THALES
Building a future we can all trust

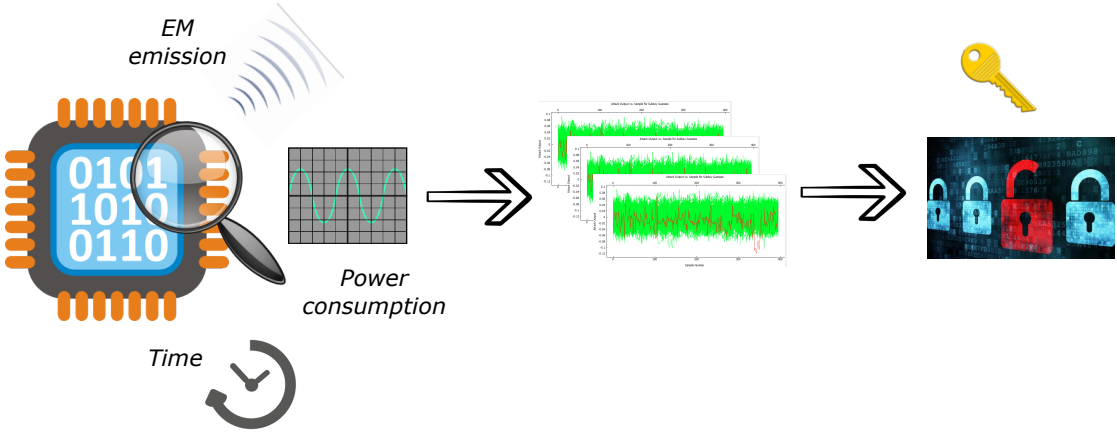
On the modelling of the processor microarchitecture for masked software verification

Karine HEYDEMANN,
Thales/Sorbonne University
Journées du GDR Sécurité Informatique, 24th June 2025

Plan

1. Background on Side-Channel Attacks
2. Masked Implementation Verification
3. Microarchitecture modelling for masked software verification

Introduction: Side-Channel Attacks

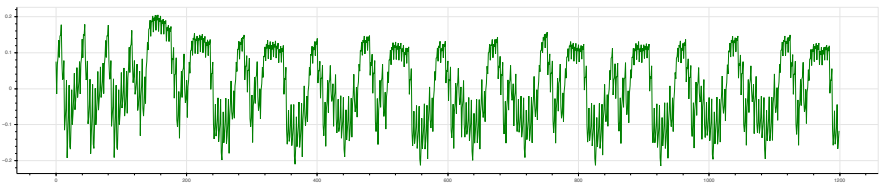


Power Consumption of Instructions

The power consumption of a CPU-based device differs with the executed instructions

- Example of a naive fast exponentiation

```
SquareMult(x, e, N):
  let e_n, ..., e_1 be the bits of e
  y ← 1
  for i = n down to 1 {
    y ← Square(y)                (S)
    y ← ModReduce(y, N)          (R)
    if e_i = 1 then {
      y ← Mult(y, x)             (M)
      y ← ModReduce(y, N)        (R)
    }
  }
  return y
```



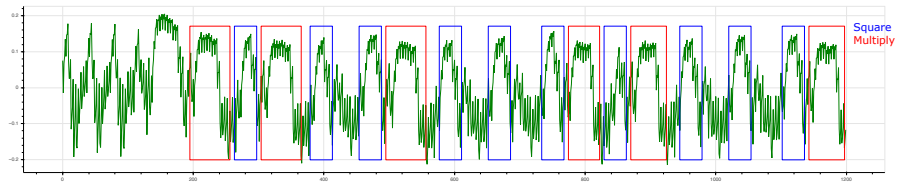
1. also used for synchronizing a fault injection

Power Consumption of Instructions

The power consumption of a CPU-based device differs with the executed instructions

- Example of a naive fast exponentiation

```
SquareMult(x, e, N):
  let e_n, ..., e_1 be the bits of e
  y ← 1
  for i = n down to 1 {
    y ← Square(y)                (S)
    y ← ModReduce(y, N)          (R)
    if e_i = 1 then {
      y ← Mult(y, x)             (M)
      y ← ModReduce(y, N)        (R)
    }
  }
  return y
```



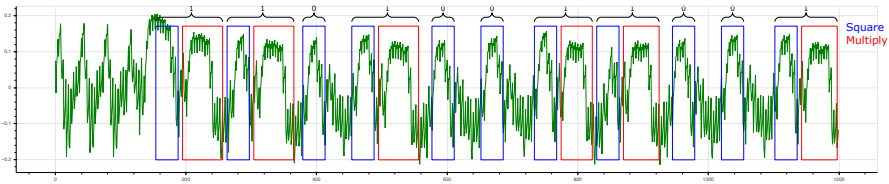
1. also used for synchronizing a fault injection

Power Consumption of Instructions

The power consumption of a CPU-based device differs with the executed instructions

- Example of a naive fast exponentiation

```
SquareMult(x, e, N):
  let e_n, ..., e_1 be the bits of e
  y ← 1
  for i = n down to 1 {
    y ← Square(y)                (S)
    y ← ModReduce(y, N)          (R)
    if e_i = 1 then {
      y ← Mult(y, x)             (M)
      y ← ModReduce(y, N)        (R)
    }
  }
  return y
```



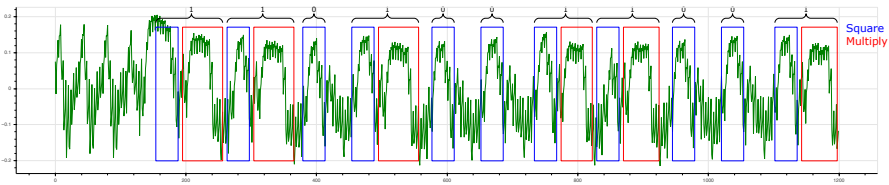
1. also used for synchronizing a fault injection

Power Consumption of Instructions

The power consumption of a CPU-based device differs with the executed instructions

- Example of a naive fast exponentiation

```
SquareMult(x, e, N):
  let e_n, ..., e_1 be the bits of e
  y ← 1
  for i = n down to 1 {
    y ← Square(y)                (S)
    y ← ModReduce(y, N)          (R)
    if e_i = 1 then {
      y ← Mult(y, x)             (M)
      y ← ModReduce(y, N)        (R)
    }
  }
  return y
```



- Typical exploitation example: Simple Power Analysis (SPA) [Mangard et al., 2010] [1]

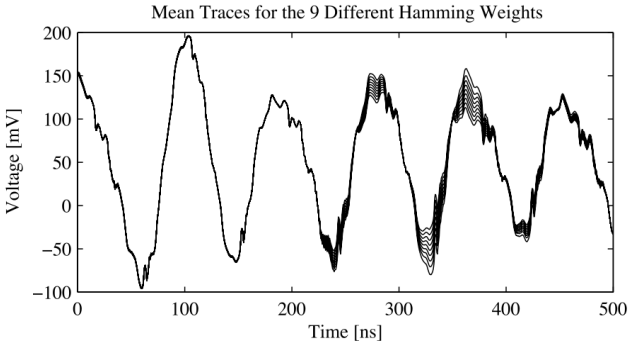
1. also used for synchronizing a fault injection

Power Consumption of Data

The power comsumption of an instruction depends on its data

- **Simple leakage model**: Hamming Weight of data

DPA Book [Mangard et al., 2010] extract

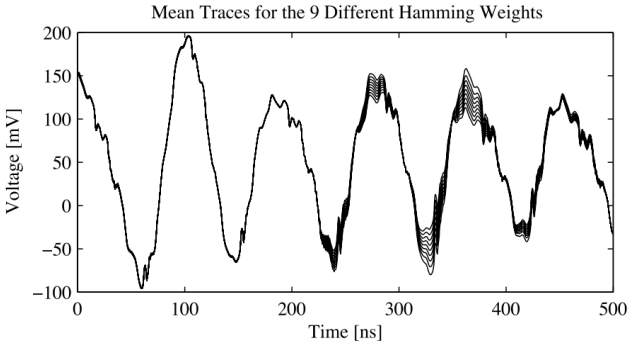


Power Consumption of Data

The power comsumption of an instruction depends on its data

- **Simple leakage model**: Hamming Weight of data

DPA Book [Mangard et al., 2010] extract



- Simple but works in practice !
- Typical exploitation examples: Differential Power Analysis [Kocher et al., 1999] and Correlation Power Analysis [Brier et al., 2004]

Counter-measure 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
- Power measurements are theoretically independent of the secret

Masking

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
  - $s_1 = s_0 \oplus s$

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

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

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

    $\Rightarrow$ 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
  - NB: c0 $\oplus$ c1 = (a0 $\oplus$ a1).(b0 $\oplus$ b1)
    = (a0.b0 $\oplus$ a0.b1 $\oplus$ a1.b0 $\oplus$ a1.b1)

    $\Rightarrow$ need to compute all the products (.) and reduce the computation (+)

    $\Rightarrow$ any reduction of two terms leads to a leakage of a or 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
  - NB: c0 $\oplus$ c1 = (a0 $\oplus$ a1).(b0 $\oplus$ b1)
    = (a0.b0 $\oplus$ a0.b1 $\oplus$ a1.b0 $\oplus$ a1.b1)

    $\Rightarrow$ need to compute all the products (.) and reduce the computation (+)

    $\Rightarrow$ 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

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

    $\Rightarrow$ need to compute all the products (.) and reduce the computation (+)

    $\Rightarrow$ 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
  - NB: c0 $\oplus$ c1 = (a0 $\oplus$ a1).(b0 $\oplus$ b1)
    = (a0.b0 $\oplus$ a0.b1 $\oplus$ a1.b0 $\oplus$ a1.b1)

    $\Rightarrow$ need to compute all the products (.) and reduce the computation (+)

    $\Rightarrow$ 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

    $\Rightarrow$ additionnal 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
  - NB: c0 $\oplus$ c1 = (a0 $\oplus$ a1).(b0 $\oplus$ b1)
    = (a0.b0 $\oplus$ a0.b1 $\oplus$ a1.b0 $\oplus$ a1.b1)

    $\Rightarrow$ need to compute all the products (.) and reduce the computation (+)

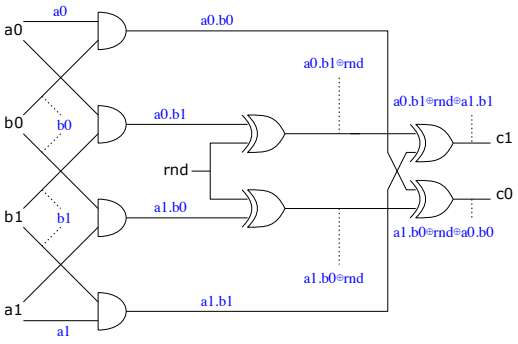
    $\Rightarrow$ any reduction of two terms leads to a leakage of a or b

    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

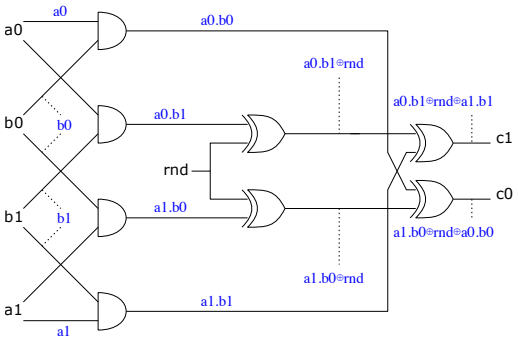
    $\Rightarrow$ additionnal randoms are necessary to make the computation secure
- Different masking schemes have been proposed ISW-AND [Ishai et al., 2003], DOM-AND [Gross et al., 2016], TI-AND [Nikova et al., 2006]

Example: Masked AND at Order 1



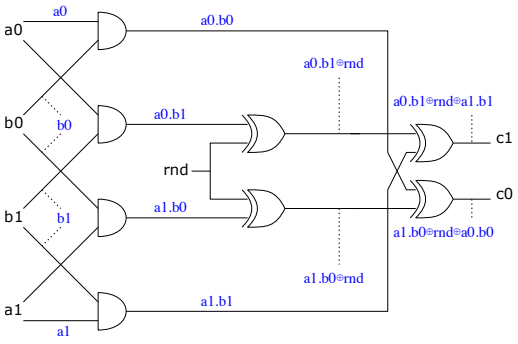

```
1  void masked_and(uint8_t a0, uint8_t a1,
2                  uint8_t b0, uint8_t b1,
3                  uint8_t rnd,
4                  uint8_t *c0, uint8_t *c1)
5
6  *c0 = ((a0 & b0) ^ rnd) ^ (a1 & b1);
7  *c1 = ((a0 & b1) ^ rnd) ^ (a1 & b0);
8  return;
```

Example: Masked AND at Order 1




```
1  void masked_and(uint8_t a0, uint8_t a1,
2                  uint8_t b0, uint8_t b1,
3                  uint8_t r,
4                  uint8_t *c0, uint8_t *c1)
5  {
6    uint8_t tmp = (a0 & b1) ^ r;
7    __asm__ __volatile__ ("" ::: "memory");
8   *c0 = tmp ^ (a1 & b0);
9   tmp = (a0 & b0) ^ r;
10   __asm__ __volatile__ ("" ::: "memory");
11  *c1 = tmp ^ (a1 & b1);
12  return;
13 }
```

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]
```

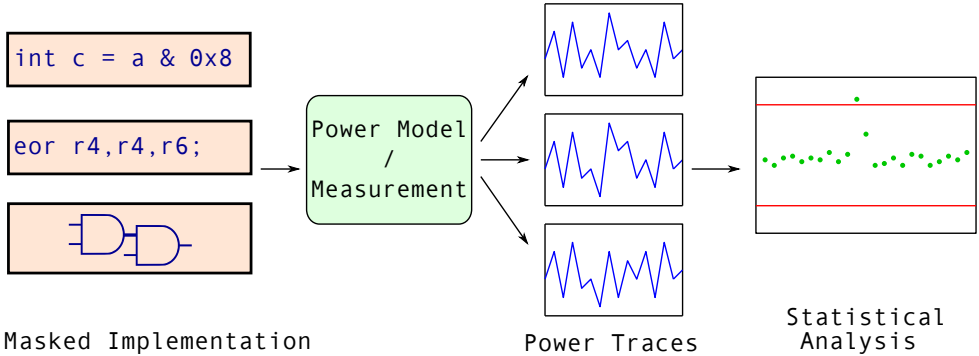
Plan

1. Background on Side-Channel Attacks
2. Masked Implementation Verification
3. Microarchitecture modelling for masked software verification

How To Verify a Masked Implementation?

Empirically

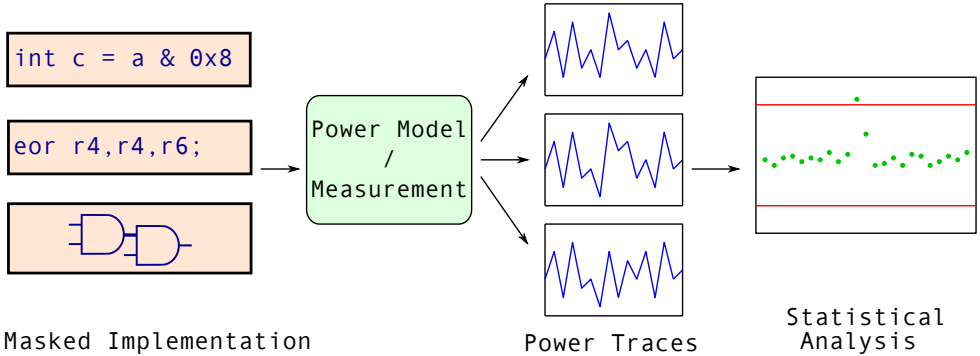
- Perform power simulations or acquisitions then use statistical metrics, such as the t-test



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, leakages are difficult to locate

- 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, and prove the absence of leakage for a chosen attacker model and a leakage model

d-probing model [Ishai et al., 2003]

- The attacker has *d* probes that can capture *d* intermediate values during the execution
- Assume a value leakage model

How To Verify a Masked Implementation ?

Formally

- Label input values as secret, mask or public, and prove the absence of leakage for a chosen attacker model and a leakage model

d-probing model [Ishai et al., 2003]

- The attacker has *d* probes that can capture *d* intermediate values during the execution
- Assume a value leakage model

Example with a method based on symbolic expression

Masked Implementation

```
tmp = (a0 & b1) ^ rnd;

*c0 = tmp ^ (a1 & b0);

tmp = (a0 & b0) ^ rnd;

*c1 = tmp ^ (a1 & b1);
```

How To Verify a Masked Implementation ?

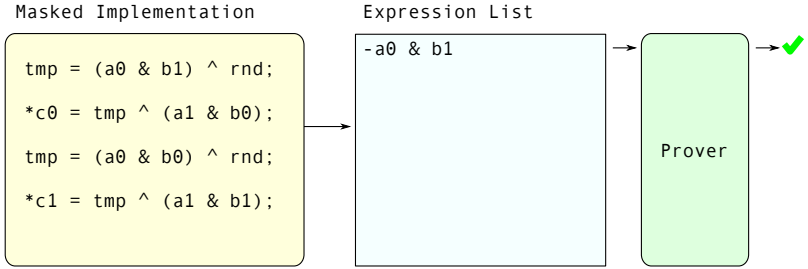
Formally

- Label input values as secret, mask or public, and prove the absence of leakage for a chosen attacker model and a leakage model

d-probing model [Ishai et al., 2003]

- The attacker has *d* probes that can capture *d* intermediate values during the execution
- Assume a value leakage model

Example with a method based on symbolic expression



How To Verify a Masked Implementation ?

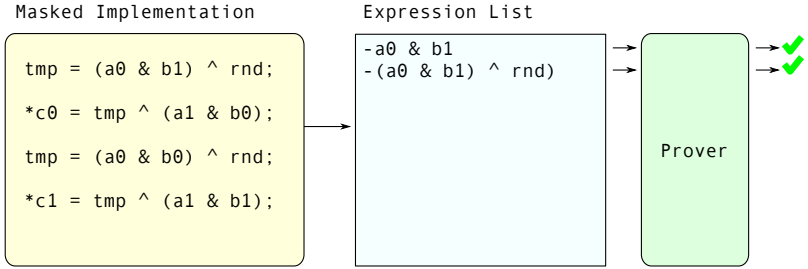
Formally

- Label input values as secret, mask or public, and prove the absence of leakage for a chosen attacker model and a leakage model

d-probing model [Ishai et al., 2003]

- The attacker has *d* probes that can capture *d* intermediate values during the execution
- Assume a value leakage model

Example with a method based on symbolic expression



How To Verify a Masked Implementation ?

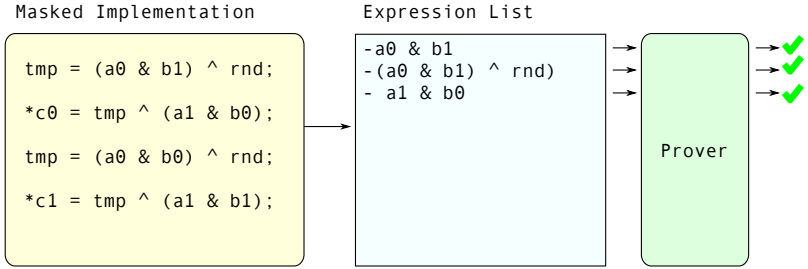
Formally

- Label input values as secret, mask or public, and prove the absence of leakage for a chosen attacker model and a leakage model

d-probing model [Ishai et al., 2003]

- The attacker has *d* probes that can capture *d* intermediate values during the execution
- Assume a value leakage model

Example with a method based on symbolic expression



How To Verify a Masked Implementation ?

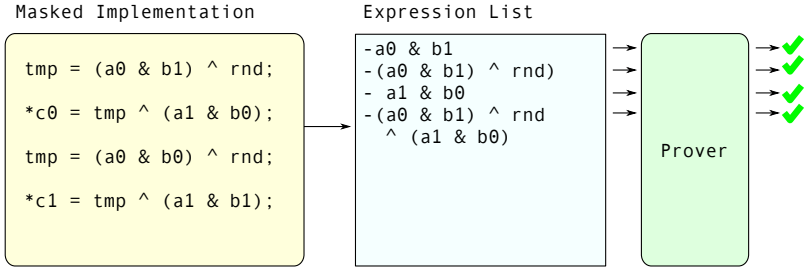
Formally

- Label input values as secret, mask or public, and prove the absence of leakage for a chosen attacker model and a leakage model

d-probing model [Ishai et al., 2003]

- The attacker has *d* probes that can capture *d* intermediate values during the execution
- Assume a value leakage model

Example with a method based on symbolic expression



How To Verify a Masked Implementation ?

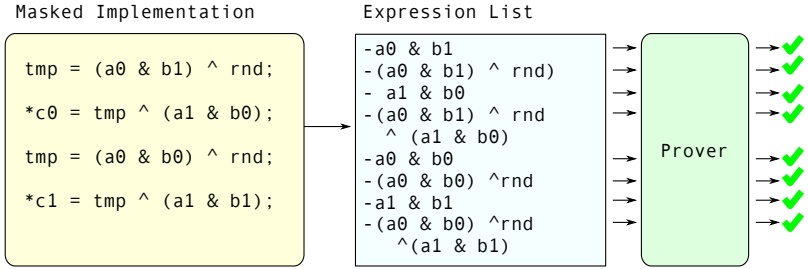
Formally

- Label input values as secret, mask or public, and prove the absence of leakage for a chosen attacker model and a leakage model

d-probing model [Ishai et al., 2003]

- The attacker has *d* probes that can capture *d* intermediate values during the execution
- Assume a value leakage model

Example with a method based on symbolic expression



How To Verify a Masked Implementation?

Formally

- Label input values as secret, mask or public, and prove the absence of leakage for a chosen attacker model and a leakage model

d-probing model [Ishai et al., 2003]

- The attacker has *d* probes that can capture *d* intermediate values during the execution
- Assume a value leakage model

Pros: Absence of secret leakage is guaranteed for the chosen model, easier to locate and undestand leakages

Cons: Scalability issues, potential false positive,
proven-secure implementations in the *d*-probing model can leak

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

How To Verify a Masked Implementation?

Formally

- Label input values as secret, mask or public, and prove the absence of leakage for a chosen attacker model and a leakage model

d-probing model with transition

- The attacker has *d* probes that can capture **transitions** during the execution
- Assume a transition leakage model

How To Verify a Masked Implementation ?

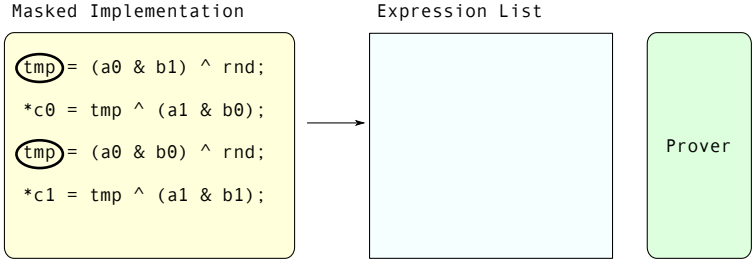
Formally

- Label input values as secret, mask or public, and prove the absence of leakage for a chosen attacker model and a leakage model

d-probing model with transition

- The attacker has *d* probes that can capture **transitions** during the execution
- Assume a transition leakage model

Example with a method based on symbolic expression



How To Verify a Masked Implementation ?

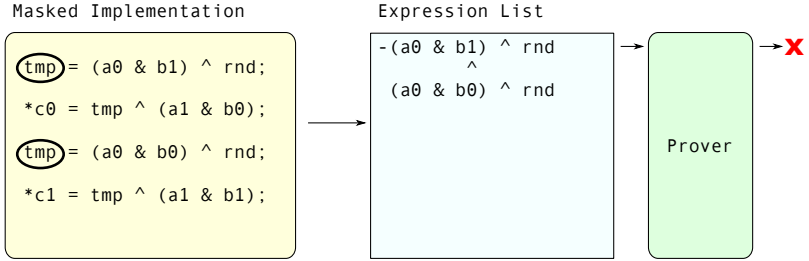
Formally

- Label input values as secret, mask or public, and prove the absence of leakage for a chosen attacker model and a leakage model

d-probing model with transition

- The attacker has *d* probes that can capture **transitions** during the execution
- Assume a transition leakage model

Example with a method based on symbolic expression



How To Verify a Masked Implementation ?

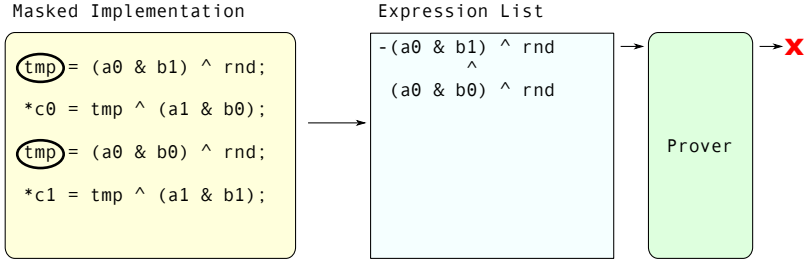
Formally

- Label input values as secret, mask or public, and prove the absence of leakage for a chosen attacker model and a leakage model

d-probing model with transition

- The attacker has *d* probes that can capture **transitions** during the execution
- Assume a transition leakage model

Example with a method based on symbolic expression



- Perform verification at the assembly level to detect vulnerabilities post-compilation

Proven Leakage-Free Implementation in Practice

- Software “ISW And” proven leakage-free at the ISA level in the value leakage model and transition leakage model (GPRs).

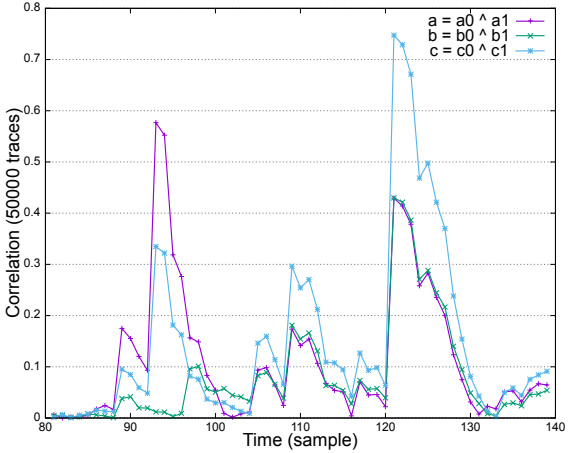
```
;r0:a0, r1:b0, r2:a1, r3:b1, r6:c[] r7:m
and.w r4, r0, r3 ; a0 & b1
eors  r4, r7     ; t0 = (a0 & b1) ^ m
and.w r5, r2, r1 ; a1 & b0
ands  r0, r1     ; a0 & b0
ands  r3, r2     ; b1 & a1
eors  r4, r5     ; t1 = t0 ^ (a1 & b0)
eors  r0, r7     ; c0 = (a0 & b0) ^ m
eors  r4, r3     ; c1 = t1 ^ (a1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

```

Proven Leakage-Free Implementation in Practice

- Software “ISW And” proven leakage-free at the ISA level in the value leakage model and transition leakage model (GPRs).

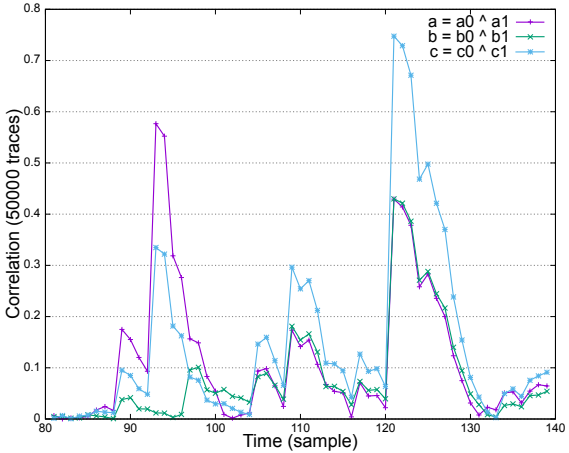
```
1  ;r0:a0, r1:b0, r2:a1, r3:b1, r6:c[] r7:m
2  and.w r4, r0, r3 ; a0 & b1
3  eors  r4, r7     ; t0 = (a0 & b1) ^ m
4  and.w r5, r2, r1 ; a1 & b0
5  ands  r0, r1     ; a0 & b0
6  ands  r3, r2     ; b1 & a1
7  eors  r4, r5     ; t1 = t0 ^ (a1 & b0)
8  eors  r0, r7     ; c0 = (a0 & b0) ^ m
9  eors 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



Proven Leakage-Free Implementation in Practice

- Software “ISW 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

Plan

1. Background on Side-Channel Attacks
2. Masked Implementation Verification
3. Microarchitecture modelling for masked software verification

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

Arnaud de Grandmaison [2], Karine Heydemann, Quentin L. Meunier [3]

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

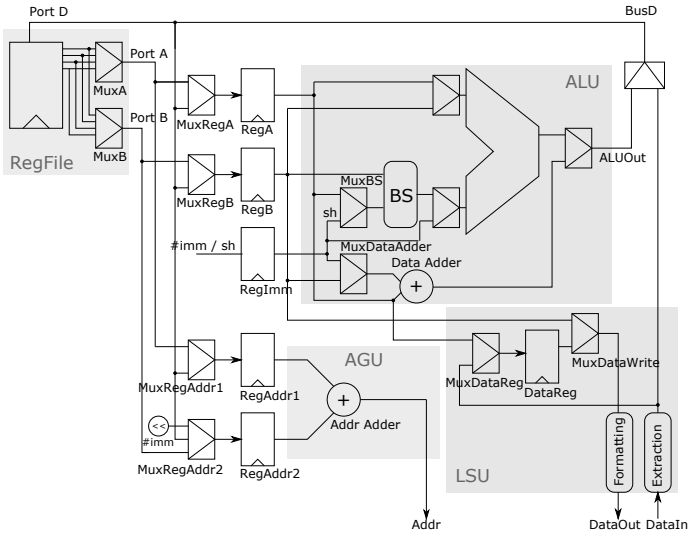
2. Arm
3. Sorbonne Université/LIP6

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

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

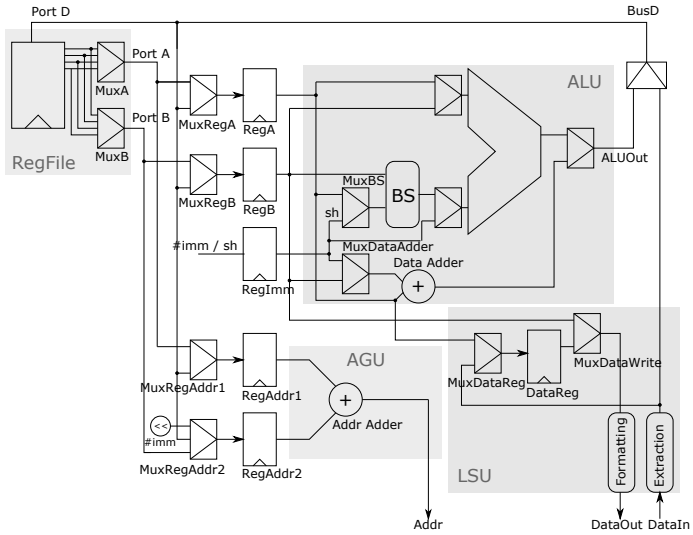
Case Study: Board STM32F1 [De Grandmaison et al., 2022]

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



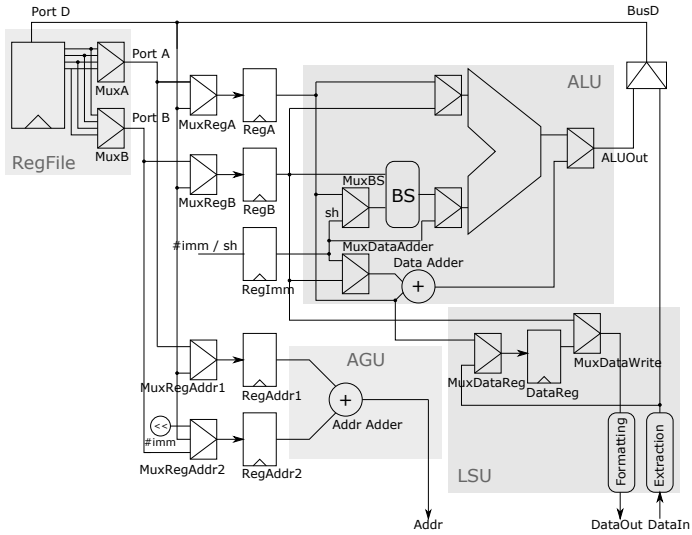
Case Study: Board STM32F1 [De Grandmaison et al., 2022]

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



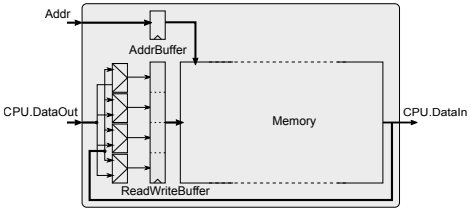
Case Study: Board STM32F1 [De Grandmaison et al., 2022]

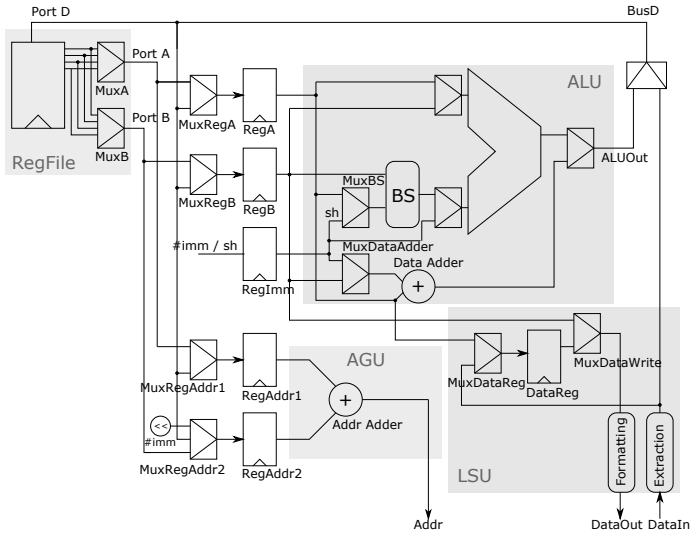
- Arm Cortex-M3: modeled from the Verilog source code
- Memoire: black-box approach (no HDL description available)
- Design of several micro-benchmarks a.k.a. “leakage test vectors”:
  - Detection of leakage sources (black-box)



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

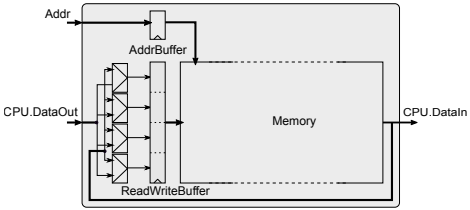
- Arm Cortex-M3: modeled from the Verilog source code
- Memoire: black-box approach (no HDL description available)
- Design of several micro-benchmarks a.k.a. “leakage test vectors”:
  - Detection of leakage sources (black-box)

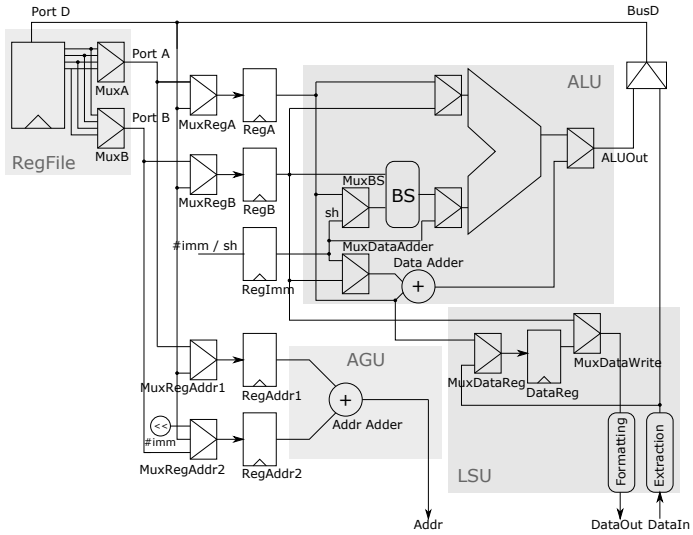




Case Study: Board STM32F1 [De Grandmaison et al., 2022]

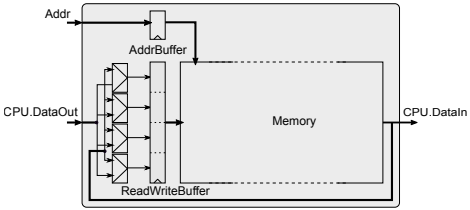
- Arm Cortex-M3: modeled from the Verilog source code
- Memoire: black-box approach (no HDL description available)
- Design of several micro-benchmarks a.k.a. “leakage test vectors”:
  - Detection of leakage sources (black-box)
  - Validation (white-box)

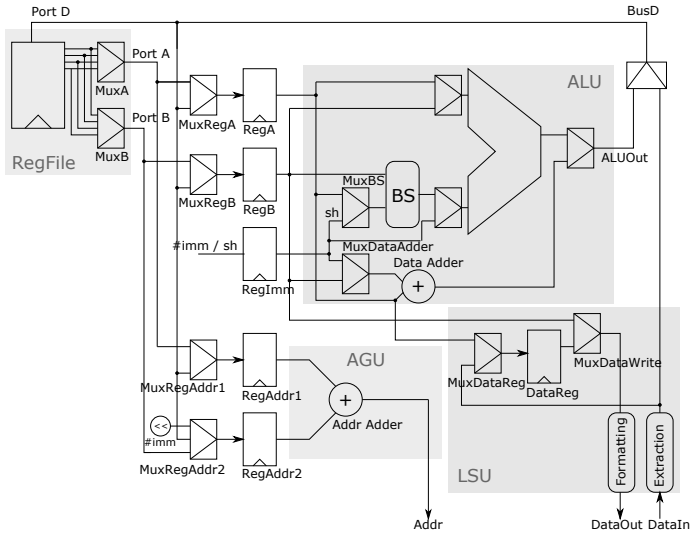




Case Study: Board STM32F1 [De Grandmaison et al., 2022]

- 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





Leakage Test Vectors

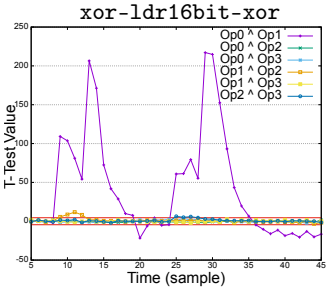
- 77 carefully designed test vectors
  - 31 for the data path components involved in each instruction
  - 5 for forwarding mechanisms
  - 7 for the writing back into the register file
  - 34 for analysis of the LSU and the modelling of the memory.
- Description and results online: `https://www-soc.lip6.fr/armistice`

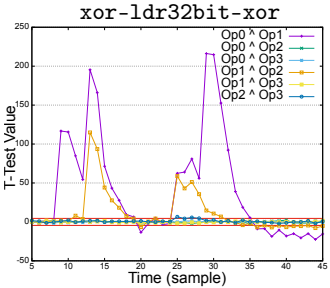
Leakage test vectors overview

Vector	Archs	Targets	Tags	Description
SeqAND.EOR.1	v6m, v7m	stm32f1	ALU, AND, DEC, EOR	Designed for testing leakage between non identical ALU instructions
SeqAND.EOR.2	v6m, v7m	stm32f1	ALU, AND, DEC, EOR	Designed for testing leakage between non identical ALU instructions
SeqEOR.ADD.1	v7m	stm32f1	ADD, ALU, DEC, ENC, IMM	Designed for testing leakage when executing a 32bit ADD with immediate instruction
SeqEOR.ADD.2	v7m	stm32f1	ADD, ALU, DEC, ENC, IMM	Designed for testing leakage when executing a 16bit ADD with immediate instruction
SeqEOR.ADD.3	v7m	stm32f1	ADD, ALU, DEC, ENC, IMM	Designed for testing leakage when execution a 16bit ADD with immediate instruction
SeqEOR.ADD.4	v7m	stm32f1	ADD, ALU	Designed for testing leakage when executing an ADD instruction
SeqEOR.AND.1	v6m, v7m	stm32f1	ALU, AND, DEC, EOR	Designed for testing leakage between non identical ALU instructions
SeqEOR.AND.2	v6m, v7m	stm32f1	ALU, AND, DEC, EOR	Designed for testing leakage between non identical ALU instructions
SeqEOR.EOR.1	v6m, v7m	stm32f1	ALU, DEC, EOR, PRE/POSTAMBLE	Designed for testing leakage when executing an EOR instruction
SeqEOR.EOR.2	v6m, v7m	stm32f1	ALU, DEC, EOR, PRE/POSTAMBLE	Designed for testing leakage when executing an EOR instruction
SeqEOR.EOR.3	v6m, v7m	stm32f1	ALU, DEC, EOR, PRE/POSTAMBLE	Designed for testing leakage when execution an EOR instruction
SeqEOR.EOR.4	v6m, v7m	stm32f1	ALU, DEC, EOR	Designed for testing leakage between non consecutive EOR instructions
SeqEOR.EOR.5	v7m	stm32f1	ALU, DEC, EOR, IMM	Designed for testing leakage when executing a XOR instruction with an immediate operand
SeqEOR.Imm.Imm	v6m, v7m	stm32f1	ADD, ALU, CMP, DEC, EOR, IMM	Designed for testing leakage between consecutive data read in RegFile in case of immediate in the decoded 32 bits instruction
SeqEOR.Imm.Imm.1	v6m, v7m	stm32f1	ADD, ALU, CMP, DEC, EOR, IMM	Designed for testing leakage between consecutive data read in RegFile in case of immediate in the decoded 32 bits instruction
SeqEOR.Imm.Imm.2	v6m, v7m	stm32f1	ADD, ALU, CMP, DEC, EOR, IMM	Designed for testing leakage between consecutive data read in RegFile in case of immediate in the decoded 32 bits instruction
SeqEOR.Imm.Imm.3	v6m, v7m	stm32f1	ADD, ALU, CMP, DEC, EOR, IMM	Designed for testing leakage between consecutive data read in RegFile in case of immediate in the decoded 32 bits instruction
SeqEOR.LD.1	v6m, v7m	stm32f1	DEC, ENC, LD	Designed for testing leakage related to 16-bit encoding of ldr
SeqEOR.LD.2	v6m, v7m	stm32f1	DEC, ENC, LD	Designed for testing leakage related to 32-bit encoding of ldr
SeqEOR.LSL.1	v6m, v7m	stm32f1	ALU, DEC, ENC, LSL	Designed for testing leakage when executing a shift by an immediate instruction (LSL)
SeqEOR.LSL.2	v6m, v7m	stm32f1	ALU, DEC, ENC, LSL	Designed for testing leakage when executing a shift by an immediate instruction (LSL)
SeqEOR.LSL.3	v6m, v7m	stm32f1	ALU, DEC, ENC, LSL	Designed for testing leakage when executing a shift by an immediate instruction (LSL)
SeqEOR.LSL.4	v6m, v7m	stm32f1	ALU, DEC, ENC, LSL	Designed for testing leakage when executing a shift by an immediate instruction (LSL)
SeqEOR.MOV.1	v7m	stm32f1	ALU, DEC, ENC, IMM, MOV	Designed for testing leakage when executing a 32bit MOV instruction with an immediate operand
SeqEOR.MOV.2	v7m	stm32f1	ALU, DEC, ENC, IMM, MOV	Designed for testing leakage when when executing a 16bit MOVS instruction with an immediate operand
SeqEOR.MOV.4	v7m	stm32f1	ALU, DEC, ENC, MOV	Designed for testing leakage when executing a 32bit MOV register instruction
SeqEOR.MOV.5	v7m	stm32f1	ALU, DEC, ENC, MOV	Designed for testing leakage when executing a 32bit MOVS register instruction
SeqEOR.MOV.6	v6m, v7m	stm32f1	ALU, DEC, ENC, MOV	Designed for testing leakage when execution a 16bit MOVS register instruction
SeqEOR.MOV.7	v6m, v7m	stm32f1	ALU, DEC, ENC, MOV	Designed for testing leakage when execution a 16bit MOV register instruction
SeqEOR.ST.1	v6m, v7m	stm32f1	DEC, ST	Designed for testing leakage related to a store instruction with an addressing mode using an immediate offset
SeqEOR.ST.2	v6m, v7m	stm32f1	DEC, ST	Designed for testing leakage related to a store instruction with an addressing mode using a register offset
SeqEOR.movw	v6m, v7m	stm32f1	ALU, DEC, MOV	Designed for testing leakage related to a MOVW instruction
SeqEOR.nop	v6m, v7m	stm32f1	ALU, DEC, NOP	Designed for testing leakage related to a nop instruction
SeqFWD.ALU.1	v7m	stm32f1	ALU, EOR, FWD	Designed for testing leakage related to the forwarding to an ALU instruction for its first operand a result from an ALU instruction
SeqFWD.ALU.2	v7m	stm32f1	ALU, EOR, FWD	Designed for testing leakage related to the forwarding to an ALU instruction for its second operand a result from an ALU instruction
SeqFWD.LSU.1	v7m	stm32f1	DATA, FWD, LD-ST, MEM	Designed for testing leakage related to the forwarding of loaded value to a store instruction with an immediate offset
SeqFWD.LSU.2	v7m	stm32f1	DATA, FWD, LD-ST, MEM	Designed for testing leakage related to forward of loaded value to a store instruction with a register offset
SeqFWD.LSU.ALU.1	v7m	stm32f1	ALU, FWD, MEM	Designed for testing leakage related to the forwarding of loaded value to a store instruction with an immediate offset
SeqMEM.ADDR.1	v7m	stm32f1	ADDR, MEM	Designed for testing leakage in the LSU in case of LDR instruction with a register offset, and related to the address bus
SeqMEM.ADDR.2	v7m	stm32f1	ADDR, MEM	Designed for testing leakage in the LSU in case of LDR instruction with a register offset, and related to the address bus
SeqMEM.ADDR.3	v7m	stm32f1	ADDR, MEM	Designed for testing leakage in the LSU in case of LDR instruction with a register offset, and related to the address bus
SeqMEM.LD.1	v7m	stm32f1	DATA, LD-LD, MEM	Designed for testing leakage related to consecutive and non consecutive loads at fixed word-aligned addresses in RAM
SeqMEM.LD.2	v7m	stm32f1	DATA, LD-LD, MEM	Designed for testing leakage related to consecutive loads at fixed word-aligned addresses in RAM
SeqMEM.LD.3	v7m	stm32f1	DATA, LD-LD, MEM	Designed for testing leakage related to consecutive and non consecutive loads (of bytes) at fixed word-aligned addresses in RAM
SeqMEM.LD.4	v7m	stm32f1	DATA, LD-LD, MEM	Designed for testing leakage related to consecutive loads (of a byte) at fixed word-aligned addresses in RAM
SeqMEM.LD.5	v7m	stm32f1	DATA, EXTRACT, LD-LD, MEM	Designed for testing leakage related to consecutive loads (of a byte) at fixed (not word-aligned) addresses in RAM
SeqMEM.LD.6	v7m	stm32f1	DATA, EXTRACT, LD-LD, MEM	Designed for testing leakage related to consecutive loads (of a byte) at fixed addresses in RAM
SeqMEM.LD.7	v7m	stm32f1	DATA, EXTRACT, LD-LD, MEM	Designed for testing leakage related to consecutive loads (of a half-word) at fixed addresses in RAM
SeqMEM.LD.8	v7m	stm32f1	DATA, EXTRACT, LD-LD, MEM	Designed for testing leakage related to consecutive loads (of a half-word) at fixed addresses in RAM
SeqMEM.LD.ST.1	v7m	stm32f1	DATA, LD-ST, MEM	Designed for testing leakage when alternating ldr and str
SeqMEM.LD.ST.2	v7m	stm32f1	DATA, LD-ST, MEM	Designed for testing leakage when alternating ldr and str
SeqMEM.LD.ST.3	v7m	stm32f1	DATA, LD-ST, MEM	Designed for testing leakage when alternating ldrb and strb
SeqMEM.LD.ST.4	v7m	stm32f1	DATA, LD-ST, MEM	Designed for testing leakage when alternating ldrb and strb
SeqMEM.LD.ST.5	v7m	stm32f1	DATA, LD-ST, MEM	Designed for testing leakage when alternating ldrb and strb
SeqMEM.LD.ST.6	v7m	stm32f1	DATA, LD-ST, MEM	Designed for testing leakage when alternating ldrb and strb
SeqMEM.LD.ST.7	v7m	stm32f1	DATA, LD-ST, MEM	Designed for testing leakage when alternating ldrb and strb
SeqMEM.LD.ST.8	v7m	stm32f1	DATA, LD-ST, MEM	Designed for testing leakage when alternating ldrb and strb
SeqMEM.LD.UA.1	v7m	stm32f1	DATA, LD-LD, MEM, UNALIGNED	Designed for testing leakage related to consecutive unaligned loads at fixed addresses in RAM
SeqMEM.LD.UA.2	v7m	stm32f1	DATA, LD-LD, MEM, UNALIGNED	Designed for testing leakage related to consecutive unaligned loads at fixed addresses in RAM separated by a XOR
SeqMEM.LDD.1	v7m	stm32f1	DATA, LDD, MEM	Designed for testing leakage related to load double instruction and to consecutive load-double instruction at fixed addresses in RAM
SeqMEM.LDD.2	v7m	stm32f1	DATA, LDD, MEM	Designed for testing leakage related to load double instruction and to consecutive load-double instruction at fixed addresses in RAM
SeqMEM.ST.1	v7m	stm32f1	DATA, MEM, ST-ST	Designed for testing leakage related to successive store instructions with different addressing mode. Aligned accesses
SeqMEM.ST.2	v7m	stm32f1	DATA, MEM, ST-ST	Designed for testing leakage related to successive store instructions with different addressing mode. Aligned accesses
SeqMEM.ST.3	v7m	stm32f1	DATA, MEM, ST-ST	Designed for testing leakage related to successive store instructions with different addressing mode. Aligned accesses
SeqMEM.ST.4	v7m	stm32f1	DATA, MEM, ST-ST	Designed for testing leakage related to successive store instructions with different addressing mode. Aligned accesses
SeqMEM.ST.5	v7m	stm32f1	DATA, MEM, ST-ST	Designed for testing leakage related to successive store instructions with different size. Aligned accesses
SeqMEM.ST.6	v7m	stm32f1	DATA, MEM, ST-ST	Designed for testing leakage related to successive store instructions with different size. Aligned accesses
SeqMEM.ST.WB.1	v7m	stm32f1	DATA, MEM, ST-ST, WB	Designed for testing leakage related to consecutive stores (of words) at the same fixed addresse Str A + 0, Str A + 64, Str A + 0, Str A + 64
SeqMEM.ST.WB.2	v7m	stm32f1	DATA, MEM, ST-ST, WB	Designed for testing leakage related to consecutive stores (of words) at the same fixed addresse Str A + 0, Str A + 64, Str A + 0, Str A + 64
SeqMEM.ST.WB.3	v7m	stm32f1	DATA, MEM, ST-ST, WB	Designed for testing leakage related to consecutive stores (of a byte) at the same fixed addresse Str A + 0, Str A + 64, Str A + 0, Str A + 64
SeqWB.r0	v7m	stm32f1	ALU, EOR, REG-WB	Designed for testing leakage related to consecutive writes in a register (R0) of the register file
SeqWB.r0.byte.1	v7m	stm32f1	ALU, EOR, REG-WB	Designed for testing leakage related to consecutive writes of one byte in registers (R0 and R6) of the register file
SeqWB.r0.byte.2	v7m	stm32f1	ALU, EOR, REG-WB	Designed for testing leakage related to consecutive writes of one byte in registers (R0 and R6) of the register file
SeqWB.r0.byte.3	v7m	stm32f1	ALU, EOR, REG-WB	Designed for testing leakage related to consecutive writes of one byte in registers (R0 and R6) of the register file
SeqWB.r4	v6m, v7m	stm32f1	ALU, EOR, REG-WB	Designed for testing leakage related to consecutive writes in a register (R4) of the register file
SeqWB.r5	v7m	stm32f1	ALU, EOR, REG-WB	Designed for testing leakage related to consecutive writes in a register (R5) of the register file
SeqWB.r7	v7m	stm32f1	ALU, EOR, REG-WB	Designed for testing leakage related to consecutive writes in a register (R7) of the register file

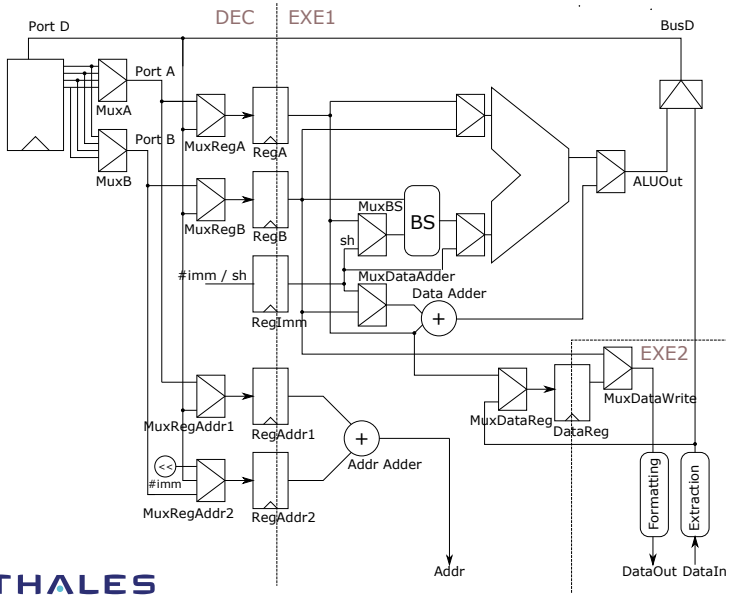
Findings Using Leakage Vectors

- 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
- Intra-word leakage in the LSU
- 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 !

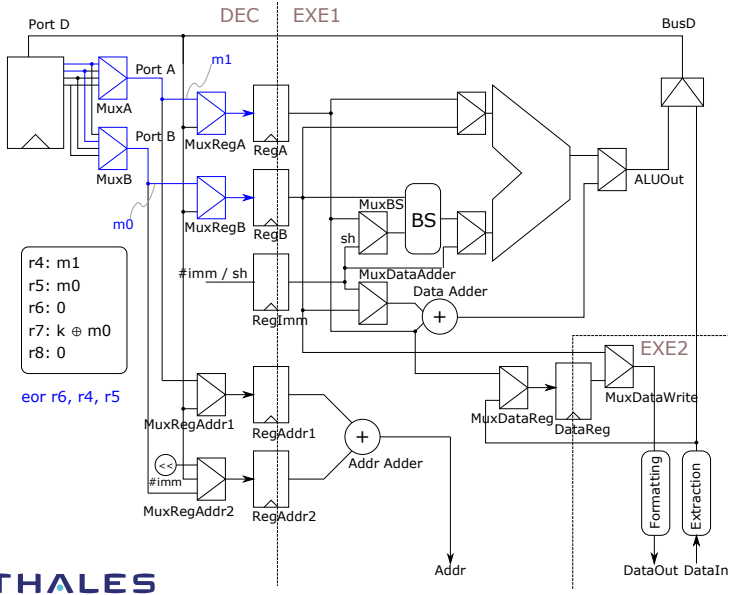




Arm Cortex-M3: Exemple

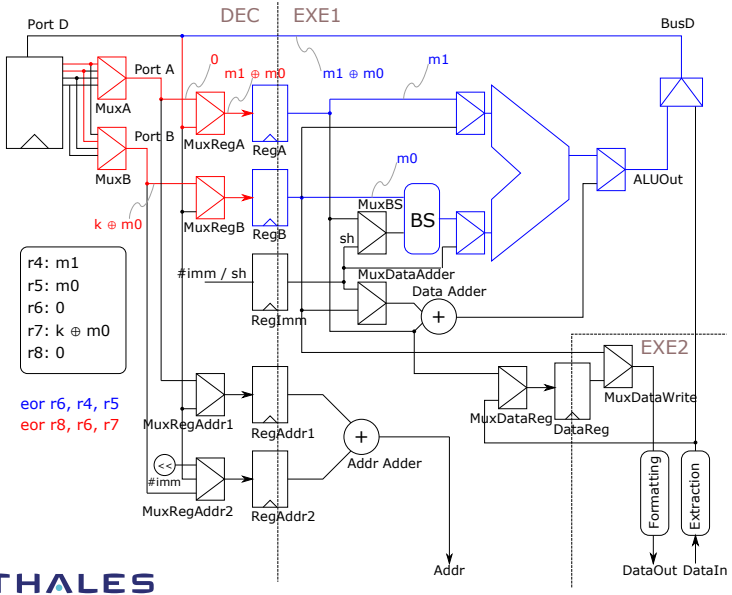


Arm Cortex-M3: Exemple



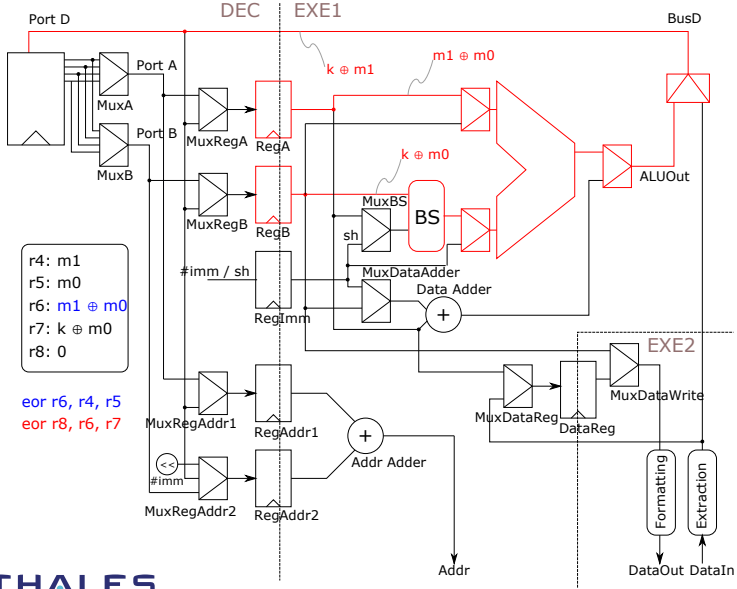
PortA: m1
PortB m0
MuxRegA / RegA: m1
MuxRegB / RegB: m0
ALUOut

Arm Cortex-M3: Exemple



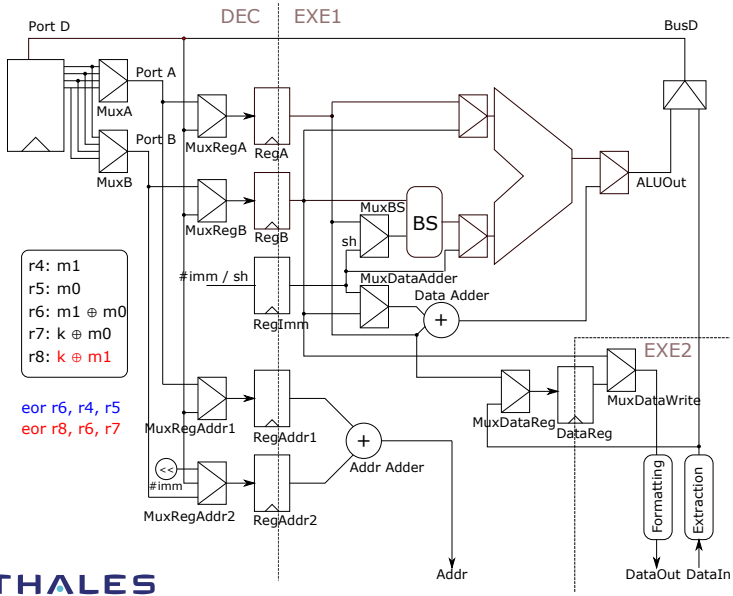
PortA:	m1	0
PortB	m0	k ⊕ m0
MuxRegA / RegA:	m1	m1 ⊕ m0
MuxRegB / RegB:	m0	k ⊕ m0
ALUOut	m1 ⊕ m0	

Arm Cortex-M3: Exemple



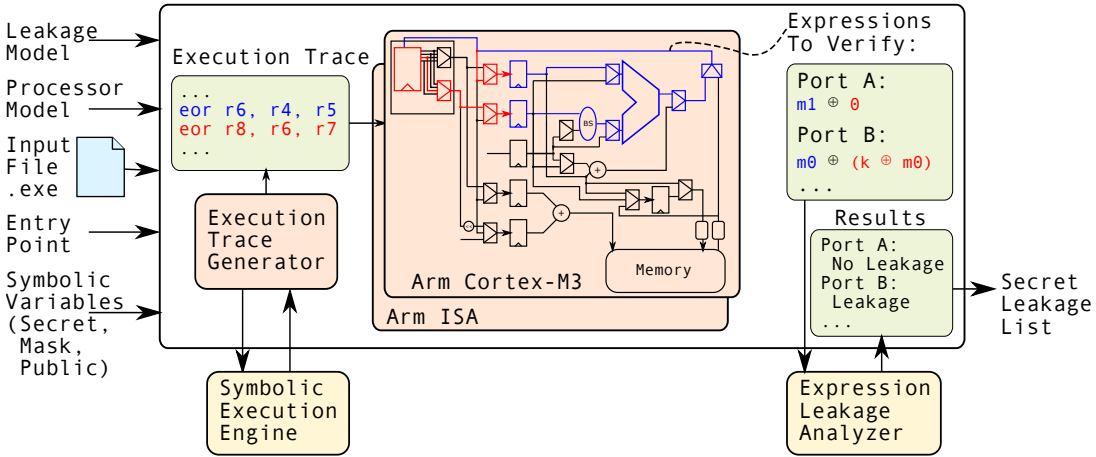
PortA:	m1	0
PortB	m0	k ⊕ m0
MuxRegA / RegA:	m1	m1 ⊕ m0
MuxRegB / RegB:	m0	k ⊕ m0
ALUOut	m1 ⊕ m0	k ⊕ m1

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

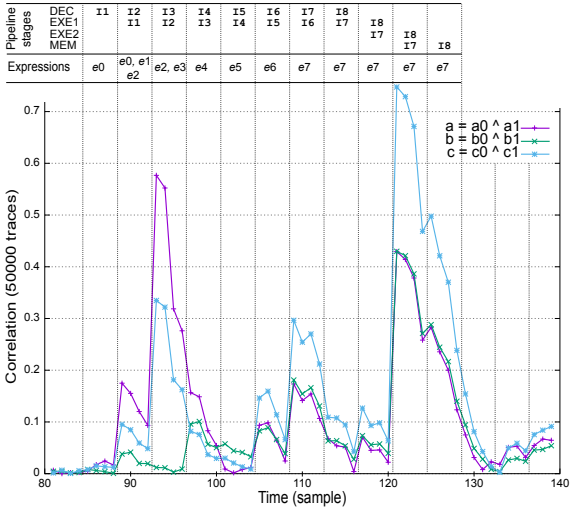
ARMISTICE Framework



Back on the Leaking “ISW And”

	Instructions	Leaks: *expr. name*
I1	`and.w r5, 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*

Nom	Expression	Fuites
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 Validation

8 masked applications from the litterature

Results summary

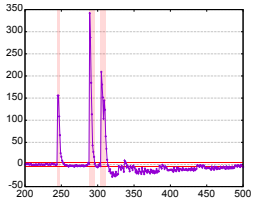
- Absence of leakage in the value based model for correct masking schemes
- At least one secret leakage due to micro-architecture in all programs, even those designed to be secured w.r.t. the Arm Cortex-M3 micro-architecture (Dilithium AND and A2B)
- At least one leakage for 22 out of 27 modeled components

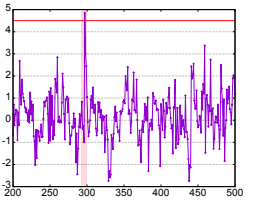
Accuracy and Exploitability (1/2)

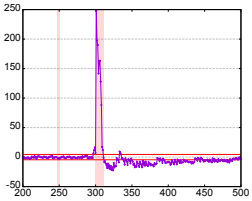
- Manual analysis of the leakage resulting from the first round of the Key Schedule
- 8 considered expressions (simplest ones)
- Experimental leakage assessment using specific t-test with 500,000 traces

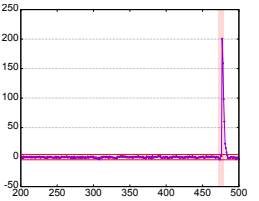
Accuracy and Exploitability (1/2)

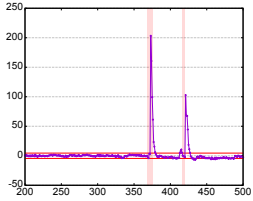
- Manual analysis of the leakage resulting from the first round of the Key Schedule
- 8 considered expressions (simplest ones)
- Experimental leakage assessment using specific t-test with 500,000 traces

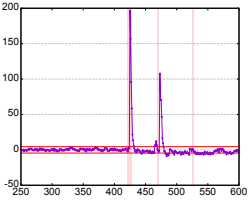


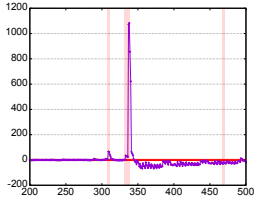


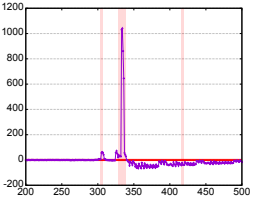












Accuracy and Exploitability (2/2)

Leakages found but not observed

- 8-bit transition in a GPR, not observable
- 8-bit transition on Bus B, not observable
- Stall cycle from the memory, could be removed with a better memory model

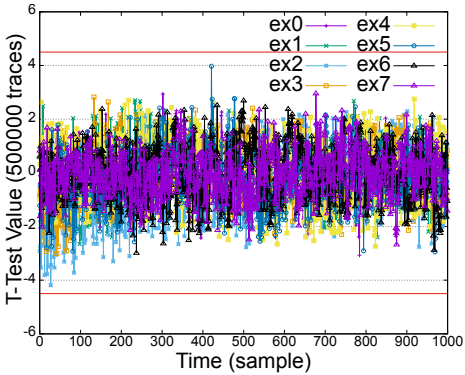
Accuracy and Exploitability (2/2)

Leakages found but not observed

- 8-bit transition in a GPR, not observable
- 8-bit transition on Bus B, not observable
- Stall cycle from the memory, could be removed with a better memory model

Leveraging `ARMISTICE` output

- Addition of carefully designed instructions to clean the part of the data path involved in the leaking transition



Conclusion and Future Work

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
- Model close to reality, good match between found leakages and observed leakages
- Locates secret leakages in time and space along with the corresponding expressions, which in turn can help remove them

Future work

- Avoid the manual generation of the micro-architecture model
  $\Rightarrow$ Automate the verification from a RTL description, a binary code and information on shares (secrets and masks) and sources of randoms
- Consider glitches

$\Rightarrow$ Noé Amiot, current PhD on this topic at LIP6, stay tuned !

Thank you

and many thanks to Quentin Meunier[4], Noé Amiot[4] and Simon Tollec[5] for their slides !

4. LIP6/Sorbonne University
5. 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.

Brier, E., Clavier, C., and Olivier, F. (2004).
Correlation power analysis with a leakage model.
In Joye, M. and Quisquater, J.-J., editors, CHES 2004, volume 3156 of LNCS, pages 16–29. Springer, Berlin, Heidelberg.

References II


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.


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.

References III


Kocher, P. C., Jaffe, J., and Jun, B. (1999).
Differential power analysis.
In Wiener, M. J., editor, CRYPTO'99, volume 1666 of LNCS, pages 388–397. Springer, Berlin, Heidelberg.


Mangard, S., Oswald, E., and Popp, T. (2010).
Power Analysis Attacks : Revealing the Secrets of Smart Cards.
Springer Publishing Company, Incorporated, 1st edition.


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.


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.

References IV

- 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.
- 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.