between the computational complexity and the number of faults:
  - Last round attack – many faults, low complexity
  - 2\textsuperscript{nd}/3\textsuperscript{rd} last round attack – fewer faults, higher complexity
Automation of DFA for Software Implementations
Why Automation of DFA?

- All the current symmetric block ciphers have been shown vulnerable against fault attacks (especially DFA).
- The question is not whether the algorithm is secure or not, but which part of it is insecure.
- Automated methods can provide an answer fast and with minimal need of human intervention.
Tool for Automated DFA on Assembly – TADA

- The main idea – feed the assembly code to the tool and get the vulnerabilities, together with a way how to exploit them
- Static analysis module analyzes the propagation of the fault and determines what information can be extracted from known data
- SMT solver module solves the DFA equations, verifying whether an attack exists
Sample Cipher and DFG Construction

<table>
<thead>
<tr>
<th>#</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LD r0 X+</td>
</tr>
<tr>
<td>1</td>
<td>LD r1 X+</td>
</tr>
<tr>
<td>2</td>
<td>LD r2 key1+</td>
</tr>
<tr>
<td>3</td>
<td>LD r3 key1+</td>
</tr>
<tr>
<td>4</td>
<td>AND r0 r1</td>
</tr>
<tr>
<td>5</td>
<td>EOR r0 r2</td>
</tr>
<tr>
<td>6</td>
<td>EOR r1 r3</td>
</tr>
<tr>
<td>7</td>
<td>ST x+ r0</td>
</tr>
<tr>
<td>8</td>
<td>ST x+ r1</td>
</tr>
</tbody>
</table>
Properties of the DFG – Explained

- **Linear edge**
  - Node r3 (3) affects node r1 (6)

- **Non-linear edge**
  - Distance between r0 (0) and r0 (4) is 1
  - Distance between r0 (0) and x+ (7) is also 1

Legend:
- Round key node
- Known node
- Ciphertext node
Real Example – DFG of AES Implementation
TADA – Detailed Process Flow

1. **Inputs**: number of round keys (m), assembly code
2. **Create Customized Data Flow Graph**
3. **Calculate/Update Known Nodes**
4. **Found Vulnerable Instruction?**
   - **Yes**: Create DFAs, formulate SMT constraints, call Z3 SMT solver
   - **No**: Go to next instruction
5. **Is Vulnerable Instruction Exploitable?**
   - **Yes**: Output the attack details
   - **No**: Recover last m round keys, analyze updated DFG
6. **Finish (Success)**
Vulnerable Instructions

- Non-linear
- For a vulnerable instruction, each of its input nodes that is not known can be a target node or/and a vulnerable node
- A fault will be injected into the vulnerable node so that it might reveal information about the target node
- TADA creates a subgraph for each pair of target and vulnerable node
## Find Vulnerable Instruction

<table>
<thead>
<tr>
<th>#</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LD r0 X+</td>
</tr>
<tr>
<td>1</td>
<td>LD r1 X+</td>
</tr>
<tr>
<td>2</td>
<td>LD r2 key1+</td>
</tr>
<tr>
<td>3</td>
<td>LD r3 key1+</td>
</tr>
<tr>
<td>4</td>
<td>AND r0 r1</td>
</tr>
<tr>
<td>5</td>
<td>EOR r0 r2</td>
</tr>
<tr>
<td>6</td>
<td>EOR r1 r3</td>
</tr>
<tr>
<td>7</td>
<td>ST x+ r0</td>
</tr>
<tr>
<td>8</td>
<td>ST x+ r1</td>
</tr>
</tbody>
</table>

Recall that r2 (2) and r3 (3) are the key nodes.

![Diagram showing the instruction sequence with nodes and edges indicating operations like AND, EOR, and storage operations.](image)
Create DFA Equations

<table>
<thead>
<tr>
<th>Correct execution</th>
<th>Faulted execution</th>
<th>Fault mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) ( r_0(4) = r_0(0) &amp; r_1(1) )</td>
<td>(d) ( r_0(4)' = r_0(0)' &amp; r_1(1) )</td>
<td>( r_0(0)' = r_0(0) \oplus \delta )</td>
</tr>
<tr>
<td>(b) ( r_0(5) = r_0(4) \oplus r_2(2) )</td>
<td>(e) ( r_0(5)' = r_0(4)' \oplus r_2(2) )</td>
<td></td>
</tr>
<tr>
<td>(c) ( r_1(6) = r_1(1) \oplus r_3(3) )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Diagram of DFA equations]

Legend:
- Vulnerable Node
- Target Node
TADA – Detailed Process Flow

- **inputs**
  - number of round keys (m)
  - assembly code

- **create customized data flow graph**

- **calculate/update known nodes**

- **found vulnerable instruction?**
  - **yes**
    - create DFA equations
    - formulate SMT constraints
    - call Z3 SMT solver
  - **no**

- **analyze updated DFG**

- **is vulnerable instruction exploitable?**
  - **yes**
    - output the attack details
  - **no**
    - **go to next instruction**

- **recover last m round keys?**
  - **yes**
    - finish (success)
  - **no**
    - calculate/update known nodes

- **finish**
Update Known Nodes

X+ (0)
  |        |
  ld (0)   ld (1)
  |        |
 r0 (0)   r1 (1)
  |              |
and (4)   and (4)
  |              |
r0 (4)     r1 (4)

key1+ (2)
  |        |
  ld (2)   ld (3)
  |        |
r2 (2)    r3 (3)

r0 (5)
  |        |
  st (7)
  |        |
x+ (7)

r1 (6)
  |        |
  st (8)
  |        |
x+ (8)

X+ (0)
  |        |
  ld (0)   ld (1)
  |        |
r0 (0)   r1 (1)
  |              |
and (4)   and (4)
  |              |
r0 (4)     r1 (4)

key1+ (2)
  |        |
  ld (2)   ld (3)
  |        |
r2 (2)    r3 (3)

r0 (5)
  |        |
  st (7)
  |        |
x+ (7)

r1 (6)
  |        |
  st (8)
  |        |
x+ (8)
TADA – Detailed Process Flow

1. inputs
   - number of round keys (m)
   - assembly code

2. create customized data flow graph

3. calculate/update known nodes

4. found vulnerable instruction?
   - yes
     - create DFA equations
     - formulate SMT constraints
     - call Z3 SMT solver
   - no

5. go to next instruction

6. is vulnerable instruction exploitable?
   - yes
     - output the attack details
   - no
     - analyze updated DFG

7. recovered last m round keys?
   - yes
     - finish (success)
   - no
     - calculate/update known nodes
One More Iteration
## Evaluation Results

<table>
<thead>
<tr>
<th>Cipher implementation</th>
<th>SIMON</th>
<th>SPECK</th>
<th>AES</th>
<th>PRIDE</th>
</tr>
</thead>
<tbody>
<tr>
<td># of lines of code (unrolled)</td>
<td>1,272</td>
<td>663</td>
<td>2,057</td>
<td>1590</td>
</tr>
<tr>
<td># of nodes in DFG</td>
<td>1,595</td>
<td>843</td>
<td>2,060</td>
<td>1763</td>
</tr>
<tr>
<td># of edges in DFG</td>
<td>2,709</td>
<td>1,562</td>
<td>3,209</td>
<td>2586</td>
</tr>
<tr>
<td>evaluation time (min)</td>
<td>17.2</td>
<td>9.8</td>
<td>298.7</td>
<td>4.6</td>
</tr>
<tr>
<td>fault attack found</td>
<td>[TBM14]</td>
<td><strong>new</strong></td>
<td>[Gir05]</td>
<td><strong>new</strong></td>
</tr>
<tr>
<td># of known nodes before attack</td>
<td>66</td>
<td>32</td>
<td>69</td>
<td>16</td>
</tr>
<tr>
<td># of known nodes after attack</td>
<td>162</td>
<td>117</td>
<td>149</td>
<td>196</td>
</tr>
<tr>
<td># of round keys found</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>


Countermeasures

How many rounds to protect?
Standard Duplication/Triplication Countermeasure

- Popular in industrial applications
- Either area or time redundancy
- Expensive overheads
- Resources can be saved in case it is not necessary to protect the entire cipher
Countermeasure implementation based on TADA

- We know which nodes are provably exploitable by TADA
- We are now trying to find the *earliest* node possible to affect the target node, such that there are no collisions
- This information will tell us what is the earliest round where the fault can be injected
Back to the Example – with 2 rounds

How can we attack r0 (5)?
- r0 (4)
- r0 (0)
- r1 (1) \(\rightarrow\) collision

As a result, we have extended the attack to the second last round

<table>
<thead>
<tr>
<th>Target node</th>
<th>Vulnerable node</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0 (5)</td>
<td>r1 (6)</td>
</tr>
<tr>
<td>r1 (6)</td>
<td>r0 (5)</td>
</tr>
</tbody>
</table>
How Many Rounds to Protect?

<table>
<thead>
<tr>
<th>Cipher implementation</th>
<th>SIMON</th>
<th>SPECK</th>
<th>AES</th>
<th>PRIDE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Earliest round attacked</td>
<td>$R - 2$</td>
<td>$R - 3$</td>
<td>$R - 3$</td>
<td>$R - 3$</td>
</tr>
</tbody>
</table>

- Resources for countermeasures can be saved as follows:
  - SIMON – over 90% (3 out of 32 rounds)
  - SPECK – over 81% (4 out of 22 rounds)
  - AES – over 60% (4 out of 10 rounds)
  - PRIDE – over 80% (4 out of 20 rounds)
Short Recap

- All the block ciphers have been shown to be vulnerable against Differential Fault Analysis
- Automated methods can help to accurately find vulnerabilities in implementations without the need of human intervention
- Application of countermeasures can be iteratively tested until the implementation is secure
Apply It

- Next week you should:
  - Identify embedded block cipher implementations that are deployed in the field and are susceptible to fault injection attacks (e.g. in IoT devices)

- In the first three months following this presentation you should:
  - Being able to automatically analyze these implementations

- Within six months you should:
  - Have a policy for applying automated analysis for every new block cipher implementation
Resources


- Future works: http://jbreier.com/research.html

Offers a complete perspective on protecting block ciphers against fault attacks – from analysis to deployment.
Thanks for attention!

Any questions?

Jakub Breier
Underwriters Laboratories, Singapore
jbreier@jbreier.com
http://jbreier.com