HDL Simulation for Masked Software Verification

Quentin Meunier

SemSecuElec Rennes – 29 Mai 2026

Sorbonne University
Laboratoire d'Informatique de Paris 6
4 Place Jussieu, 75252 Paris, France

Outline

Side-Channel Attacks

Masking and Verification

Armistice: Masking Verification at the Micro-Architectural Level

aLEAKator: Taking HDL into account

Conclusion

Outline

Cryptographic protocols are based on mathematical algorithms supposed to be safe

- Difficult/Impossible to find secret information in a reasonable time

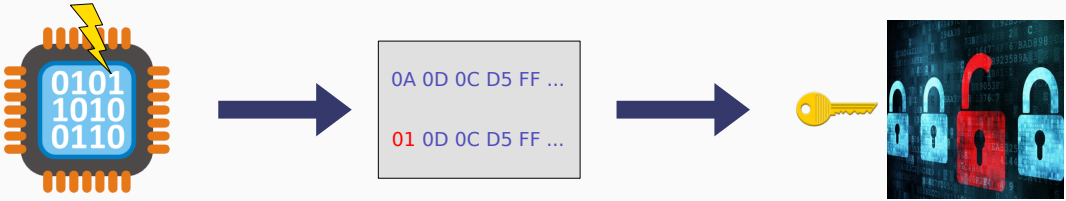
Implementations of safe protocols can be vulnerable

- Attacks linked with the software and hardware on which the algorithms are deployed

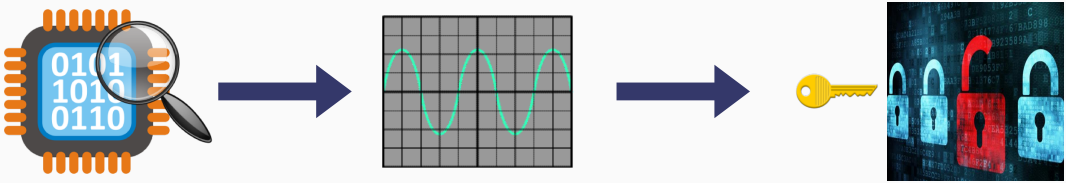
$\Rightarrow$ Information obtained during the execution of an implementation can help find secret data and break security

Non-Invasive Attacks

- Active / Fault attacks: clock glitch, voltage glitch, EM impulsion



- Passive / Observation attacks: time, dissipated power, electromagnetic emission (EM)



Simple Power Attacks (SPA): Example of the Modular Exponentiation

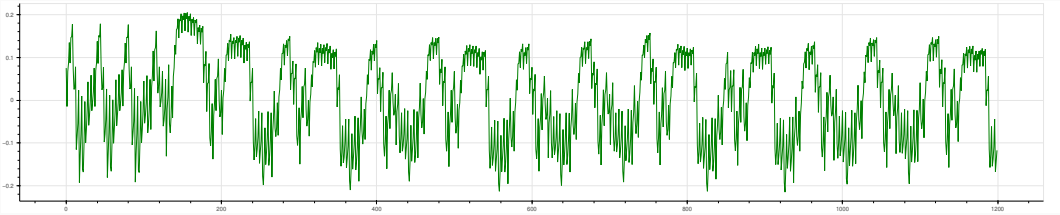
- Simple Power Attack: when information can be deduced by looking directly at the trace
- Pseudo-code of the modular exponentiation

```
1  uint64_t exp(uint64_t b, uint64_t e, uint64_t m) {
2    uint64_t res = 1;
3    for (int32_t i = 64; i >= 0; i -= 1) {
4      res = square(res);
5      res = modulo(res, m);
6      if (get_bit(e, i) == 1) {
7        res = mult(res, b);
8        res = modulo(res, m);
9      }
10   }
11   return res;
12 }
```

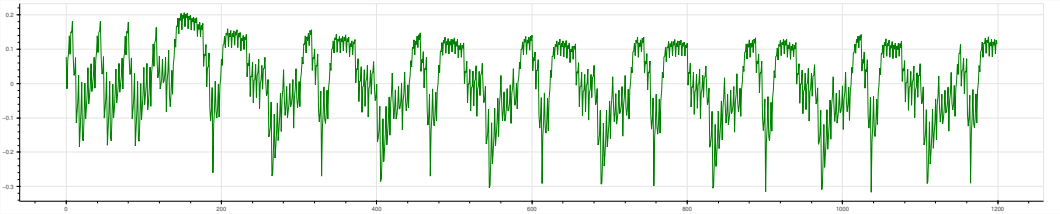
- Using the execution path, or an execution trace (sequence of all instructions executed by the processor), the power trace can reveal the value of all the exponent bits
- In RSA, this computation is made with the secret key as exponent
- The secure implementations are now always constant-time if the exponent is secret

SPA: Example of the Modular Exponentiation

- Square and Multiply

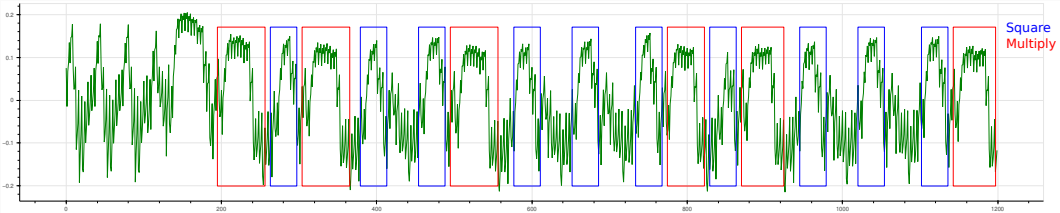


- Square and Multiply always

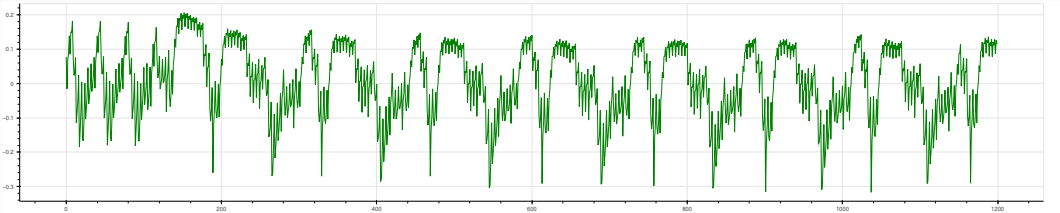


SPA: Example of the Modular Exponentiation

- Square and Multiply

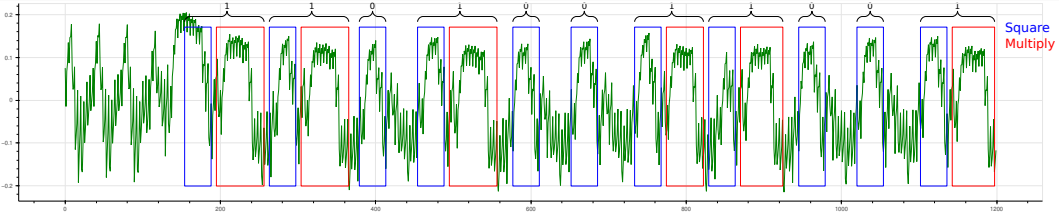


- Square and Multiply always

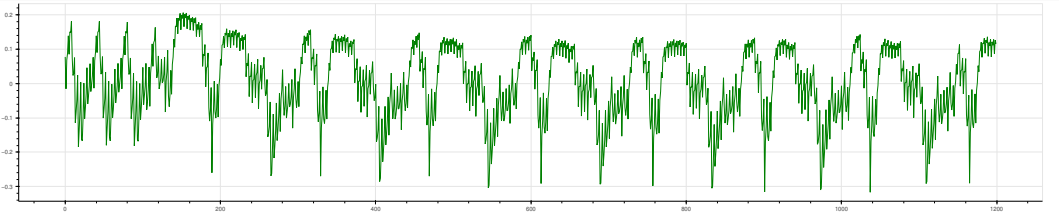


SPA: Example of the Modular Exponentiation

- Square and Multiply

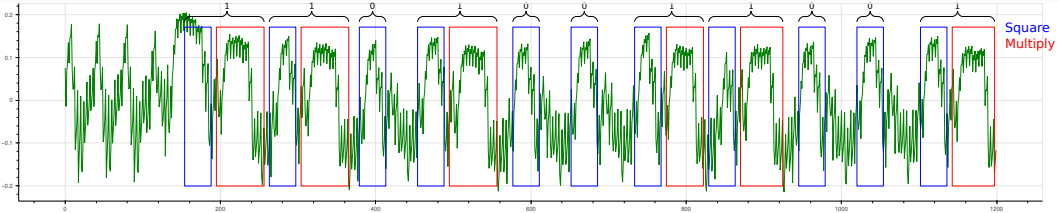


- Square and Multiply always

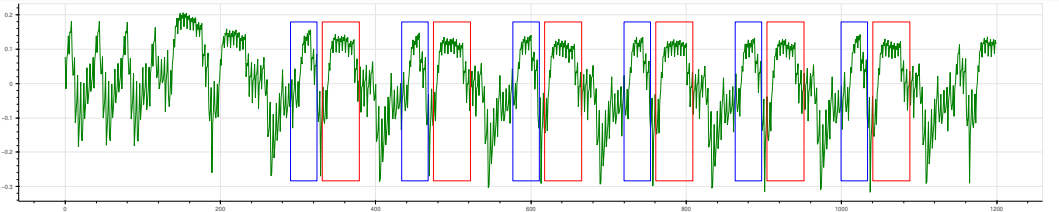


SPA: Example of the Modular Exponentiation

- Square and Multiply

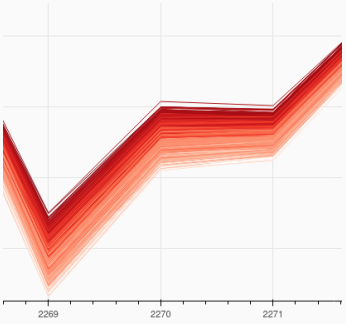


- Square and Multiply always



Variation in Power Related to Data

- In addition to instructions, data also influence power consumption
- In that case, the Hamming Weight can be a good power model



- Traditionally, SPA target power related to instructions
- Power related to data can be used in differential attacks (Differential Power Analysis, Correlation Power Analysis), using several / a lot of traces
- Possible to make a SPA by looking at data-related power consumption?

Modular Exponentiation in RSA

Input: Base **c**, exponent **d** $= (d_{k-1}...d_0)_2$, modulus **N**
Output: **r** $= \mathbf{c}^{\mathbf{d}} \bmod \mathbf{N}$

Original version

```
function ModularExponentiation(c, d, N)
    r ← 1
    for i from k − 1 to 0 do
        r ← r × r mod N
        if d_i = 1 then
            r ← r × c mod N
    return r
```

Traditional (constant-time) version

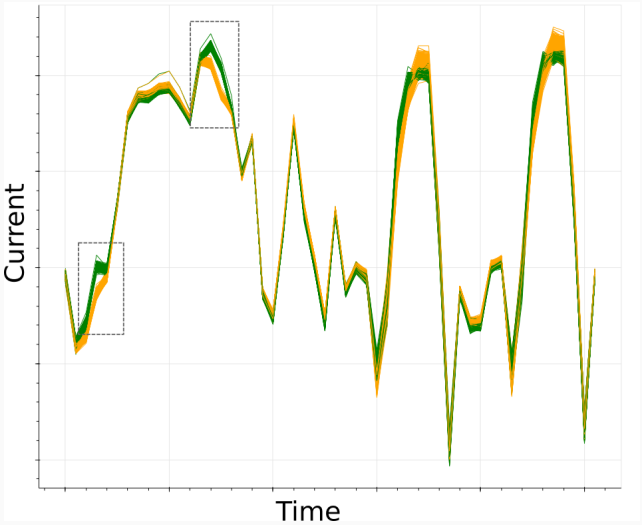
```
function ModularExponentiation(c, d, N)
    r ← 1
    for i from k − 1 to 0 do
        r ← r × r mod N
        x ← r × c mod N
        r ← SetCond(r, x, d_i)
    return r
```

- The mask computation either changes all bits or none, depending on the condition

```
function SetCond(v, u, c)                    ▷ if c = 0 returns v, else returns u
    msk0 ← wzero − c                         ▷ wzero is a word containing only '0' bits
    msk1 ← c − wone                          ▷ wone is a word containing the value 1 (0b0...01)
    for i from 0 to nlimbs − 1 do            ▷ nlimbs is the number of limbs
        r_i ← (msk0 & u_i) | (msk1 & v_i)    ▷ u_i and v_i are the i^th limbs of u and v
    return r
```

Blind-folded: Power Consumption of the SetCond function

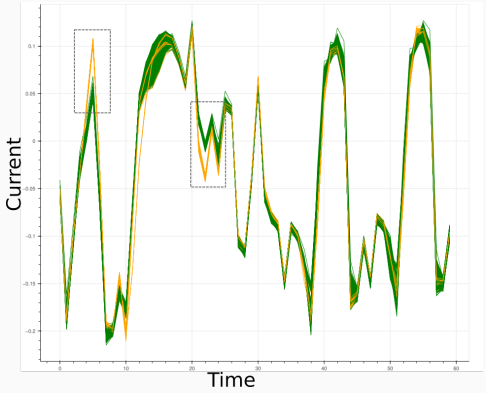
- Captures on the implementation of the `libgcrypt` library, on an Arm Cortex-M4 processor
- Folding the power trace to overlay the different calls to the SetCond function
- Different color depending of the condition value: masks computation visible
- A single sample is enough to recover the condition value
- Threshold computed from the biggest gap in the sorted values
  - Ongoing PhD thesis of Xunyue Hu



X. Hu, Q. L. Meunier, E. Encrenaz (2025). Blind-Folded: Simple Power Analysis Attacks using Data with a Single Trace and no Training. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2025(1), 475-496.

Blind-folded: Results

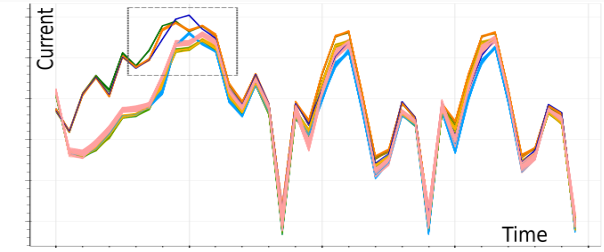
- Use of patterns to find the masks' computation (some non constant-time parts)
- On 30 traces corresponding to 30 different 2048-bit keys
  - 1 single incorrect bit (61,439/61,440 correct bits)
- Same attack on another version of the function using a sliding window
  - All bits correct

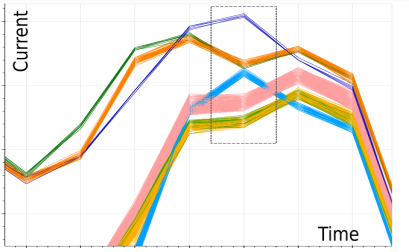


1: **rp** ← `result`;
2: **for** `i` from `j` to 0 **do**
3: **for** `k` from 0 to $(2^{w-1} - 1)$ **do**
4: **u** ← `precomp[k]`
5: SetCond(**w**, **u**, `k` = e0)
6: SetCond(**w**, **rp**, `i` ≠ 0)
7: **xp** ← MulMod(**rp**, **w**, **mp**)
8: **rp** ← **xp**

Blind-folded: Attacking ECDSA from the BearSSL library

- Implementation using a similar constant-time technique






```
1: function LookupGwin(idx)          ▷ returns idx*G
2:     P ← 0
3:     for i from 0 to 14 do
4:         ▷ EQ(x, y) returns 1 if x = y, 0 otherwise
5:         mask ← -EQ(idx, i + 1)
6:         for j from 0 to 17 do
7:             ▷ Gwin[i][j] contains word j of (i+1)*G
8:             P[j] ← P[j] | (mask & Gwin[i][j])
9:     return P
```

- Results: 1000 independent attacks on a single trace, all the `idx` values found

Outline

Side-Channel Attacks

Masking and Verification

Armistice: Masking Verification at the Micro-Architectural Level

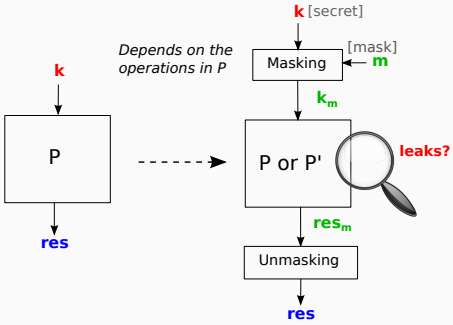
aLEAKator: Taking HDL into account

Conclusion

Outline

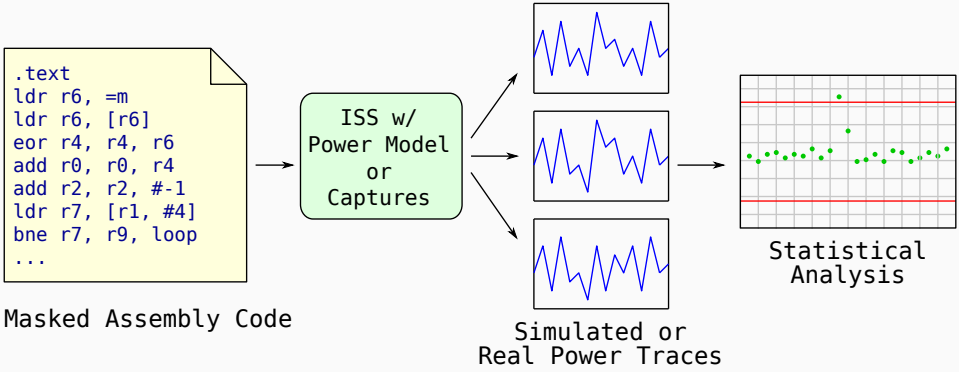
Masking Principle

- Combine secret variables with random variables called masks in order not to directly manipulate secret variables in expressions, for example using the $\oplus$ operation
- Make the distribution of values of t intermediate expressions independent from secret values: Threshold Probing Security (TPS) at order t
- Can be applied at the software or hardware (circuit) level



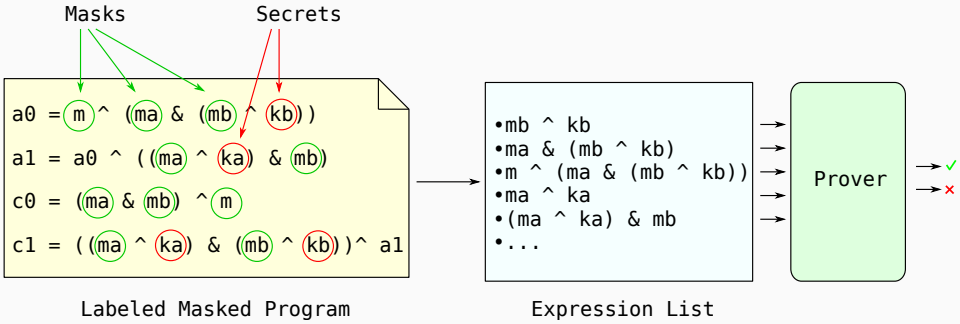
How to Verify a Masking Scheme?

- Empirically: Make simulations or captures, and use statistical tests (e.g. t-test)
- Drawback: Does not give any guarantee (depends on implementation factors)



How to Verify a Masking Scheme?

- Formally: Verify that all intermediate values have a secret independent distribution (TPS)



Formal Masking Verification

- Exhaustive approach: Enumerate all input values, for masks, secret and public variables, and check if the distributions obtained are the same for all secret values

`e = (k ⊕ m1) & m2`

k	m_1	m_2	e	
0	0	0	0	$P(e=0\|k=0)=\frac{3}{4}$
	0	1	0	
	1	0	0	$P(e=1\|k=0)=\frac{1}{4}$
	1	1	1	
1	0	0	0	$P(e=0\|k=1)=\frac{3}{4}$
	0	1	1	
	1	0	0	$P(e=1\|k=1)=\frac{1}{4}$
	1	1	0	

`e'= (k ⊕ m1) & m1`

e'	
0	$P(e'=0\|k=0)=\frac{1}{2}$
0	
1	$P(e'=1\|k=0)=\frac{1}{2}$
1	
0	$P(e'=0\|k=1)=1$
0	
0	$P(e'=1\|k=1)=0$
0	

- Different families of techniques, relying on symbolic variables in order to scale:
  - Inference methods [El Ouahma et al., 2017, Gao, 2020]
  - Substitution methods [Barthe et al., 2015]
  - Correlation sets based methods [Gigerl et al., 2021]
  - Encoding expressions using ROBDD [Knichel et al., 2020, Zhou et al., 2025]
- Challenges to address:
  - Expressiveness: supported operations, e.g. +, -, *, &, |, <<...
  - Scalability: maximum program size and expression size, verification time
  - Accuracy: if the method can produce false positives or false negatives (generally the case)

Substitution Methods

- Background: Substitution Algorithm as described in [Barthe et al., 2018]

---

procedure ThresholdProbingSecurity(e)

Inputs: tuple of expressions $V = (v_1, ..., v_n)$, flag *simplified* $= 0$, set of masks $M = \emptyset$

Step 1: if a secret k is involved in the computation of at least one expression in V then go to Step 2. Otherwise return True.

Step 2: while there exists a mask $m \notin M$ involved in the computation of an expression v_i of V, then find a sub-expression e in v_i such that $m \rightarrow e + m$ is bijective and substitute m by $e + m$ in all expressions. Extend M with $\{m\}$. If at least such a transformation occurred, go to Step 1. Otherwise go to Step 3.

Step 3: if *simplified* $\neq 0$, then return False. Otherwise, mathematically simplify the expressions in V. Then, set *simplified* to one and go back to Step 1.

- Lets a lot of implementation choices:
  - Data structures and memory implantation
  - Selection of the mask occurrence at step 2
  - Simplification rules

First Proposal: LeakageVerif

- Python library for the verification of masked symbolic expressions, method based on the substitution approach
- Benefits from Python control flow
- Scalable and easy-to-use tool, overcomes the limitations of some other tools
- Possible to use concrete values for simulation
- Each variable is tagged secret, mask or public
- Variables and constants have a fixed arbitrary width (e.g. 9 bits)
- Support for boolean and arithmetic operations

```
# 8-bit variable named 'm0' of type Mask
m0 = symbol('m0', 'M', 8)
# 8-bit variable named 'm1' of type Mask
m1 = symbol('m1', 'M', 8)
# 8-bit variable named 'k0' of type Secret
k0 = symbol('k0', 'S', 8)
# 8-bit variable named 'k1' of type Secret
k1 = symbol('k1', 'S', 8)
# expression computation
exp = (m0 ^ k0) & (m0 ^ m1 ^ k1)
# check for leakage in the expression value
res = checkTpsVal(exp)
```

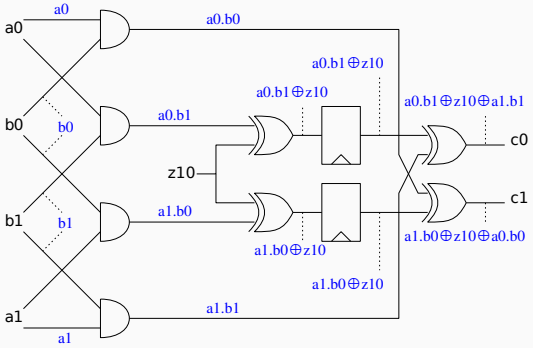
$\Rightarrow$ Can verify a full masked AES (Herbst scheme) in 0.5 second with symbolic multiplication, and in 10 seconds with a non-symbolic multiplication

- Experiments made on 40+ programs show that `LeakageVerif` performs well regarding expressiveness, performance and accuracy compared to 3 state-of-the-art tools

Q. L. Meunier, E. Pons, K. Heydemann (2023). Leakageverif: Efficient and scalable formal verification of leakage in symbolic expressions. IEEE Transactions on Software Engineering, 49(6), 3359-3375.

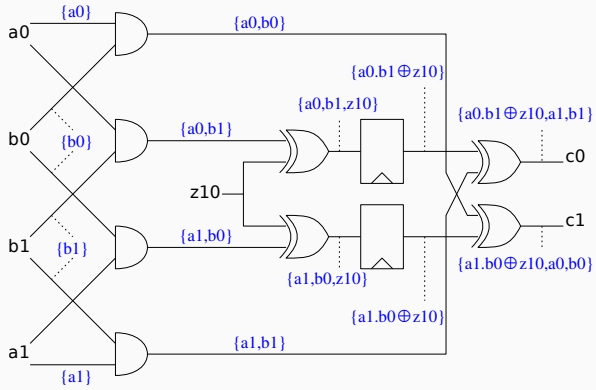
Share-based Security Properties (at order t)

- Different way of defining masking: splitting a secret $\mathtt{k}$ into several parts called shares, such that $\mathtt{k} = \mathtt{k}0 \star ... \star \mathtt{k}_{n-1}$ (e.g. $\star = \oplus$)
- Implies different security properties, based on shares, for example: Non-Interference (NI) [Barthe et al., 2015], Strong Non-Interference (SNI) [Barthe et al., 2016], Probe Isolating Non-Interference (PINI) [Cassiers and Standaert, 2020]
- Non-Interference: the distribution of each tuple of t expressions depends at most on t shares per input
- DOM-AND [Groß et al., 2017] verifies NI: each expression distribution depends at most on 1 share per input



Glitches

- Consider that transient values on wires are sufficient to leak information, and included in the leakage model
- Modifies the leakage associated to each wire
- Orthogonal to the security property



Higher Order

- Attack at order n: an attack at order n can use n probes (e.g. wires)
  - Idea: the measures from the different wires can be combined
  - In practice, can result from a single global power measure
- Security at order n: security at order n provides resistance against attacks at order n
  - Implies sharing at order $t > n$
- Security at order n requires to enumerate all possible tuples of size n, and show secret independence for each
- Order 2 DOM-AND circuit is not secure at order 2: the joint distribution of the expressions ($\texttt{ma}$, $\texttt{ma} \oplus \texttt{a}$) is not independent from $\texttt{a}$

VerifMSI: Verification of Masking Schemes Implementations

- Extension of `LeakageVerif`, available at `https://github.com/quentin-meunier/VerifMSI`
- Both shares and secrets representations
- Hardware constructs with glitches (gates, registers)
- Implements several security properties
- Verifications at higher orders using optimisations to reduce the probe number
- Example: Order 1 DOM-AND gadget in `VerifMSI`

```
a    = symbol('a', 'S', 1) # 1-bit secret
b    = symbol('b', 'S', 1) # 1-bit secret
z10 = symbol('z10', 'M', 1) # 1-bit mask
# Do the sharing for 'a' and 'b'
a0, a1 = getRealShares(a, 2)
b0, b1 = getRealShares(b, 2)
# Create input gates
a0  = inputGate(a0)
a1  = inputGate(a1)
b0  = inputGate(b0)
b1  = inputGate(b1)
z10 = inputGate(z10)
# Cross products
a0b0 = andGate(a0, b0)
a0b1 = andGate(a0, b1)
a1b0 = andGate(a1, b0)
a1b1 = andGate(a1, b1)
# Remaining gates and registers
a1b0 = xorGate(a1b0, z10)
a1b0 = Register(a1b0)
a0b1 = xorGate(a0b1, z10)
a0b1 = Register(a0b1)
c0   = a0b0
c0   = xorGate(c0, a0b1)
c1   = a1b1
c1   = xorGate(c1, a1b0)
# Check the NI security property
checkSecurity(order, wGlitches, 'ni', c0, c1)
```

Q. L. Meunier, A. Taleb (2023). VerifMSI: Practical verification of hardware and software masking schemes implementations. In 20th International Conference on Security and Cryptography. SciTePress.

Outline

Side-Channel Attacks

Masking and Verification

Armistice: Masking Verification at the Micro-Architectural Level

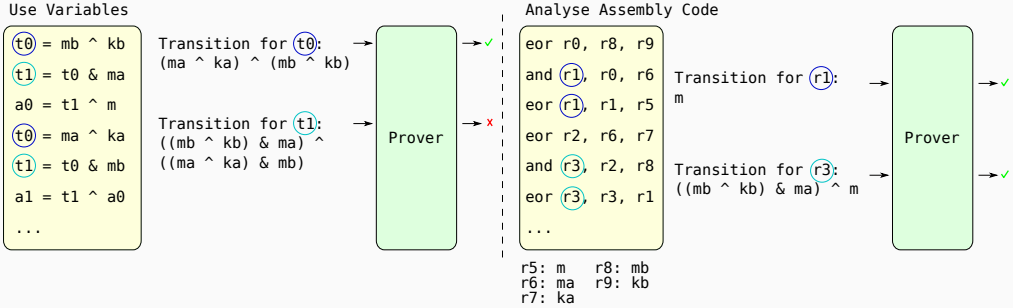
aLEAKator: Taking HDL into account

Conclusion

Outline

Reducing the Gap Between the Analysed Expressions and Real Leakages

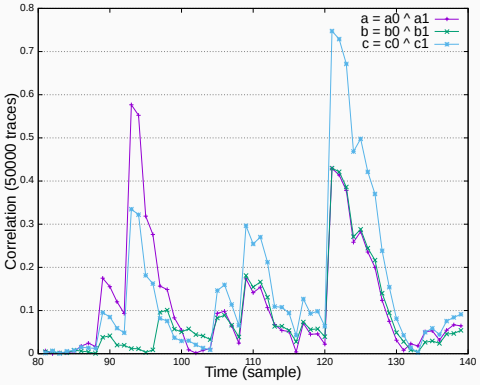
- Expressions considered for the proof do not necessary model well real leakages
- $\Rightarrow$ A proven masking scheme can leak secrets in practice
- 2 verification techniques have been proposed to replace the verification of intermediate values
  - Verify the transitions between Variables
  - Verify the transitions between General Purpose Registers (GPR) at assembly level



Proven "Leakage Free"?

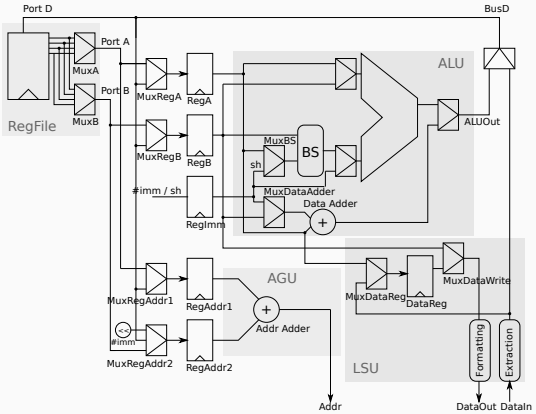
- Example of the "ISW And" [Ishai et al., 2003]
  - Proven leakage-free in the GPR value model and in the GPR transition model

```
; 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]
```



- How to better model real leakages?

Armistice: Micro-Architectural Leakage Modeling for Masked Software Formal Verification

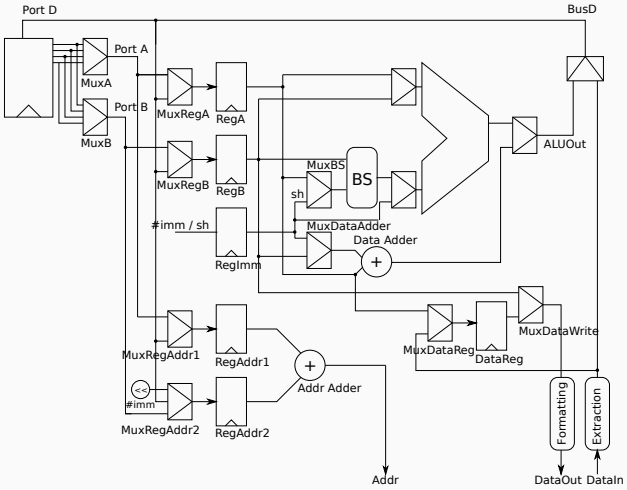


- Arm Cortex-M3: Manual model from a RTL description
  - Stateful circuit ⇒ Simulation (not in VerifMSI)
- Memory: Black-box model (no RTL)
- Uses `VerifMSI` for formal verification, experimental validation of reported leakaged on a STM32F1 Board
- Design and implementation of many "leakage test vectors":
  - Leakage sources detection
  - Validation
  - Ranking

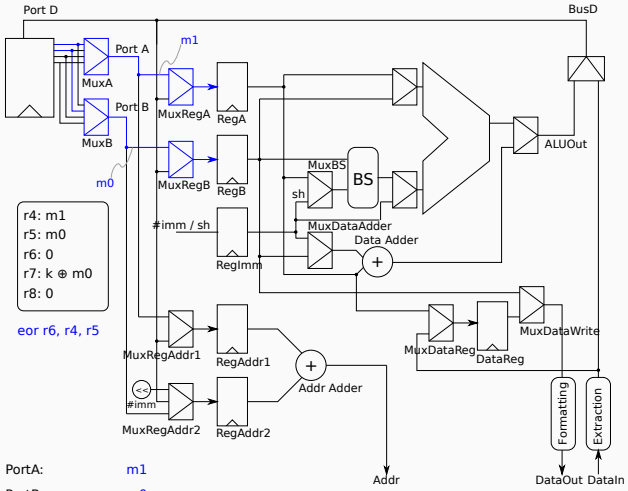
- Accessible online: `https://www-soc.lip6.fr/armistice`

A. De Grandmaison, K. Heydemann, Q. L. Meunier (2022). ARMISTICE: Microarchitectural Leakage Modeling for Masked Software Formal Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 41(11), 3733-3744.

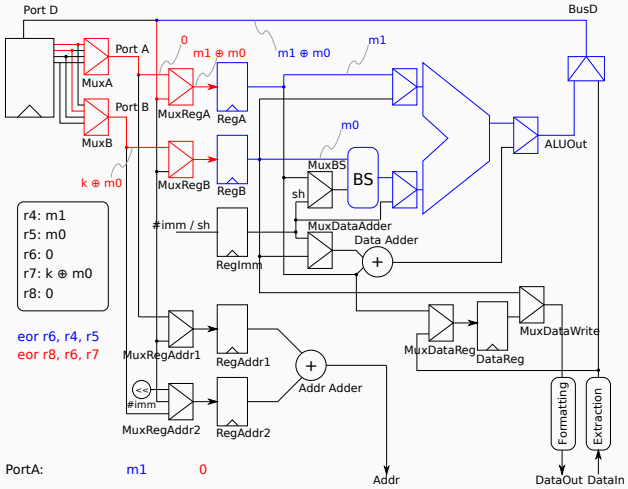
Arm Cortex-M3: Example



Arm Cortex-M3: Example

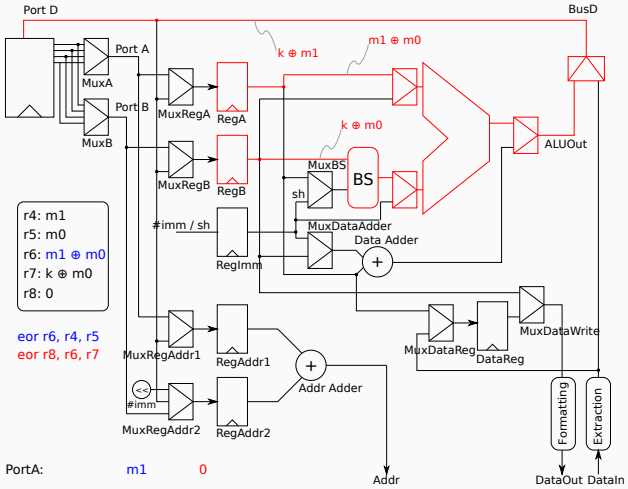


Arm Cortex-M3: Example



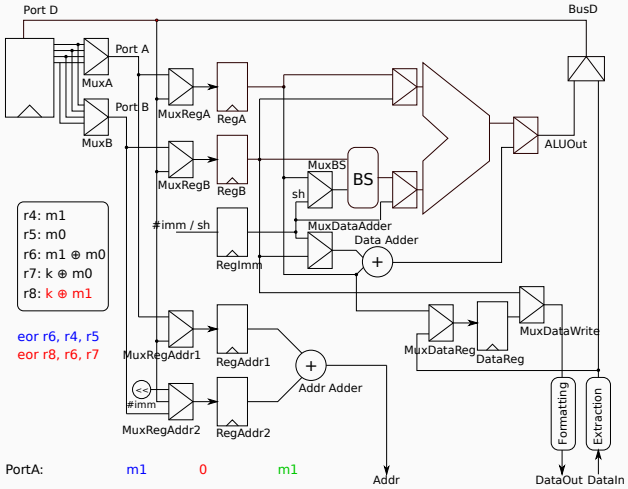
PortA:	m1	0
PortB	m0	k ⊕ m0
MuxRegA / RegA:	m1	m1 ⊕ m0
MuxRegB / RegB:	m0	k ⊕ m0
ALUOut	m1 ⊕ m0	

Arm Cortex-M3: Example



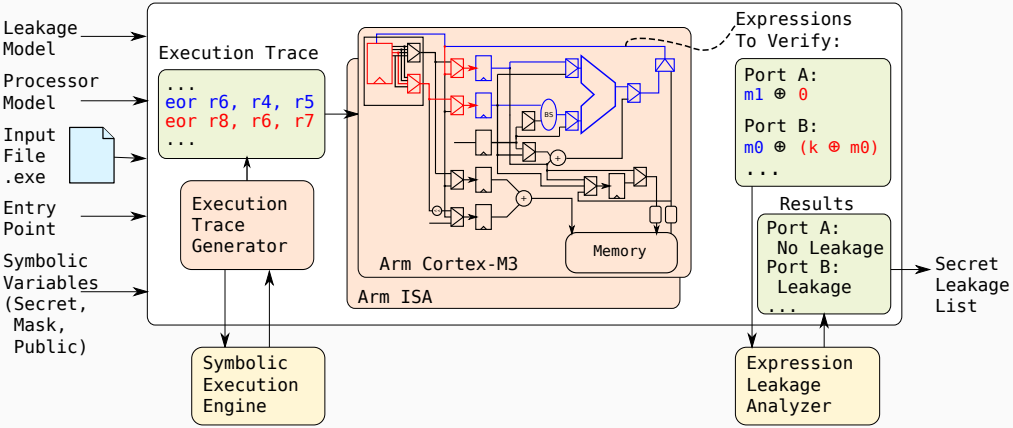
PortA:	m1	0
PortB	m0	k ⊕ m0
MuxRegA / RegA:	m1	m1 ⊕ m0
MuxRegB / RegB:	m0	k ⊕ m0
ALUOut	m1 ⊕ m0	k ⊕ m1

Arm Cortex-M3: Example



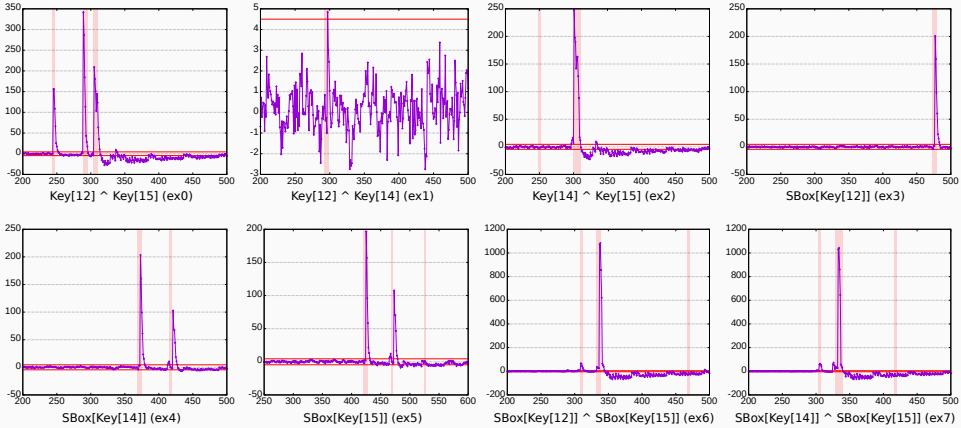
PortA:	m1	0	m1
PortB	m0	k ⊕ m0	k
MuxRegA / RegA:	m1	m1 ⊕ m0	m0
MuxRegB / RegB:	m0	k ⊕ m0	k
ALUOut	m1 ⊕ m0	k ⊕ m1	k ⊕ m0

Armistice Framework



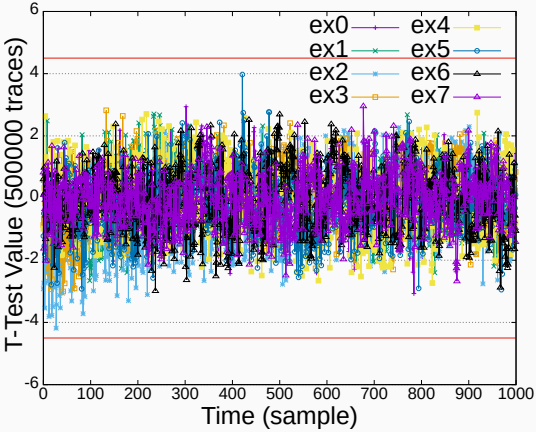
Accuracy and Exploitability (1/2)

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



Accuracy and Exploitability (2/2)

- Using Armistice output to suppress all observable leakages
- Manual addition of carefully designed instructions to clean the parts of the data path involved in the leaking transitions



Outline

Side-Channel Attacks

Masking and Verification

Armistice: Masking Verification at the Micro-Architectural Level

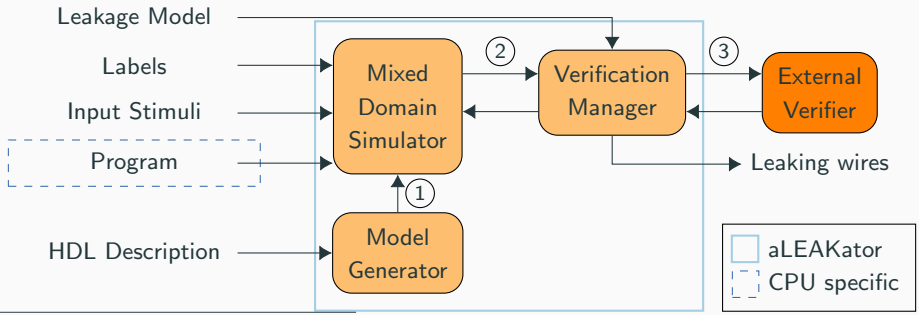
aLEAKator: Taking HDL into account

Conclusion

Outline

Extending Armistice

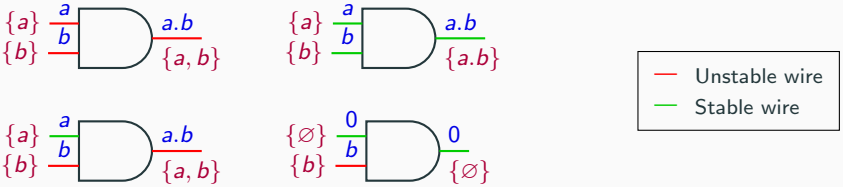
- The work done in Armistice is useful, but relies on a manual model of the datapath, which is laborious to write, incomplete and error prone
- Ideally, we would like the processor model to be an input of the framework
- Idea: Take the processor RTL description and extend the RTL simulation with symbolic expressions
- Overview of this new framework, named aLEAKator:



N. Amiot, Q. L. Meunier, E. Encrenaz, K. Heydemann. aLEAKator: : HDL Mixed-Domain Simulation for Masked Hardware & Software Formal Verification. *accedpted in Transactions on Cryptographic Hardware and Embedded Systems (TCHES) 2026*

Leakage Model and Stability

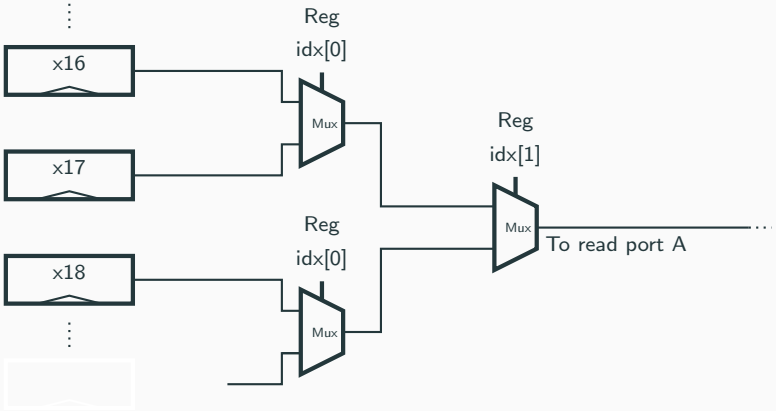
- Stateful simulation allows to take into account transitions and glitches, but can also leverage signal stability: property expressing that the value on a wire has remained unchanged all along the current cycle
- $\rightarrow$ in this case glitches are not possible [Gigerl et al., 2021, Müller and Moradi, 2024]



- [Müller and Moradi, 2024] also introduces the notion of Robust but Relaxed (RR) probing model as the leakage model considering transitions and glitches, along with stability
- In the following, in our use of RR, we also consider the RTL signal granularity, meaning that a n-bit signal in the source code has to be verified as a single word – what we call `sw-probing-model`

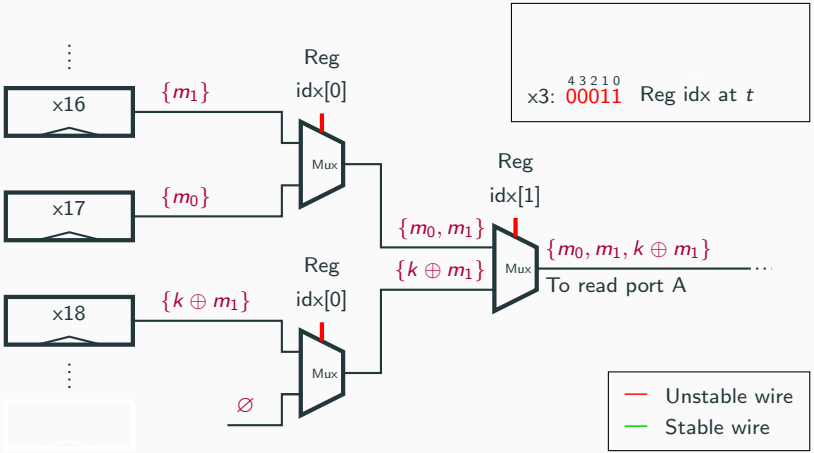
Stability Example on the Register Bank

- Leakage analysis with glitches when reading the register bank



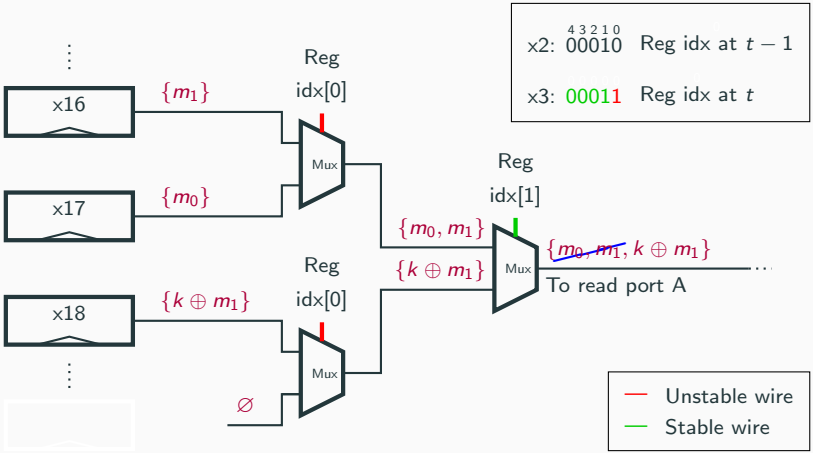
Stability Example on the Register Bank

- Leakage analysis with glitches when reading the register bank without stability



Stability Example on the Register Bank

- Leakage analysis with glitches when reading the register bank with stability



Gates and Cores

- List of considered gates after synthesis in the simulation backend:
  `Not`, `And`, `Or`, `Xor`, `Mux`, `Ucmp`, `Scmp`, `Equal`, `Add`, `Sub`, `Neg`, `Mul`, `Sll`, `Srl`, `Sra`, `(R)trunc`, `(R)zext`, `Sext`, `Blit`, `Repeat`, `MemRead`, `MemWrite`, `Register`
- $\Rightarrow$ Maps more or less directly on symbolic expressions in `VerifMSI`
- Characteristics of the considered cores after synthesis:

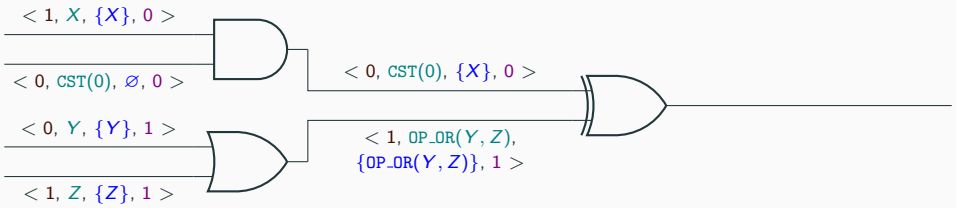
Processor	Number of gates	Number of signals	Number of bits
Cortex-M3	10 435	14 661	41 776
Cortex-M4	11 438	15 946	94 000
Ibex	2 389	4 034	18 692
CV32E40P	3 019	8 010	30 846

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:

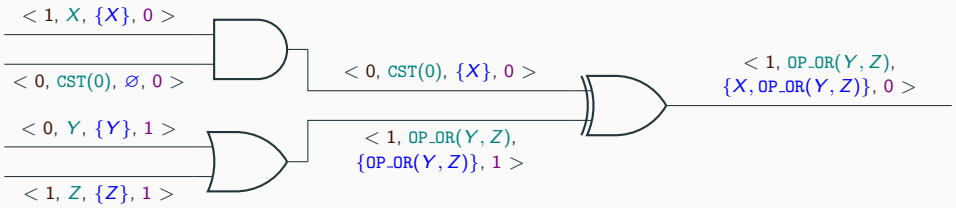
Mixed-Domain Simulation: Extension of Concrete Simulation

- 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:

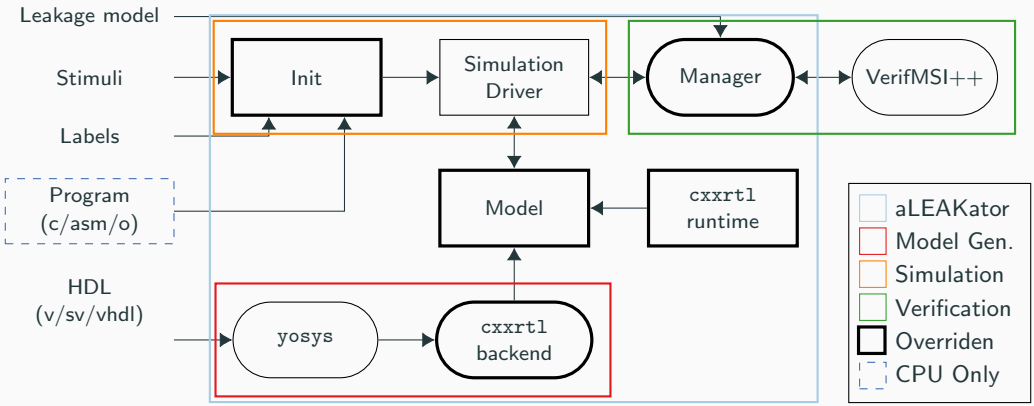


Mixed-Domain Simulation: Extension of Concrete Simulation

- 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:

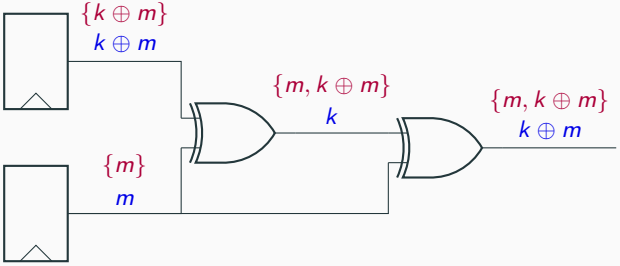


aLEAKator Implementation



Optimizing the Number of Verifications

- To limit verifying costs in CPU time and memory, aLEAKator:
  - Skips the verification of constant expressions or expressions without secret variables (most of signals)
  - Uses a cache for the verification verdict (approximately 95% hit rate)
  - Exploits properties to skip the verification on wires whose leakages are fully captured by other wires (average 2.2× less verifications)



- Without stability, glitchy expression sets can only grow in combinatorial gates → only verify primary outputs and registers inputs
- More specific rules are needed when considering stability

Results: Software AES Verification on 4 Processors (1/2)

- C implementation of AES masked with Herbst scheme [Herbst et al., 2006], iteratively verified and patched at the source level to prevent secret leakages in the `1-probing model`
- Patches include memory barriers, data reorganization and pipeline flushes

Processor	Cycles	Leaking cycles	Time	Ram (Go)	Unique verified expressions
Cortex-M3	10.1K	0	5m04	43.7	90K
Cortex-M4	10.1K	0	5m48	45.3	93K
Ibex	8.8K	0	21m41	226.8	185K
CV32E40P	8.8K	0	43m44	423.5	199K

- Complete formal verification on the four CPUs (including the Key Schedule), highlighting the versatility and scalability of aLEAKator

Results: Software AES Verification on 4 Processors (2/2)

- What about the `RR 1-probing model`?
- Too difficult to patch, but aLEAKator is still able to perform the complete verification, despite using more time and memory

Processor	Cycles	Leaking cycles	Time	Ram (Go)	Unique verified expressions
Cortex-M3	10.1K	8 329	18m52	208.0	148K
Cortex-M4	10.1K	8 329	20m14	211.5	147K
Ibex	8.8K	6 651	22m08	301.0	90K
CV32E40P	8.8K	6 705	39m01	333.7	86K

- Other software verified: DOM-AND [Groß et al., 2017], Secmult [Rivain and Prouff, 2010]
- Some HW accelerators have been verified as well

Experimental Validation Against Real Traces

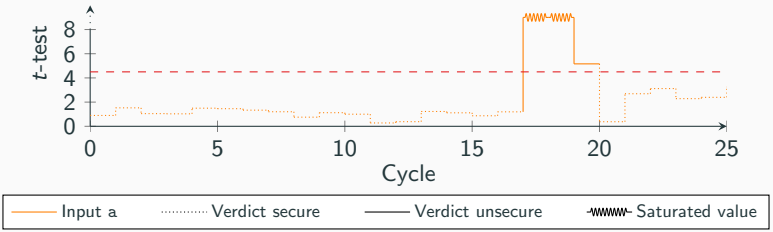
- Example program: Refreshing both shares of a secret on the Cortex-M4

```
ldr r0, =a0
ldr r1, =a1
ldr r2, =m
ldr r3, =blk
ldr r4, [r0] // a0 in r4
ldr r5, [r2] // m in r5
```

```
// Refresh a0
eor r4, r4, r5
str r4, [r0]
// Clear write LSU path
str r3, [r3]
eor r4, r4, r4 // Clear r4
mov r4, 0x0 // Clear ALU
```

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

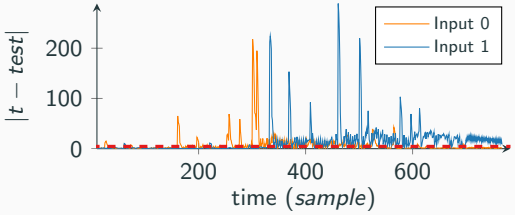
- Comparison between detected leakages in aLEAKator and observed leakages



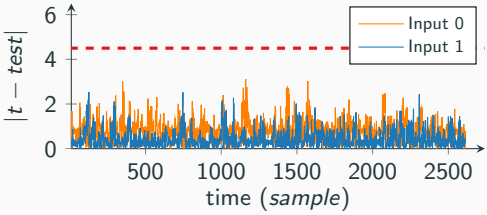
Experimental Validation Against Real Traces

- aLEAKator can detect secret leakages which are not observable: considered glitches that do not happen in practice, power consumption too weak on leaking wire, etc.
- Claim / Hope: when no secret leakages are detected by aLEAKator, no secret leakages are observed in practice
- Experiments made with the programs DOM-AND and Secmult on the Cortex-M3 and Cortex-M4, with two versions each:
  - DOM-AND: secure in GPR with transitions (*unsecure*) and RR 1-probing secure (no leakage detected)
  - Secmult: `-O3` compiled (*unsecure*) and RR 1-probing secure (no leakage dectected)
- Results for the Secmult program for the Cortex-M3:

(a) `Secmult unsecure`



(b) `Secmult secure`



Outline

Side-Channel Attacks

Masking and Verification

Armistice: Masking Verification at the Micro-Architectural Level

aLEAKator: Taking HDL into account

Conclusion

Outline

Conclusion

- In summary:
  - aLEAKator is a framework for formally proving the absence of secret leakages in masked software, with a configurable leakage model, using symbolic HDL simulation
  - Versatile method: 5 CPUs evaluated (Cortex-M3, Cortex-M4, CV32E40P, Ibex, Coco-Ibex)
  - Efficient: a lot faster than the State-of-the-Art for tested SW implementations
  - Also efficient for verifying coprocessor designs (to a certain extent), faster than existing solutions
  - Helps identify the secret leakage sources and fix them by modifying the hardware or software
  - Available as an open-source project[1]
- Future work:
  - Iteratively and automatically patch assembly code
  - Use the approach for faults

[1] https://github.com/noeamiot/aLEAKator

Thank you!

Contact:
Email: quentin.meunier@lip6.fr

Barthe, G., Belaïd, S., Dupressoir, F., Fouque, P.-A., Grégoire, B., and Strub, P.-Y. (2015).
Verified proofs of higher-order masking.
In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 457–485. Springer.

Barthe, G., Belaïd, S., Fouque, P.-A., and Grégoire, B. (2018).
maskverif: automated analysis of software and hardware higher-order masked implementations.
Technical report, Technical Report 562.

Barthe, G., Belaïd, S., Dupressoir, F., Fouque, P.-A., Grégoire, B., Strub, P.-Y., and Zucchini, R. (2016).
Strong non-interference and type-directed higher-order masking.
In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 116–129.


Cassiers, G. and Standaert, F.-X. (2020).
Trivially and efficiently composing masked gadgets with probe isolating non-interference.
IEEE Transactions on Information Forensics and Security, 15:2542–2555.


El Ouahma, I. B., Meunier, Q. L., Heydemann, K., and Encrenaz, E. (2017).
Symbolic approach for side-channel resistance analysis of masked assembly codes.
In *6th International Workshop on Security Proofs for Embedded Systems (PROOFS)*.


Gao, P. (2020).
Formal verification of masking countermeasures for arithmetic programs.
In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 1385–1387.


Gigerl, B., Hadzic, V., Primas, R., Mangard, S., and Bloem, R. (2021).
Coco:{Co-Design} and {Co-Verification} of masked software implementations on {CPUs}.
In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1469–1468.


Groß, H., Mangard, S., and Korak, T. (2017).
An efficient side-channel protected aes implementation with arbitrary protection order.
In *Topics in Cryptology–CT-RSA 2017: The Cryptographers Track at the RSA Conference 2017, San Francisco, CA, USA, February 14–17, 2017, Proceedings*, pages 95–112. Springer.


Herbst, C., Oswald, E., and Mangard, S. (2006).
An aes smart card implementation resistant to power analysis attacks.
In *ACNS*, volume 3989, pages 239–252. Springer.


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


Knichel, D., Sasdrich, P., and Moradi, A. (2020).
Silver–statistical independence and leakage verification.
In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 787–816. Springer.


Müller, N. and Moradi, A. (2024).
Robust but relaxed probing model.
IACR Transactions on Cryptographic Hardware and Embedded Systems, 2024(4):451–482.


Rivain, M. and Prouff, E. (2010).
Provably secure higher-order masking of aes.
In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 413–427. Springer.


Zhou, F., Chen, H., and Fan, L. (2025).
Prover-toward more efficient formal verification of masking in probing model.
IACR Transactions on Cryptographic Hardware and Embedded Systems, 2025(1):552–585.