UCLouvain

UCLouvain
Crypto-group

Low Latency Hardware Masking

On how to build (relatively) low-cost and ***order-independent-latency*** *masked designs*

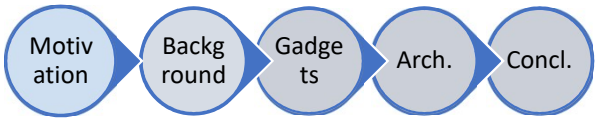
Itamar Levi,
June 2019

UCLouvain

- *Based on a work by* (available on eprint)*:*
  - G.Cassiers, B.Gregoire, I. Levi, F-X. Standaert.

- *In Short:*
  - *Hardware efficient masking…*
    - *low-latency, low rand. requirements, easy-to-use code/design*

- *On the long run:*
  - *We are using these methodologies and primitives with other technique,*
    - *to mask Clyde (from the NIST submission Spook) and,*
    - *for an on going work on composability…*



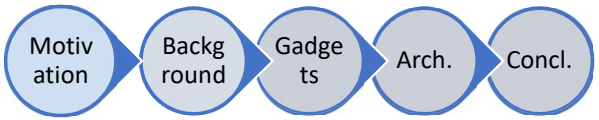
UCLouvain

Outline:

Motivation

- Necessary background – masking, in HW/SW & tradeoffs
- Improved gadgets on HW
- Building a BC and evaluating its performance

Discussion



UCLouvain

- Main motivation:
  - Building ***physically****-secured* devices

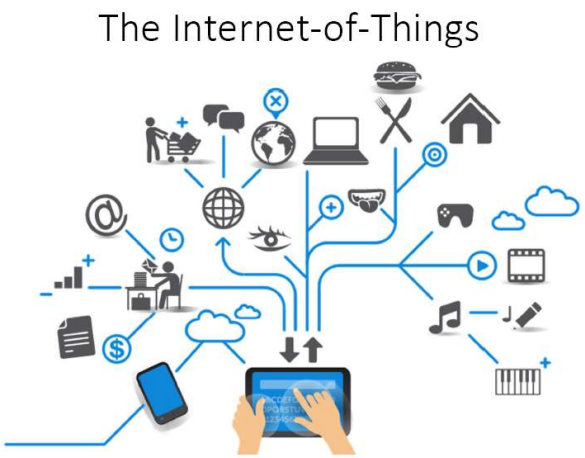
  Currently, it is "possible". But, the cost is ***high***…

The Internet-of-Things

UCLouvain

- Main motivation:
  - Building ***physically**-secured* devices

  Currently, it is "possible". But, the cost is ***high***…
- The limiting factors:
  - Cheap
  - low energy/area requirements
  - Low-latency
  - **Devices are highly accessed and exposed**
- Now, **zooming-in** on **high-order masking countermeasures..**



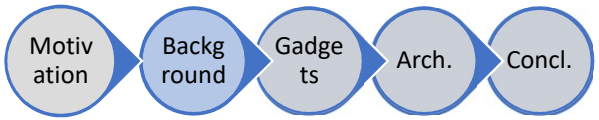
UCLouvain

Outline:

Motivation

- Necessary background – masking, in HW/SW & tradeoffs
- Improved gadgets on HW
- Building a BC and evaluating its performance

Discussion



UCLouvain

Masking: $\boldsymbol{x} = \overbrace{x_1 \oplus x_2 \oplus \ldots}^{\text{rand.}} \oplus x_d$ + independence condition.

We need to build logic (e.g. *S*−box …) $z = S(\boldsymbol{x} \oplus \boldsymbol{k}) = S(\boldsymbol{y})$

- Linear operations (i.e. XOR): $O(d)$
- Non-linear operations (i.e. AND): $O(d^2)$
  - Quadratic overheads + ↑#rand.

$$\begin{bmatrix} a_1b_1 & a_1b_2 & a_1b_3 \\ a_2b_1 & a_2b_2 & a_2b_3 \\ a_3b_1 & a_3b_2 & a_3b_3 \end{bmatrix} + \begin{bmatrix} 0 & r_1 & r_2 \\ -r_1 & 0 & r_3 \\ -r_2 & r_3 & 0 \end{bmatrix} \Rightarrow \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix}$$

partial products refreshing compression

- ISW[Crypto03]: Secured AND gate
- Many extensions/ improvements

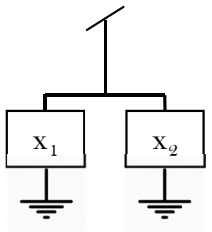
UCLouvain

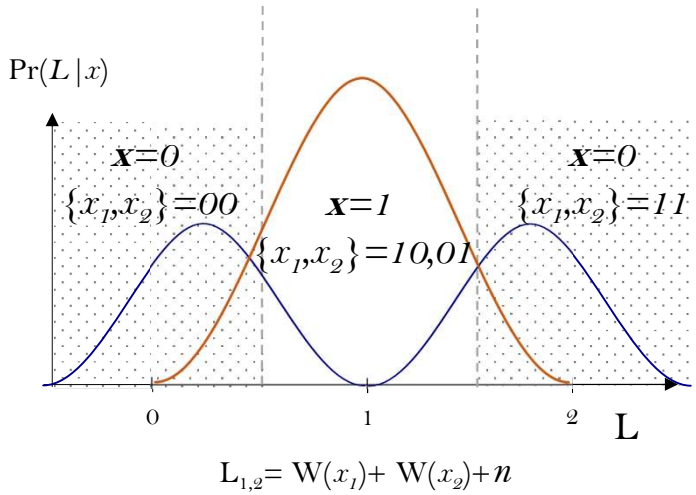
Masking: $\boldsymbol{x} = x_1 \oplus x_2$

1-bit, *d*=2-shares example, Assuming:

- Independent leakages
- $n \sim N(\mu, \sigma)$.
- Univariate

$$L = W(x_1) + W(x_2) + n$$





Secret independent means .. $\mu_1 = \mu_2$

Secret dependent variance .. $\sigma_1 \neq \sigma_2$

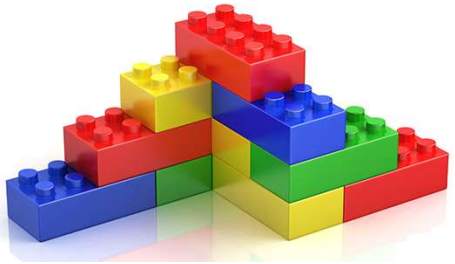
d=2 → "secured" up to the (d-1) statistical order→ μ

UCLouvain

- Security models and definitions:
  - Probing model: The adversary cannot learn anything unless she measures with (at least) ***d*** *probes*

$$\boldsymbol{x} = x_1 \oplus x_2 \oplus \ldots \oplus x_d \rightarrow \boldsymbol{x} = x_1 \oplus x_2 \oplus \ldots \oplus x_d$$

  - Proving a specific gadget is probing secure is not enough to *compose complex circuits with those instances and guaranty security*
    - *e.g. shares-refreshing is needed between gadgets*
    - *What if we want to save some randomness/ remove registers… Better definitions are needed.*



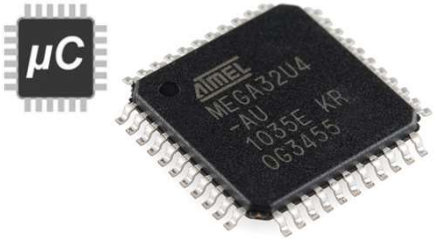
  - Many simulation based definitions were propose to aid in this task: NI, SNI, **MIMO-SNI, PINI and f-NI**
  - The goal: if a gadget meets (one/some of) these definitions we can compose with it..

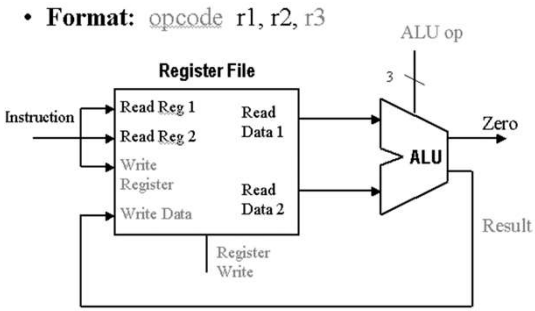
  - <u>In this work we only make use of such composable gadgets</u>

UCLouvain

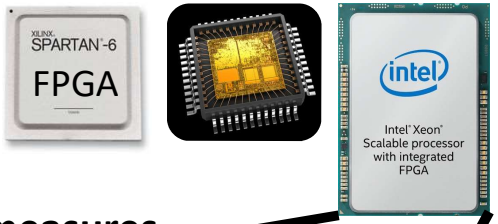
How to implement ?

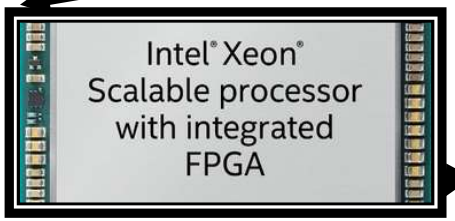
- SW
  - Slow
  - Hard to get “cheap” randomness
  - Less energy efficient than dedicated HW
  - Hard to add special SCA countermeasures

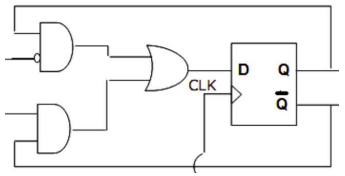




- HW / future ‘SW’..
  - Fast
  - Cheap True randomness
  - **Flexible – we can stack additional countermeasures**

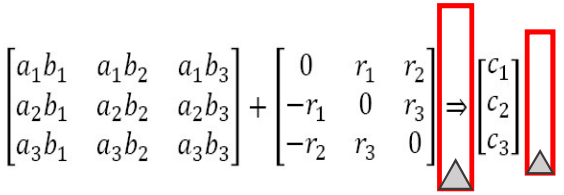




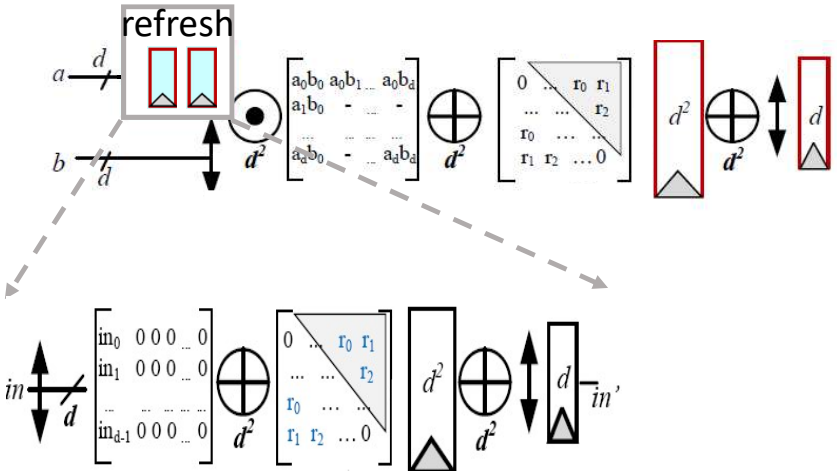


UCLouvain

- [ISW] – “as is”.
  - Is an “SNI” mult. Gadget
  - How to refresh/How to compose ?



- *Double-SNI / Faust et-al.*
  - *One input refresh is enough..*
  - *Use another SNI-refresh* (trivially, just use ISW)



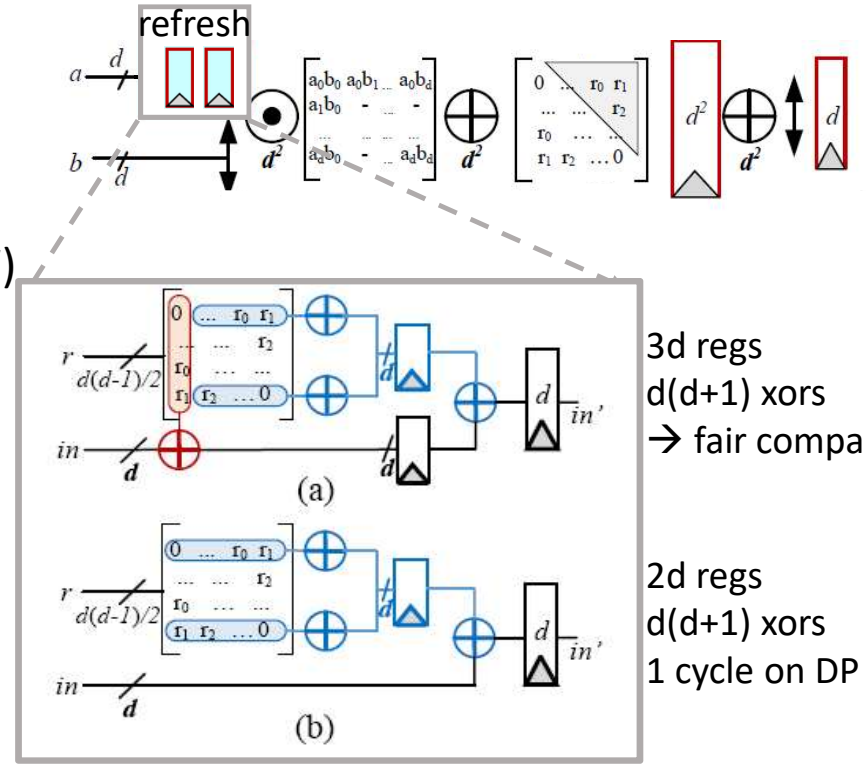
  - **How to implement this in HW ?**

UCLouvain

- [ISW] – “as is”.
  - Is an “SNI” mult. Gadget
  - How to refresh/How to compose ?

- *Double-SNI / Faust et-al.*
  - *One input refresh is enough..*
  - *Use another SNI-refresh* (trivially, just use ISW)

  - **How to implement this in HW ?**



3d regs
d(d+1) xors
→ fair comparison

2d regs
d(d+1) xors
1 cycle on DP

UCLouvain

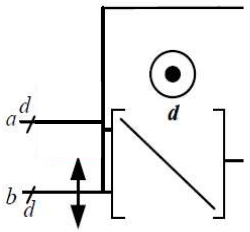
- **How to implement this on SW/HW ?**

$$\begin{bmatrix} a_1b_1 & a_1b_2 & a_1b_3 \\ a_2b_1 & a_2b_2 & a_2b_3 \\ a_3b_1 & a_3b_2 & a_3b_3 \end{bmatrix} + \begin{bmatrix} 0 & r_1 & r_2 \\ -r_1 & 0 & r_3 \\ -r_2 & r_3 & 0 \end{bmatrix} \Rightarrow \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix}$$

- SW – operands are too big…
  - **Need to chunks things up**

This is the (serial) HW counterpart..

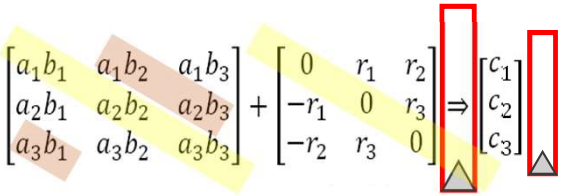
1: $c_1 = a \cdot b$



(b) Serial Barth. *et al* . + one-input refresh (adapted to HW context)

UCLouvain

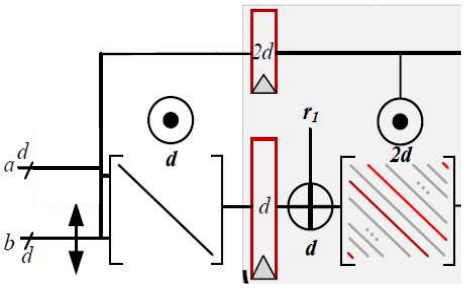
- **How to implement this on SW/HW ?**



- SW – operands are too big…
  - **Need to chunks things up**

1: $c_1 = a \cdot b$
2: $c_2 = a \cdot \mathrm{rot}(b, 1)$
3: $c_3 = \mathrm{rot}(a, 1) \cdot b$

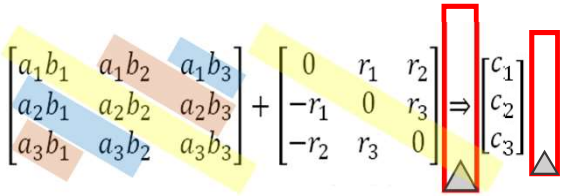
This is the (serial) HW counterpart..



(b) Serial Barth. *et al* . + one-input refresh (adapted to HW context)

UCLouvain

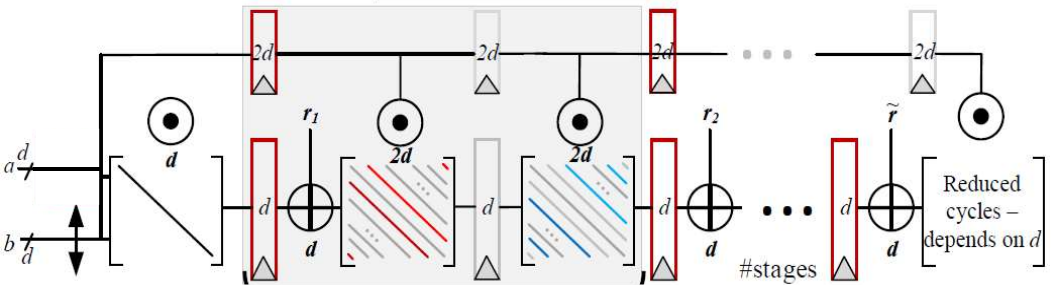
- **How to implement this on SW/HW ?**



- SW – operands are too big…
  - **Need to chunks things up**

1: $c_1 = a \cdot b$
2: $c_2 = a \cdot \mathrm{rot}(b, 1)$
3: $c_3 = \mathrm{rot}(a, 1) \cdot b$
4: $d_1 = c_1 \oplus r$
5: $d_2 = d_1 \oplus c_2$
6: $d_3 = d_2 \oplus c_3$
7: $d_4 = d_3 \oplus \mathrm{rot}(r, 1)$
8: $x = d_4$
9: return x

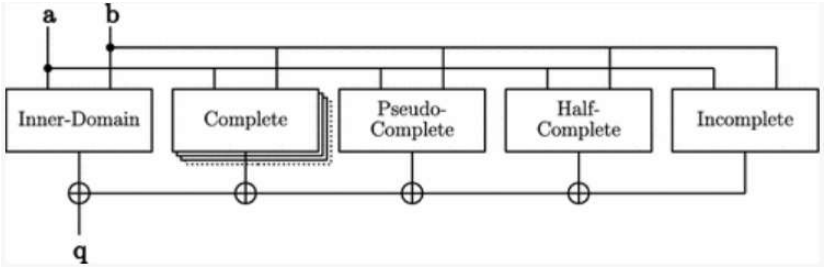
This is the (serial) HW counterpart..

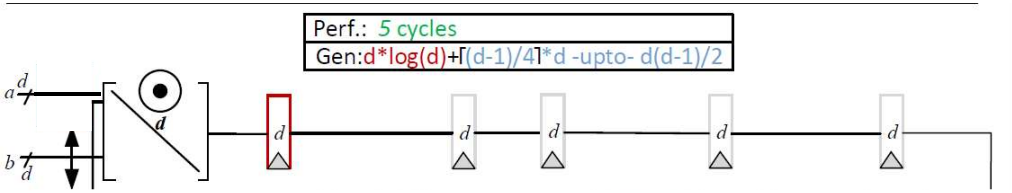


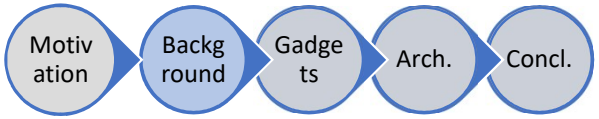
(b) Serial Barth. *et al* . + one-input refresh (adapted to HW context)

UCLouvain

- **How to implement this on HW ?**
- HW – operands are too big ….
  - Cool – **let's go parallel.. UMA**
  - **Did not consider needed refreshing …**

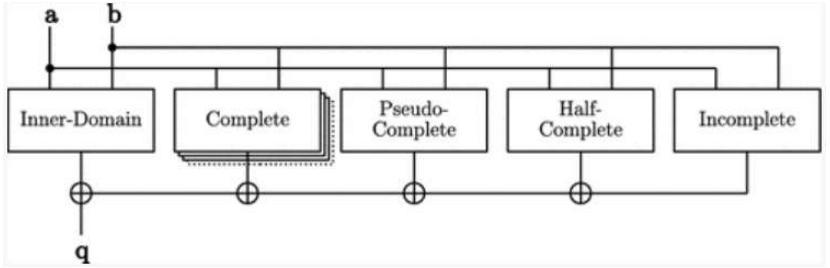


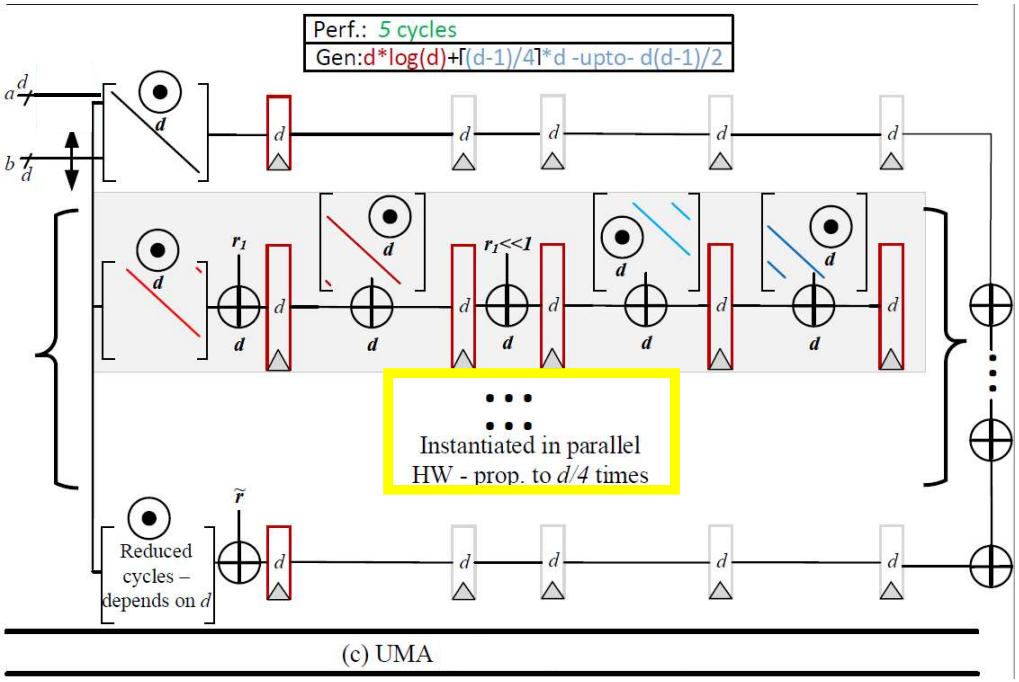




UCLouvain

- **How to implement this on HW ?**
- HW – operands are too big ….
  - Cool – **let's go parallel.. UMA**
  - **Did not consider needed refreshing …**

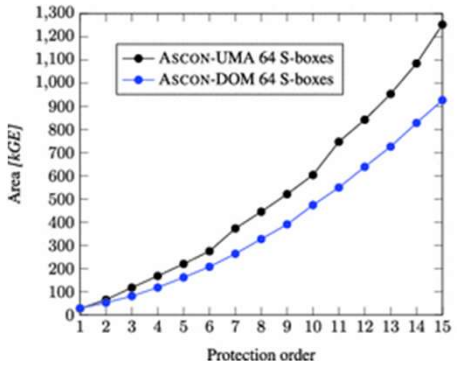


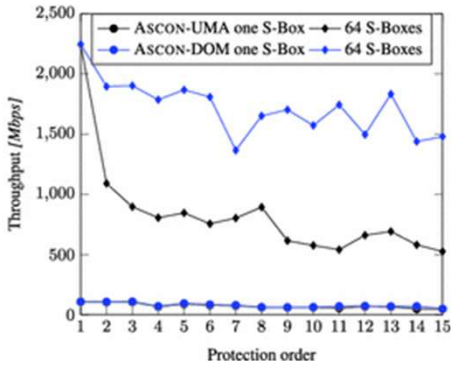


UCLouvain

Generally, this is the tradeoff we will get:

- HW
  - ~Order-indep. latency
  - Pay on that in Area..





A unified masking approach
[Hannes Gross, Stefan Mangard]

- SW
  - Unacceptable latency with increasing *d* (for certain applications)

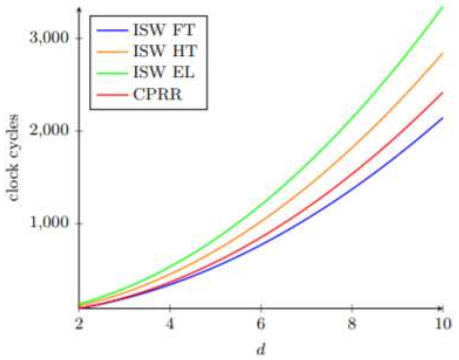
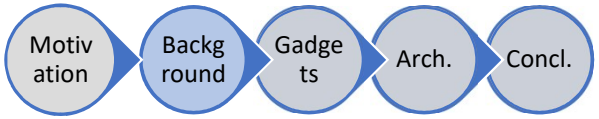
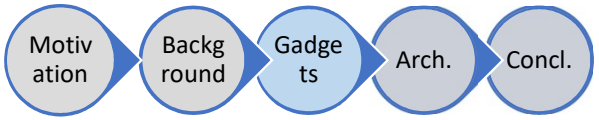


Fig. 2. Timings of ISW and CPRR schemes.

How Fast Can Higher-Order Masking Be in Software?
[D. Goudarzi, M. Rivain]



UCLouvain



Outline:

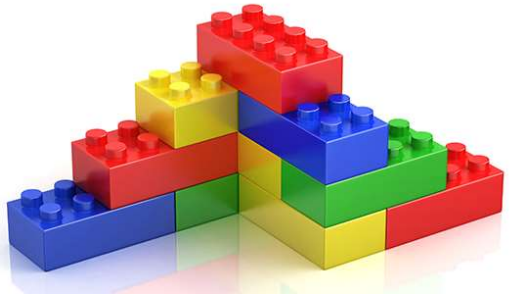
Motivation

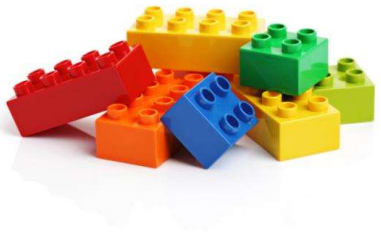
- Necessary background – masking, in HW/SW & tradeoffs
- Improved gadgets on HW
- Building a BC and evaluating its performance

Discussion

UCLouvain

- **So what will we do ?**
  1. Build more HW efficient gadgets
     1. Multiplication
     2. Refresh
  2. Reduce randomness cost
  3. Utilize the asymmetry of the 1-input refresh for more efficient Sboxes
  4. Build a nice and generic/modular code & Implement a system

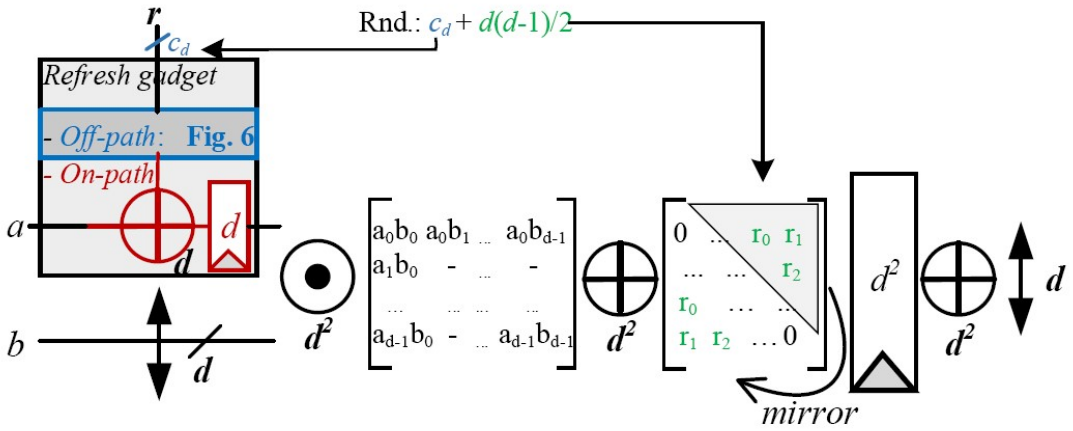




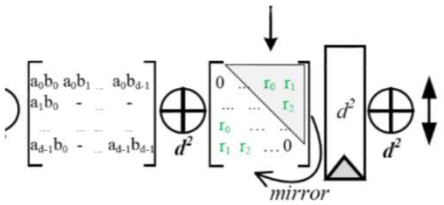
UCLouvain

Multiplication

- Removing output register – still composable



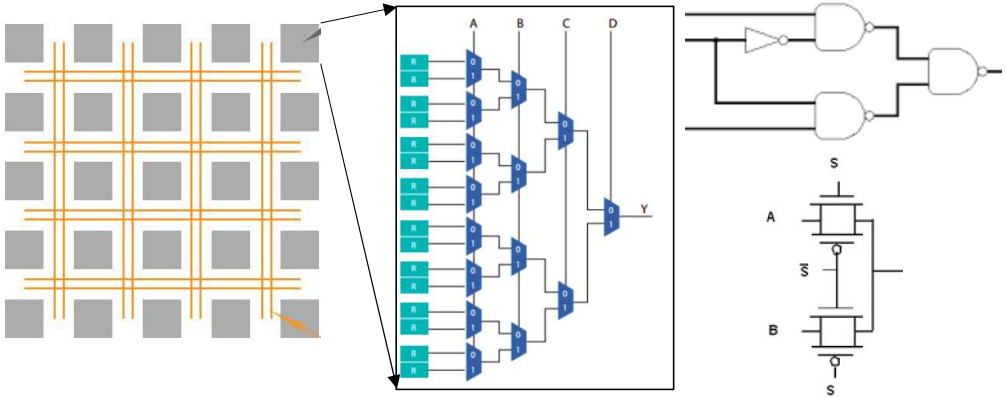
UCLouvain



Multiplication

A word on composability on FPGA and making sure we do not do dangerous things …

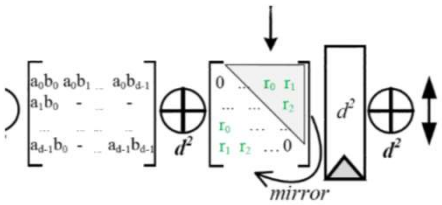
Actel
XILINX
ALTERA
now part of Intel



$a_1, a_2, a_3 \rightarrow a_1a_2, a_1a_3$ in a single LUT is DANGEROUS ! We have no control..

On many parts of the design we need to restrict the tool optimization

UCLouvain



Multiplication

$a_1, a_2, a_3 \rightarrow a_1a_2, a_1a_3$ in a single LUT is DANGEROUS ! We have no control..

On many parts of the design we need to restrict the tool optimization.

- On the tool - prevent *LUTs merging* and *FFs optimization*, and on the design:

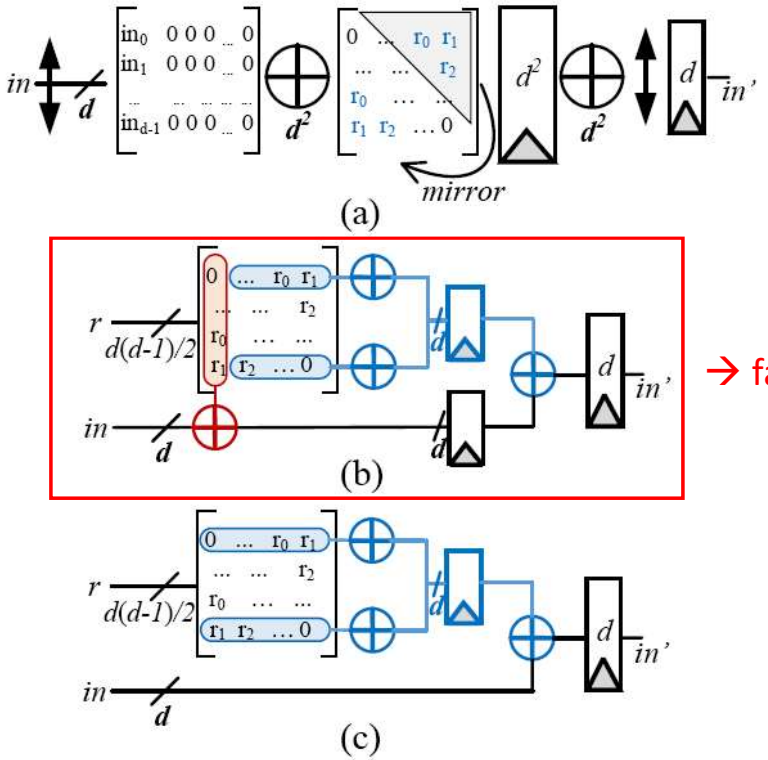
```
`ifdef ISW

    genvar i,j;
    generate
        for(i=0; i<d; i=i+1)
            begin: ParProdI
            for(j=0; j<d; j=j+1)
                begin: ParProdJ
                (* keep = "true" *) wire mult_wire ;
                        assign mult_wire = ina[i] & inb[j];
                end
            end
    endgenerate
```

```
generate
    for(i=0; i<d; i=i+1)
        begin: RefreshedI
        for(j=0; j<d; j=j+1)
            begin: RefreshedJ
            (* keep = "true" *) wire rfrsh_wire ;
                if (i!=j) begin
                    if (i>j) begin
                        assign rfrsh_wire = ParProdI[i].ParProdJ[j].mult_wire ^ rin_mat[i][j];
                    end
                    else begin
                        assign rfrsh_wire = ParProdI[i].ParProdJ[j].mult_wire ^ rin_mat[i][j];
                    end
                end
                else begin
                    assign rfrsh_wire = ParProdI[i].ParProdJ[j].mult_wire; end
                end
            end
endgenerate
```

UCLouvain

- Refresh – we compare to..
  - Low cost HW version of ISW
  - The randomness cost is d(d-1)/2

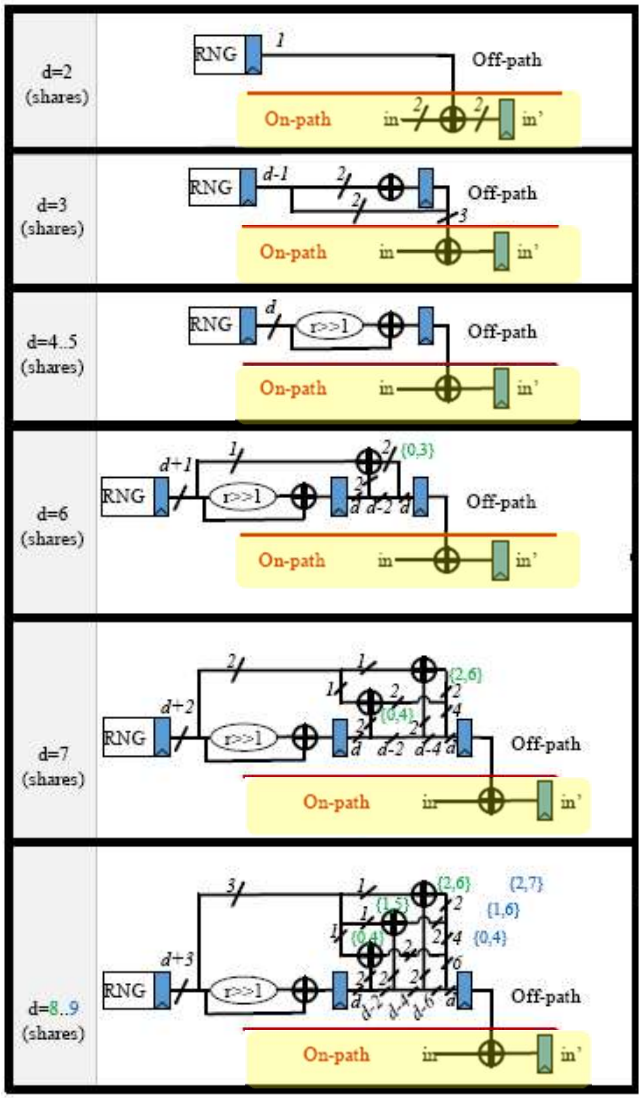


→ fair comparison

UCLouvain

Refresh

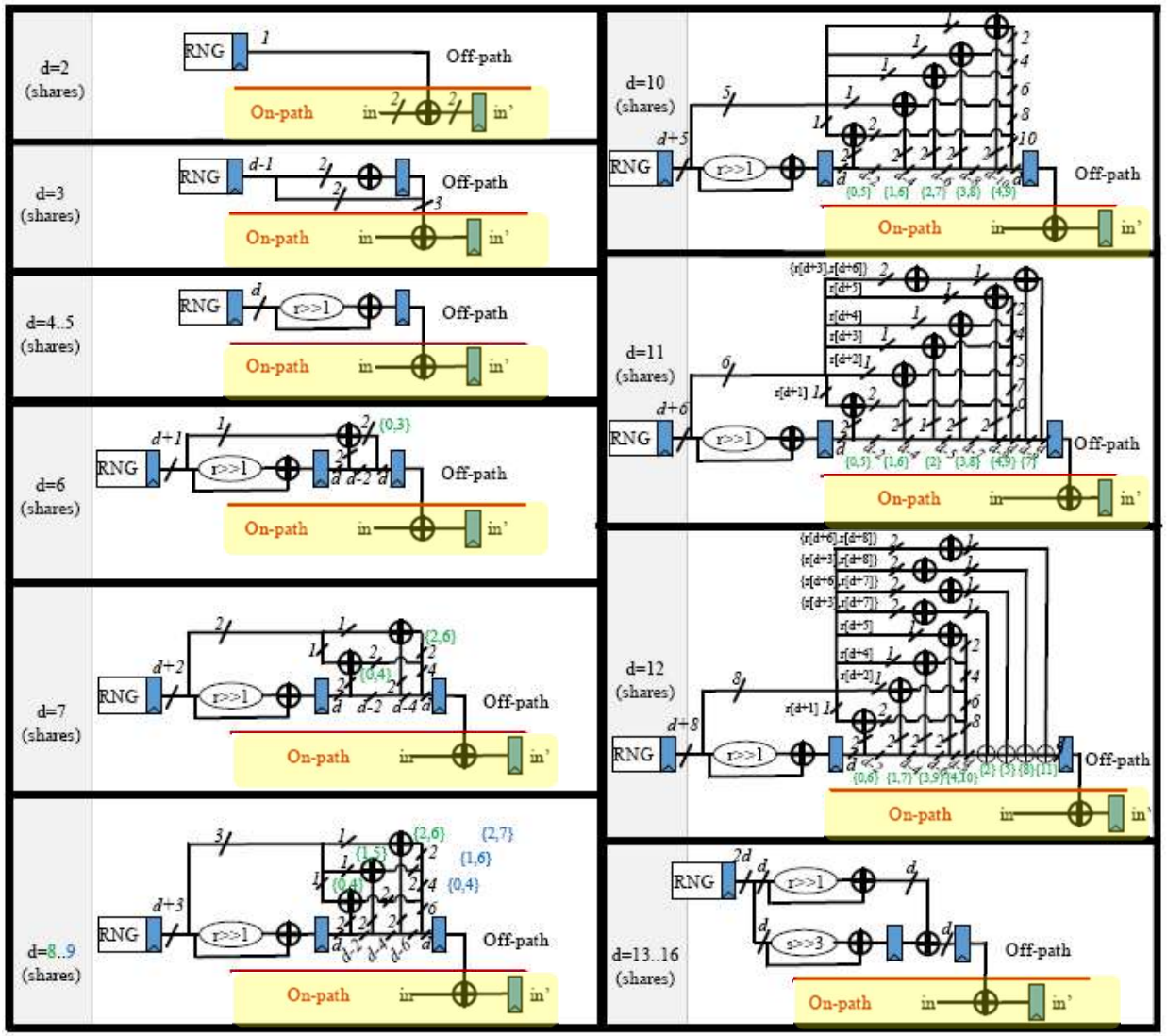
- Saving randomness (thanks Gaetan.C/Benjamin)
  - Some intuition
  - Verifying with *MaskVerif*
- Removing unnecessary computations off the critical path



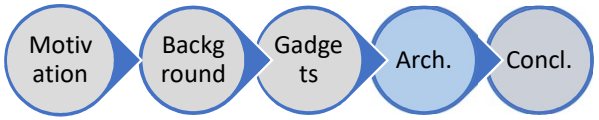
UCLouvain

Refresh

- Saving randomness (thanks Gaetan.C/Benjamin)
  - Some intuition
  - Verifying with *MaskVerif*
- Removing unnecessary computations off the critical path



UCLouvain



Outline:

Motivation

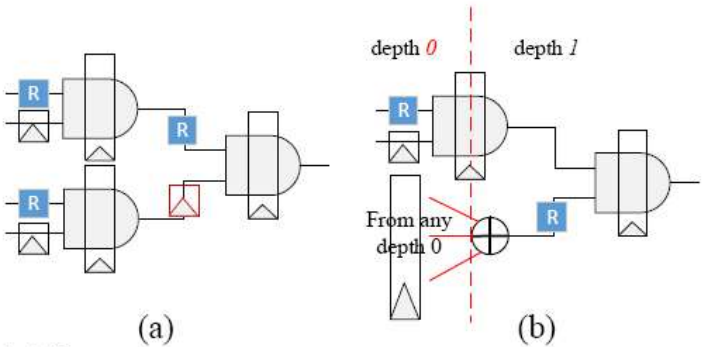
- Necessary background – masking, in HW/SW & tradeoffs
- Improved gadgets on HW
- Building a BC and evaluating its performance

Discussion

Objectives :

Building up Sboxes

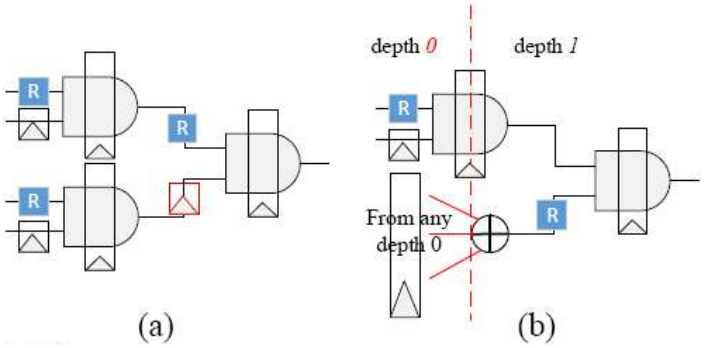
- Make use of inherent asymmetry of one input refresh

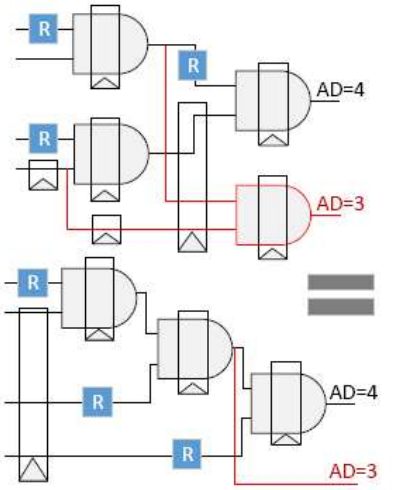


Objectives :

Building up Sboxes

- Make use of inherent asymmetry of one input refresh



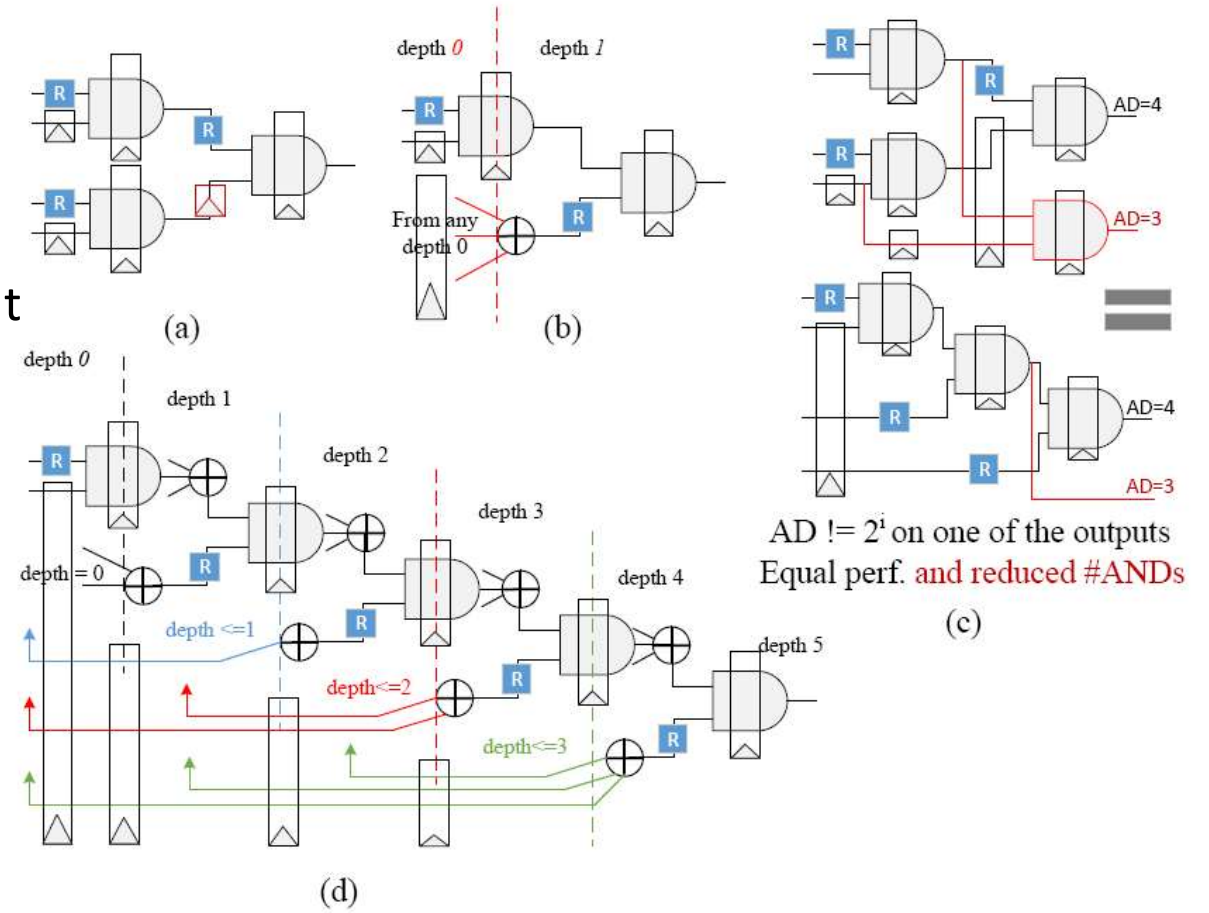


AD != 2^i on one of the outputs
Equal perf. and reduced #ANDs
(c)

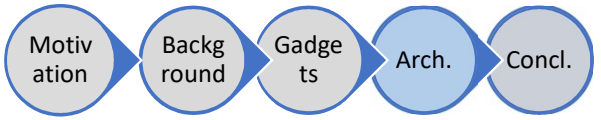
Objectives :

Building up Sboxes

- Make use of inherent asymmetry of one input refresh
  - To reduce latency
  - or, to reduce and counts

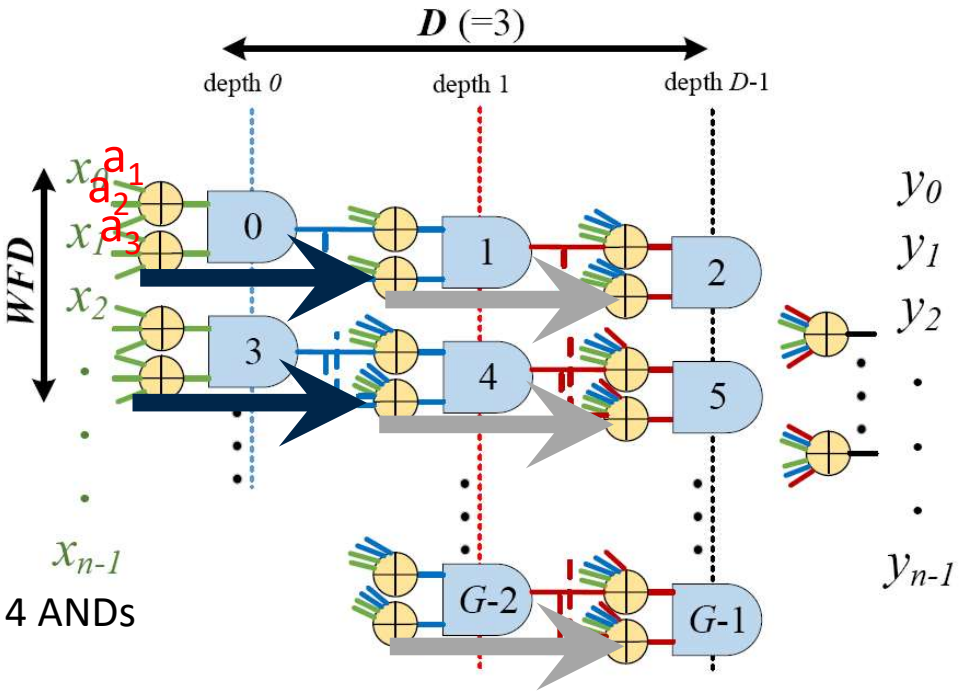


Objectives :



Sboxes are typically logical (LUT) and not topological…

- Ko Stoffelen - build a tool to generate circuits representations
- Input to a SAT solver
  - What the tool generates
    - Constrained structure equations with unknown coefficients coefficients
  - What does the SAT solve
    - Coefficients to match the LUT x,y pairs..
  - Let's modify the tool ….
  - Well start with simple 4bit Sboxes - Depth (MD) 2, 4 ANDs



Objectives :

- Our test case - PRESENT

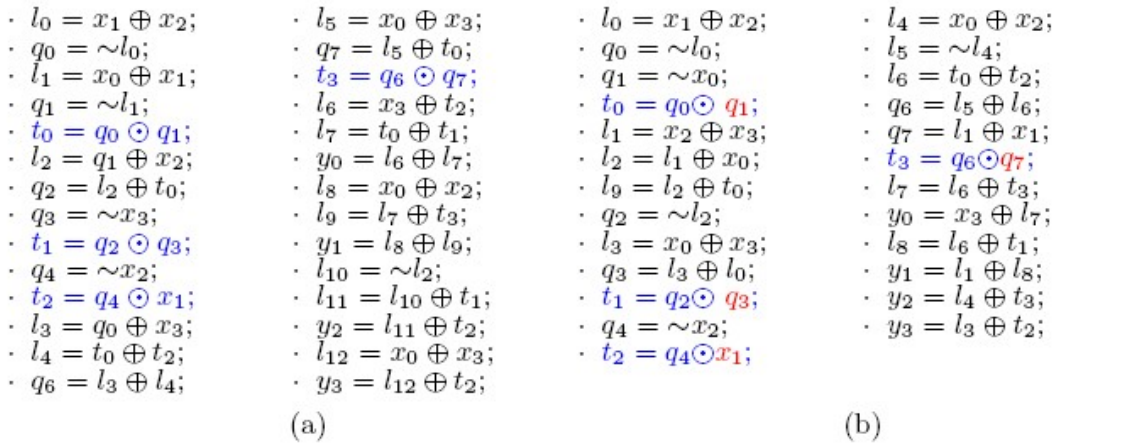
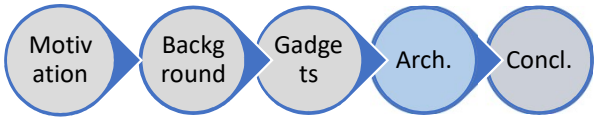


Fig. 8: PRESENT S-box AND depth 2 and 4 AND gates. (a) SAT solution without logical manipulation. (b) SAT solution with *asymmetric-structure*. XOR, AND and NOT are symbolized by $\oplus$, $\odot$ and $\sim$, respectively.

Objectives :



- It is a generic solution
  - For many good Sboxes
  - Finding solutions with the tool for larger ones /higher Depth - becomes hard.

C Additional AND depth 2, 4 ANDs, 4-bit (manipulated) S-boxes

Rectangle S.

- $q_0 = \sim x_0;$
- $l_0 = x_0 \oplus x_2$
- $l_1 = x_0 \oplus x_1$
- $l_3 = l_0 \oplus x_1$
- $q_1 = \sim(l_0);$
- $t_0 = q_0 \odot q_1;$
- $q_2 = \sim(x_0 \oplus x_3 \oplus t_0);$
- $q_3 = \sim(l_3);$
- $t_1 = q_2 \odot q_3;$
- $q_4 = \sim(l_0 \oplus x_3);$
- $q_5 = \sim x_2;$
- $t_2 = q_4 \odot q_5;$
- $q_6 = l_0 \oplus x_1 \oplus t_2;$
- $q_7 = l_1 \oplus x_3;$
- $t_3 = q_6 \odot q_7;$
- $l_2 = t_1 \oplus t_2;$
- $y_0 = l_0 \oplus t_0 \oplus l_2;$
- $y_1 = l_3 \oplus l_2 \oplus t_3;$
- $y_2 = l_1 \oplus x_3 \oplus t_0;$
- $y_3 = l_1 \oplus t_0 \oplus t_2;$

Rectangle S^{-1}.

- $l_0 = x_1 \oplus x_2;$
- $l_1 = l_0 \oplus x_3;$
- $l_2 = x_0 \oplus l_1;$
- $t_0 = x_0 \odot x_3;$
- $q_2 = \sim l_0 \oplus t_0;$
- $q_3 = \sim t_0 \oplus x_2;$
- $t_1 = q_2 \odot q_3;$
- $q_4 = \sim x_0 \oplus x_1;$
- $t_2 = q_4 \odot x_3;$
- $q_6 = l_0 \oplus t_2;$
- $q_7 = \sim x_2;$
- $t_3 = q_6 \odot q_7;$
- $y_0 = l_2 \oplus t_1 \oplus t_2;$
- $y_1 = l_2 \oplus t_0;$
- $y_2 = l_1 \oplus t_2;$
- $y_3 = l_1 \oplus t_1 \oplus t_3;$

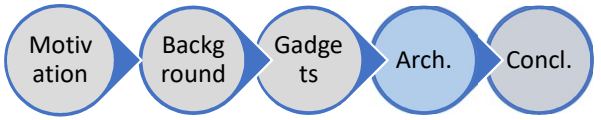
Class-13 S.

- $l_0 = x_0 \oplus x_1;$
- $l_1 = l_0 \oplus x_2;$
- $q_0 = x_1 \oplus x_3;$
- $l_2 = q_0 \oplus x_2;$
- $q_1 = \sim l_2;$
- $t_0 = q_0 \odot q_1;$
- $q_2 = l_1 \oplus x_3 \oplus t_0;$
- $q_3 = \sim x_3;$
- $t_1 = q_2 \odot q_3;$
- $q_4 = \sim x_3;$
- $t_2 = q_4 \odot x_2;$
- $l_3 = t_0 \oplus t_2;$
- $q_6 = l_1 \oplus t_2;$
- $q_7 = \sim x_0;$
- $t_3 = q_6 \odot q_7;$
- $y_0 = l_2 \oplus t_2 \oplus t_3;$
- $y_1 = l_0 \oplus l_3;$
- $y_2 = l_1 \oplus t_1 \oplus l_3;$
- $y_3 = x_1 \oplus x_2 \oplus t_2;$

Class-13 S^{-1}.

- $l_0 = x_1 \oplus x_3;$
- $l_1 = l_0 \oplus x_2;$
- $l_2 = x_0 \oplus x_3;$
- $q_0 = \sim l_0;$
- $t_0 = q_0 \odot x_1;$
- $q_2 = l_2 \oplus x_2 \oplus t_0;$
- $t_1 = q_2 \odot x_2;$
- $q_4 = \sim l_2;$
- $q_5 = x_0 \oplus l_0;$
- $t_2 = q_4 \odot q_5;$
- $l_3 = t_0 \oplus t_2;$
- $l_4 = l_3 \oplus t_1;$
- $q_6 = x_2 \oplus t_0;$
- $q_7 = x_0 \oplus x_2;$
- $t_3 = q_6 \odot q_7;$
- $y_0 = x_2 \oplus x_3 \oplus l_4 \oplus t_3;$
- $y_1 = l_1 \oplus l_4;$
- $y_2 = x_1 \oplus x_2 \oplus l_3;$
- $y_3 = l_2 \oplus t_0;$

Objectives :



- It is a generic solution
  - For many good Sboxes
  - Finding solutions with the tool for larger ones /higher Depth - becomes hard.
  - And of coarse we have the SPOOK Sbox..

Skinny S.

- $q_1 = x_0 \oplus x_2;$
- $t_0 = x_3 \odot q_1;$
- $q_2 = x_0 \oplus x_1 \oplus t_0;$
- $t_1 = q_2 \odot x_0;$
- $q_4 = x_3;$
- $q_5 = \sim x_0 \oplus x_3;$
- $t_2 = q_4 \odot q_5;$
- $l_0 = t_1 \oplus t_2;$
- $q_7 = x_1 \oplus x_3;$
- $q_6 = \sim q_1 \oplus q_7 \oplus t_2;$
- $t_3 = q_6 \odot q_7;$
- $y_0 = x_0 \oplus x_3 \oplus l_0;$
- $y_1 = l_0 \oplus t_3;$
- $y_2 = x_1 \oplus t_0 \oplus t_2;$
- $y_3 = x_2 \oplus t_2;$

Skinny S^{-1}.

- $q_1 = x_1 \oplus x_3;$
- $q_0 = q_1 \oplus x_2;$
- $t_0 = q_0 \odot q_1;$
- $l_0 = x_0 \oplus x_1;$
- $q_2 = l_0 \oplus t_0;$
- $q_3 = \sim x_3;$
- $t_1 = q_2 \odot q_3;$
- $l_1 = x_2 \oplus x_3;$
- $q_5 = \sim l_1;$
- $t_2 = x_2 \odot q_5;$
- $q_6 = l_0 \oplus x_2 \oplus t_2;$
- $q_7 = x_0 \oplus x_2;$
- $t_3 = q_6 \odot q_7;$
- $y_0 = x_1 \oplus t_2;$
- $y_1 = q_1 \oplus t_0 \oplus t_1 \oplus t_3;$
- $y_2 = x_0 \oplus l_1 \oplus t_3;$
- $y_3 = x_0 \oplus q_1 \oplus t_0;$

iClass13 S.

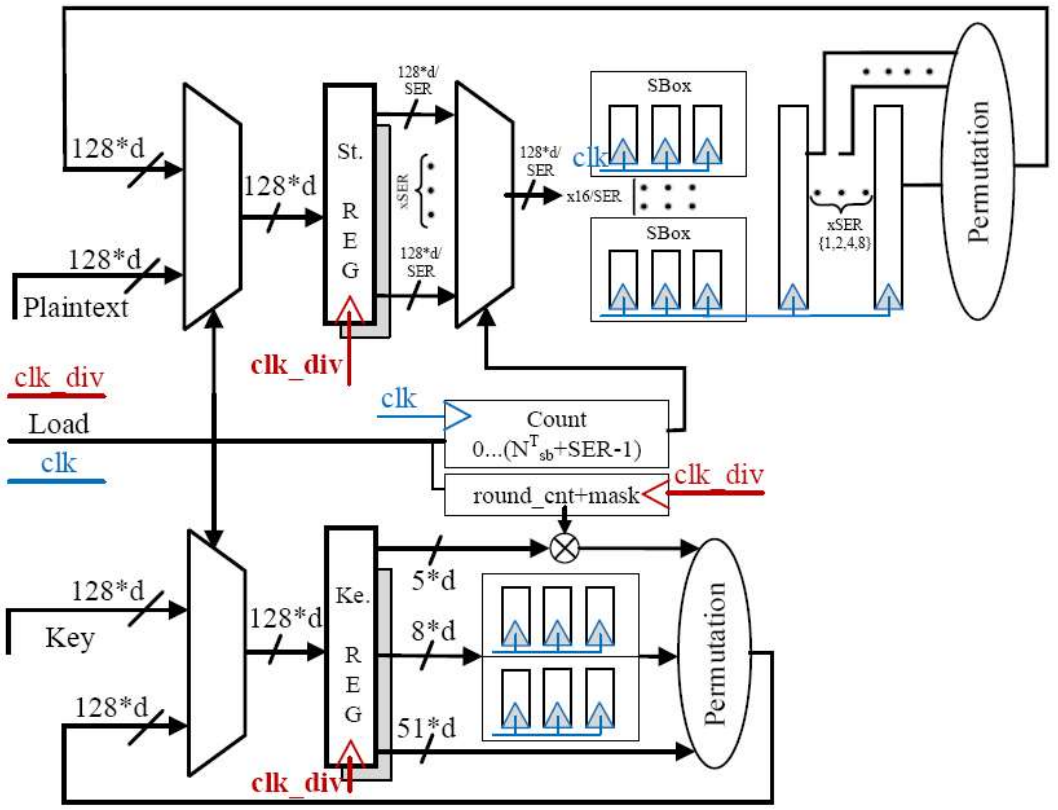
- $l_0 = x_2 \oplus x_3;$
- $l_2 = x_0 \oplus x_3;$
- $q_0 = \sim x_1;$
- $t_0 = q_0 \odot x_3;$
- $q_2 = l_2 \oplus t_0;$
- $q_3 = \sim x_2;$
- $t_1 = q_2 \odot q_3;$
- $q_4 = l_0;$
- $q_5 = \sim x_1;$
- $t_2 = q_4 \odot q_5;$
- $l_1 = t_0 \oplus t_2;$
- $q_6 = \sim x_0 \oplus x_2 \oplus l_1;$
- $q_7 = x_0 \oplus l_0;$
- $t_3 = q_6 \odot q_7;$
- $y_0 = q_7 \oplus x_1 \oplus t_1 \oplus t_2;$
- $y_1 = q_7 \oplus t_2;$
- $y_2 = l_0 \oplus l_1;$
- $y_3 = l_2 \oplus t_1 \oplus t_3;$

Prøst S.

- $q_1 = x_0 \oplus x_2;$
- $q_0 = q_1 \oplus x_1;$
- $t_0 = q_0 \odot q_1;$
- $q_2 = \sim q_1 \oplus x_3 \oplus t_0;$
- $t_1 = q_2 \odot x_0;$
- $q_4 = x_0 \oplus x_1;$
- $q_5 = \sim x_0;$
- $t_2 = q_4 \odot q_5;$
- $l_1 = t_0 \oplus t_2;$
- $q_6 = q_0 \oplus t_2;$
- $t_3 = q_6 \odot x_3;$
- $y_0 = x_1 \oplus x_2 \oplus t_2;$
- $y_1 = q_0 \oplus x_3 \oplus l_1;$
- $y_2 = q_1 \oplus t_0 \oplus t_1 \oplus t_3;$
- $y_3 = q_1 \oplus l_1 \oplus t_3;$

UCLouvain

- And, a full architecture..
  - Parametric:
    - **SER**IALIZATION / d / refr.

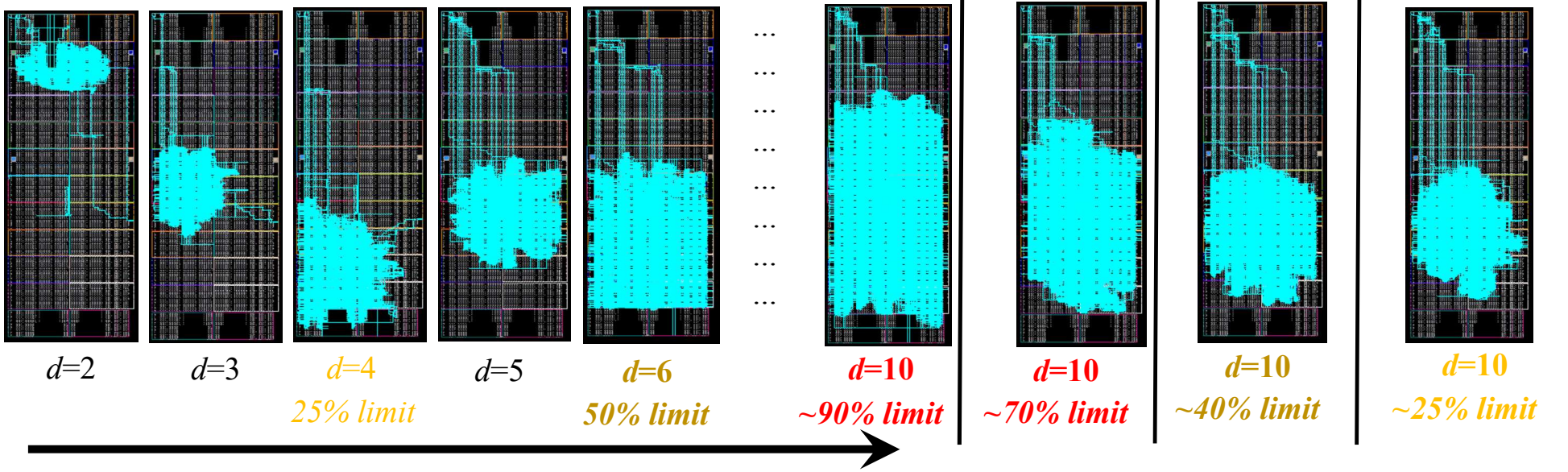


Area bottleneck :

Spartan 6 LX75 FPGA, Xilinx

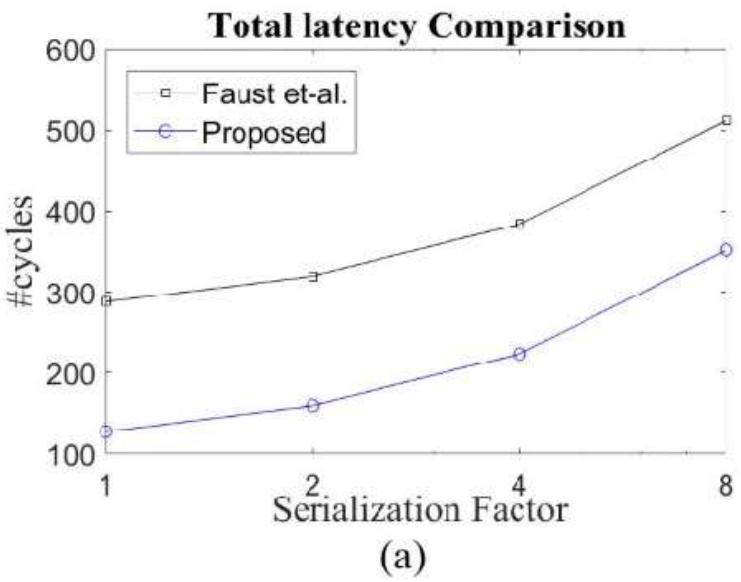
- **SER**IALIZATION
- Fully parallel (SER=1)
  - 5 cycles/round

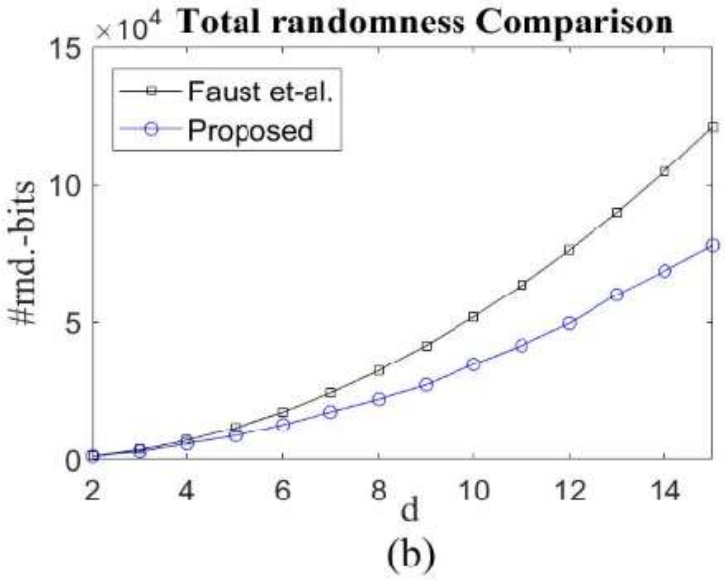
SER=2 6 cycles/round | SER=4 8 cycles/round | SER=8 12 cycles/round



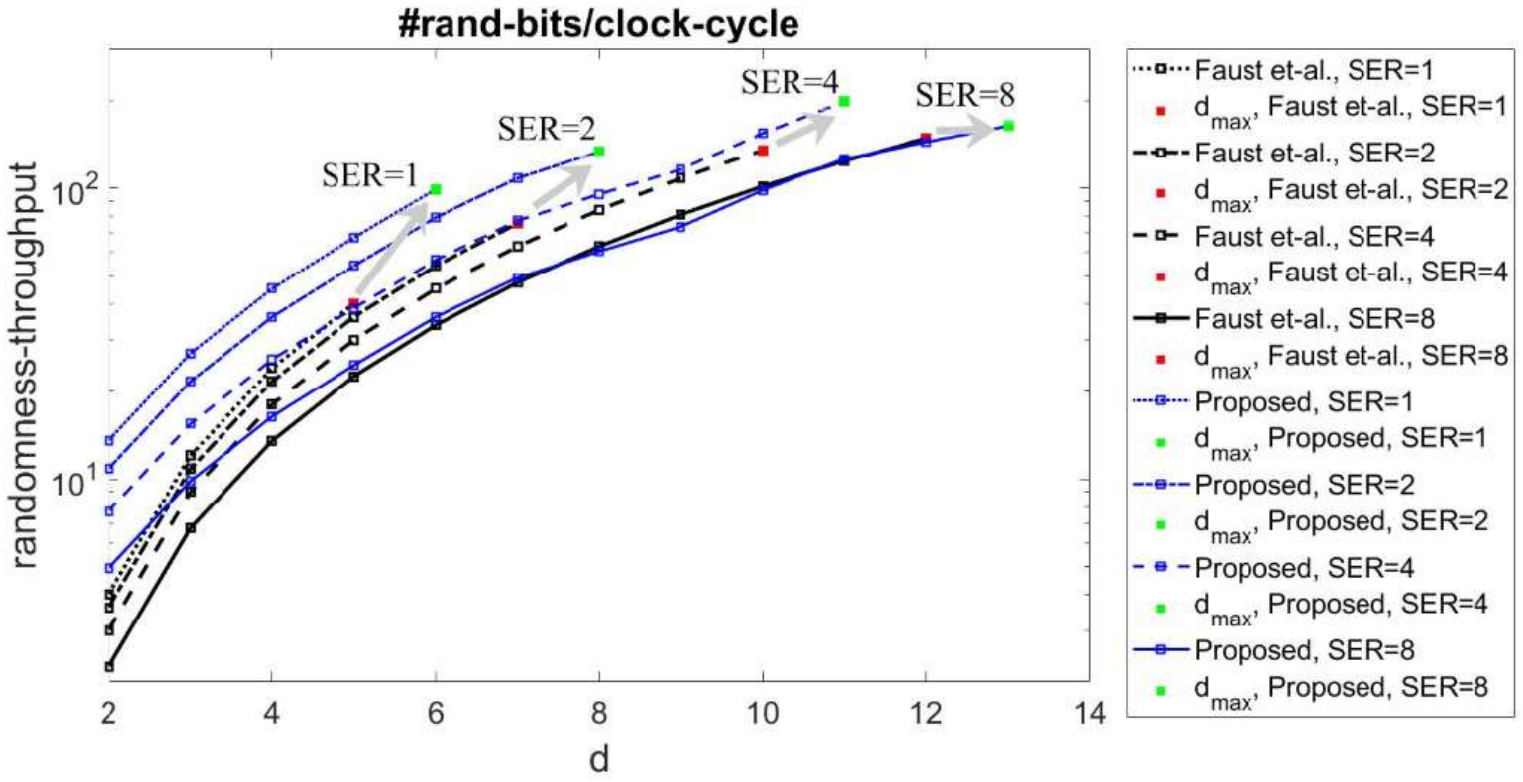
Latency and randomness :

- Considerable latency and randomness savings for the whole cypher



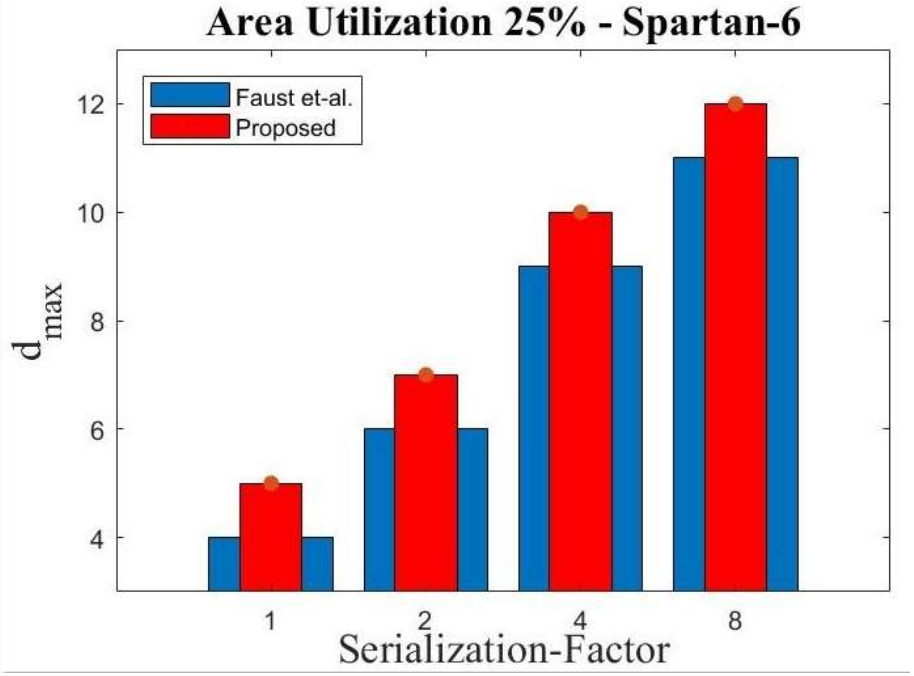


Randomness throughput :

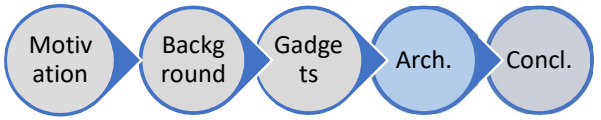


Higher security per area:

- We can get more per area, not just latency..



Latency break-down:



- Less on computation and considerably less on refresh

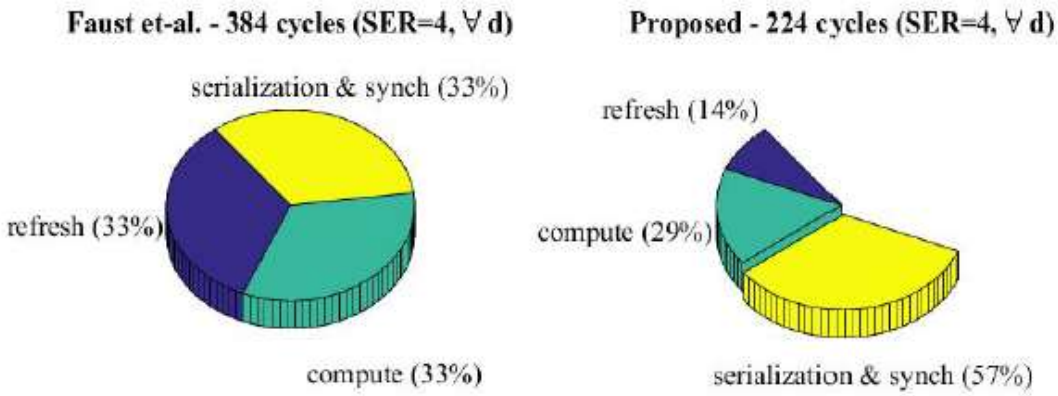
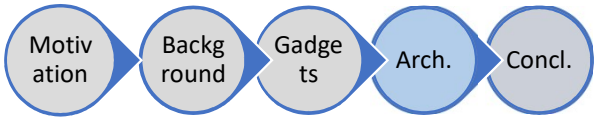
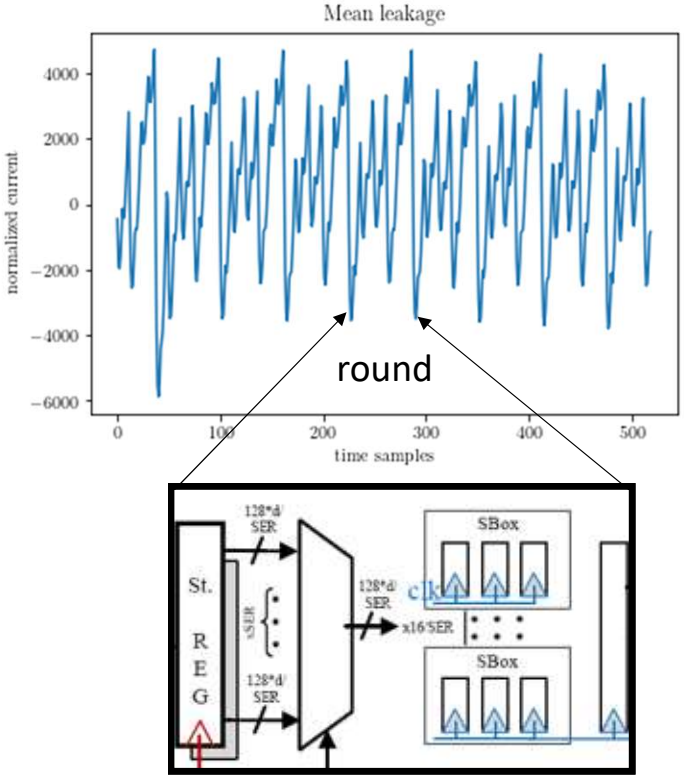


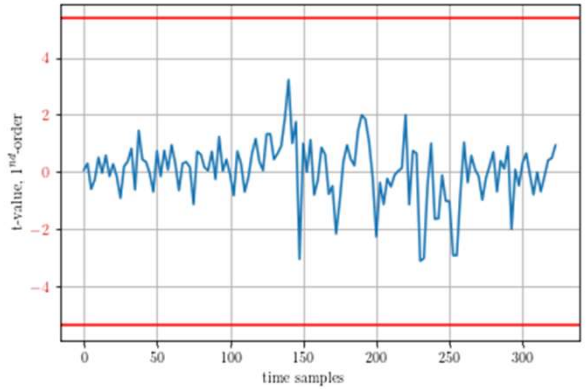
Fig. 13: Total latency cost (#Cycles) break-down to {*computation*, *refreshing*, and *synchronization*} components vs. *SER* for a full encryption: (a) Faust et al. design (b) The proposed architecture.

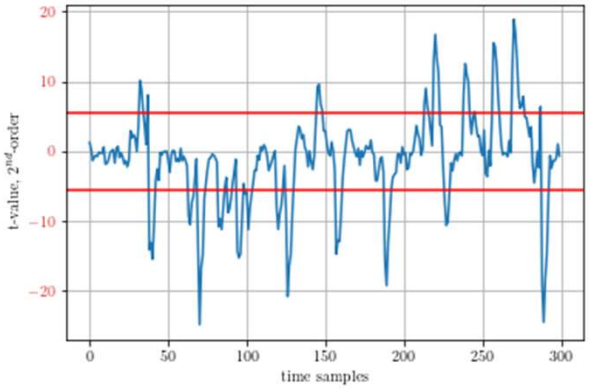
Security evaluation:



- Just a taste… (d=2)
  - A fully parallel design ..







Conclusions:

- Ultra low-latency masking from composable gadgets
- The gadgets enable considerable savings on higher hierarchical levels
- Reduced area utilization:
  - Larger security order /area
- Reduced randomness cost and randomness throughput (from e.g. a TRNG)

Thank you