Software Countermeasures Against \textbf{DPA} Attacks: Masking vs. Dual-Rail with Precharge Logic
(for automatic and formally proven insertion)

Pablo Rauzy
rauzy@enst.fr
with Sylvain Guilley and Jean-Luc Danger

\textbf{Telecom} ParisTech
LTCI / COMELEC / SEN

March 8, 2013
Cryptosystems’ software should be bug-free and rely as little as possible on hand-written code for critical parts.

We need formally verified tools such as a certified compiler which would automatically add the necessary protections against SCA.
Countermeasures can be classified in two categories [MOP06]:

- those that use randomness to make the leakage statistically independent from sensitive data (like *masking* [CG00]);
- those that make the leakage indistinguishable (like *dual-rail with precharge logic* [HDD11] (*DPL*)).

Automated masking has already been explored [MOPT12] but most efforts have yet to be done for *DPL*. 
Countermeasures

Masking

- Needs randomness (hard to formalize).
- Assumes shares are not interfering neither logically (opcode’s effect depending on previous ones) nor physically (glitches, cross-coupling).
- Assumes the data and operations in the algorithm are embedded within a group (for instance $(\mathbb{F}_2^n, \oplus)$ for Boolean additive masking).
- Protection depends on the linearity of the operation.
- Masking S-Boxes is difficult in general [PR07].
Assumes that (at least) two equivalent (in terms of leakage) resources exist.

Protection depends less on the algorithm.

Algorithms can be bitsliced [Bih97], which leads to a simple Turing machine like model that operates at the bit level.

Since it seems easier, we chose to start working on automatic insertion of countermeasures with DPL.
We want to be able to formally prove two properties on automatically applied countermeasures.

- The semantics of the code must not be unaltered by the transformation that adds countermeasures (correctness).
  ⇒ Exactly what a formally proven compiler does.
- The countermeasure must be efficient (security).
  ⇒ We need formal models of the possible leakages, and then use them to prove that the obtained code is protected against those leakages.
The necessity of both these properties is obvious. Moreover such **proofs will enable optimizations**. Indeed, an optimization of a protected piece of code should not damage the protection. Formally proven code transformations and security can guarantee the validity of an optimization.

The formal model in which we will work also has to be explicit about its hypotheses, in particular those we have to make about the hardware. This has the effect of yielding an exhaustive **list of assumptions on the hardware** that will have to be tested in lab.
source

bitslice compiler

DPL bitwise macros

abstract CPU

CPU
The abstract CPU

code (read only)

registers

finite state machine

memory

PC
We start with the formal study of a DPL implementation of PRESENT.

As leakage model, we chose the Hamming distance of updates of the memory write, read, and address buses, and of the the registers write, read, and address buses.
What we currently have

Abstract CPU

- Able to keep track of information necessary to compute leakages during evaluation.

⇒ Enable to experimental tests.
The “symbolic abstract CPU” is the same as the “abstract CPU” but does symbolic evaluation.

No assumptions are made on the initial values of the bits of the key and the plaintext (values in registers, memory and buses are sets of possible values).

We can compute all the possible leakages and verify if there actually is only one.
What we currently have

Symbolic abstract CPU

- The “symbolic abstract CPU” is the same as the “abstract CPU” but does symbolic evaluation.
- No assumptions are made on the initial values of the bits of the key and the plaintext (values in registers, memory and buses are sets of possible values).

⇒ We can compute all the possible leakages and verify if there actually is only one.
What we currently have
Symbolic abstract CPU

Example: PRESENT sbox

\[ y_0y_1y_2y_3 = \text{sbox}(x_0x_1x_2x_3) \]

;;;; bitwise PRESENT sbox in 14 operations \[\text{[CHM11]}\]

xor r0 @2 @1 ; t0 = x_2 \text{ ^ } x_1
and r1 @1 r0 ; t1 = x_1 \text{ & } t0
xor r2 @0 r1 ; t2 = x_0 \text{ ^ } t1
xor @7 @3 r2 ; y_3 = x_3 \text{ ^ } t2
and r1 r0 r2 ; t1 = t0 \text{ & } t2
xor r0 r0 @7 ; t0 = t0 \text{ ^ } y_3
xor r1 r1 @1 ; t1 = t1 \text{ ^ } x_1
orr r3 @3 r1 ; t3 = x_3 \text{ | } t1
xor @6 r0 r3 ; y_2 = t0 \text{ ^ } t3
xor r3 @3 #1 ; t3 = \sim x_3
xor r1 r1 r3 ; t1 = t1 \text{ ^ } t3
xor @4 @6 r1 ; y_0 = y_2 \text{ ^ } t1
orr r1 r1 r0 ; t1 = t1 \text{ | } t0
xor @5 r2 r1 ; y_1 = t2 \text{ ^ } t1
What we currently have / Symbolic abstract CPU

Example: **PRESENT** sbox leakages trace

<table>
<thead>
<tr>
<th>instructions</th>
<th>ri</th>
<th>ro</th>
<th>r@</th>
<th>mi</th>
<th>mo</th>
<th>m@</th>
</tr>
</thead>
<tbody>
<tr>
<td>xor r0 @2 @1</td>
<td>0,1</td>
<td>0</td>
<td>0</td>
<td>0,1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>and r1 @1 r0</td>
<td>0,1</td>
<td>0,1</td>
<td>0,1</td>
<td>0,1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>xor r2 @0 r1</td>
<td>0,1</td>
<td>0,1</td>
<td>0,1</td>
<td>0,1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>xor @7 @3 r2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0,1</td>
<td>2,3</td>
<td></td>
</tr>
<tr>
<td>and r1 r0 r2</td>
<td>0,1</td>
<td>0,1</td>
<td>0,1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xor r0 r0 @7</td>
<td>0,1</td>
<td>0,1</td>
<td>0,1</td>
<td>0,1</td>
<td>0,1</td>
<td></td>
</tr>
<tr>
<td>xor r1 r1 @1</td>
<td>0,1</td>
<td>1</td>
<td>1</td>
<td>0,1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>orr r3 @3 r1</td>
<td>0,1</td>
<td>0,1</td>
<td>1,2</td>
<td>0,1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>xor @6 r0 r3</td>
<td>0,1,2</td>
<td>0,2</td>
<td>0,1</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>xor r3 @3 #1</td>
<td>0,1</td>
<td>0,2</td>
<td>2</td>
<td>0,1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>xor r1 r1 r3</td>
<td>0,1</td>
<td>0,1</td>
<td>1,2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xor @4 @6 r1</td>
<td>0,1</td>
<td>1</td>
<td>0,1</td>
<td>0,1</td>
<td>0,1</td>
<td></td>
</tr>
<tr>
<td>orr r1 r1 r0</td>
<td>0,1</td>
<td>0,1</td>
<td>0,1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xor @5 r2 r1</td>
<td>0,1</td>
<td>0,1</td>
<td>0,1</td>
<td>0,1</td>
<td>0,1</td>
<td></td>
</tr>
</tbody>
</table>
Eli Biham.
A Fast New DES Implementation in Software.

Jean-Sébastien Coron and Louis Goubin.
On Boolean and Arithmetic Masking against Differential Power Analysis.

Nicolas Courtois, Daniel Hulme, and Theodosis Mourouzis.
Solving Circuit Optimisation Problems in Cryptography and Cryptanalysis.

Philippe Hoogvorst, Guillaume Duc, and Jean-Luc Danger.
Software Implementation of Dual-Rail Representation.
In *COSADE*, 2011.

Stefan Mangard, Elisabeth Oswald, and Thomas Popp.
*Power Analysis Attacks: Revealing the Secrets of Smart Cards*.

Andrew Moss, Elisabeth Oswald, Dan Page, and Michael Tunstall.
Compiler Assisted Masking.

Emmanuel Prouff and Matthieu Rivain.
That’s it. Questions?

Automatic insertion of countermeasures

Countermeasures
  - Masking
  - Dual-Rail with Precharge Logic (DPL)

Formally proven countermeasures
  - Bonus

Master plan

The abstract CPU

What we currently have
  - Abstract CPU
  - Symbolic abstract CPU
    - Example: PRESENT sbox
    - Example: PRESENT sbox leakages trace