Fault injection attacks on cryptographic devices and countermeasures – Part 2

Israel Koren

Department of Electrical and Computer Engineering
University of Massachusetts
Amherst, MA

Countermeasures - Examples

- Must first detect injected fault, then prevent attacker from observing erroneous output
  - Block the output (e.g., generate all zeroes output), or
  - Produce a random output misleading the attacker, or/and
  - Erase the secret key after a certain number of attacks

- 1. Active protection - use sensors to detect variations in voltage, frequency, light etc

- 2. Duplicate encryption (decryption) process (hardware or time redundancy) and compare results - injected faults transient and will manifest differently
  - Spatial duplication - redundant encryption unit or use decryption unit & compare to original plaintext
  - Temporal duplication - reuse hardware or re-execute software
  - Above techniques may incur high hardware and/or time penalty
3. Error Detecting Codes (EDCs)

- **Error-detection codes** may require less overhead but possibly have a lower coverage
  - Code generator, prediction circuits and comparator(s)

- **Common Codes (separable):**
  1. Parity codes
  2. Residue codes
  3. Error correcting codes (e.g., Hamming code)

- **Specialized codes**
  - AES - linear code for the linear part and inverse calculation for the non-linear part (S-Box)
  - A nonlinear code allowing a fault coverage vs. hardware overhead tradeoff

- **Performance and area overheads**
  - Performance 30% - 100%
  - Area 15% - 170%

---

Error Detecting Codes (EDCs)

- First generate check bits
- For each operation within encryption predict check bits
- Periodically compare predicted check bits to generated ones
- Predicting check bits for each operation - most complex step
  - Should be compared to duplication

Input text

- Check bit(s) Generator
- Operation(s)
- Check bit(s) Generator
- Intermediate or final Ciphertext
- Error
- Predicted check bits

- Predictor(s)
Example: Parity prediction for AES

- Byte-level parity is natural - a total of 16 parity bits
- ShiftRows: rotating the parity bits
- AddRoundKey: add parity bits of state to those of key
- SubBytes: Expand Sbox to 256×9 - add output parity bit; to propagate incoming errors (rather than having to check) expand to 512×9 - put incorrect parity bit for inputs with incorrect parity
- MixColumns: Expressions below where $s_{i,j}^{(7)}$ is msb of state byte $i,j$

<table>
<thead>
<tr>
<th>Transformation Input (input state matrix)</th>
<th>Transformation Result (output state matrix)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_{0,j} = p_{0,j} \oplus p_{2,j} \oplus p_{3,j} \oplus s_{0,j}^{(7)} \oplus s_{1,j}^{(7)}$</td>
<td></td>
</tr>
<tr>
<td>$p_{1,j} = p_{0,j} \oplus p_{1,j} \oplus p_{3,j} \oplus s_{1,j}^{(7)} \oplus s_{2,j}^{(7)}$</td>
<td></td>
</tr>
<tr>
<td>$p_{2,j} = p_{0,j} \oplus p_{1,j} \oplus p_{2,j} \oplus s_{2,j}^{(7)} \oplus s_{3,j}^{(7)}$</td>
<td></td>
</tr>
<tr>
<td>$p_{3,j} = p_{1,j} \oplus p_{2,j} \oplus p_{3,j} \oplus s_{3,j}^{(7)} \oplus s_{0,j}^{(7)}$</td>
<td></td>
</tr>
</tbody>
</table>

AES – Scheduling of Checks

- Comparing predicted to generated parity bits
  - After each operation
  - After each round
  - At end of encryption - smallest hardware & time overheads
  - Should not mask error indication (Error propagation matrix)
Error Coverage – Parity bits

100% coverage of single faults

![Bar chart showing percentage of undetected faults for different numbers of injected faults.]

Error Detection Overheads

<table>
<thead>
<tr>
<th>Design I (S-Box)</th>
<th>Area ($\mu m^2$)</th>
<th>Latency (ns)</th>
<th>AT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>233,095</td>
<td>8.88</td>
<td>2,069,883</td>
</tr>
<tr>
<td>w/ Error Detection</td>
<td>271,245</td>
<td>11.06</td>
<td>2,999,969</td>
</tr>
<tr>
<td>Overhead</td>
<td>+16.36%</td>
<td>+24.55%</td>
<td>+44.93%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Design II (S-Box)</th>
<th>Area ($\mu m^2$)</th>
<th>Latency (ns)</th>
<th>AT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>226,099</td>
<td>9.32</td>
<td>2,107,242</td>
</tr>
<tr>
<td>w/ Error Detection</td>
<td>259,631</td>
<td>12.27</td>
<td>3,185,672</td>
</tr>
<tr>
<td>Overhead</td>
<td>+14.83%</td>
<td>+31.62%</td>
<td>+51.18%</td>
</tr>
</tbody>
</table>

- Latency overhead is mainly due to the code comparator
  - Can be reduced by moving comparator out of the critical path
- Common design improvements can be followed
  - E.g., pipelining to hide latency
Reducing the Performance Overhead

- Apply complete temporal redundancy to AES but
- Drastically reduce the performance penalty
- Double-Data-Rate (DDR) technique
- Perform the two encryptions rounds (that would be compared) during the same clock cycle
  - Use rising and falling clock edge
- Lower maximum clock frequency
  - No penalty if embedded in a slow system
- Detection relies on the two computations not affected by the same fault which can be a multi-cycle one
  - Authors claim: small percentage (~6%) of undetected faults; goes up to 39% for 6-cycle faults

Maistri and Leveugle, 2008

EDCs for other Block Ciphers

- Other ciphers use different basic operations, e.g.,
  - Bit-oriented operations (DES)
  - Modular arithmetic with unusual modulus (IDEA)
- Determine the "best" EDC for a given cipher, for example:

<table>
<thead>
<tr>
<th>Cipher</th>
<th>Suggested Code</th>
</tr>
</thead>
<tbody>
<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>Twofish</td>
<td>Parity, per byte</td>
</tr>
</tbody>
</table>
Detection Coverage - RC5 (Residue)

Protecting RSA

- Randomized multiplicative masking - use random integers \( r_1 \) and \( r_2 \)
  \[
  M_p^* = S^{d_p} \mod (p \cdot r_1) \quad M_q^* = S^{d_q} \mod (q \cdot r_2)
  
  M_1 = S^{d_p \mod \phi(r_1)} \mod r_1 \
  M_2 = S^{d_q \mod \phi(r_2)} \mod r_2
  
  \text{If} \quad M_1 = M_p^* \mod r_1 \quad \text{and} \quad M_2 = M_q^* \mod r_2
  \]
  \[
  \text{output} \quad M = CRT(M_p^*, M_q^*) \quad \text{else} \quad \text{Error detected}
  \]

- A fault injected during the CRT combination not detected

- Another option: Use Residue codes
  - Fits modular arithmetic
  - The result residue check bits of any operation can be easily obtained from the check bits of the operands

\( Shamir, 1999 \)
Residue code for RSA: Overheads

<table>
<thead>
<tr>
<th>Key Length (bits)</th>
<th>Global Overheads</th>
<th>Area Memory Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>768</td>
<td>+17.8%</td>
<td>+14.3%</td>
</tr>
<tr>
<td>1024</td>
<td>+17.8%</td>
<td>+11.1%</td>
</tr>
<tr>
<td>1536</td>
<td>+17.8%</td>
<td>+7.7%</td>
</tr>
<tr>
<td>2048</td>
<td>+17.8%</td>
<td>+5.9%</td>
</tr>
</tbody>
</table>

The residue modulus is $2^{16} - 1$

The aliasing probability (i.e., error not detected) is $\approx 2^{-16}$

Countermeasures - passive SCAs

Approaches

- Since side channel attacks leak information about the computation on an unintended side channel, the countermeasures focus on two aspects:
  - **Refactoring the algorithm**: These countermeasures try to rearrange the algorithm to avoid the leakage on a side channel
  - **Eliminate the information**: These countermeasures try to remove the correlation between the ongoing computation and the side channel measurements
  - **Suppress the side channel**: These countermeasures try to physically suppress the side channel

- Usually one of the countermeasures is enough to avoid the leakage on a channel, but due to the large number of side channels, more than one is usually in place (e.g. one for SPA, one for DPA, one for faults)
### Countermeasures - passive SCAs

**SPA mitigation**

- Simple Power analysis relies on dependencies of the control flow on the key values
- **Solution:** remove conditional statements (if constructs) turning them into predicated instructions
- *Predicated Instructions* are an architectural feature which allows the retirement of the result of an instruction depending on a particular condition (e.g. add together two register only if a third one is zero)
- ARM and Power ISAs support predicated instructions natively and have been doing so for a long time
- For architectures not supporting predicated instruction, it is possible to rewrite the conditional statements into predicated constructs through arithmetic masking

### Countermeasures - passive SCAs

**SPA mitigation**

<table>
<thead>
<tr>
<th>Plain code</th>
<th>ARM-Style predicated instructions</th>
<th>SW predication code</th>
</tr>
</thead>
<tbody>
<tr>
<td>if(R1==0)</td>
<td>and R1 R1 R1; addeq R2 R2 R3</td>
<td>R4=(R1!=0)*0xFFFFFFF</td>
</tr>
<tr>
<td>R2=R2+R3;</td>
<td></td>
<td>R5=(R1==0)*0xFFFFFFF</td>
</tr>
</tbody>
</table>

*Assembly translation:*

- and R1 R1 R1;
- jne after;
- addeq R2 R2 R3;
- after: rest of the code

Predicated instructions always have the same control flow, so no SPA leakage occurs
Countermeasures - passive SCAs

DPA mitigation
- Mitigating DPA can be done through the following ways:
  - **Hiding the sensitive operation in time**: introduce non-determinism in the time instant when the sensitive operation is performed
  - **Avoiding the knowledge of input values**: introduce so called masks to prevent the attacker from making a prediction on an intermediate value
  - **Design a circuit with data independent power consumption**: alter the logic of the circuit to remove data dependencies from the power consumption
  - **Continuously change the algorithm implementation**: employing different (semantically equivalent) variants of the algorithm stops the attacker from making a model of the power consumption

Countermeasures - passive SCAs

Hiding in software
- The typical method to perform hiding in software is represented by random delays inserted in the algorithm
- The simplest way to insert a random delay is a loop of nop operations where the number of iterations is picked at random each time (f.i. via an on-die hardware RNG) keeping it within a reasonable limit
- The insertion of random delays misaligns the time instants where the sensitive operations happen effectively lowering the correlation of the instantaneous power consumption
- However hiding **does not** remove the correlation: as the possible delays inserted are finite, an attacker collecting a large amount of traces will still be able to find a correlation with its power model and the consumption of the circuit of all the time instants where the sensitive operation has been performed
Countermeasures - passive SCAs

Hiding in software - Counterattacks and strategies

- If NOP instructions are used as delays, the attacker may be able to identify them through SPA like techniques and remove them from the measurements.
- The most practical way to counteract this is to use actual idempotent instructions (e.g. AND R1 R1 R1) as padding instead of NOPs.
- To compensate for the introduced delay, the attacker may compute the correlation on the sum of the power consumption of $n$ clock cycles:
  - The correlation of the sum of the power consumptions with the same model is $\frac{\beta_{\text{single}}}{\sqrt{n}}$, where $\beta_{\text{single}}$ is the correlation of the attack on the correct time instant.
  - Computing the correlation on the raw measurements yields a correlation of $\frac{\beta_{\text{single}}}{\sqrt{n}}$ only, as only one over $n$ measurements has the sensitive operation performed in the same time instant.

*Assume a maximum delay of $n$ clock cycles.

Countermeasures - passive SCAs

Hiding in hardware

- Performing hiding in hardware is usually done via employing a randomized clock for the chip:
  - Small disturbance on the clock generator: the randomized clock is obtained via changing by small amounts (<10%) the frequency generated by the clock, resulting in a global misalignment of the measured operations.
  - Different clock generators: the clock tree of the chip is hooked to different clock generators, which are switched during the computations: as the circuit is very small the performance penalty is acceptable. Typically the different frequencies are obtained splitting the main clock generator one.
- These techniques achieve an effect similar to the introduction of dummy instructions (i.e. hide the operation on the time axis) but, due to the higher variances in the introduced delays, they are more difficult to counteract.
Countermeasures - passive SCAs

**Masking in software**

- The key principle of adding masks to the computation is to prevent the attacker from knowing the actual input values to the key addition operations.
- **Key idea:** add a random quantity to both the input and the key via xor so that the key addition will remove it implicitly.
- The result of this countermeasure is that the key addition is performed as \((p \oplus r) \oplus (k \oplus r)\) and the attacker is no longer able to predict the switching activity of any of the xor operations as the value of \(r\) changes every time and he does not know it.
- Different masking schemes have been proposed employing different operations depending on the cipher but the aforementioned one is still the most common one for computational cost reasons.

Countermeasures - passive SCAs

**Masking in hardware and attacks against masking**

- Masking in hardware can be directly performed at single logic gate level, adding the random quantities right before the sensitive operations are performed.
- The time overhead of masking in software is translated in both a time (extra delays on the computation path) and area (extra logic gates and wires) on the masked circuits.
- The main attack strategy against masking implies the construction of a power model dependent predicting the combination (f.i. sum/difference) of the power consumption of all the operations involved in the masking. These attacks are known as high order DPA.
- The key idea is that the attacker needs to devise a model which is independent from the random values involved through combining the predicted power consumptions.
Countermeasures - passive SCAs

Data independent power consumption - Differential logic styles

- A way to hide the correlation in power consumption is to employ complementary logic styles (once employed for fault tolerance)
- Complementary logic styles (such as WDDL) employ two signals to represent the value of one bit, encoding a value \( x \) as complementary values on the two wires (i.e. storing \( x \) and \( \bar{x} \))
- All the computations are performed for both the correct and complementary wire, thus resulting in a constant (in terms of number of logic gates) switching activity during the computation
- The net effect is to remove the data dependency from the number of logic gate switches
- Differential logics can be attacked, exploiting the small differences in wiring and size of the logic gates, but the attack is significantly more difficult to perform

Countermeasures - passive SCAs

Data independent power consumption - Current mode logic styles

- Current mode logic styles are one of the oldest logic styles and rely on representing ones and zeros as different current levels on wires
- The switching is performed through changing the part of the circuit in which the current flows: the transitions are extremely fast
- Since it is only a current flow being redirected on different wires which computes the result CM logic styles have a perfectly flat dynamic power consumption: DPA is practically impossible on CM
- **Drawbacks:**
  - CM logic continuously draws power: the global power consumption of the circuit is greater than \( 5 \times \) w.r.t. its CMOS equivalent
  - CM logic cannot be embedded in a voltage mode logic circuit without proper voltage-current converters at the boundary of the CM region
Countermeasures - passive SCAs

Hindering the modelling - SW - Code morphing

- An effective way to impede SCA is to continuously change the way the computation of a cryptographic primitive is performed
- **Key Idea**: there are (countably) infinite different ways to perform the same computation which are semantically equivalent
- If the procedure computing the algorithm is changed every single time, the attacker does never collect more than a single measurement fitting the same power model: modify the executable code each time!
- To increase the number of possible variants, it is possible to do per-instruction substitution with semantically equivalent alternatives
- Bonus point: all the previous software countermeasures are different, semantically equivalent ways to compute a part of a cipher ⇒ they can be used as variants of the code, exploiting their advantages

\[\text{\textsuperscript{a}Think at all the possible bivariate logic expressions which compute a XOR}\]

---

Fault Detection not always sufficient

- **Example - RSA**
  - Can be protected against fault injection using residue check or encrypting M & comparing to S
  - Still vulnerable to power analysis - more power consumed if \( d_i = 1 \)
  - Modified to use same power + fault detection
  - Inject fault in calculation of \( b \) - if correct M, one bit of \( d \) is deduced
  - Faulty result not needed!
  - "Shut down" if several faults detected!

\[
\text{Decryption Algorithm}._1(S, N, (d_{n-1}, d_{n-2}, \ldots, d_0))
\]

begin

\[
a = S \\
\text{for } i \text{ from } n - 2 \text{ to } 0 \text{ do} \\
a = a^2 \mod N \\
\text{if } d_i = 1 \text{ then } a = S \cdot a \mod N \\
\text{end}
\]

M = a

\[
\text{Decryption Algorithm}._2(S, N, (d_{n-1}, d_{n-2}, \ldots, d_0))
\]

begin

\[
a = S \\
\text{for } i \text{ from } n - 2 \text{ to } 0 \text{ do} \\
a = a^2 \mod N \\
b = S \cdot a \mod N \\
\text{if } d_i = 1 \text{ then } a = b \text{ else } a = a \\
\text{end}
\]

if (no error has been detected) then M = a

end

*Yen and Joye, 2000*
Montgomery-step Algorithm

- Intermediate values of $a$ & $b$ are used and an injected fault will be detected.
- Provides another way to detect faults: $a$ & $b$ must be of form $(M,SM)$.
- Checking this relation detects most faults except - bits of $d$ or number of loop iterations - these must be checked separately (e.g., EDC).

Joye and Yen, 2002

Safe-error Resistant Algorithm

- Avoid decision tests.
- Check errors in $d$ and loop counter.
- Error detected if $a_i \neq S \cdot a_0 \mod N$.

Giraud, 2005
AES - Successful attack even if faults detected

- Provide all-zero input to AES encryption
- An initial round key is added (XOR): \( \text{state} = \text{key} \)
- Before SubBytes inject a stuck-at-0 fault into bit \( j \)
  - If result is correct then bit \( j \) of key is 0
- Even duplicating the encryption will not help - it does not matter whether the fault was detected or not
  - Unless the number of allowed faults is limited
- Attack is complicated - exact timing and precise location of fault and fault type
  - If strict timing and location are not practical - repeating the experiment many times will allow extracting the secret key
- Attack can be done if a byte (or several bytes) are reset to 0
  - If key byte \( j \) is reset to 0, perform 256 encryptions with byte \( j \) of message assuming values 0 to 255 - the one that matches the faulty ciphertext reveals byte \( j \) of key (a.k.a Collision Fault attack)  

Blomer and Seifert, 2003

Combining Passive and Active Attacks

- Many current cryptographic devices include separate countermeasures against power attacks and fault injection attacks
- Two new questions/challenges
  - Can a countermeasure against one type of attacks make the other one simpler to execute?
  - What happens if the attacker uses a combination of passive and active attacks?
Can the presence of error checking circuitry make a power attack simpler?

- Correlation Power Analysis (CPA)
- Based on linear relationship between power and Hamming weight of data processed
- AES implementation with no error check circuit
- Correct key distinguishable after 160 traces

Differential Power Analysis

1. Collect many ciphertexts and the power traces for the last round
2. Guess a byte of the final round key
3. Calculate the target byte based on the guess
4. Select one bit of the target byte B
5. Divide the power traces into 2 sets: those for B=1 and those for B=0
6. Calculate the averages of the 2 sets and the difference between the averages
7. If the average depends on B there will be a spike in the data indicating correlation
8. If the guess (of the key byte) is correct the power should depend on the value of B
Correlation Power analysis

* Construct a power model to estimate the power for every value of one byte of the last round key

* Calculate the correlation between the estimated power and the power traces

* The highest correlation indicates the correct key byte

Power Attacks in the presence of error checking

- AES with a parity bit per byte

- Correct key distinguishable after 130 traces

- For residue mod 3 code - correct key distinguishable after 100 traces
Fault Injections that make DPA feasible

- Circuit techniques to protect HW implementations against DPA have been developed
  - Specially designed balanced gates for which the power consumption is equal for all data
- Faults injected in the “balancing” part of the circuit will imbalance it but will not cause a logical error
  - Can not be detected by any redundancy scheme
  - If 4 out of 137 gates were made imbalanced (through fault injection) the protected circuit was as vulnerable to DPA as an unprotected circuit
- A possible countermeasure is adding differential current comparators that would detect the imbalances

Kulikowski, Karpovsky and Taubin, 2006

Protecting RSA against DPA and Faults

- The “fault resistant” algorithm is multiplicatively blinded by a random number $r$ making it DPA resistant as well
- The increase in execution time vs. Algorithm_4 is about 45%
- A fault is not detected if injected during the computation of $a_2 = a_2^2 \mod N$
- Modified algorithm developed in 2008 by Kim & Quisquater

Algorithm_5 $(S, N, d_{n-1}, ..., d_0)$

Select a random number $r$

$a_0 = r; \quad a_1 = rS; \quad a_2 = r^{-1}$

for i from n-1 to 1 do

$\quad a_{d_i} = a_{d_i}^2 \mod N$
$\quad a_{d_i} = a_{d_i}^2 \mod N$
$\quad a_2 = a_2^2 \mod N$

$a_i = a_i a_0 \mod N$
$a_0 = a_0^2 \mod N$
$a_2 = a_2^2 \mod N$

return $(a_2 a_0, a_2 a_1)$

Fumaroli and Vigilant, 2006
Protecting AES against DPA and Faults

- To protect an fault-resistant AES implementation against DPA one can mask all 16 data bytes with a random number $r_1$ and all key bytes with $r_2$
- Injecting faults in the first XOR operation using the Collision Fault Attack allowed recovering the key with 112 fault injections (Amiel, Clavier and Tunstal, 2006)
- To protect against the above use 16 different random masks instead of one
  - Can not use S-Boxes (would need 16 tables for each value of $r$)
  - This implementation increased the area by ~40% and the latency by ~250%

AES DPA and DFA – Recent results

- Recently (2010) a modified collision fault attack was developed requiring ~1568 faults to be injected
- One countermeasure that has been suggested - duplicate the AES rounds that are exposed to the attack
  - The duplicated rounds should be performed with two different masks
  - Bytes should be processed in a random order
  - The first 3 and last 3 rounds duplicated leading to
  - Latency overhead of about 400% vs. the previous 250%
Conclusions

- The need to protect cryptographic devices against passive and active side channel attacks is well established
  - A strong cipher is insufficient
  - Hardware and/or software aids must be included in the design to counteract side channel attacks
  - Current techniques incur a high overhead

- Interactions among different side channel attacks must be further investigated
  - Separately protecting against individual side channel attacks is insufficient
  - The currently known techniques to counter both passive and active attacks have a high overhead