Software Implementation of the Dual-Rail Representation

Philippe Hoogvorst Jean-Luc Danger Guillaume Duc

Centre National de la Recherche Scientifique / Telecom-ParisTech

Thursday, February 24^{th}, 2011

Side-Channel Attacks.

A side-channel attack consists in two major steps:

- Data gathering:
  - Electric measurements,
  - Electromagnetic measurements,
- Statistical processing:
  - DPA / CPA,
  - MIA,
  - template attacks,
  - etc...

Such an attack can be successful because the power consumption of non-protected devices is related to the computations being performed.

Countermeasures.

Two general types of countermeasures:

- Masking: perform 2 operations:
  - One on the real data, scrambled by a random value,
  - One on the "descrambling values", needed to end with the correct results.
  - expensive on hardware when LUT are used,
  - Subjects to "counter-attacks" (higher-order attacks)
- Rendering the power consumption independent of the data:
  - "Dual-Rail" with precharge representation,
  - Roughly doubles the hardware.
  - Also subject to "counter-attacks".

"Dual-Rail" representation.

"Dual-Rail" representation of binary values

A binary datum x is represented by 2 electrical signals (x_0, x_1):

x_0	x_1	Value	Meaning
V_{ss}: "physical" '0'	V_{ss}: "physical" '0'	0	Precharge
V_{dd}: "physical" '1'	V_{ss}: "physical" '0'	1	Logical "0"
V_{ss}: "physical" '0'	V_{dd}: "physical" '1'	2	Logical "1"
V_{dd}: "physical" '1'	V_{dd}: "physical" '1'	3	Illegal state.

Legal states: Logical 0, Precharge and Logical 1.

Legal transitions: Logical 0 $\leftrightarrow$ Precharge $\leftrightarrow$ Logical 1.

Note that the choices:

- "$V_{ss} \leftrightarrow 0, V_{dd} \leftrightarrow 1$" and
- $(0,0) \leftrightarrow$ Precharge, $(1,1) \leftrightarrow$ Illegal

are arbitrary.

"Dual-Rail" representation (2).

Operations on "Dual-Rail" values.

$$
\begin{aligned}
\text{NOT} \quad & \overline{(x_0, x_1)} = (x_1, x_0) \text{ (wire crossing only)}, \\
\text{AND} \quad & (x_0, x_1) \wedge (y_0, y_1) = (x_0 \vee y_0, x_1 \wedge y_1), \\
\text{IOR} \quad & (x_0, x_1) \vee (y_0, y_1) = (x_0 \wedge y_0, x_1 \vee y_1), \\
\text{NAND} \quad & (x_0, x_1) \wedge (y_0, y_1) = (x_1 \wedge y_1, x_0 \vee y_0), \\
\text{NOR} \quad & (x_0, x_1) \vee (y_0, y_1) = (x_1 \vee y_1, x_0 \wedge y_0), \\
\text{XOR} \quad & (x_0, x_1) \oplus (y_0, y_1) = \\
& ((x_0 \wedge y_0) \vee (x_1 \wedge y_1), (x_1 \wedge y_0) \vee (x_0 \wedge y_1)) \\
\text{NXOR} \quad & (x_0, x_1) \oplus (y_0, y_1) = \\
& ((x_1 \wedge y_0) \vee (x_0 \wedge y_1), (x_0 \wedge y_0) \vee (x_1 \wedge y_1))
\end{aligned}
$$

Note that:

- $\wedge$ and $\vee$ are increasing functions, thus:
- only transitions $V_{ss} \rightarrow V_{dd}$ occur during computation and
- only transitions $V_{dd} \rightarrow V_{ss}$ occur during precharge.

"Dual-Rail" representation (3).

Weaknesses of the "Dual-Rail" representation:

- Electrical dissymetry of the x_0 and x_1 wires:
  - goal of the "Dual-Rail" representation: to render power consumption & radiation independent of the data but...
  - the design of the silicon must ensure this independence,
  - it is only nearly possible.
- Timing dissymetry of the x_0 and x_1 wires:
  For Ex.: During the computation of AND(x,y):
  - If the x is ready to be used "a long time" before y,
  - If x="0" the result is evaluated **immediately**.
  - If x="1" the result is evaluated **when** y **is ready**.

  This "early evaluation" flaw has been successfully used to attack "Dual-Rail" representation.

  The countermeasure is to synchronize the computation to the arrival of both operands.

"Dual-Rail" representation (4).

Performance:

The increased security of "Dual-Rail" comes at a price:

Time Each step of computation takes 2 clock cycles:

- 1 clock cycle for the precharge and
- 1 clock cycle for the computation.

Silicon The surface of a "Dual-Rail" operator is significantly larger than its single-rail counterpart:

- There is *a priori* twice more transistors,
- In fact, a lot more: a NOR gate cannot be built with less than 24 transistors,
- there are more constraints on the placement of gates and the routing of the signals.

This price tends to confine the "Dual-Rail" representation to hih-end systems. Can it be cheaper?

Target Applications.

- Non-cost-bound applications needing "Dual-Rail" will include a hardware coprocessor.
- Cost-bound applications will prefer a standard, cheap (i.e. without special protection devices) μ-processor.

However, just not any cheap μ-processor can be used.

On which processor?.

Constraints on the μ-processor:

- 2 types of operations:
  - Transfers Registers $\leftrightarrow$ RAM,
  - Arithmetic operations between registers.
- Each of them imply a precharge of either the destination register or the target memory **AND** the RAM data and address buses.
- The target μ-processor must feature a "Harvard" structure.
- This is true for modern cheap μ-processors: 8051, AVR, PIC, etc. . . , which are used in low-cost applications.

From now on we assume that the μ-processor is:

- an 8-bit μ-processor,
- with a Harvard structure.

Representation of data.

The hardware is standard, with no special protection against SCA.
The "Dual-Rail" representation is implemented in s/w only.

- As in h/w 1 logical bit = 2 physical bits.
- The same coding is used:

  Precharge : $(x_1, x_0) = (0, 0)$ (Decimal: 0) ,
  Logical 0 : $(x_1, x_0) = (1, 0)$ (Decimal: 2) and
  Logical 1 : $(x_1, x_0) = (0, 1)$ (Decimal: 1) .

- The same transitions are allowed: 0 $\leftrightarrow$ precharge $\leftrightarrow$ 1.

Consequences:

- An 8-bit byte can hold 4 logical bits.
- The handling of data will be more complex thus slower.
- No miracle: the lower price of a software implementation will have a price in terms of performance.

Computing with "Dual-Rail" representation.

Goal:

- Compute with a power consumption independent of the data.
- Compute in time also independent of the data.

Consequences:

- Operations must contain no test, which can alter their timing,
- The number of bits changing state at any time must be independent of the data.
- This excludes the use of the μ-processor's arithmetic unit if the instruction model is "2-operands" as no precharge state can be inserted before the result overwrites the destination operand.
- μ-processors with the 3-operands model do not have this limitation for all instructions.

Using μ-processors with a 2-operands model.

- we cannot use the built-in arithmetic & logic unit.
- Any function on n bits can be expressed with a 2^n-entry look-up table.
- For practical reasons:
  - Size of the LUT,
  - Addressing capability without using the arithmetic unit,

  4 operands seem a maximum on a 8-bit μ-processor.
- As the look-up will return an 8-bit byte, the result of a singke look-up operation can be 4-bit wide.
- This allows the implementation of:
  - logic AND, NAND, OR, NOR, XOR, NXOR, NOT,
  - ADD, ADD with carry, SUB, SUB with carry.
  - S-Boxes up to 4-bit input & 4-bit output.
  - And, more generally, any function $\{0,1\}^4 \mapsto \{0,1\}^4$.

2-bit input, 1-bit output operation.

- Each operand is part of a 4-logical-bit byte.
- We assume they are both in physical bits 1:0.
- Generalization is obvious.
- Generalization to more operands is also simple.

The skeleton of the computation is:

1. Transfer the operands into registers.
2. Isolate the operands into temporaries using ‘AND” operations.
3. Shift one of these temporaries 2 positions to the left.
4. Combine the operands into an index using the ‘IOR” operation.
5. Read the result from the table.

We assume that:

- All operations are performed in the register bank.
- There are “enough” registers.

2-bit input, 1-bit output operation (2).

Communication to/from RAM

- The transfer to/from the RAM is very power-greedy, compared to register-to-register operations.
- It is thus necessary to precharge:
  - The destination (in RAM for a STORE, the reg. for a LOAD),
  - The data lines of the internal bus and
  - The address lines of the internal bus.
- Moving zero to a 2^n-aligned byte clears the n low-order addr. and the 8 data bits.
- We reserve a properly aligned byte, Ω, where the value 0 will be stored when a precharge of the data bus is necessary.

2-bit input, 1-bit output operation (3).

LOAD/STORE operations.

- LOAD register R from RAM byte b:

```
XOR     R, R    // Precharge R.
STORE   R, Ω    // Precharge address and data BUS.
LOAD    b, R    // Load R from b.
```

- STORE register R to RAM byte b:

```
XOR     R', R'  // use a temporary register.
STORE   R', Ω   // Precharge address and data BUS.
STORE   R', b   // Precharge b.
STORE   R, b    // Store data to b.
```

- Note that:
  1. The precharge of the BUS would be destroyed by the reading of the instructions on a non-Harvard μ-processor.
  2. The interrupts must be masked during sensitive operations.
  3. The absence of cache on cheap μ-processors is here an advantage.

2-bit input, 1-bit output operation (4).

Ancillary operations on data

Isolation :

- Use the AND operation with the proper mask.
- causes 1 bit flip per erased logical bit.

Putting in place in the register :

- Is applied **after isolation** of 1 logical bit.
- Use the "SHIFT left / right" operation.
- Causes 2 bit flips / shift.

Combining data :

- Uses the "OR" operation.
- Is applied after putting in place.
- The μ-processor's OR always perform either $b \leftarrow 0 \vee x$ or $b \leftarrow b \vee 0$.
- The added bit is always in front of a $00 \Rightarrow 1$ bit flip.

Computing the Result.

- The operands have been combined into a single byte.
- The destination register and the BUS are precharged.
- The combined word is used to index in a look-up table.
- The table must be aligned so that the indexation cause no internal carry within the address adder.
- Given that:
  - the numeric value of a logical "1" is $01_b = 1$ and
  - the numeric value of a logical "0" is $10_b = 2$

  The valid values of the index X are:
  - operands ("0", "0") $\Rightarrow X = 10$,
  - operands ("0", "1") $\Rightarrow X = 9$,
  - operands ("1", "0") $\Rightarrow X = 6$,
  - operands ("1", "0") $\Rightarrow X = 5$,
  - all other entries are 0, which is the precharge value.

Sum-up of a 2-operand instruction..

// Save data cache & interrupt modes.	
// Disable data cache & interrupts.	
// R_0 is assumed to be zero.	
$D \leftarrow R_0$	; Precharge: nb. of bit flips is irrelevant.
$R_1 \leftarrow R_0$	; Precharge.
$R_1 \leftarrow X$	; 1 bit flip per logical bit in X.
$R_1 \leftarrow R_1 \wedge 00000011_b$	; 1 bit flip per erased logical bit.
$R_1 \leftarrow R_1 << 1$	; Bring the x operand dots to bits 3..2.
$R_1 \leftarrow R_1 << 1$	; 2 bit flips for each shift.
$D \leftarrow R_0$	; Precharge.
$R_2 \leftarrow R_0$	; Precharge.
$R_2 \leftarrow Y$	; 1 bit flip per logical bit in Y.
$R_2 \leftarrow R_2 \wedge 3$	; 1 bit flip per erased logical bit.
$R_1 \leftarrow R_1 \vee R_2$	; Combine operands: 1 bit flip.

Sum-up of a 2-operand instruction. (2).

$R_3 \leftarrow R_0$	; Precharge. R_0 is assumed to be zero.
$R_3 \leftarrow T[R_1]$	; "Execute" operation: 1 bit flip.
$D \leftarrow R_0$	; Precharge RAM data & address bus.
$D \leftarrow R_3$	; store result into RAM.
$R_1 \leftarrow R_0$	; Clear sensitive information.
$R_2 \leftarrow R_0$	; Clear sensitive information.
$R_3 \leftarrow R_0$	; Clear sensitive information.
$D \leftarrow R_0$	; Clear sensitive information.
// Restore data cache and interrupt modes.	

Remarks on the Look-Up tables.

- The size of the LUT is a drawback:
  - They are 16-, 64- or 256- bytes wide but
  - only 4, 8 or 16 values are used.
  - This is indeed a waste of memory but...

Remarks on the Look-Up tables.

- The size of the LUT is a drawback:
  - They are 16-, 64- or 256- bytes wide but
  - only 4, 8 or 16 values are used.
  - This is indeed a waste of memory but...
- It is also a strength in front of a fault induction:
  - Any 1-bit fault will bring an "invalid" precharge value into the destination.
  - In fact, only errors which change a $(0, 1)$ into a $(1, 0)$ (or the reverse.) will cause an incorrect but "valid" result.
  - This "invalid" value will then propagate througout the computation with no possible recovery.
  - Thus the attacker will obtain no result, which could be used to discover secret data.

The Present Cryptosystem.

- "Lightweight" cryptosystem, targeted to "low-cost" applications.
- Substitution-Permutation Network,
- Blocksize is 64 bits.
- 31 rounds + 1 last key addition.
- Simple key schedule.
- 1 round consists of:
  1. Key addition,
  2. S-Boxes (16 identical S-Boxes, 4-bit in, 4-bit out.
  3. Bit shuffling to ensure diffusion.

Our implementation of PRESENT.

- The μ-processor is a state-of-the-art Atmel's ATmega128,
- However Atmel does not claim any kind of intrisic security,
- 1-chip system,
- Harvard architecture,
- 128 K-bytes Flash memory for the program,
- 32 general purpose registers,
- 8 K-bytes RAM.
- UART for communication with the host.
- Timers to measure the execution time, if necessary.
- Other peripheral devices, useless for our purpose.

Performance.

Memory :

- Due to the structure of PRESENT, the computation is performed in the registers
  $\Rightarrow$ no memory overhead for the working set.
- The precomputed key schedule taked 512 bytes instead of 256.
- one 256-byte table for the S-Box,
- one 256-byte table for the XOR operations, performed 2 bits by 2 bits.

Time :

- Each instruction takes 1 clock cycle. $\Rightarrow$ the duration of the computation is evaluated by counting the instructions.
- No test, apart from the loop count for the round,
- The time to encipher a 64-bit block is multiplied by 9.

Further Work.

- Evaluation of security is under way for both
  - a non-protected version and
  - the protected version.
- Other cryptosystems will be implemented and evaluated.
- Implementations on embedded 32-bit processors (e.g. LM-32).

Extension to AES.

- Key addition can be implemented just as for PRESENT.
- ShiftRows is just a permutation of names & indices.
- MixCoumns is linear operation, which can easily be discomposed into 4-bit input functions.
- SubBytes is more tricky, some solutions:
  - use 16-byte LUT and program 4 layers of 2-to-1 multiplexors.
  - represent the $GF(256)$ as a vector space over $GF(16)$ and use 4-input, 4-output LUT.
- The extension to AES is possible.
- However its cost remains to be evaluated.

Conclusion.

- Programming methodology which could help thwarting SCA.
- Yet it remains to be experimentally proved.
- The loss of performance is about 10.
- Yet it may not really matter in many cases if an encryption is performed in 10 ms instead of 1 ms. . . .
- The penalty as for memory space is 2: each physical byte can contain only 4 logical bits.

Last Word.

Thank you for your attention!