Software Countermeasures Against 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

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.

$\Rightarrow$ 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.

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].

Dual-Rail with Precharge Logic (DPL)

- Assumes that (at least) two equivalent (in term of leakage) resources exists.
- 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 countermeasure with DPL.

We want to be able to formally prove two properties on automatically applied countermeasures.

- The semantic of the code must not be unaltered by the transformation that adds countermeasure (correctness).

$\Rightarrow$ Exactly what a formally proven compiler does.

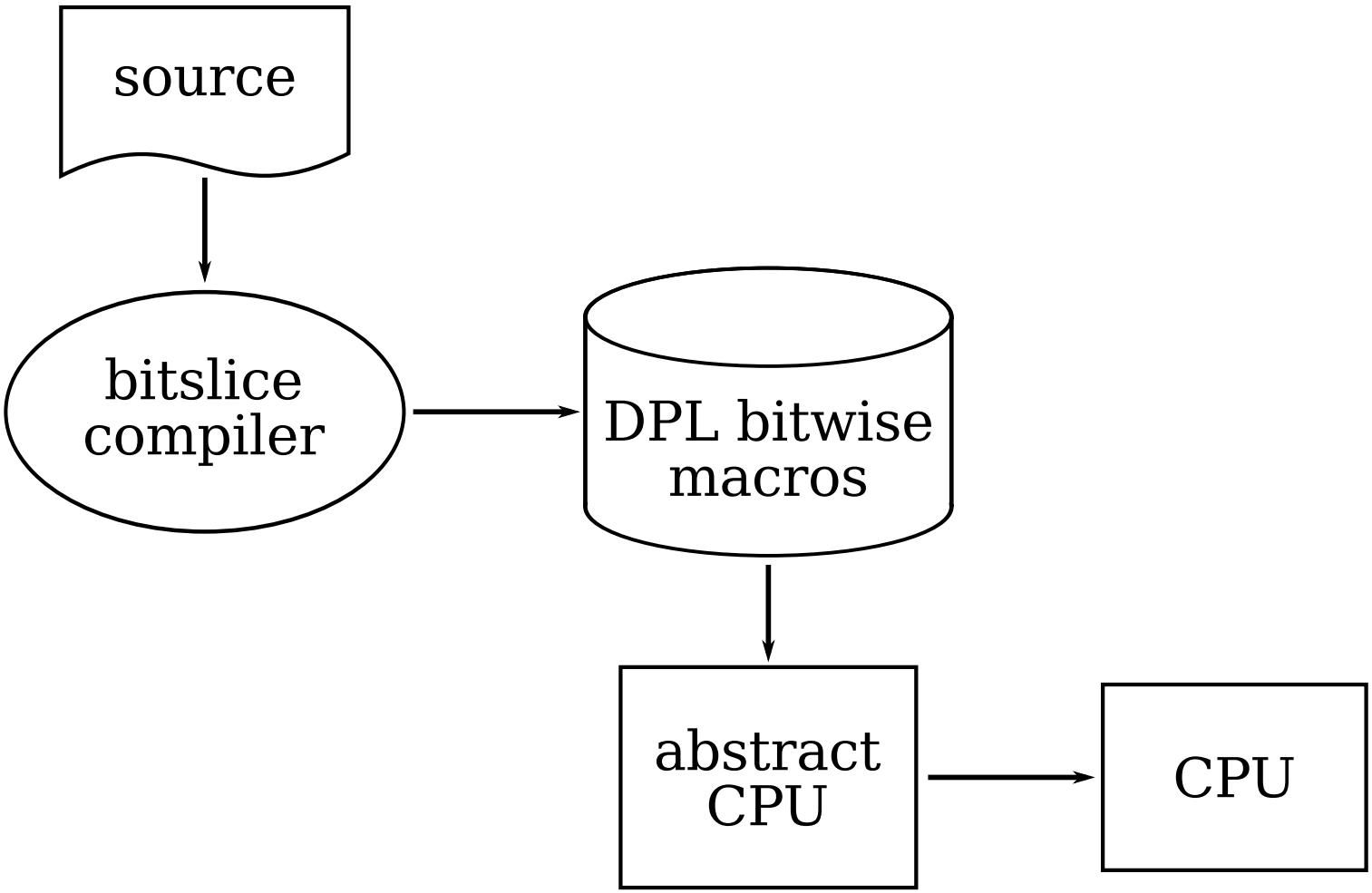
- The countermeasure must be efficient (security).

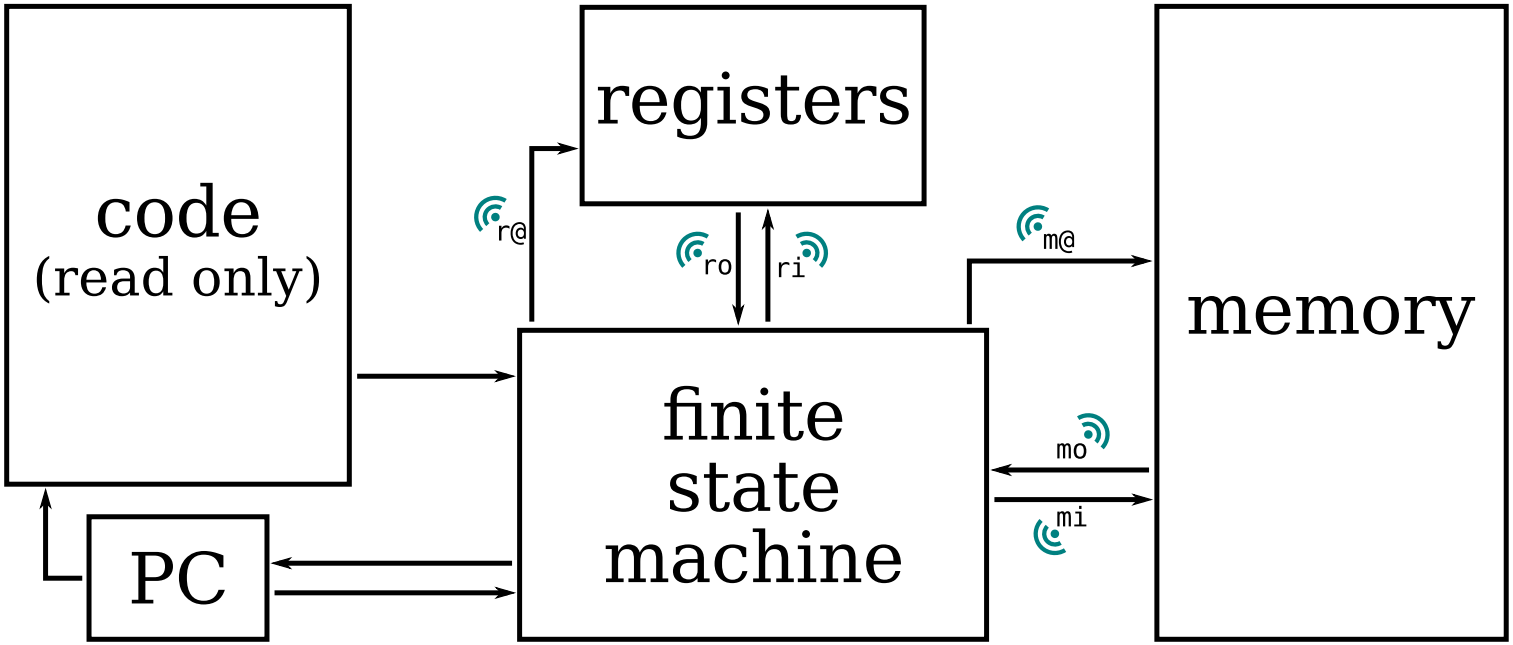
$\Rightarrow$ We need formal model of the possible leakages, and then use them to prove that the obtained code is protected against those leakages.

Bonus

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 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 of the hardware** that will have to be tested in lab.


source
bitslice compiler
DPL bitwise macros
abstract CPU
CPU


code
(read only)
registers
r@
ro
ri
m@
memory
finite
state
machine
mo
mi
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.

Abstract CPU

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

$\Rightarrow$ Enable to experimental tests on multiple exemple.

Symbolic abstract CPU

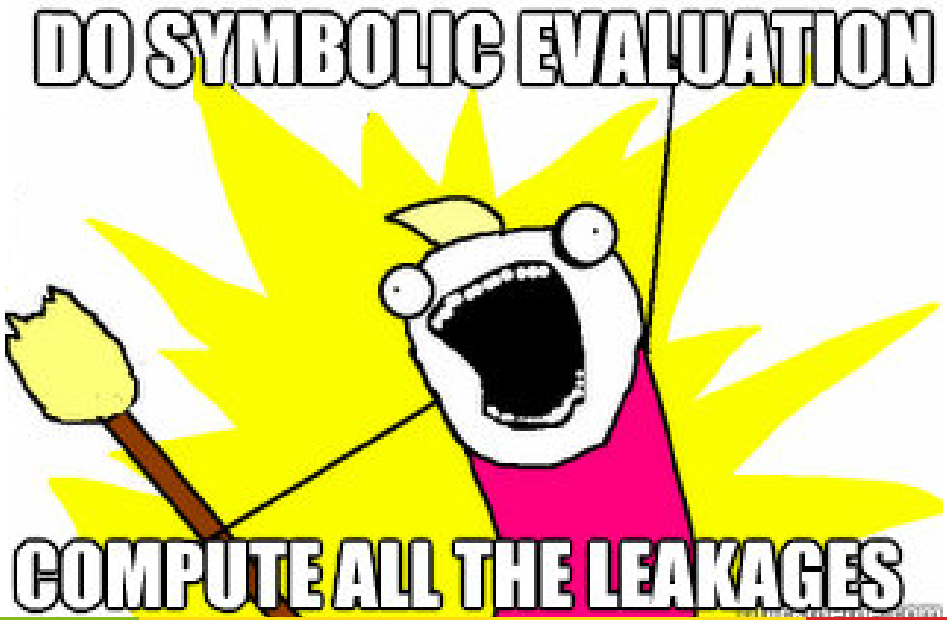
- The “symbolic abstract CPU” is the same as the “abstract CPU” but does symbolic evaluation.
- No assumptions are made on the initial values of the bits of the key and the plaintext (values in registers, memory and buses are sets of possible values).

$\Rightarrow$ We can compute all the possible leakages and verify if there actually is only one.

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).

$\Rightarrow$ We can compute all the possible leakages and verify if there actually is only one.



Example: PRESENT sbox

$$y_0 y_1 y_2 y_3 = sbox(x_0 x_1 x_2 x_3)$$

```
;;; bitwise PRESENT sbox in 14 operations [CHM11]
xor  r0  @2  @1  ; t0 = x2 ^ x1
and  r1  @1  r0  ; t1 = x1 & t0
xor  r2  @0  r1  ; t2 = x0 ^ t1
xor  @7  @3  r2  ; y3 = x3 ^ t2
and  r1  r0  r2  ; t1 = t0 & t2
xor  r0  r0  @7  ; t0 = t0 ^ y3
xor  r1  r1  @1  ; t1 = t1 ^ x1
orr  r3  @3  r1  ; t3 = x3 | t1
xor  @6  r0  r3  ; y2 = t0 ^ t3
xor  r3  @3  #1  ; t3 = ~x3
xor  r1  r1  r3  ; t1 = t1 ^ t3
xor  @4  @6  r1  ; y0 = y2 ^ t1
orr  r1  r1  r0  ; t1 = t1 | t0
xor  @5  r2  r1  ; y1 = t2 ^ t1
```

Example: PRESENT sbox leakages trace

instructions	ri	ro	r@	mi	mo	m@
`xor r0 @2 @1`	0,1		0		0,1	1
`and r1 @1 r0`	0,1	0,1	0,1		0,1	0
`xor r2 @0 r1`	0,1	0,1	0,1		0,1	1
`xor @7 @3 r2`		0	0		0,1	2,3
`and r1 r0 r2`	0,1	0,1	0,1			
`xor r0 r0 @7`	0,1	0,1	0,1		0,1	0,1
`xor r1 r1 @1`	0,1	1	1		0,1	2
`orr r3 @3 r1`	0,1	0,1	1,2		0,1	1
`xor @6 r0 r3`		0,1,2	0,2	0,1		0
`xor r3 @3 #1`	0,1	0,2	2		0,1	0
`xor r1 r1 r3`	0,1	0,1	1,2			
`xor @4 @6 r1`		0,1	1	0,1	0,1	0,1
`orr r1 r1 r0`	0,1	0,1	0,1			
`xor @5 r2 r1`		0,1	0,1	0,1		0,1

Bibliography

Eli Biham.

A Fast New DES Implementation in Software.
In *FSE*, pages 260–272, 1997.

Jean-Sébastien Coron and Louis Goubin.

On Boolean and Arithmetic Masking against Differential Power Analysis.
In *CHES*, volume 1965 of *Lecture Notes in Computer Science*, pages 231–237. Springer, August 17-18 2000.

Nicolas Courtois, Daniel Hulme, and Theodosis Mourouzis.

Solving Circuit Optimisation Problems in Cryptography and Cryptanalysis.
IACR Cryptology ePrint Archive, 2011:475, 2011.

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.
`http://www.springer.com/Springer`, December 2006.
ISBN 0-387-30857-1, `http://www.dpabook.org/`.

Andrew Moss, Elisabeth Oswald, Dan Page, and Michael Tunstall.

Compiler Assisted Masking.
In *CHES*, pages 58–75, 2012.

Emmanuel Prouff and Matthieu Rivain.

A Generic Method for Secure SBox Implementation.
In *WISA*, pages 227–244, 2007.

That was 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