Formally Proved Security of Assembly Code Against Power Analysis:
A Case Study on Balanced Logic

Pablo Rauzy
rauzy@enst.fr
pablo.rauzy.name

Sylvain Guilley
guilley@enst.fr
perso.enst.fr/~guilley

Zakaria Najm
znajm@enst.fr

Telecom ParisTech
CNRS LTCI / COMELEC / SEN

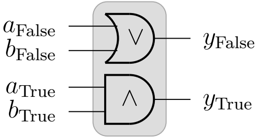
PROOFS 2014

Third Workshop on Security Proofs for Embedded Systems

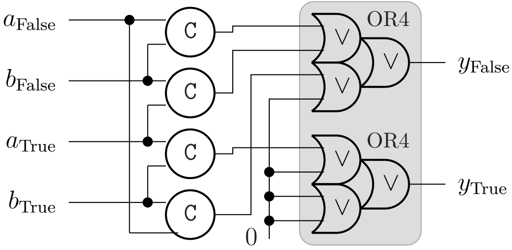
September 27, 2014 @ Busan, Korea

IACR ePrint 2013/554

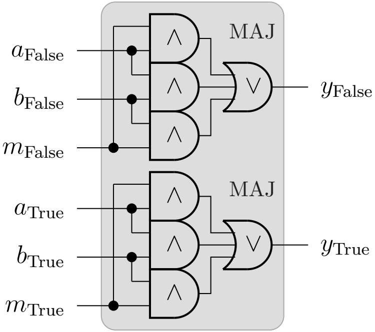
WDDL:



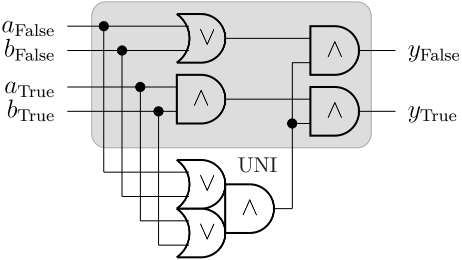
SecLib:

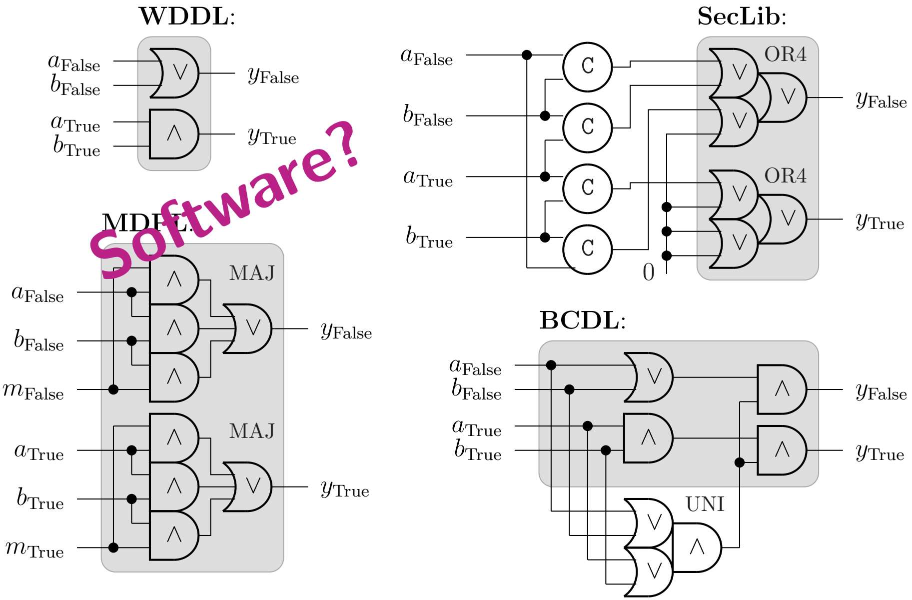


MDPL:

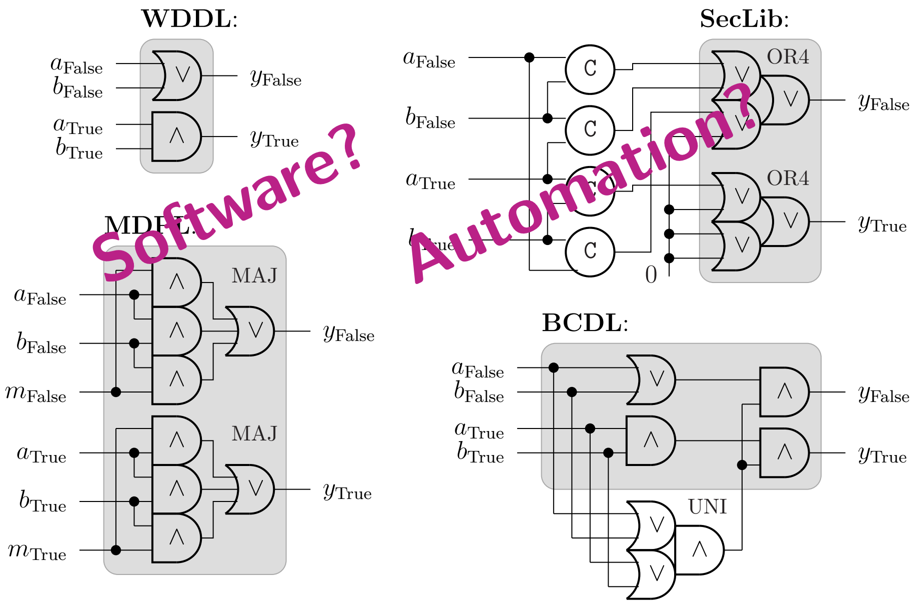


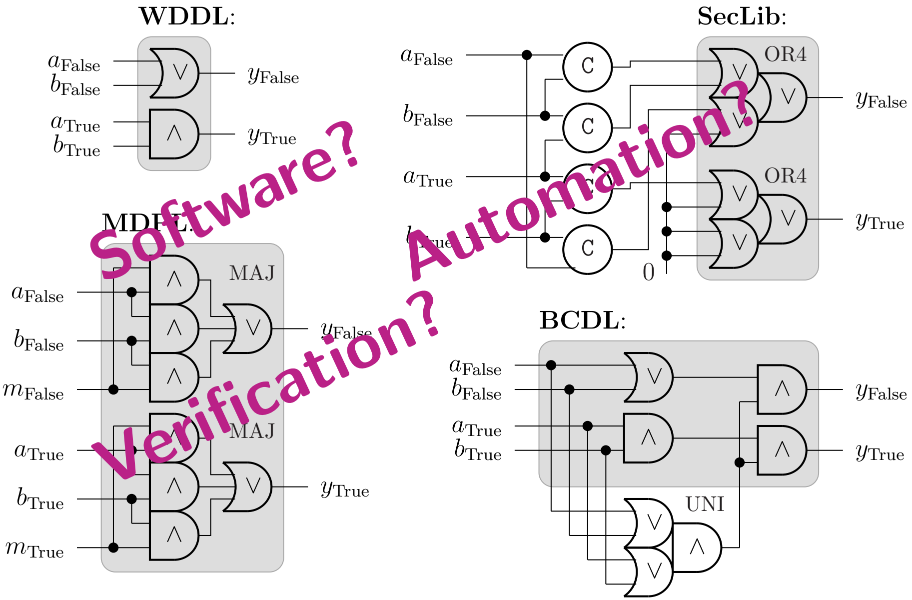
BCDL:

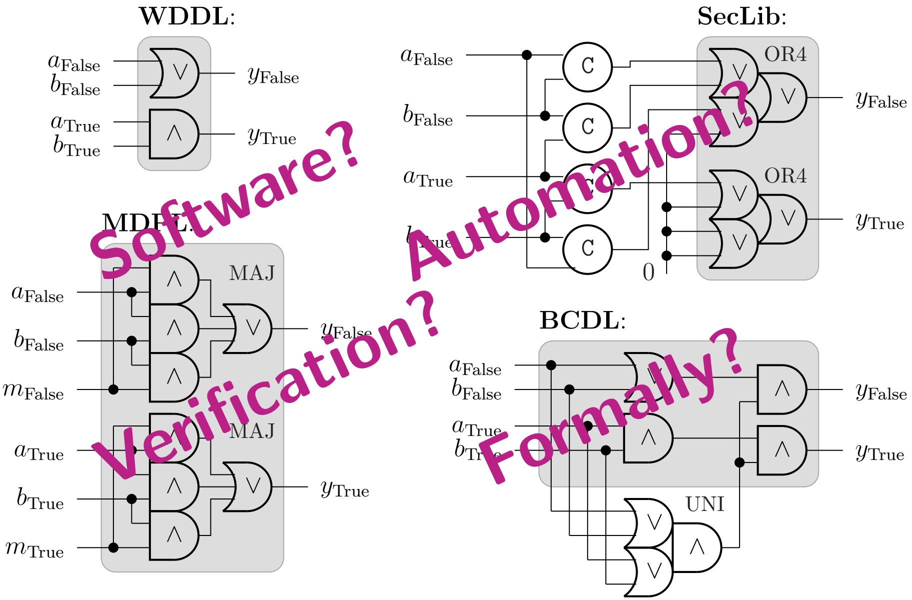




WDDL:
a_{False}
b_{False}
a_{True}
b_{True}
y_{False}
y_{True}
Software?
MDPL:
MAJ
a_{False}
b_{False}
m_{False}
y_{False}
MAJ
a_{True}
b_{True}
m_{True}
y_{True}
SecLib:
a_{False}
b_{False}
a_{True}
b_{True}
C
C
C
C
0
OR4
OR4
y_{False}
y_{True}
BCDL:
a_{False}
b_{False}
a_{True}
b_{True}
y_{False}
y_{True}
UNI


WDDL:
a_{False}
b_{False}
a_{True}
b_{True}
y_{False}
y_{True}
SecLib:
OR4
OR4
C
0
MDPL:
MAJ
MAJ
m_{False}
m_{True}
BCDL:
UNI
Software?
Automation?


WDDL:
SecLib:
MDPL:
BCDL:
OR4
OR4
MAJ
MAJ
UNI
C
0
a_{False}
b_{False}
a_{True}
b_{True}
m_{False}
m_{True}
y_{False}
y_{True}
Software?
Automation?
Verification?


WDDL:
SecLib:
MDPL:
BCDL:
a_{False}
b_{False}
a_{True}
b_{True}
m_{False}
m_{True}
y_{False}
y_{True}
OR4
MAJ
UNI
C
0
Software?
Automation?
Verification?
Formally?

- Our goal is to be able to *formally assess the security of a cryptosystem against power analysis attacks*.
- But, *formal methods work with models, not implementations*.
- Yet, *side-channel attacks are an implementation-level threat*.

→ We want to apply formal methods on the implementation.

- Power analysis is a form of side-channel attack in which the attacker measures the *power consumption* of a cryptographic device.
- Power consumption is modeled by the *Hamming weight* of values and the *Hamming distance* of updates.

- Unprotected implementation leaks at every step.
- Thwarting side-channel analysis is a complicated task.

Countermeasures

- In practice, there are two ways to protect cryptosystems.
- *Palliative countermeasures* attempt to make the attack more difficult, however without a theoretical foundation:
  - variable clock,
  - operation shuffling,
  - dummy encryptions, *etc.*
- *Curative countermeasures* aim at providing a leak-free implementation based on a security rationale:
  - decorrelate the leakage from the manipulated data, or
  - make the leakage constant, irrespective of the manipulated data.

Masking — Definition

Mix the computation with *random* numbers to make the leakage (at least in average) independent of the sensitive data.

- Pros:
  - independence with respect to the leakage behavior of the hardware,
  - existence of provably secure masking schemes.
- Cons:
  - greedy requirement for randomness,
  - randomness is hard to formalize,
  - hardware *glitches* are likely to depend on more than one sensitive data, hence being high-order.
  - possibility of high-order attacks.

Balancing — Definition

Follow a *dual-rail* protocol to make the leakage *constant*, irrespective of the manipulated data.

DPL (*Dual-rail with Precharge Logic*) — Definition

Compute on redundant representation on two *indistinguishable* resources, so that the attacker cannot know which one has been set (which depends on the bit value).

- Pros:
  - no randomness necessary,
  - simple protocol easily captured formally.
- Cons:
  - strongly depends on assumption on the hardware leakage.

- The DPL countermeasure consists in computing on a redundant representation: each bit y is implemented as a pair $(y_{\mathsf{False}}, y_{\mathsf{True}})$.
- The bit pair is then used in a protocol made up of two phases:
  1. a *precharge* phase, during which all the bit pairs are zeroized $(y_{\mathsf{False}}, y_{\mathsf{True}}) = (0, 0)$, such that the computation starts from a known reference state;
  2. an *evaluation* phase, during which the $(y_{\mathsf{False}}, y_{\mathsf{True}})$ pair is equal to $(1, 0)$ if it carries the logical value 0, or $(0, 1)$ if it carries the logical value 1.

DPL in Software

- Historically, DPL has been designed for implementation at hardware level.
- But we want to run DPL on an off-the-shelf processor.
- Therefore, we must:
  - identify two similar resources that can hold true and false values in an indiscernible way for a side-channel attacker;
  - play the DPL protocol by ourselves, in software.
- Then, to reproduce the DPL protocol in software we have to:
  - work at the bit level, and
  - duplicate (in positive and negative logic) the bit values.

DPL Macro

- Each sensitive instruction should replaced by a *DPL macro*.
- The DPL macro assumes that the system is in a valid DPL state.
- And leaves it in a valid DPL state to make the macros chainable.

- The basic idea is to concatenate two DPL encoded values.
- Then use the result as an index in a look-up table.

Example Using the Two Least Significant Bit

- In this example we use the two LSB.
- Logical value 1 is 1 (`01`).
- Logical value 0 is 2 (`10`).
- Precharge phases (activity: 1 if sensitive)
- Evaluation phases (activity: 1)
- Masks (activity: normally 0)
- Shifts (activity: 2)
- Concatenation (activity: 1)
- Look-up (activity: $1 + 2$)

$$
\begin{array}{rcl}
r_1 & \leftarrow & r_0 \\
r_1 & \leftarrow & a \\
r_1 & \leftarrow & r_1 \wedge 3 \\
r_1 & \leftarrow & r_1 \ll 1 \\
r_1 & \leftarrow & r_1 \ll 1 \\
r_2 & \leftarrow & r_0 \\
r_2 & \leftarrow & b \\
r_2 & \leftarrow & r_2 \wedge 3 \\
r_1 & \leftarrow & r_1 \vee r_2 \\
r_3 & \leftarrow & r_0 \\
r_3 & \leftarrow & op[r_1] \\
d & \leftarrow & r_0 \\
d & \leftarrow & r_3
\end{array}
$$

DPL macro for
$d = a \text{ op } b$

Example Using the Two Least Significant Bit

- In this example we use the two LSB.
- Logical value 1 is 1 ($\texttt{01}$).
- Logical value 0 is 2 ($\texttt{10}$).

- Precharge phases (activity: 1 if sensitive)
- Evaluation phases (activity: 1)
- Masks (activity: normally 0)
- Shifts (activity: 2)
- Concatenation (activity: 1)
- Look-up (activity: $1 + 2$)

r_1	$\leftarrow$	r_0
r_1	$\leftarrow$	a
r_1	$\leftarrow$	$r_1 \wedge 3$
r_1	$\leftarrow$	$r_1 \ll 1$
r_1	$\leftarrow$	$r_1 \ll 1$
r_2	$\leftarrow$	r_0
r_2	$\leftarrow$	b
r_2	$\leftarrow$	$r_2 \wedge 3$
r_1	$\leftarrow$	$r_1 \vee r_2$
r_3	$\leftarrow$	r_0
r_3	$\leftarrow$	$op[r_1]$
d	$\leftarrow$	r_0
d	$\leftarrow$	r_3

DPL macro for
$d = a \text{ op } b$

Example Using the Two Least Significant Bit

- In this example we use the two LSB.
- Logical value 1 is 1 (`01`).
- Logical value 0 is 2 (`10`).
- Precharge phases (activity: 1 if sensitive)
- Evaluation phases (activity: 1)
- Masks (activity: normally 0)
- Shifts (activity: 2)
- Concatenation (activity: 1)
- Look-up (activity: $1+2$)

r_1	$\leftarrow$	r_0
r_1	$\leftarrow$	a
r_1	$\leftarrow$	$r_1 \wedge 3$
r_1	$\leftarrow$	$r_1 \ll 1$
r_1	$\leftarrow$	$r_1 \ll 1$
r_2	$\leftarrow$	r_0
r_2	$\leftarrow$	b
r_2	$\leftarrow$	$r_2 \wedge 3$
r_1	$\leftarrow$	$r_1 \vee r_2$
r_3	$\leftarrow$	r_0
r_3	$\leftarrow$	$op[r_1]$
d	$\leftarrow$	r_0
d	$\leftarrow$	r_3

DPL macro for
$d = a \text{ op } b$

Example Using the Two Least Significant Bit

- In this example we use the two LSB.
- Logical value 1 is 1 ($\texttt{01}$).
- Logical value 0 is 2 ($\texttt{10}$).
- Precharge phases (activity: 1 if sensitive)
- Evaluation phases (activity: 1)
- Masks (activity: normally 0)
- Shifts (activity: 2)
- Concatenation (activity: 1)
- Look-up (activity: $1 + 2$)

$$
\begin{aligned}
r_1 &\leftarrow r_0 \\
r_1 &\leftarrow a \\
r_1 &\leftarrow r_1 \wedge 3 \\
r_1 &\leftarrow r_1 \ll 1 \\
r_1 &\leftarrow r_1 \ll 1 \\
r_2 &\leftarrow r_0 \\
r_2 &\leftarrow b \\
r_2 &\leftarrow r_2 \wedge 3 \\
r_1 &\leftarrow r_1 \vee r_2 \\
r_3 &\leftarrow r_0 \\
r_3 &\leftarrow op[r_1] \\
d &\leftarrow r_0 \\
d &\leftarrow r_3
\end{aligned}
$$

DPL macro for
$d = a \text{ op } b$

Example Using the Two Least Significant Bit

- In this example we use the two LSB.
- Logical value 1 is 1 (`01`).
- Logical value 0 is 2 (`10`).
- Precharge phases (activity: 1 if sensitive)
- Evaluation phases (activity: 1)
- Masks (activity: normally 0)
- Shifts (activity: 2)
- Concatenation (activity: 1)
- Look-up (activity: $1 + 2$)

$$
\begin{aligned}
r_1 &\leftarrow r_0 \\
r_1 &\leftarrow a \\
r_1 &\leftarrow r_1 \wedge 3 \\
r_1 &\leftarrow r_1 \ll 1 \\
r_1 &\leftarrow r_1 \ll 1 \\
r_2 &\leftarrow r_0 \\
r_2 &\leftarrow b \\
r_2 &\leftarrow r_2 \wedge 3 \\
r_1 &\leftarrow r_1 \vee r_2 \\
r_3 &\leftarrow r_0 \\
r_3 &\leftarrow op[r_1] \\
d &\leftarrow r_0 \\
d &\leftarrow r_3
\end{aligned}
$$

DPL macro for $d = a \text{ op } b$

Example Using the Two Least Significant Bit

- In this example we use the two LSB.
- Logical value 1 is 1 (`01`).
- Logical value 0 is 2 (`10`).
- Precharge phases (activity: 1 if sensitive)
- Evaluation phases (activity: 1)
- Masks (activity: normally 0)
- Shifts (activity: 2)
- Concatenation (activity: 1)
- Look-up (activity: $1 + 2$)

$$
\begin{aligned}
r_1 &\leftarrow r_0 \\
r_1 &\leftarrow a \\
r_1 &\leftarrow r_1 \wedge 3 \\
r_1 &\leftarrow r_1 \ll 1 \\
r_1 &\leftarrow r_1 \ll 1 \\
r_2 &\leftarrow r_0 \\
r_2 &\leftarrow b \\
r_2 &\leftarrow r_2 \wedge 3 \\
r_1 &\leftarrow r_1 \vee r_2 \\
r_3 &\leftarrow r_0 \\
r_3 &\leftarrow op[r_1] \\
d &\leftarrow r_0 \\
d &\leftarrow r_3
\end{aligned}
$$

DPL macro for
$d = a \text{ op } b$

Example Using the Two Least Significant Bit

- In this example we use the two LSB.
- Logical value 1 is 1 ($\texttt{01}$).
- Logical value 0 is 2 ($\texttt{10}$).
- Precharge phases (activity: 1 if sensitive)
- Evaluation phases (activity: 1)
- Masks (activity: normally 0)
- Shifts (activity: 2)
- Concatenation (activity: 1)
- Look-up (activity: 1 + 2)

$$
\begin{array}{rcl}
r_1 & \leftarrow & r_0 \\
r_1 & \leftarrow & a \\
r_1 & \leftarrow & r_1 \wedge 3 \\
r_1 & \leftarrow & r_1 \ll 1 \\
r_1 & \leftarrow & r_1 \ll 1 \\
r_2 & \leftarrow & r_0 \\
r_2 & \leftarrow & b \\
r_2 & \leftarrow & r_2 \wedge 3 \\
r_1 & \leftarrow & r_1 \vee r_2 \\
r_3 & \leftarrow & r_0 \\
r_3 & \leftarrow & op[r_1] \\
d & \leftarrow & r_0 \\
d & \leftarrow & r_3
\end{array}
$$

DPL macro for
$d = a \text{ op } b$

- We want to *automatically insert* this countermeasure in assembly code.
- To be as universal as possible, we use a *generic assembly language* which can be mapped to and from virtually any actual assembly language.

Generic Assembly Language

```
Prog    ::= ( Label? Inst? ( ';' <comment> )? '\n' )*
Label   ::= <label-name> ':'
Inst    ::= Opcode0
          | Branch1 Addr
          | Opcode2 Lval Val
          | Opcode3 Lval Val Val
          | Branch3 Val Val Addr
Opcode0 ::= 'nop'
Branch1 ::= 'jmp'
Opcode2 ::= 'not' | 'mov'
Opcode3 ::= 'and' | 'orr' | 'xor' | 'lsl' | 'lsr'
          | 'add' | 'mul'
Branch3 ::= 'beq' | 'bne'
Val     ::= Lval | '#' <immediate-value>
Lval    ::= 'r' <register-number>
          | '@' <memory-address>
          | '!' Val ( ',' <offset> )?
Addr    ::= '#' <absolute-code-address>
          | <label-name>
```

DPL Macro Using the Two Least Significant Bit

`mov r1 r0`	$r_1 \leftarrow r_0$
`mov r1` *a*	$r_1 \leftarrow a$
`and r1 r1 #3`	$r_1 \leftarrow r_1 \wedge 3$
`lsl r1 r1 #1`	$r_1 \leftarrow r_1 \ll 1$
`lsl r1 r1 #1`	$r_1 \leftarrow r_1 \ll 1$
`mov r2 r0`	$r_2 \leftarrow r_0$
`mov r2` *b*	$r_2 \leftarrow b$
`and r2 r2 #3`	$r_2 \leftarrow r_2 \wedge 3$
`orr r1 r1 r2`	$r_1 \leftarrow r_1 \vee r_2$
`mov r3 r0`	$r_3 \leftarrow r_0$
`mov r3 !r1,`*op*	$r_3 \leftarrow op[r_1]$
`mov` *d* `r0`	$d \leftarrow r_0$
`mov` *d* `r3`	$d \leftarrow r_3$

Code Transformation

1. Bitslice code.
2. DPL macros expansion.
3. Look-up tables.

1. Bitslicing Code

- Always possible (by Turing machines equivalence theorem)
- But, hard to do automatically in practice.
- However, there are a lot of already (manually) bitsliced implementations, since it is a common optimization technique.

→ We take already bitsliced code as input.

Sensitive value — Definition

A *value* is said *sensitive* if it depends on sensitive data. A sensitive data depends on the secret key or the plaintext.

Sensitive instruction — Definition

An *instruction* is said *sensitive* if it may modify the Hamming weight of a sensitive value.

- All the sensitive instructions must be expanded to a DPL macro.
- Thus, all the sensitive data must be transformed too.

2.2. Which Instructions are Sensitive?

- Bitsliced code means that only the logical (bit level) operators, except shifts, are used in sensitive instructions.
- DPL protocol implies that `not` instructions are replaced by `xor`.

→ Only `and`, `or`, and `xor` instructions need to be expanded to DPL macros.

3. Look-Up Tables

- Addresses of the look-up tables are sensitive too: their indices are sensitive values.
- Thus, the addresses bits corresponding to the accessed cell must be 0.
- In our example, the look-up table addresses must be multiple of 16.

index	0000	0001	0010	0011	0100	0101	0110	0111
and	00	00	00	00	00	01	10	00
or	00	00	00	00	00	01	01	00
xor	00	00	00	00	00	10	01	00

index	1000	1001	1010	1011	1100	1101	1110	1111
and	00	10	10	00	00	00	00	00
or	00	01	10	00	00	00	00	00
xor	00	01	10	00	00	00	00	00

3. Look-Up Tables

- Addresses of the look-up tables are sensitive too: their indices are sensitive values.
- Thus, the addresses bits corresponding to the accessed cell must be 0.
- In our example, the look-up table addresses must be multiple of 16.

index	0000	0001	0010	0011	0100	0101	0110	0111
and	00	00	00	00	00	01	10	00
or	00	00	00	00	00	01	01	00
xor	00	00	00	00	00	10	01	00

index	1000	1001	1010	1011	1100	1101	1110	1111
and	00	10	10	00	00	00	00	00
or	00	01	10	00	00	00	00	00
xor	00	01	10	00	00	00	00	00

Correct DPL transformation — Definition

Let S be a valid state of the system (values in registers and memory).

Let c be a sequence of instructions of the system.

Let $\widehat{S}$ be the state of the system after the execution of c with state S, we denote that by $S \xrightarrow{c} \widehat{S}$.

We write $dpl(S)$ for the DPL state equivalent to the state S.

We say that c' is a *correct DPL transformation* of the code c if
$S \xrightarrow{c} \widehat{S} \implies dpl(S) \xrightarrow{c'} dpl(\widehat{S})$.

Correctness of our code transformation — Proposition

The expansion of the sensitive instructions into DPL macros is a correct DPL transformation.

- Proof in the paper.

- Example execution for `and`.

a, b	10, 10				Sensitive activity
	d	r1	r2	r3	
mov r1 r0	?	0	?	?	0
mov r1 a	?	10	?	?	1
and r1 r1 #3	?	10	?	?	0
shl r1 r1 #1	?	100	?	?	2
shl r1 r1 #1	?	1000	?	?	2
mov r2 r0	?	1000	0	?	0
mov r2 b	?	1000	10	?	1
and r2 r2 #3	?	1000	10	?	0
orr r1 r1 r2	?	1010	10	?	1
mov r3 r0	?	1010	10	0	0
mov r3 !r1,and	?	1010	10	10	3
mov d r0	0	1010	10	10	0
mov d r3	10	1010	10	10	1

- Example execution for `and`.

a, b	10, 01				Sensitive activity
	d	r1	r2	r3	
`mov r1 r0`	?	0	?	?	0
`mov r1` a	?	10	?	?	1
`and r1 r1 #3`	?	10	?	?	0
`shl r1 r1 #1`	?	100	?	?	2
`shl r1 r1 #1`	?	1000	?	?	2
`mov r2 r0`	?	1000	0	?	0
`mov r2` b	?	1000	01	?	1
`and r2 r2 #3`	?	1000	01	?	0
`orr r1 r1 r2`	?	1001	01	?	1
`mov r3 r0`	?	1001	01	0	0
`mov r3 !r1,` and	?	1001	01	10	3
`mov` d `r0`	0	1001	01	10	0
`mov` d `r3`	10	1001	01	10	1

- Example execution for `and`.

a, b	01, 10				Sensitive activity
	d	r1	r2	r3	
mov r1 r0	?	0	?	?	0
mov r1 a	?	01	?	?	1
and r1 r1 #3	?	01	?	?	0
shl r1 r1 #1	?	010	?	?	2
shl r1 r1 #1	?	0100	?	?	2
mov r2 r0	?	0100	0	?	0
mov r2 b	?	0100	10	?	1
and r2 r2 #3	?	0100	10	?	0
orr r1 r1 r2	?	0110	10	?	1
mov r3 r0	?	0110	10	0	0
mov r3 !r1, and	?	0110	10	10	3
mov d r0	0	0110	10	10	0
mov d r3	10	0110	10	10	1

- Example execution for `and`.

a, b	01, 01				Sensitive activity
	d	r1	r2	r3	
mov r1 r0	?	0	?	?	0
mov r1 a	?	01	?	?	1
and r1 r1 #3	?	01	?	?	0
shl r1 r1 #1	?	010	?	?	2
shl r1 r1 #1	?	0100	?	?	2
mov r2 r0	?	0100	0	?	0
mov r2 b	?	0100	01	?	1
and r2 r2 #3	?	0100	01	?	0
orr r1 r1 r2	?	0101	01	?	1
mov r3 r0	?	0101	01	0	0
mov r3 !r1,and	?	0101	01	01	3
mov d r0	0	0101	01	01	0
mov d r3	01	0101	01	01	1

Computed Proof of Constant Activity

- Our tool does this verification automatically for the whole program.
- It uses *symbolic computations* to keep track of possible leakages.

- The strategy is to *simulate* a CPU and memory in software, and *compute with sets* of values.
- Initially, all sensitive data values can be either 0 or 1.
- At each cycle and for each possible combination of actual values:
  - it looks at the Hamming weight of values and Hamming distance of updates in registers, memory, and addresses; and
  - if one can have different values, it reports a leak.

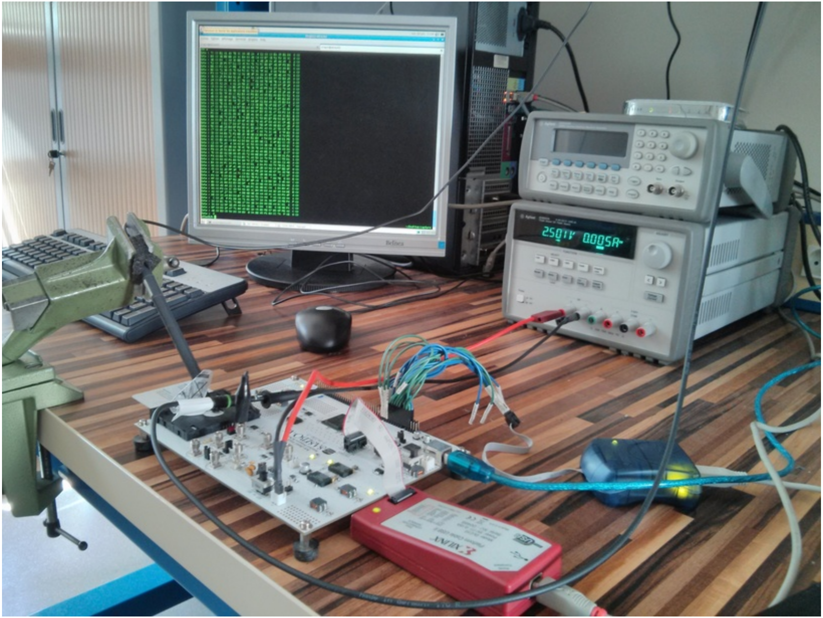
- This verification is independent from the code transformation.

Hardware Characterization

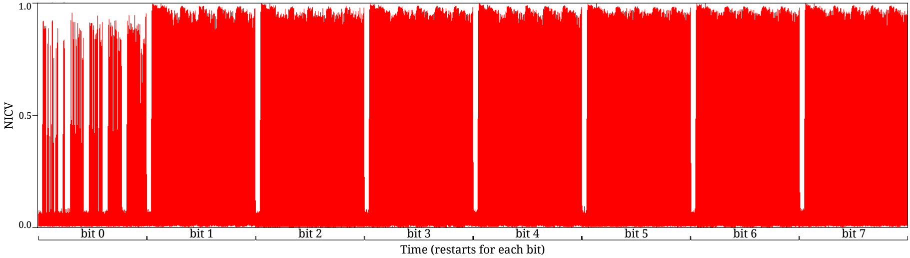
- The DPL countermeasure heavily relies on the indistinguishable resources hypothesis on the hardware.
- This property is generally not true in non-specialized hardware.

- Using the bits whose leakage are the most similar will maximize the relevancy of our leakage model.
- Profiling the hardware allows to find them.

Case Study: PRESENT on an AVR Micro-Controller



Profiling the AVR Micro-Controller



Leakage level during unprotected encryption for each bit of the *ATmega163*.

Generating Balanced AVR Assembly

$$
\begin{aligned}
r_1 &\leftarrow r_0 \\
r_1 &\leftarrow a \\
r_1 &\leftarrow r_1 \wedge 6 \\
r_1 &\leftarrow r_1 \ll 1 \\
r_1 &\leftarrow r_1 \ll 1 \\
r_2 &\leftarrow r_0 \\
r_2 &\leftarrow b \\
r_2 &\leftarrow r_2 \wedge 6 \\
r_1 &\leftarrow r_1 \vee r_2 \\
r_3 &\leftarrow r_0 \\
r_3 &\leftarrow op[r_1] \\
d &\leftarrow r_0 \\
d &\leftarrow r_3
\end{aligned}
$$

DPL macro for $d = a \text{ op } b$ on the *ATmega163*.

Cost of the Countermeasure

	bitslice	DPL	cost
code (B)	1620	3056	$\times 1.88$
RAM (B)	288	352	$+64$
#cycles	$78,403$	$235,427$	$\times 3$

DPL cost.

Attacks

- We attacked three implementations:
  - a bitsliced but unprotected one,
  - a DPL protected one using the two less significant bits,
  - a DPL protected one taking the hardware characterization into account.
- We took $100,000$ execution traces.
- We computed the success rate of using *monobit CPA* of the output of the S-Box as a model.

Results

- The unprotected implementation breaks using about 400 traces.
- The poorly balanced one is still not broken using $100,000$ traces.

→ But we want to show that the hardware characterization is beneficial!

- Let's make the attacker "cheat".
- We used our knowledge of the key to select a narrow part of the traces where we knew that the attack would work.
- We used the NICV to select the point where the signal-to-noise ratio of the CPA attack is the highest.

Results

- The unprotected implementation breaks using about 400 traces.
- The poorly balanced one is still not broken using $100,000$ traces.

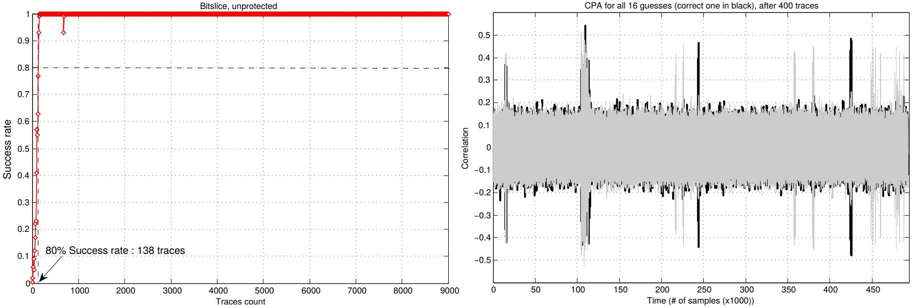
→ But we want to show that the hardware characterization is beneficial!

- Let's make the attacker "cheat".
- We used our knowledge of the key to select a narrow part of the traces where we knew that the attack would work.
- We used the NICV to select the point where the signal-to-noise ratio of the CPA attack is the highest.

Results for the "Cheating Attacker"

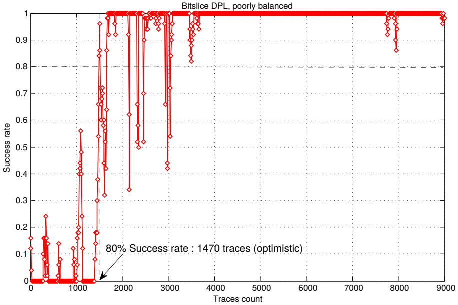
- The unprotected implementation breaks using 138 traces.
- The poorly balanced one breaks using $1,470$ traces.
- The better balanced one breaks using $4,810$ traces.

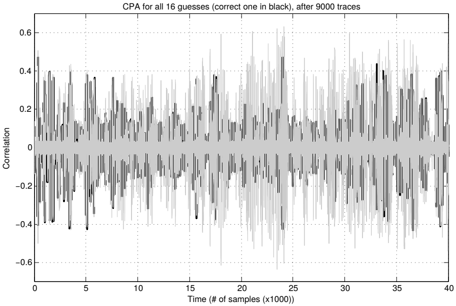
Results for the "Cheating Attacker": unprotected



Monobit CPA attack on unprotected bitslice implementation.

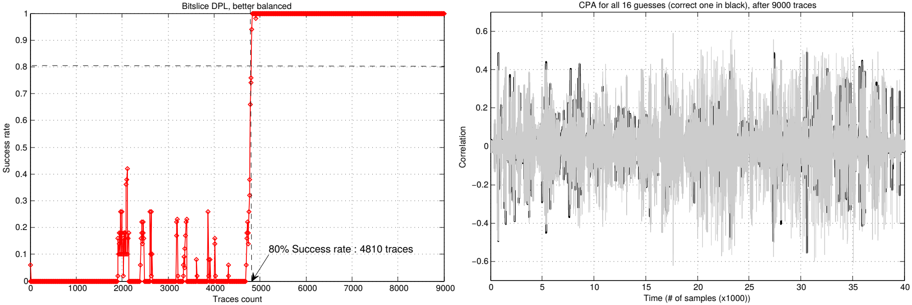
Results for the "Cheating Attacker": poorly balanced





Monobit CPA attack on poorly balanced DPL implementation (bits 0 and 1).

Results for the "Cheating Attacker": better balanced



Monobit CPA attack on better balanced DPL implementation (bits 1 and 2).

- Automatic and proven correct code protection.
- Independent formal proof of constant activity according to a leakage model.
- Hardware characterization method to increase the leakage model relevancy.

- Provably balanced DPL protected implementation or PRESENT:
  - At least 250 times more resistant to power analysis attacks.
  - SNR divided by at least 16.
  - Only 3 (or 24) times slower.

→ Software balancing countermeasures are realistic.

http://pablo.rauzy.name/sensi/paioli.html

- The pair of bits used for the DPL protocol could change during the execution or chosen at random for each execution.
- Unused bits could be randomized instead of being zero in order to add noise on top of balancing.
- Randomness could be used to mask the computation.
- Also:
  - our methods and tools need to be further tested in other experimental settings;
  - although the mapping from the internal assembly of our tool to the concrete assembly is straightforward, it would be better to have a formal correctness proof of the mapping;
  - our work would also benefit from automated bitslicing.

We believe formal methods have a bright future concerning the certification of side-channel attacks countermeasures for trustable cryptosystems.

That was it. Questions?

rauzy@enst.fr

Open access and always up-to-date version of the paper:
IACR ePrint 2013/554