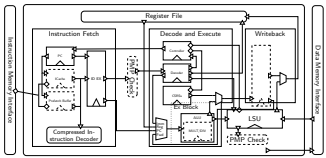
Micro-architecture des processeurs et vérification d'implémentations masquées

Karine HEYDEMANN, Thales/Sorbonne Université

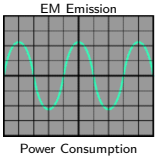
Sécurité pour et par le matériel, 4 Mai 2026

Side-Channel Attacks

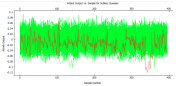
Processor / Hardware



Side-Channels



Trace(s)



Secret Leakage Analysis

Counter-measures Against SCA

Hiding

- Add noise to reduce the signal to noise ratio
- Examples: dummy instruction, instruction or loop shuffling, semantic variants (function or instruction)
- Does not remove leakage but makes it harder to exploit (more traces are needed)

Masking

- Make the manipulated data statistically independent from the secret values
- Can be formally proven
- Measurements are theoretically independent of the secret

Masking Principle

At order d

- Split a secret s into $d+1$ parts (a.k.a shares) s_0, s_1,..., s_d such than $s = s_0 \star s_1 \star ... \star s_d$
  - $s_0, ..., s_{d-1}$ are d uniform randoms (a.k.a "masks")
  - $s_d = s \star s_0 \star s_1 \star ... \star s_{d_1}$
- Any combination of less than d shares is statistically independant from the secret
- First-order boolean masking:
  - s_0 is a uniform random
  - $s_1 = s_0 \oplus s$
- Algorithms must be rewritten to work with shared data

Masking of a "AND" Operation

- Consider 2 boolean shared values $a = (a0, a1)$ and $b = (b0, b1)$ at order 1
- How to **securely** compute c, also shared, such that $c = a.b$?

Masking of a "AND" Operation

- Consider 2 boolean shared values a = (a0, a1) and b = (b0,b1) at order 1
- How to **securely** compute c, also shared, such that c = a.b ?
  - We want c0 and c1 such that c0 $\oplus$ c1 = (a0 $\oplus$ a1).(b0 $\oplus$ b1) without computing a and b

$$c0 \oplus c1 = (a0.b0 \oplus a0.b1 \oplus a1.b0 \oplus a1.b1)$$

Masking of a "AND" Operation

- Consider 2 boolean shared values a = (a0, a1) and b = (b0,b1) at order 1
- How to **securely** compute c, also shared, such that c = a.b ?
  - We want c0 and c1 such that c0 $\oplus$ c1 = (a0 $\oplus$ a1).(b0 $\oplus$ b1) without computing a and b

    c0 $\oplus$ c1 = (a0.b0 $\oplus$ a0.b1 $\oplus$ a1.b0 $\oplus$ a1.b1)
  - We need to compute all the products (.) and reduce the computation ($+$)

Masking of a "AND" Operation

- Consider 2 boolean shared values a = (a0, a1) and b = (b0,b1) at order 1
- How to **securely** compute c, also shared, such that c = a.b ?
  - We want c0 and c1 such that c0 $\oplus$ c1 = (a0 $\oplus$ a1).(b0 $\oplus$ b1) without computing a and b

    c0 $\oplus$ c1 = (a0.b0 $\oplus$ a0.b1 $\oplus$ a1.b0 $\oplus$ a1.b1)
  - We need to compute all the products (.) and reduce the computation ($+$)
  - But any reduction of two terms leads to a leakage of a or b

    e.g c0 = a0.b0 $\oplus$ a0.b1, c1 = a1.b0 $\oplus$ a1.b1 leaks b

    e.g c0 = a0.b0 $\oplus$ a1.b1, c1 = a1.b0 $\oplus$ a0.b1 leaks a and b

Masking of a "AND" Operation

- Consider 2 boolean shared values a = (a0, a1) and b = (b0,b1) at order 1
- How to **securely** compute c, also shared, such that c = a.b ?
  - We want c0 and c1 such that c0 $\oplus$ c1 = (a0 $\oplus$ a1).(b0 $\oplus$ b1) without computing a and b

    c0 $\oplus$ c1 = (a0.b0 $\oplus$ a0.b1 $\oplus$ a1.b0 $\oplus$ a1.b1)
  - We need to compute all the products (.) and reduce the computation ($+$)
  - But any reduction of two terms leads to a leakage of a or b

    e.g c0 = a0.b0 $\oplus$ a0.b1, c1 = a1.b0 $\oplus$ a1.b1 leaks b

    e.g c0 = a0.b0 $\oplus$ a1.b1, c1 = a1.b0 $\oplus$ a0.b1 leaks a and b
  - **Additional randoms** are necessary to make the computation secure

Masking of a "AND" Operation

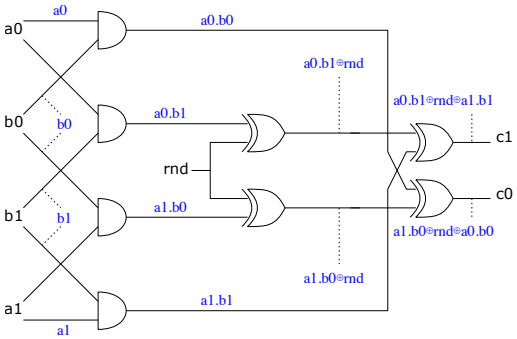
- Consider 2 boolean shared values a = (a0, a1) and b = (b0,b1) at order 1
- How to **securely** compute c, also shared, such that c = a.b ?
  - We want c0 and c1 such that c0 $\oplus$ c1 = (a0 $\oplus$ a1).(b0 $\oplus$ b1) without computing a and b

    c0 $\oplus$ c1 = (a0.b0 $\oplus$ a0.b1 $\oplus$ a1.b0 $\oplus$ a1.b1)
  - We need to compute all the products (.) and reduce the computation ($+$)
  - But any reduction of two terms leads to a leakage of a or b

    e.g c0 = a0.b0 $\oplus$ a0.b1, c1 = a1.b0 $\oplus$ a1.b1 leaks b

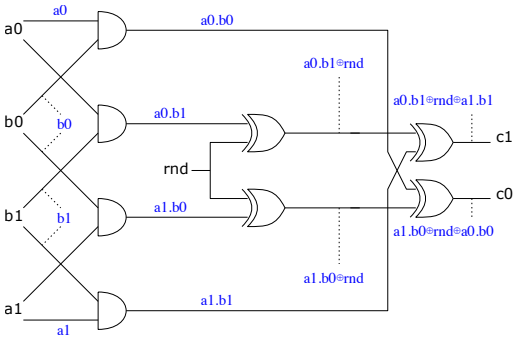
    e.g c0 = a0.b0 $\oplus$ a1.b1, c1 = a1.b0 $\oplus$ a0.b1 leaks a and b
  - **Additional randoms** are necessary to make the computation secure
- Different masking schemes have been proposed ISW-AND [Ishai et al., 2003], DOM-AND [Gross et al., 2016], TI-AND [Nikova et al., 2006]

Example: Masked AND at Order 1



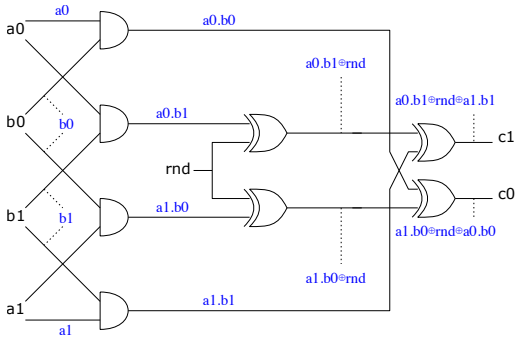

```
1  void masked_and(uint8_t a0, uint8_t a1,
2                  uint8_t b0, uint8_t b1,
3                  uint8_t rnd,
4                  uint8_t *c0, uint8_t *c1)
5
6  *c0 = ((a0 & b0) ^ rnd) ^ (a1 & b1);
7  *c1 = ((a0 & b1) ^ rnd) ^ (a1 & b0);
8  return;
```

Example: Masked AND at Order 1




```c
void masked_and(uint8_t a0, uint8_t a1,
                uint8_t b0, uint8_t b1,
                uint8_t r,
                uint8_t *c0, uint8_t *c1)
{
  uint8_t tmp = (a0 & b1) ^ r;
  __asm__ __volatile__ ("" ::: "memory");
 *c0 = tmp ^ (a1 & b0);
 tmp = (a0 & b0) ^ r;
  __asm__ __volatile__ ("" ::: "memory");
 *c1 = tmp ^ (a1 & b1);
 return;
}
```

Example: Masked AND at Order 1

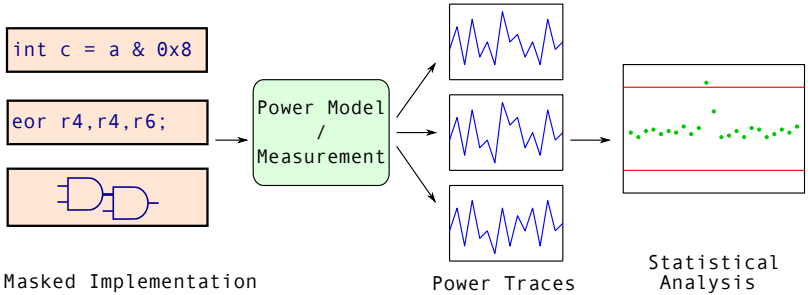



```
1  ;r0:a0, r1:b0, r2:a1, r3:b1, r6:c[] r7:r
2
3  and.w r4, r0, r3 ; a0 & b1
4  eors  r4, r7     ; t0 = (a0 & b1) ^ r
5  and.w r5, r2, r1 ; a1 & b0
6  ands  r0, r1     ; a0 & b0
7  ands  r3, r2     ; b1 & a1
8  eors  r4, r5     ; t1 = t0 ^ (a1 & b0)
9  eors  r0, r7     ; c0 = (a0 & b0) ^ r
10 eors  r4, r3     ; c1 = t1 ^ (a1 & b1)
11 str   r0, [r6, #0]
12 str   r4, [r6, #4]
```

How To Verify a Masked Implementation?

Empirically

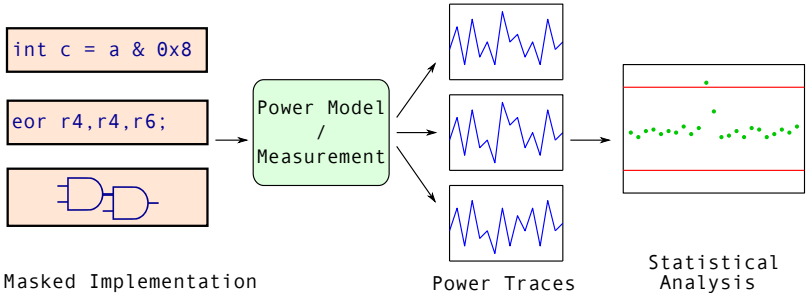
- Perform power simulations or acquisitions then use statistical metrics, such as the t-test



How To Verify a Masked Implementation?

Empirically

- Perform power simulations or acquisitions then use statistical metrics, such as the t-test



Pros : Complex circuits/software analysis

Cons : No guarantee, difficult leakage location identification

- MAPS [Corre et al., 2018], PROLEAD [Müller and Moradi, 2022], ELMO [McCann et al., 2017] or ROSITA [Shelton et al., 2021]

How To Verify a Masked Implementation?

Formally

- Label input values as secret, mask or public value
- Prove the absence of leakage for a chosen **leakage model** for any input values

How To Verify a Masked Implementation?

Formally

- Label input values as secret, mask or public value
- Prove the absence of leakage for a chosen **leakage model** for any input values

Leakage model

- Information that can be observed by an attacker
- **Value-based leakage model**: value of intermediate computations, registers, wires
- **Transition-based leakage**: transition in variables, registers or wires
- **Glitch-based leakage**: all intermediate values of a gate (hardware only)

How To Verify a Masked Implementation?

Formally

- Label input values as secret, mask or public value
- Prove the absence of leakage for a chosen **leakage model** for any input values

Leakage model

- Information that can be observed by an attacker
- **Value-based leakage model**: value of intermediate computations, registers, wires
- **Transition-based leakage**: transition in variables, registers or wires
- **Glitch-based leakage**: all intermediate values of a gate (hardware only)

Security property

- d-**probing security** [Ishai et al., 2003]: secure for d probes observing values
- d-**probing security with transition**: d probes observing transitions
- **Robust** d-**probing security** [Faust et al., 2018]: also observing glitches (hardware only)

How To Verify a Masked Implementation?

Formally

- Model what can be observed, verify a security property

How To Verify a Masked Implementation?

Formally

- Model what can be observed, verify a security property

Example for 1-probing security

1. build symbolic expressions representing intermediate computations
2. prove their statistical independence with secret

Masked Implementation

```
tmp = (a0 & b1) ^ rnd;
*c0 = tmp ^ (a1 & b0);
tmp = (a0 & b0) ^ rnd;
*c1 = tmp ^ (a1 & b1);
```

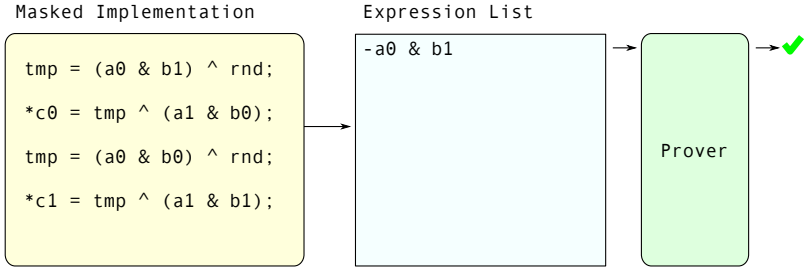
How To Verify a Masked Implementation?

Formally

- Model what can be observed, verify a security property

Example for 1-probing security

1. build symbolic expressions representing intermediate computations
2. prove their statistical independence with secret



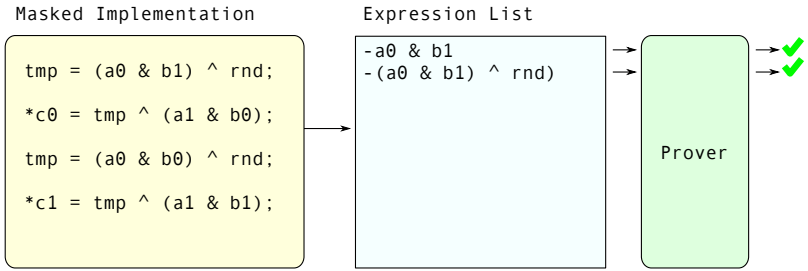
How To Verify a Masked Implementation?

Formally

- Model what can be observed, verify a security property

Example for 1-probing security

1. build symbolic expressions representing intermediate computations
2. prove their statistical independence with secret



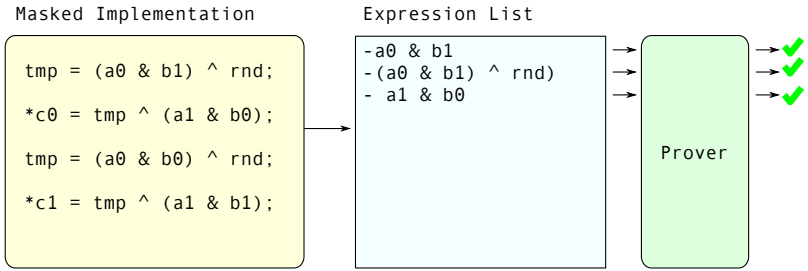
How To Verify a Masked Implementation?

Formally

- Model what can be observed, verify a security property

Example for 1-probing security

1. build symbolic expressions representing intermediate computations
2. prove their statistical independence with secret



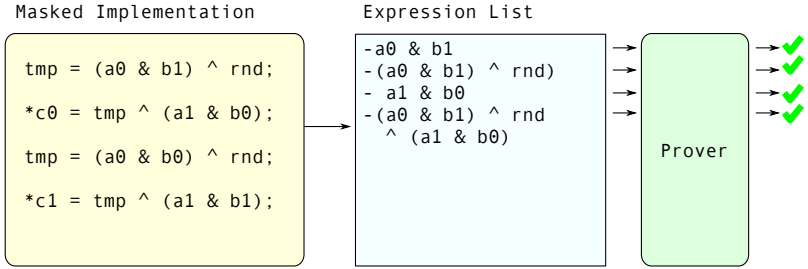
How To Verify a Masked Implementation?

Formally

- Model what can be observed, verify a security property

Example for 1-probing security

1. build symbolic expressions representing intermediate computations
2. prove their statistical independence with secret



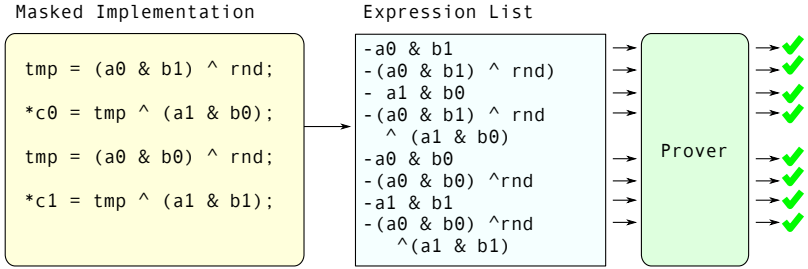
How To Verify a Masked Implementation?

Formally

- Model what can be observed, verify a security property

Example for 1-probing security

1. build symbolic expressions representing intermediate computations
2. prove their statistical independence with secret



How To Verify a Masked Implementation?

Formally

- Model what can be observed, verify a security property

Example for 1-probing security with transition

1. build symbolic expressions representing transitions in variables
2. prove their statistical independence with secret

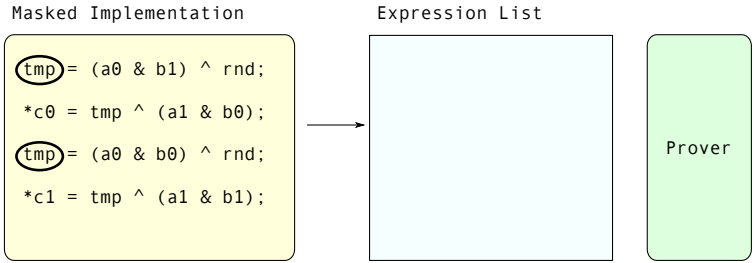
How To Verify a Masked Implementation?

Formally

- Model what can be observed, verify a security property

Example for 1-probing security with transition

1. build symbolic expressions representing transitions in variables
2. prove their statistical independence with secret



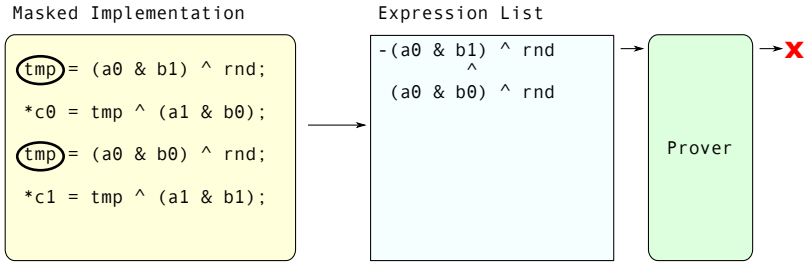
How To Verify a Masked Implementation?

Formally

- Model what can be observed, verify a security property

Example for 1-probing security with transition

1. build symbolic expressions representing transitions in variables
2. prove their statistical independence with secret



Formal Verification of Masked Implementations

Pros

- Guarantee for the chosen leakage model
- Easier to locate and understand leakages

Cons

- Scalability issues
- Potential false positive

Existing verifiers

- MaskVerif [Barthe et al., 2019]
- ARISTI [Ben El Ouahma et al., 2019]
- LeakageVerif [Meunier et al., 2023], VerifMSI [Meunier and Taleb, 2023]
- ...

Proven Leakage-Free Implementation in Practice

- Masked software AND proven leakage-free at the ISA level in the value leakage model and transition leakage model (GPRs).

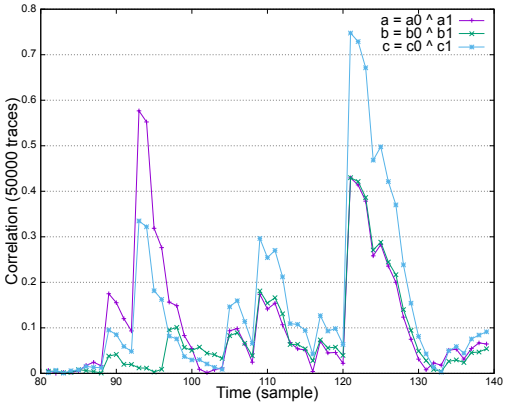
```
;r0:a0, r1:b0, r2:a1, r3:b1, r6:c[] r7:m
and.w r4, r0, r3 ; a0 & b1
eors  r4, r7     ; t0 = (a0 & b1) ^ m
and.w r5, r2, r1 ; a1 & b0
ands  r0, r1     ; a0 & b0
ands  r3, r2     ; b1 & a1
eors  r4, r5     ; t1 = t0 ^ (a1 & b0)
eors  r0, r7     ; c0 = (a0 & b0) ^ m
eors  r4, r3     ; c1 = t1 ^ (a1 & b1)
str   r0, [r6, #0]
str   r4, [r6, #4]

```

Proven Leakage-Free Implementation in Practice

- Masked software AND proven leakage-free at the ISA level in the value leakage model and transition leakage model (GPRs).

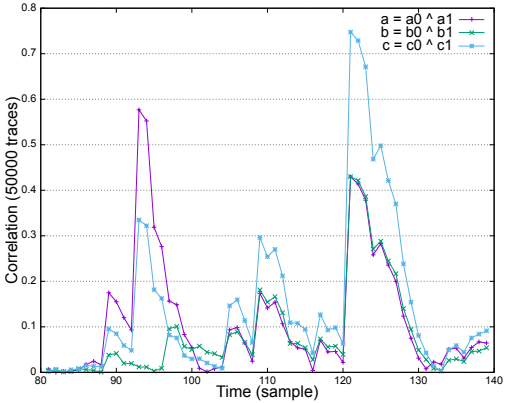
```
1  ;r0:a0, r1:b0, r2:a1, r3:b1, r6:c[] r7:m
2  and.w r4, r0, r3 ; a0 & b1
3  eors  r4, r7     ; t0 = (a0 & b1) ^ m
4  and.w r5, r2, r1 ; a1 & b0
5  ands  r0, r1     ; a0 & b0
6  ands  r3, r2     ; b1 & a1
7  eors  r4, r5     ; t1 = t0 ^ (a1 & b0)
8  eors  r0, r7     ; c0 = (a0 & b0) ^ m
9  eors  r4, r3     ; c1 = t1 ^ (a1 & b1)
10 str   r0, [r6, #0]
11 str   r4, [r6, #4]
12
```



Proven Leakage-Free Implementation in Practice

- Masked software AND proven leakage-free at the ISA level in the value leakage model and transition leakage model (GPRs).

```
1  ;r0:a0, r1:b0, r2:a1, r3:b1, r6:c[] r7:m
2  and.w r4, r0, r3 ; a0 & b1
3  eors  r4, r7     ; t0 = (a0 & b1) ^ m
4  and.w r5, r2, r1 ; a1 & b0
5  ands  r0, r1     ; a0 & b0
6  ands  r3, r2     ; b1 & a1
7  eors  r4, r5     ; t1 = t0 ^ (a1 & b0)
8  eors  r0, r7     ; c0 = (a0 & b0) ^ m
9  eors  r4, r3     ; c1 = t1 ^ (a1 & b1)
10 str   r0, [r6, #0]
11 str   r4, [r6, #4]
12
```



- Need for modelling leakage happening in the circuit at the micro-architectural level while software is executed to capture leakage that can not be modeled at ISA level

ARMISTICE: Micro-Architectural Leakage Modelling for Masked Software Formal Verification

Arnaud de Grandmaison[1], Karine Heydemann, Quentin L. Meunier[2]

published in IEEE Transaction Computer-Aided Design 2022 and presented at the conference CASES 2022

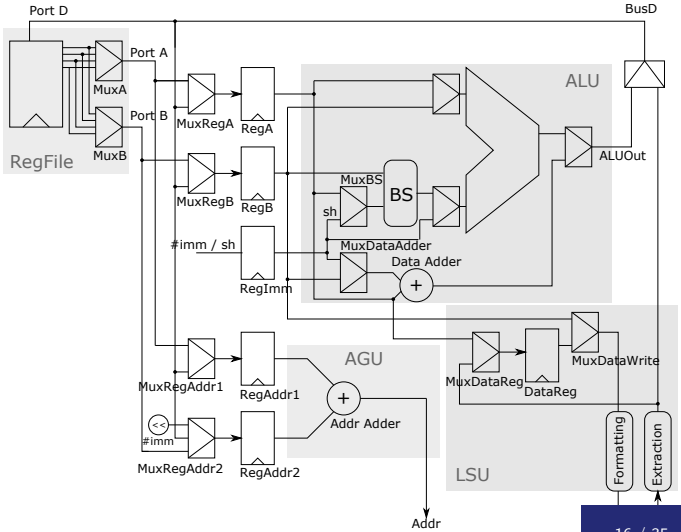
[1]Arm
[2]Sorbonne Université/LIP6

Case Study: Board STM32F1 [De Grandmaison et al., 2022]

- Arm Cortex-M3: modeled from the Verilog source code

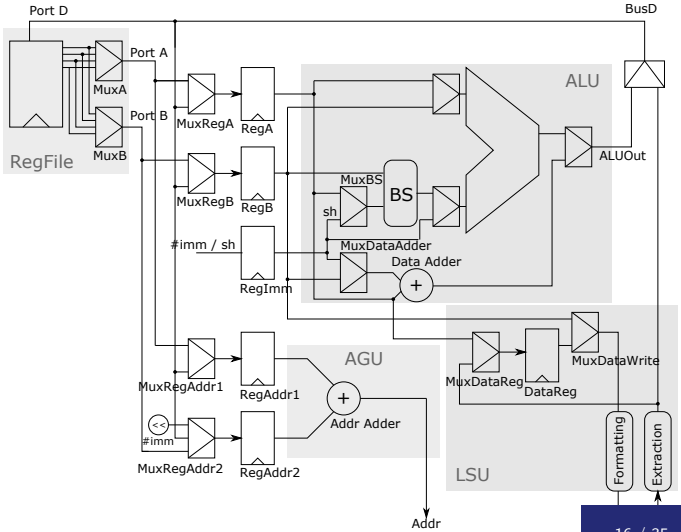
Case Study: Board STM32F1 [De Grandmaison et al., 2022]

- Arm Cortex-M3: modeled from the Verilog source code



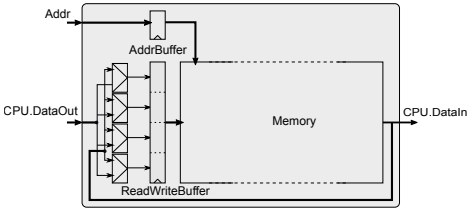
Case Study: Board STM32F1 [De Grandmaison et al., 2022]

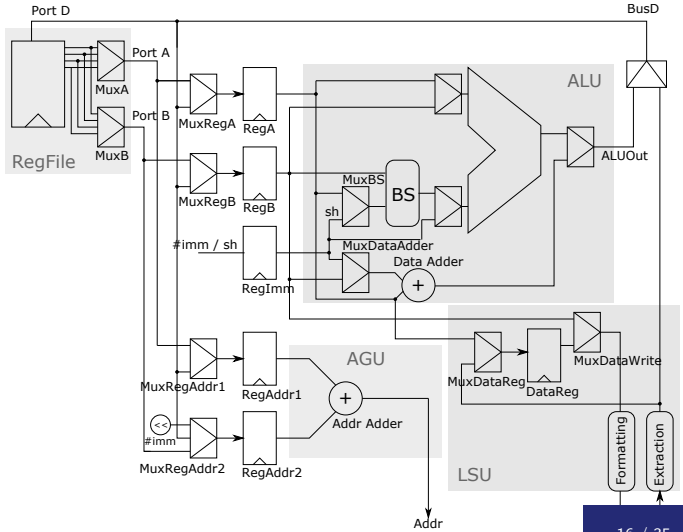
- Arm Cortex-M3: modeled from the Verilog source code
- Memoire: black-box approach (no HDL description available)



Case Study: Board STM32F1 [De Grandmaison et al., 2022]

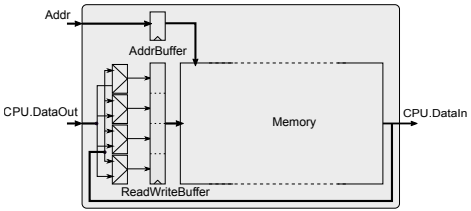
- Arm Cortex-M3: modeled from the Verilog source code
- Memoire: black-box approach (no HDL description available)
- Design of several micro-benchmarks a.k.a. "leakage test vectors":
  - Detection of leakage sources (black-box)

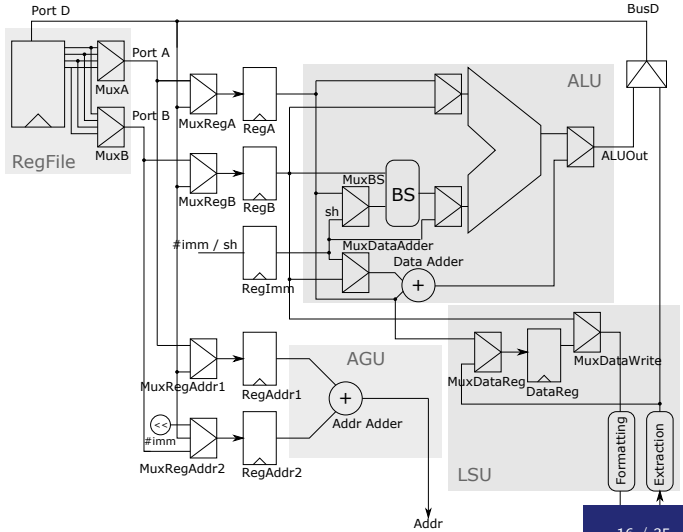




Case Study: Board STM32F1 [De Grandmaison et al., 2022]

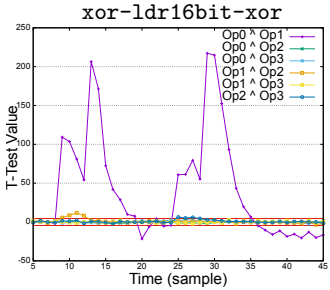
- Arm Cortex-M3: modeled from the Verilog source code
- Memoire: black-box approach (no HDL description available)
- Design of several micro-benchmarks a.k.a. “leakage test vectors”:
  - Detection of leakage sources (black-box)
  - Validation (white-box)
  - Ranking

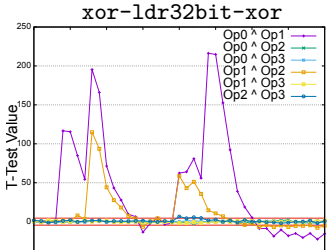




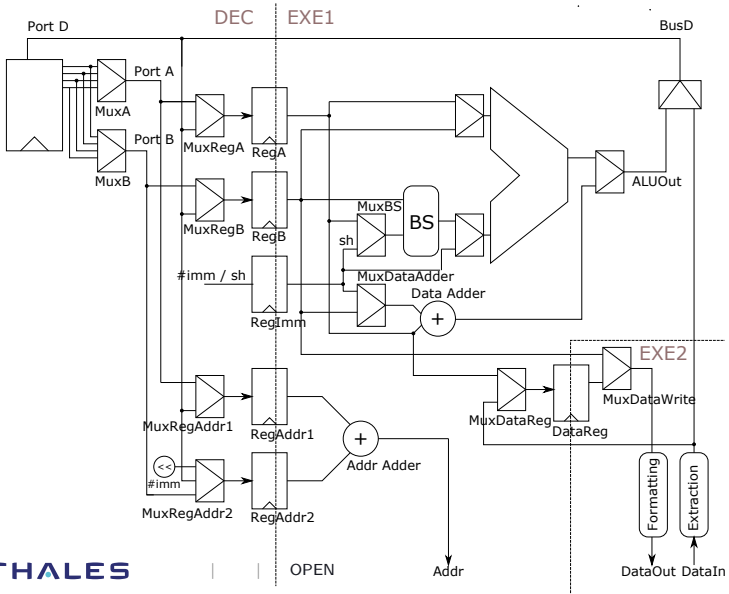
Findings Using Leakage Vectors

- Leakage without any link to the data manipulated by instructions !
  - Instruction encodings (16-bit versus 32-bit) can impact leakage
  - Part of immediate in the encoding can be used to read the register bank
  - Forwarding mechanism
  - ...
- The required number of traces varies with the source of leakage
- We did not have the RTL version corresponding to the CPU of our target !

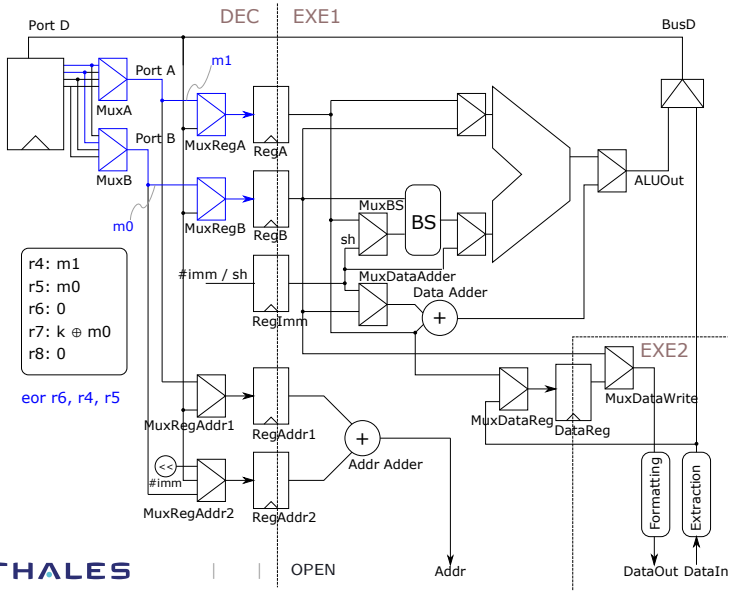




Arm Cortex-M3: Exemple



Arm Cortex-M3: Exemple



PortA: m1

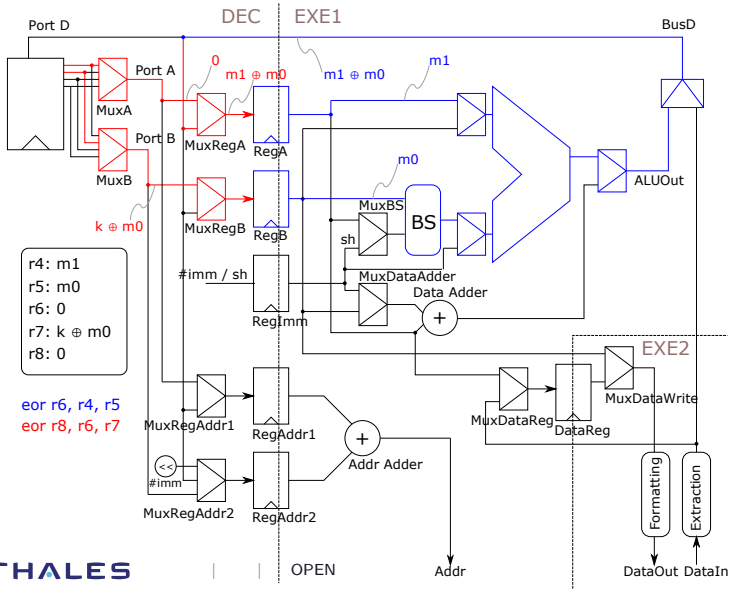
PortB m0

MuxRegA / RegA: m1

MuxRegB / RegB: m0

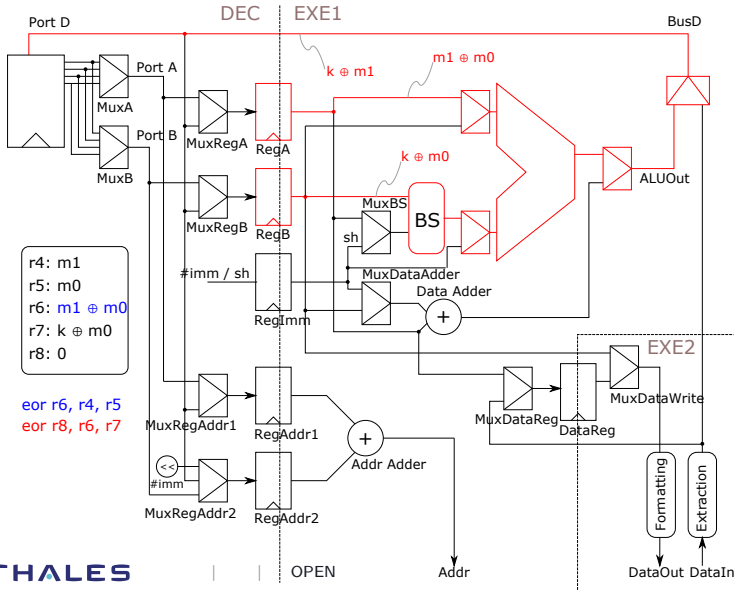
ALUOut

Arm Cortex-M3: Exemple



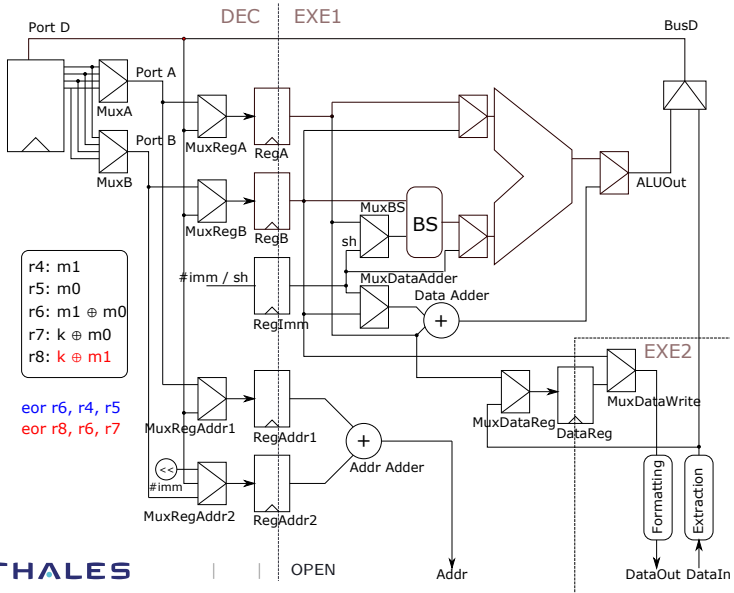
PortA:	m1	0
PortB	m0	k ⊕ m0
MuxRegA / RegA:	m1	m1 ⊕ m0
MuxRegB / RegB:	m0	k ⊕ m0
ALUOut	m1 ⊕ m0	

Arm Cortex-M3: Exemple



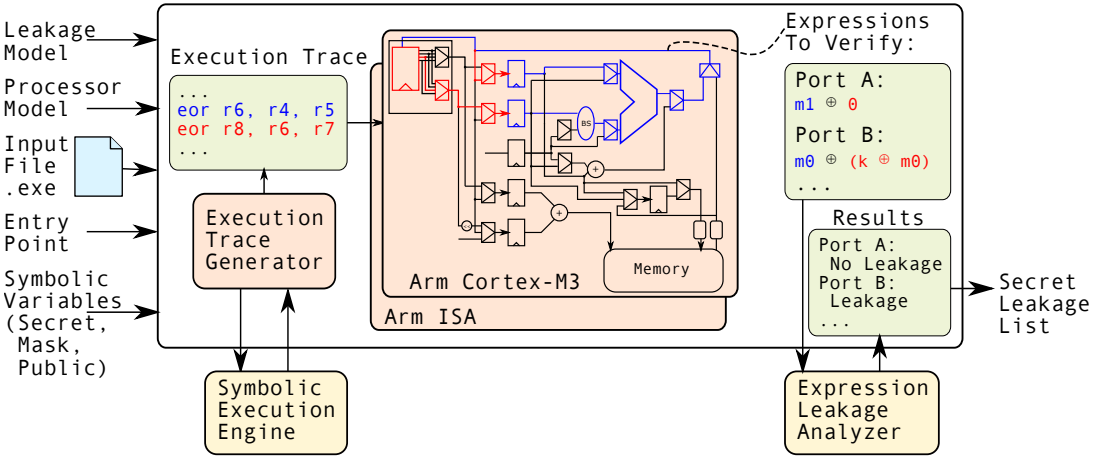
PortA:	m1	0
PortB	m0	k ⊕ m0
MuxRegA / RegA:	m1	m1 ⊕ m0
MuxRegB / RegB:	m0	k ⊕ m0
ALUOut	m1 ⊕ m0	k ⊕ m1

Arm Cortex-M3: Exemple



PortA:	m1	0	m1
PortB	m0	k ⊕ m0	k
MuxRegA / RegA:	m1	m1 ⊕ m0	m0
MuxRegB / RegB:	m0	k ⊕ m0	k
ALUOut	m1 ⊕ m0	k ⊕ m1	k ⊕ m0

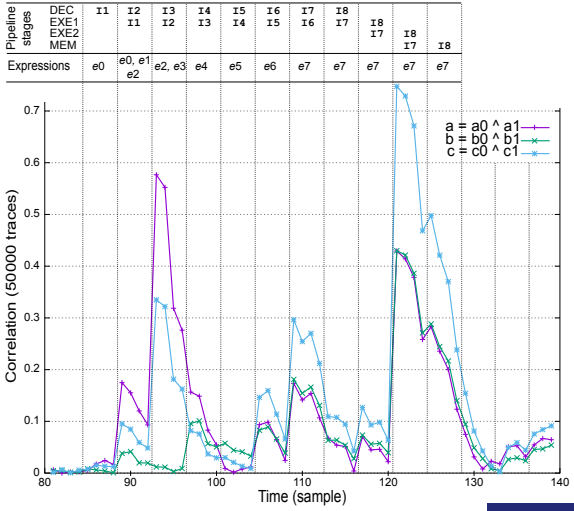
ARMISTICE Framework



Back on the Leaking Masked And

	Instructions	Leaks: *expr. name*
I1	`and.wr5, r2, r1`	MuxRegA, RegA: $e0$ RegB: $e1$
I2	`ands r0, r1`	PortA, RegA: $e2$ AluOut: $e3$
I3	`ands r3, r2`	AluOut: $e4$
I4	`eors r4, r5`	RegB: $e5$
I5	`eors r0, r7`	AluOut: $e6$
I6	`eors r4, r3`	AluOut: $e7$
I7	`str r0, [r6, #0]`	-
I8	`str r4, [r6, #4]`	PortB, RegB, DataReg, DataOut, BufferMem: $e7$

Name	Expression	Leaks
$e0$	$a0 \cdot b1 \oplus a1$	*a, c*
$e1$	$a0 \cdot b1 \oplus b0$	*b, c*
$e2$	$a0 \oplus a1$	*a, c*
$e3$	$a0 \cdot b0 \oplus a1 \cdot b0$	*a, c*
$e4$	$a0 \cdot b0 \oplus a1 \cdot b1$	*a, b, c*
$e5$	$a1 \cdot b0 \oplus b1$	*b, c*
$e6$	$a0 \cdot b0 \oplus a0 \cdot b1 \oplus a1 \cdot b0$	*a, b, c*
$e7$	$a0 \cdot b0 \oplus a0 \cdot b1 \oplus a1 \cdot b0 \oplus a1 \cdot b1$	*a, b, c*



ARMISTICE Results

8 correctly masked applications from the litterature

- Application proven 1-probing secure (value-based leakage model)
- At least one secret leakage due to micro-architecture in all programs

ARMISTICE Results

8 correctly masked applications from the litterature

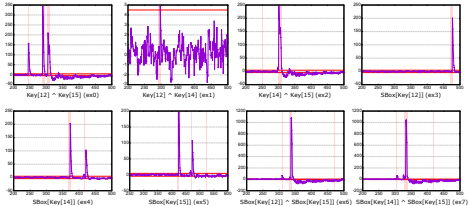
- Application proven 1-probing secure (value-based leakage model)
- At least one secret leakage due to micro-architecture in all programs
- Manual investigation of 8 reported leakages in a masked AES

ARMISTICE Results

8 correctly masked applications from the litterature

- Application proven 1-probing secure (value-based leakage model)
- At least one secret leakage due to micro-architecture in all programs
- Manual investigation of 8 reported leakages in a masked AES

Experimental leakage assessment
(specific t-test with the leaking expression)

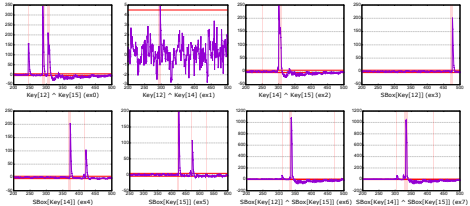


ARMISTICE Results

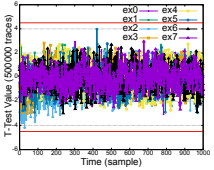
8 correctly masked applications from the litterature

- Application proven 1-probing secure (value-based leakage model)
- At least one secret leakage due to micro-architecture in all programs
- Manual investigation of 8 reported leakages in a masked AES

Experimental leakage assessment
(specific t-test with the leaking expression)



Leakage assessment after manual patching using `ARMISTICE`'s output



Conclusion on ARMISTICE

ARMISTICE

- A framework for formally proving the absence of secret leakage in a masked code
- Based on the micro-architectural details of a Arm Cortex-M3 core and a memory model
- Good match between found leakages and experimentally observed leakages
- Locates secret leakages in time and space along with the corresponding expressions, which in turn can help remove them

Take away

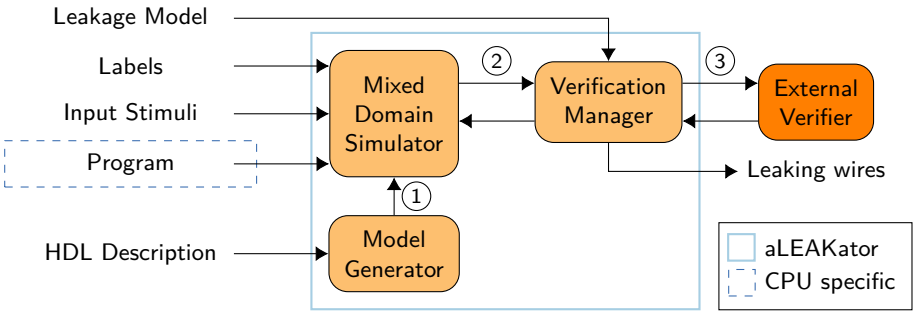
- Avoid the manual generation of the micro-architecture model
- Also consider glitches

$\Rightarrow$ aLEAKator, an approach proposed by Noé Amiot (PhD student at LIP6/Sorbonne University)

- Automate code patching

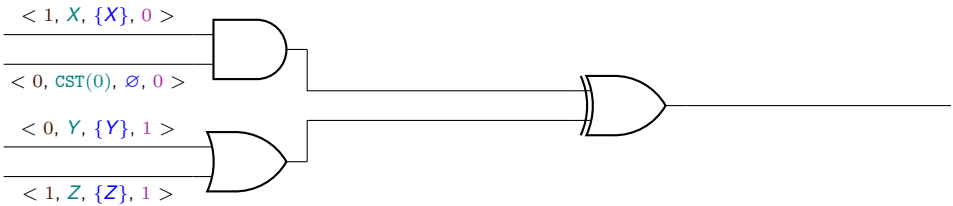
aLEAKator Overview

- Automatically exploit the HDL description of both cryptographic hardware accelerators and general-purpose processors
- Performs mixed-domain simulation to extract information to be verified
- Formally prove the absence of leakages in different leakage models



Mixed-Domain Simulation: Extension of Concrete Simulation

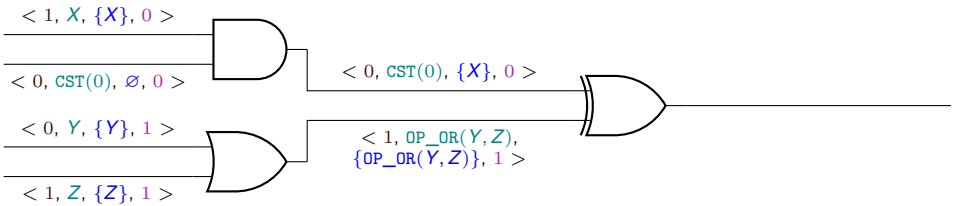
- Instead of only representing a concrete value, a wire w at cycle t is the tuple composed of:
  1. A concrete value
  2. A symbolic expression
  3. A glitchy expression set, representing all possibly observable glitchy values
  4. A stability attribute
- Each of these elements is defined in a domain for which we define a set of rules to compute the evolution for all supported hardware gates



Stateless circuit example propagation with X, Y and Z three symbolic expressions

Mixed-Domain Simulation: Extension of Concrete Simulation

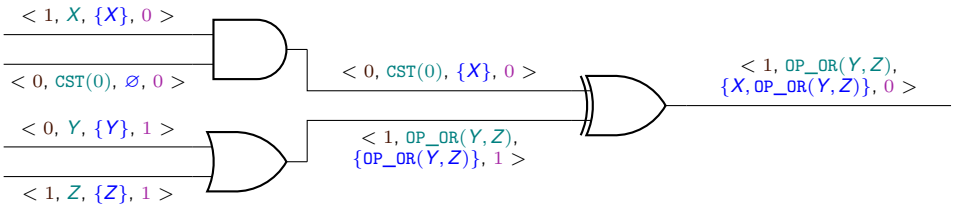
- Instead of only representing a concrete value, a wire w at cycle t is the tuple composed of:
  1. A concrete value
  2. A symbolic expression
  3. A glitchy expression set, representing all possibly observable glitchy values
  4. A stability attribute
- Each of these elements is defined in a domain for which we define a set of rules to compute the evolution for all supported hardware gates



Stateless circuit example propagation with X, Y and Z three symbolic expressions

Mixed-Domain Simulation: Extension of Concrete Simulation

- Instead of only representing a concrete value, a wire w at cycle t is the tuple composed of:
  1. A concrete value
  2. A symbolic expression
  3. A glitchy expression set, representing all possibly observable glitchy values
  4. A stability attribute
- Each of these elements is defined in a domain for which we define a set of rules to compute the evolution for all supported hardware gates



Stateless circuit example propagation with X, Y and Z three symbolic expressions

aLEAKator Implementation

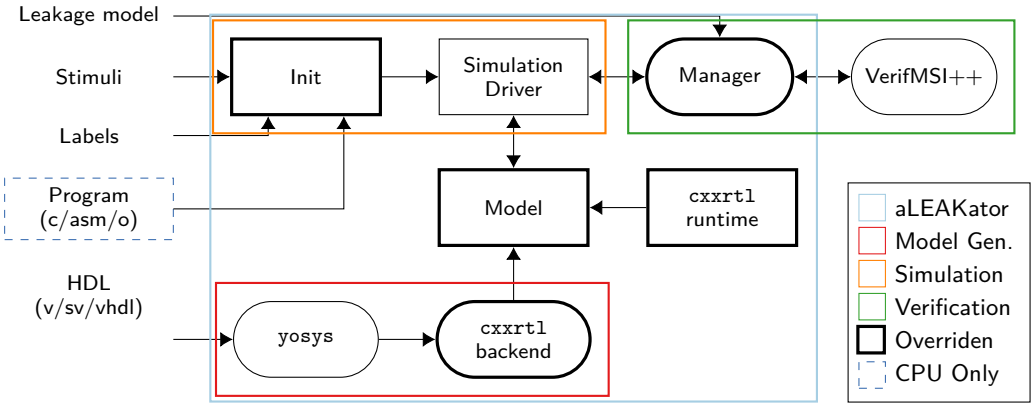


Figure: Implementation of aLEAKator.

Results

Verification of cryptographic hardware

- Validation of the approach on (5) already studied HW cryptographic blocks with SoA tools
- aLeakator finds known vulnerabilities, and faster

Results

Verification of cryptographic hardware

- Validation of the approach on (5) already studied HW cryptographic blocks with SoA tools
- aLeakator finds known vulnerabilities, and faster

Verification of masked software

- Validation of the approach with several CPUs and different security properties
- Different masked programs among which a full tabulated masked AES

Results

Verification of cryptographic hardware

- Validation of the approach on (5) already studied HW cryptographic blocks with SoA tools
- aLeakator finds known vulnerabilities, and faster

Verification of masked software

- Validation of the approach with several CPUs and different security properties
- Different masked programs among which a full tabulated masked AES

Experimental validation of found leakage

- Proven not leaking software does not leak in practice
- Experimentally each visible leakage matches with a secret leakage found by aLeakator

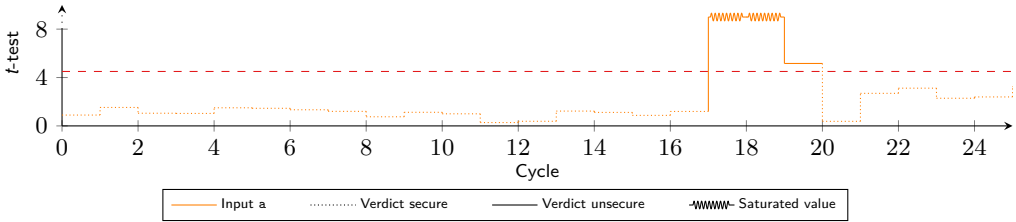
Observable Leakage versus aLeakator Verdict

`Refresh`: A simple program refreshing both shares of a secret a.

```
1  ldr r0, =a0
2  ldr r1, =a1
3  ldr r2, =m
4  ldr r3, =blk
5  ldr r4, [r0] // a0 in r4
6  ldr r5, [r2] // m in r5
```

```
7  eor r4, r4, r5 // Refresh a0
8  str r4, [r0]
9
10 // Clear write LSU path
11 str r3, [r3]
12 eor r4, r4, r4 // Clear r4
13 mov r4, 0x0 // Clear alu path?
```

```
14 // LSU Read path is cleared
15 ldr r6, [r1] // a1 in r12
16
17 eor r6, r6, r5 // Refresh a1
18 str r6, [r2]
19 mov r6, 0x0
```



Cycle's max t-test value and corresponding aLEAKator `RR 1-probing` verdict, Cortex-M4.

- Each leakage observable with the t-test is associated with a leakage reported by aLEAKator

Masked AES Implementation on Various CPUs

- C implementation of AES masked with HERBST scheme (6 masks), iteratively verified and patched at the source level to prevent leaks in the `1-probing model`
- Patches include memory barriers, data reorganization and pipeline flushes

Masked AES Implementation on Various CPUs

- C implementation of AES masked with HERBST scheme (6 masks), iteratively verified and patched at the source level to prevent leaks in the `1-probing model`
- Patches include memory barriers, data reorganization and pipeline flushes

Processor	#KGates	Cycles	Leaking cycles	Time	Ram (Go)	Unique verified expressions
Arm Cortex-M3	10.4	10 113	0	5m04	43.7	90 053
Arm Cortex-M4	11.4	10 113	0	5m48	45.3	93 279
RISCV IBEX	2.4	8 868	0	21m41	226.8	185 508
RISCV CV32E40P	3.0	8 868	0	43m44	423.5	199 405

- Different optimisations to verify the minimal set of wires

Masked AES Implementation on Various CPUs

- C implementation of AES masked with HERBST scheme (6 masks), iteratively verified and patched at the source level to prevent leaks in the `1-probing model`
- Patches include memory barriers, data reorganization and pipeline flushes

Processor	#KGates	Cycles	Leaking cycles	Time	Ram (Go)	Unique verified expressions
Arm Cortex-M3	10.4	10 113	0	5m04	43.7	90 053
Arm Cortex-M4	11.4	10 113	0	5m48	45.3	93 279
RISCV IBEX	2.4	8 868	0	21m41	226.8	185 508
RISCV CV32E40P	3.0	8 868	0	43m44	423.5	199 405

- Different optimisations to verify the minimal set of wires
- First formal verification of a full masked AES-128 tabulated implementation, on four different CPUs

Masked AES Implementation on Various CPUs

- aLEAKator is able to verify the same AES program, in the `RR 1-probing model`.

Processor	#KGates	Cycles	Leaking cycles	Time	Ram (Go)	Unique verified expressions
Arm Cortex-M3	10.4	10 113	8 329	18m52	208.0	148 409
Arm Cortex-M4	11.4	10 113	8 329	20m14	211.5	147 766
RISCV IBEX	2.4	8 868	6 651	22m08	301.0	90 050
RISCV CV32E40P	3.0	8 868	6 705	39m01	333.7	86 259

- As expected, lot of leaks

Masked AES Implementation on Various CPUs

- aLEAKator is able to verify the same AES program, in the `RR 1-probing model`.

Processor	#KGates	Cycles	Leaking cycles	Time	Ram (Go)	Unique verified expressions
Arm Cortex-M3	10.4	10 113	8 329	18m52	208.0	148 409
Arm Cortex-M4	11.4	10 113	8 329	20m14	211.5	147 766
RISCV IBEX	2.4	8 868	6 651	22m08	301.0	90 050
RISCV CV32E40P	3.0	8 868	6 705	39m01	333.7	86 259

- As expected, lot of leaks
- Lastest result: a masked AES automatically patched using aLEAKator has been proven secure !

Conclusion

- Need for micro-architectural leakage source modeling for masked software verification
- aLEAKator, a framework for formally proving the absence of secret leakages
  1. With a configurable and state-of-the-art leakage model
  2. Based on mixed-domain simulation
  3. Usable on cryptographic hardware accelerators as well as masked programs executed on CPUs
  4. Able to handle various CPUs such as the Cortex-M3, Cortex-M4, IBEX, CV32E40P and the COCO-IBEX
- Accepted as a long paper in TCHES 2026 volume 2
- Available at `https://github.com/noeamiot/aLEAKator`
- Next step: automated patching using aLeakator's output

Thank you

and many thanks to Quentin Meunier[4], Noé Amiot[3] and Simon Tollec[4] for their slides !

[3] LIP6/Sorbonne University
[4] Thales

References I

Barthe, G., Belaïd, S., Cassiers, G., Fouque, P.-A., Grégoire, B., and Standaert, F.-X. (2019).
maskverif: Automated verification of higher-order masking in presence of physical defaults.
In Computer Security – ESORICS 2019: 24th European Symposium on Research in Computer Security, Luxembourg, September 23–27, 2019, Proceedings, Part I, page 300–318, Berlin, Heidelberg. Springer-Verlag.

Ben El Ouahma, I., Meunier, Q. L., Heydemann, K., and Encrenaz, E. (2019).
Side-channel robustness analysis of masked assembly codes using a symbolic approach.
Journal of Cryptographic Engineering, 9:231–242.

Corre, Y. L., Großschädl, J., and Dinu, D. (2018).
Micro-architectural power simulator for leakage assessment of cryptographic software on ARM Cortex-M3 processors.
In Fan, J. and Gierlichs, B., editors, COSADE 2018, volume 10815 of LNCS, pages 82–98. Springer, Cham.

De Grandmaison, A., Heydemann, K., and Meunier, Q. L. (2022).
Armistice: Microarchitectural leakage modeling for masked software formal verification.
IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 41(11):3733–3744.

References II


Faust, S., Grosso, V., Merino Del Pozo, S., Paglialonga, C., and Standaert, F.-X. (2018).
Composable masking schemes in the presence of physical defaults & the robust probing model.
IACR TCHES, 2018(3):89–120.


Gross, H., Mangard, S., and Korak, T. (2016).
Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order.
In Proceedings of the 2016 ACM Workshop on Theory of Implementation Security, TIS '16, page 3, New York, NY, USA. Association for Computing Machinery.


Ishai, Y., Sahai, A., and Wagner, D. (2003).
Private circuits: Securing hardware against probing attacks.
In Annual International Cryptology Conference, pages 463–481. Springer.


McCann, D., Oswald, E., and Whitnall, C. (2017).
Towards practical tools for side channel aware software engineering: 'grey box' modelling for instruction leakages.
In Kirda, E. and Ristenpart, T., editors, USENIX Security 2017, pages 199–216. USENIX Association.

References III


Meunier, Q. and Taleb, A. (2023).
Verifmsi: Practical verification of hardware and software masking schemes implementations.
In 20th International Conference on Security and Cryptography, volume 1, pages 520–527. SciTePress.


Meunier, Q. L., Pons, E., and Heydemann, K. (2023).
Leakageverif: Efficient and scalable formal verification of leakage in symbolic expressions.
volume 49, page 3359,Äì3375. IEEE Press.


Müller, N. and Moradi, A. (2022).
PROLEAD A probing-based hardware leakage detection tool.
IACR TCHES, 2022(4):311–348.


Nikova, S., Rechberger, C., and Rijmen, V. (2006).
Threshold implementations against side-channel attacks and glitches.
In Ning, P., Qing, S., and Li, N., editors, Information and Communications Security, pages 529–545, Berlin, Heidelberg. Springer Berlin Heidelberg.

References IV

Shelton, M. A., Samwel, N., Batina, L., Regazzoni, F., Wagner, M., and Yarom, Y. (2021).
Rosita: Towards automatic elimination of power-analysis leakage in ciphers.
In NDSS 2021. The Internet Society.