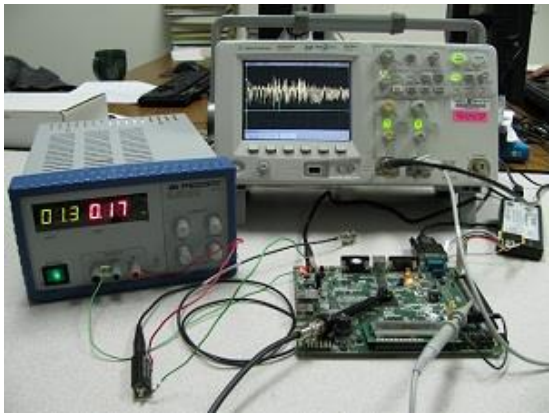
From Trivial Composition to Full Verification and an Application to Masking in Hardware

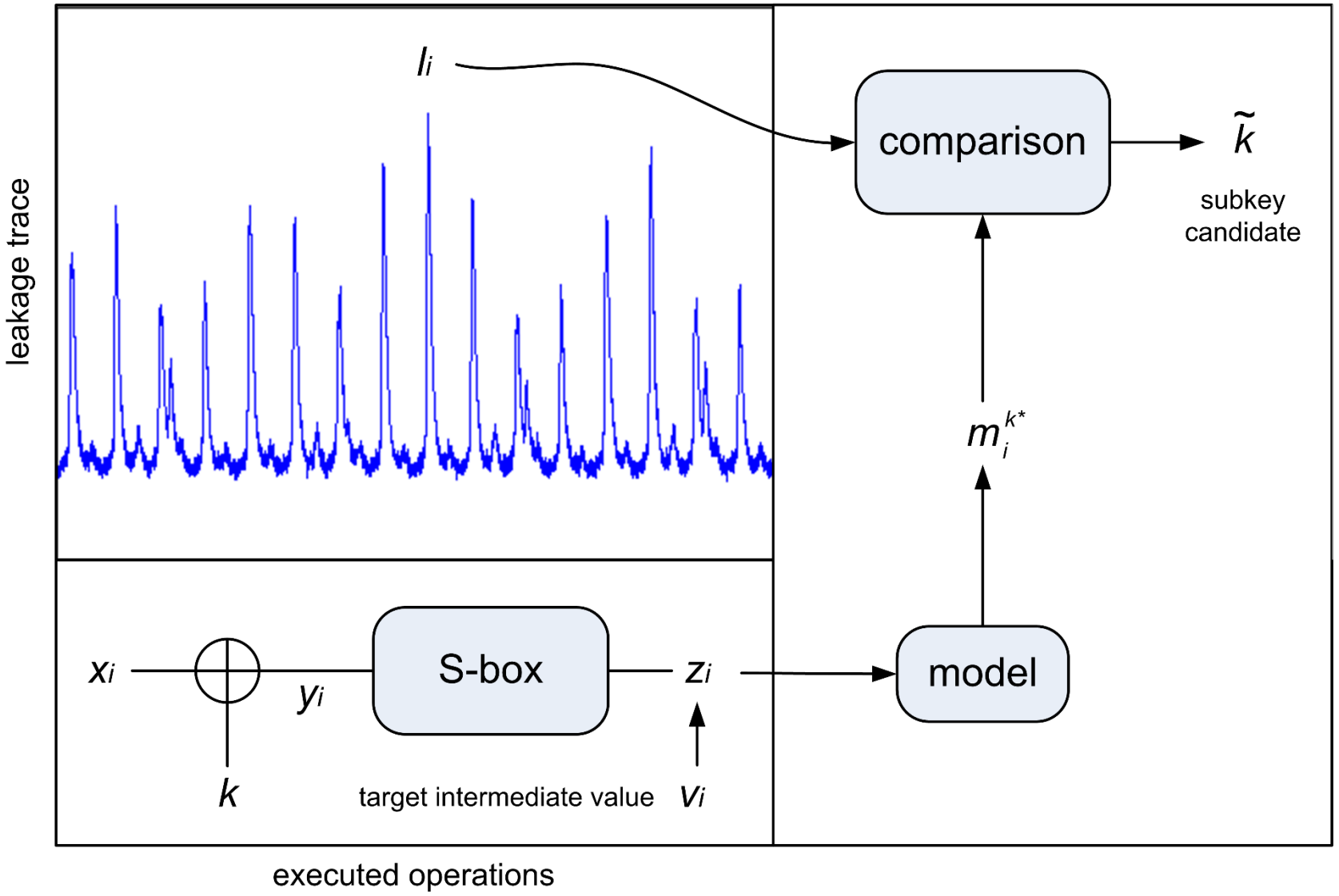


Gaëtan Cassiers, François-Xavier Standaert

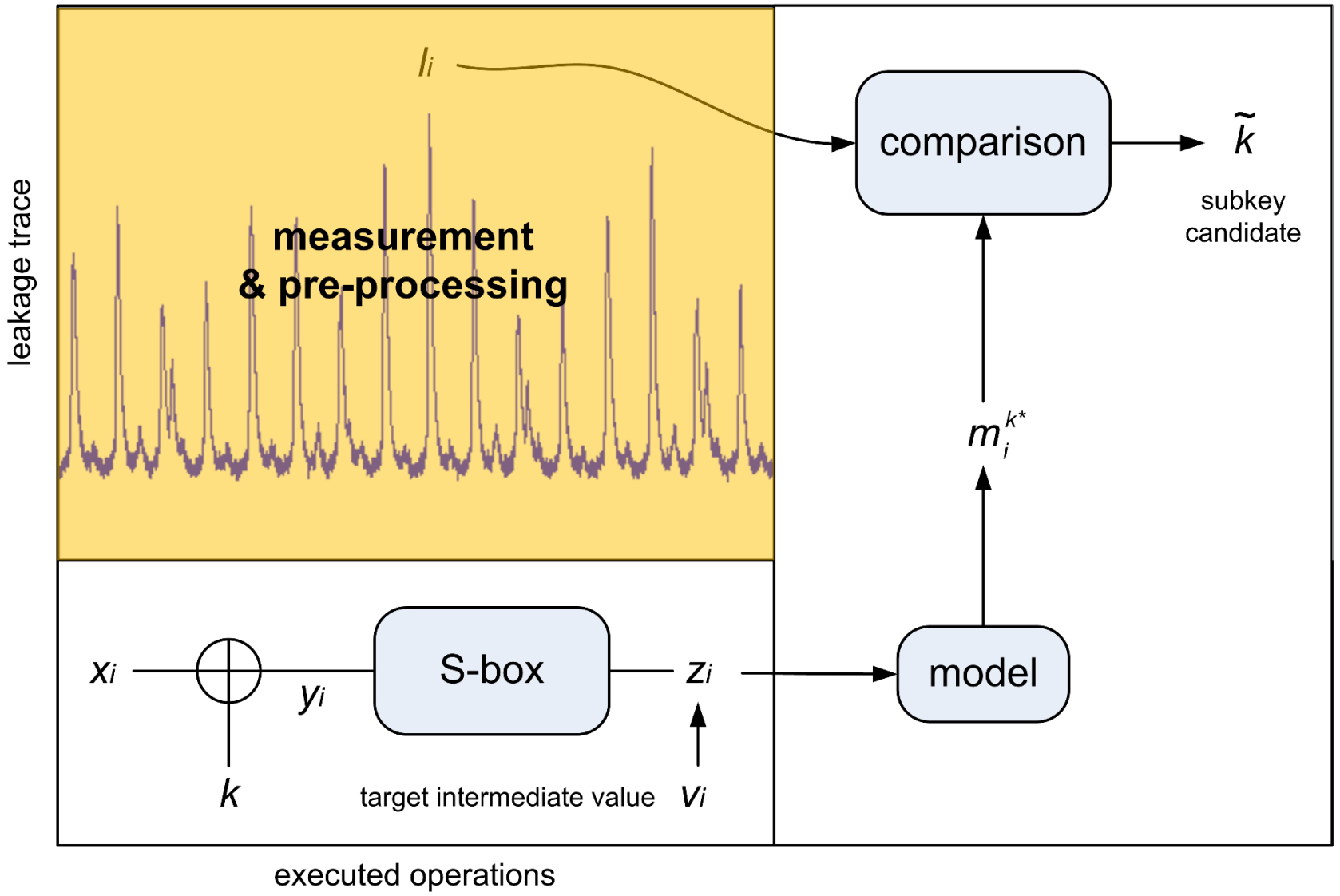
UCLouvain (Belgium)

VeriSiCC Seminar, Paris, France, September 2019

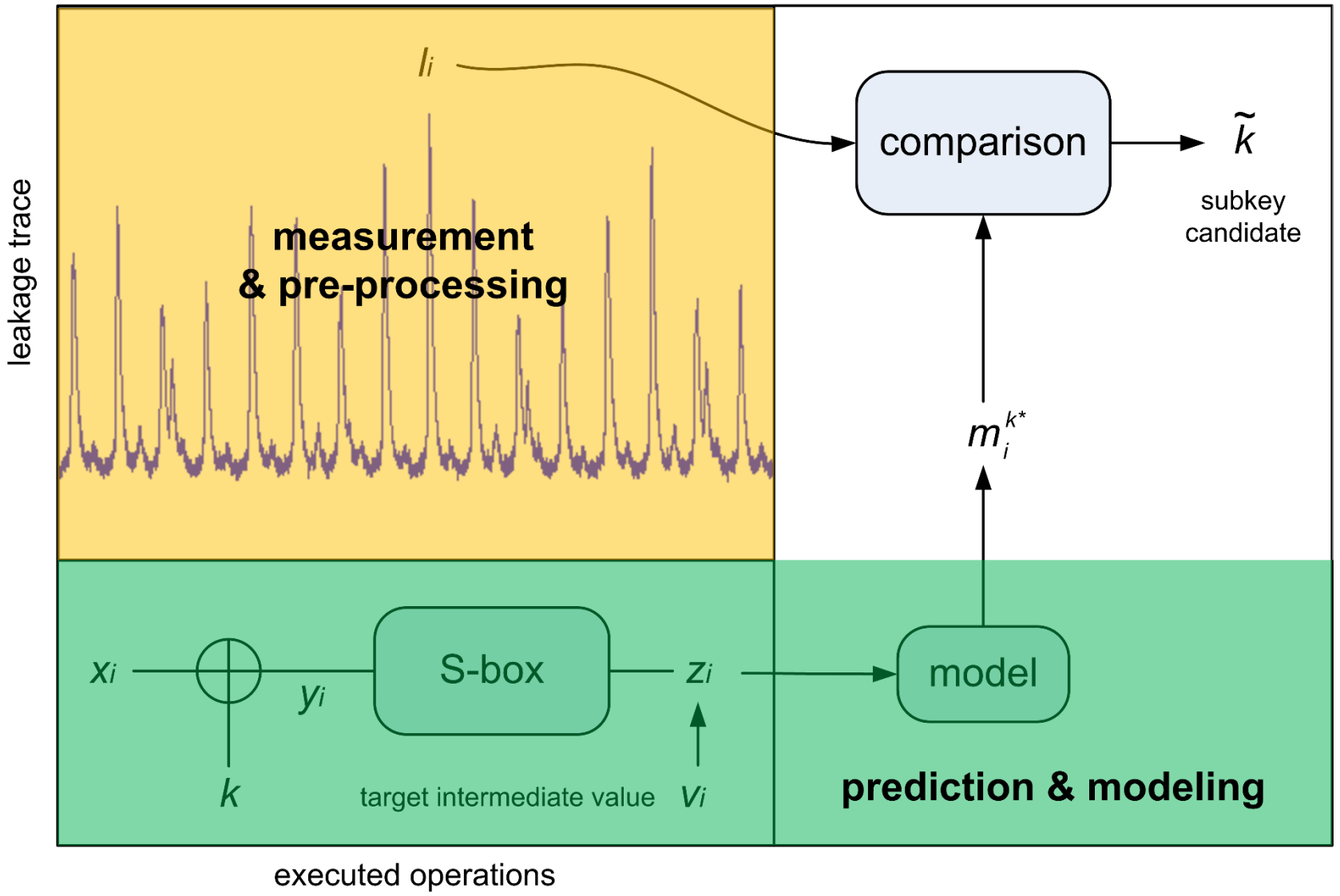
Side-Channel Analysis



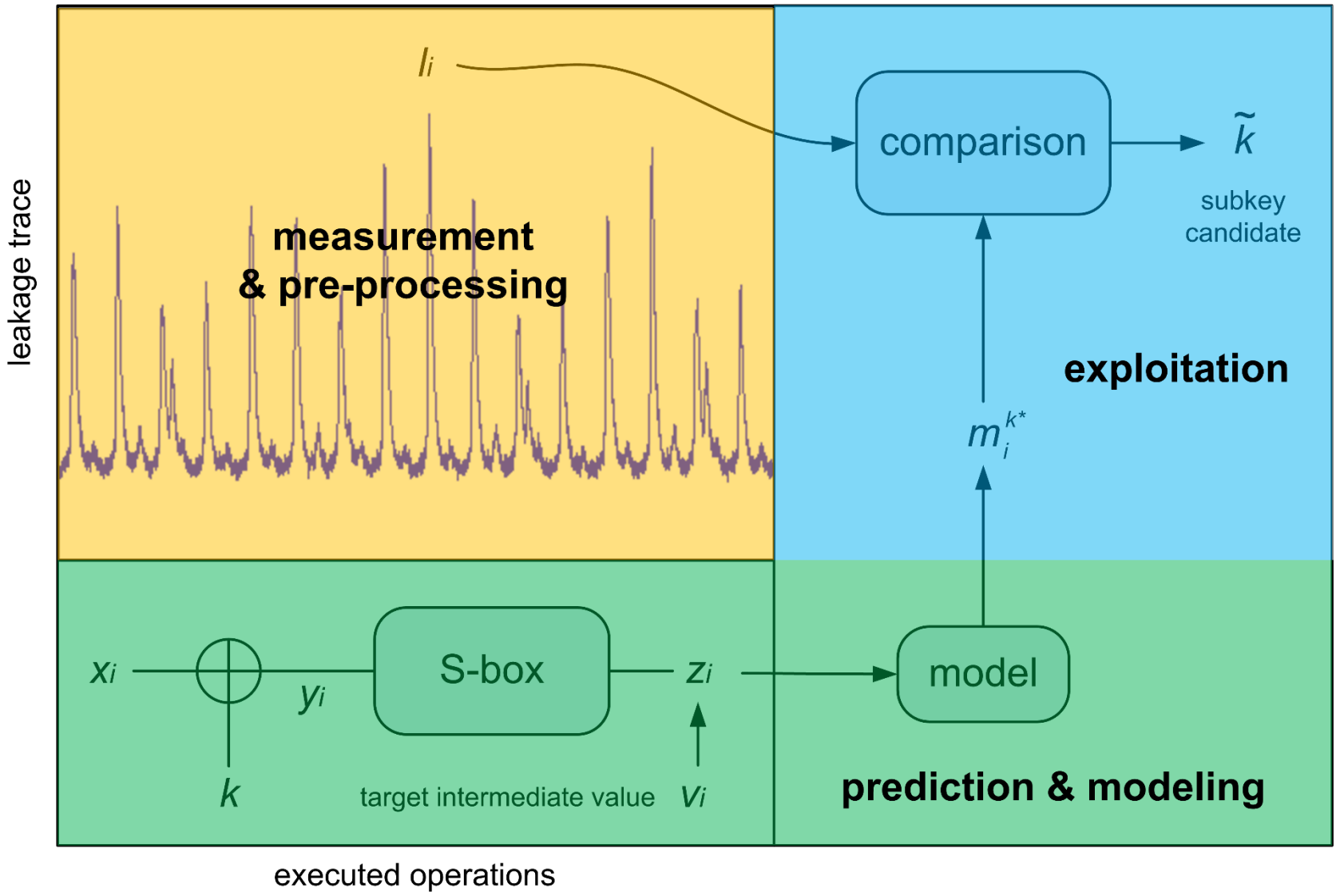
Side-Channel Analysis



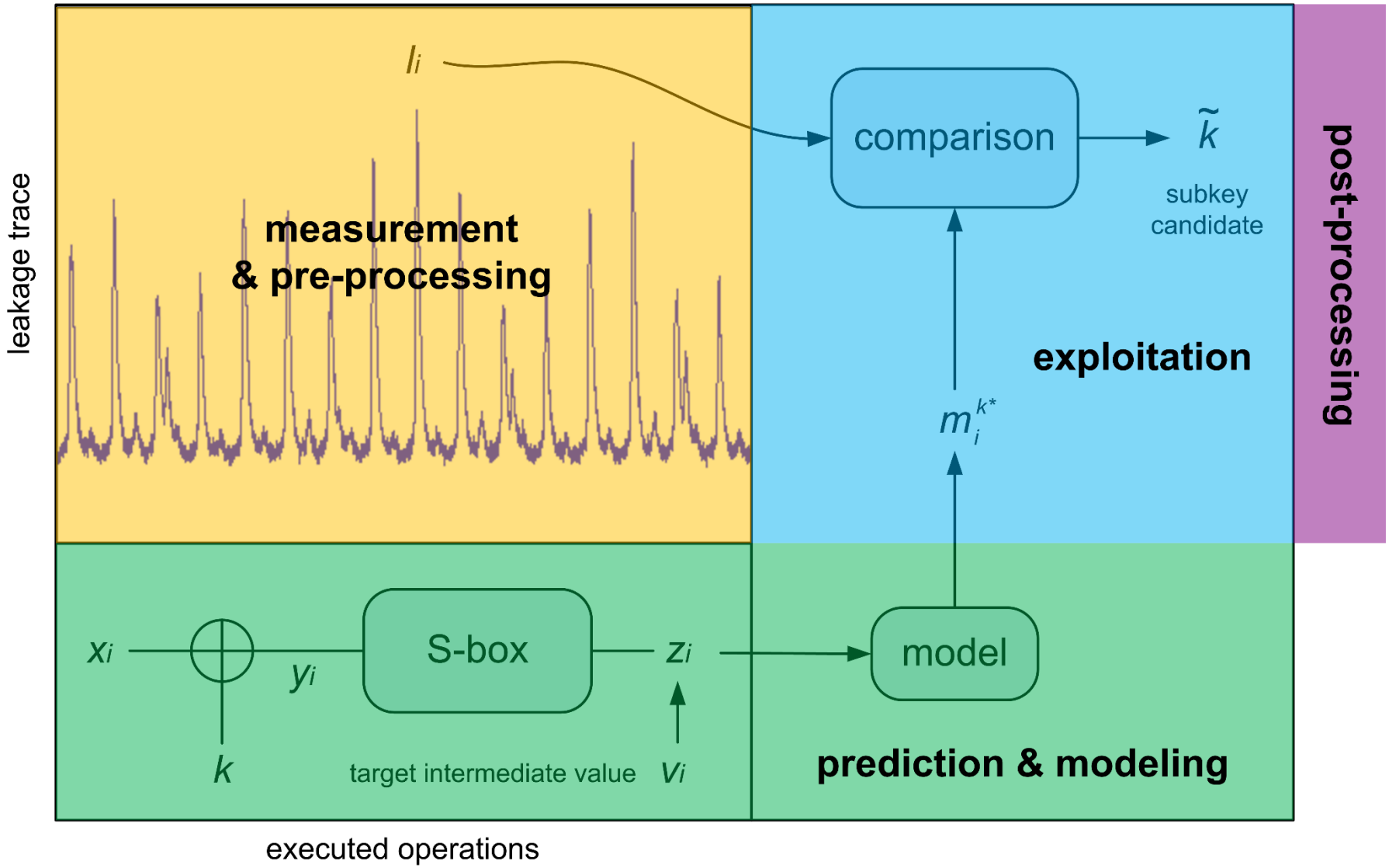
Side-Channel Analysis



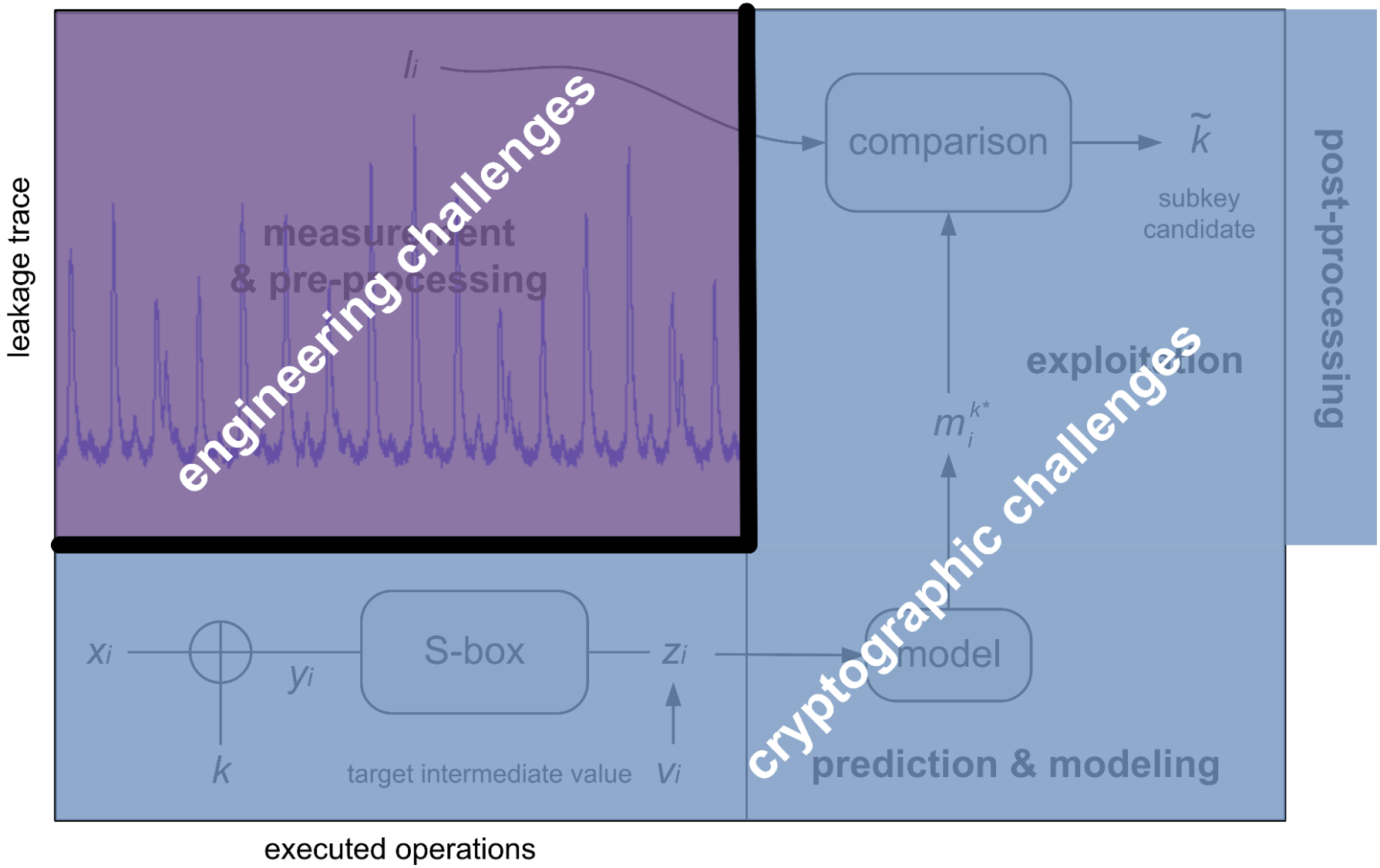
Side-Channel Analysis



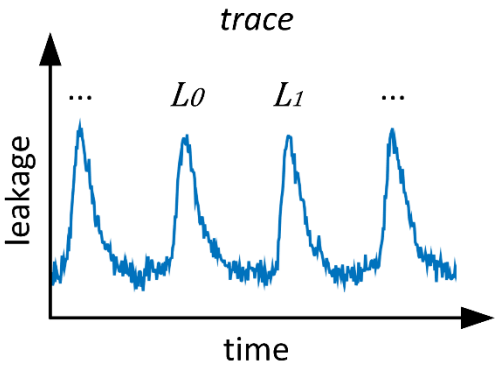
Side-Channel Analysis



Side-Channel Analysis



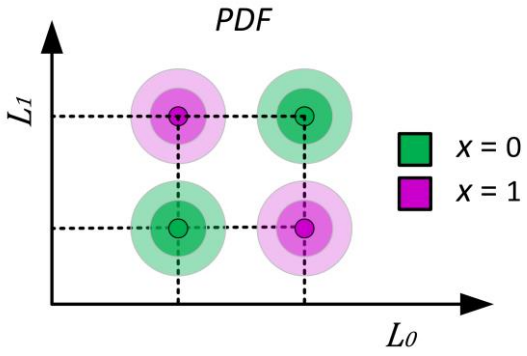
Masking (e.g., Boolean $x = x_0 + x_1 + \cdots + x_d$)



Noisy leakages security: $N \propto \frac{c}{\mathrm{MI}(X;\boldsymbol{L})}$

Goal (ideally): $\mathrm{MI}(X;\boldsymbol{L}) < \mathrm{MI}(X_i;L_i)^d$

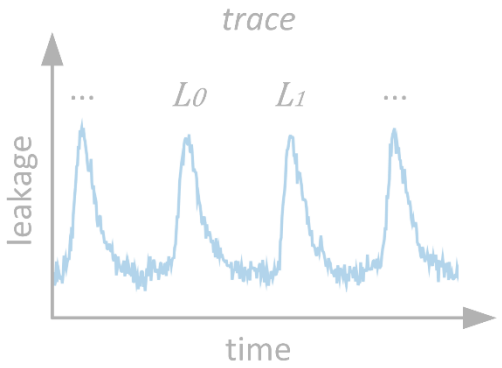
Masking (e.g., Boolean $x = x_0 + x_1 + \cdots + x_d$)



Bounded moment security:

$$\prod_{i_1, i_2, \ldots, i_{d-1}} L_i \perp\!\!\!\perp X$$

(d-1)th order statistical moment (ideally)

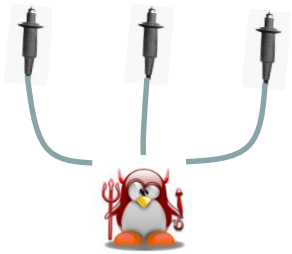


Noisy leakages security: $N \propto \frac{c}{\mathrm{MI}(X;\boldsymbol{L})}$

Goal (ideally): $\mathrm{MI}(X; \boldsymbol{L}) < \mathrm{MI}(X_i; L_i)^d$

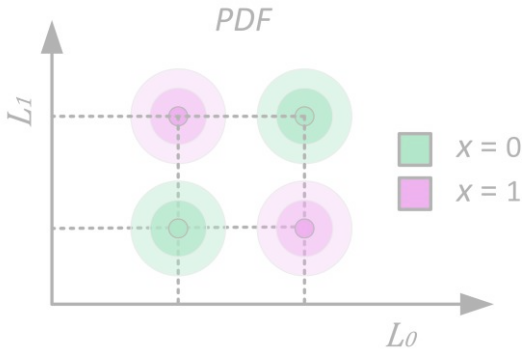
Masking (e.g., Boolean $x = x_0 + x_1 + \cdots + x_d$)

$x = x_0 + x_1 + \cdots + x_d$



Probing security:

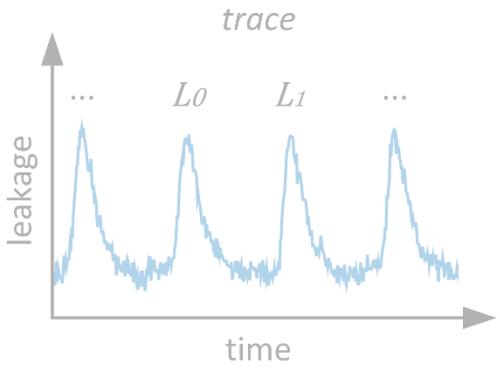
Sets of (d-1) probes are $\perp\!\!\!\perp$ of X (ideally)



Bounded moment security:

$$\prod_{i_1,i_2,\ldots,i_{d-1}} L_i \quad \perp\!\!\!\perp \quad X$$

(d-1)th order statistical moment (ideally)

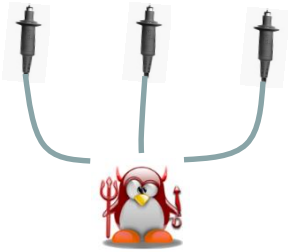


Noisy leakages security: $N \propto \frac{c}{\mathrm{MI}(X;\boldsymbol{L})}$

Goal (ideally): $\mathrm{MI}(X;\boldsymbol{L}) < \mathrm{MI}(X_i;L_i)^d$

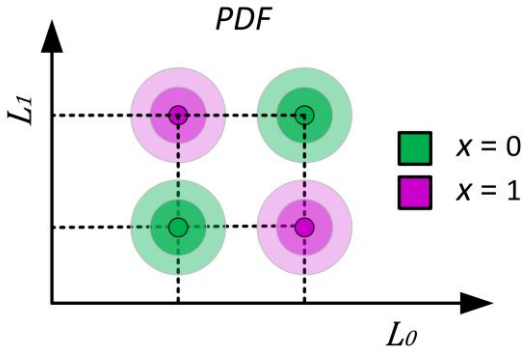
Security reductions

$$x = x_0 + x_1 + \cdots + x_d$$



probing
abstract-qualitative

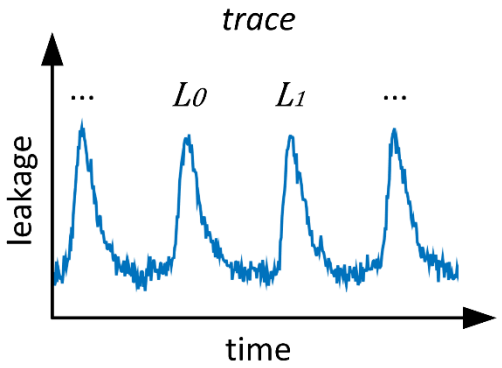
[Barthe et al., Eurocrypt 2017]



bounded moment
physical-qualitative

independence assumption

[Duc et al., Eurocrypt 2014]



noisy leakages
physical-quantitative

independence & noise assumptions

What can go wrong? (e.g., when computing $a.b$)

Issue #1. Lack of randomness (can break the independence assumption)

$$\begin{pmatrix} a_1b_1 & a_1b_2 & a_1b_3 \\ a_2b_1 & a_2b_2 & a_2b_3 \\ a_3b_1 & a_3b_2 & a_3b_3 \end{pmatrix} \Rightarrow \begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix}$$

Example: probing $c_1 = a_1.(b_1 + b_2 + b_3)$ reveals information on b (when $c_1 = 1$)

What can go wrong? (e.g., when computing $a.b$)

Issue #1. Lack of randomness (can break the independence assumption)

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

- mitigated by adding «refreshing gadgets »
- can be analyzed in the probing model

What can go wrong? (e.g., when computing $a.b$)

Issue #1. Lack of randomness (can break the independence assumption)

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

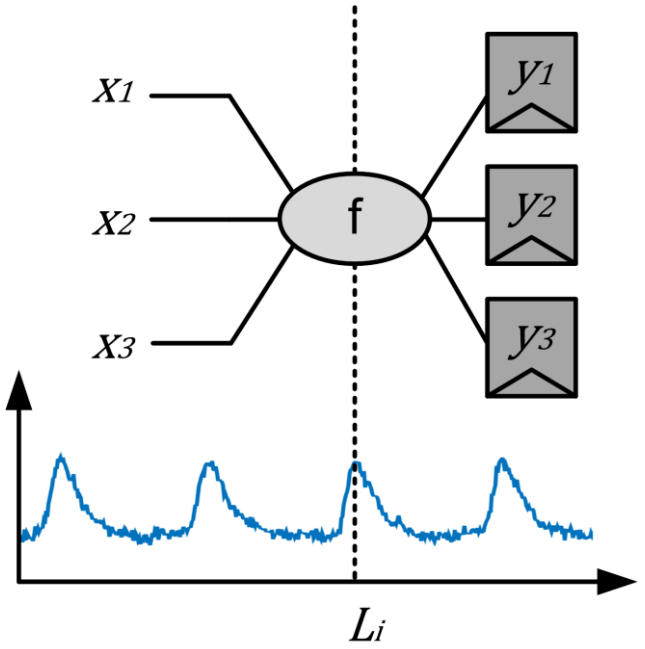
- mitigated by adding «refreshing gadgets »
- can be analyzed in the probing model

Issue #2. Physical defaults
(can break the independence assumption)

Example: glitches (transcient values)
« re-combine » the shares such that:

$$L_i = \delta(x_1 \cdot x_2 \cdot x_3)$$

(detected in the bounded moment model)



What can go wrong? (e.g., when computing $a.b$)

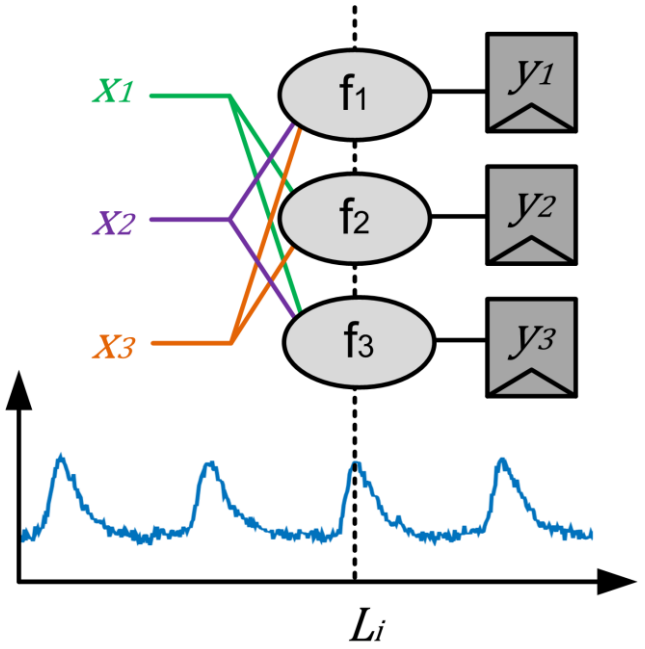
Issue #1. Lack of randomness (can break the independence assumption)

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

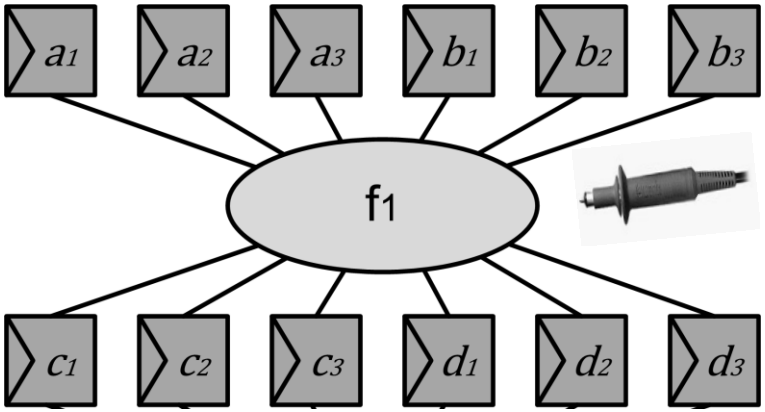
- mitigated by adding «refreshing gadgets »
- can be analyzed in the probing model

Issue #2. Physical defaults
(can break the independence assumption)

- mitigated by adding a « non-completeness » property [≈ Theshold Implementations]
- *abstract property: can be analyzed in the probing model!*

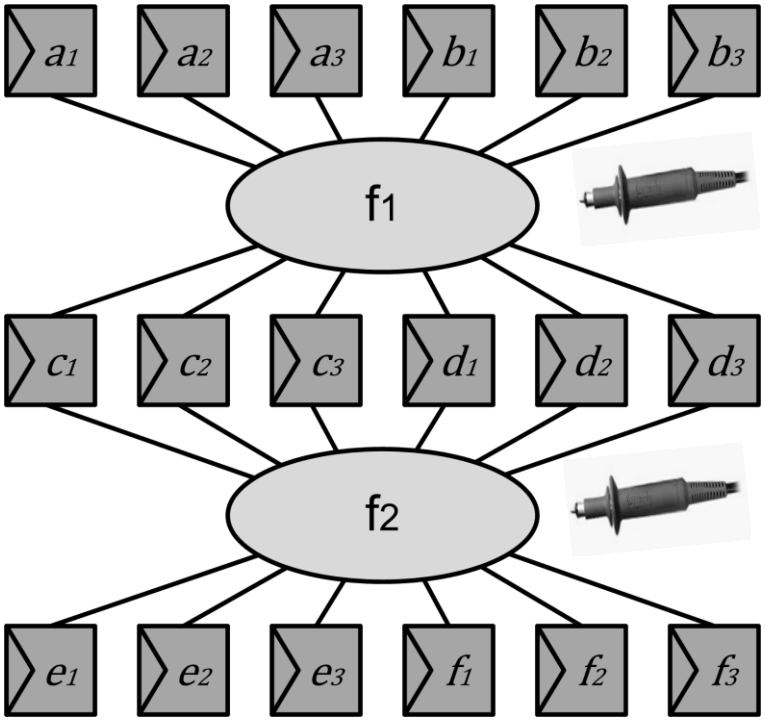


Technical challenge: scalability



***q*-probing security** [ISW, 2004]**:** any q-tuple of shares in the protected circuit is independent of any sensitive variable

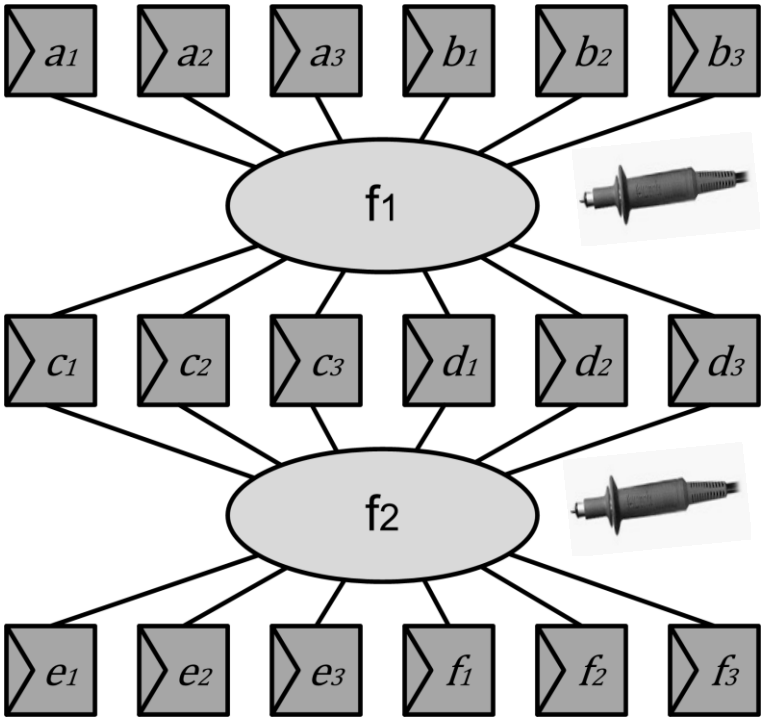
Technical challenge: scalability



***q*-probing security** [ISW, 2004]**:** any q-tuple of shares in the protected circuit is independent of any sensitive variable

Problem: the cost of testing probing security increases (very) fast with circuit size and the # of shares (since ∃ many tuples)

Technical challenge: scalability

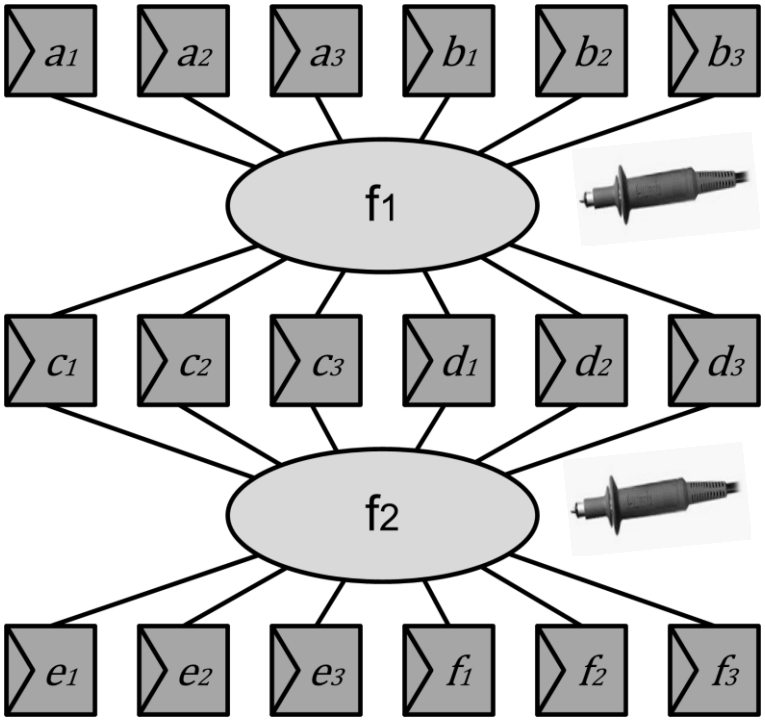


***q*-probing security** [ISW, 2004]**:** any q-tuple of shares in the protected circuit is independent of any sensitive variable

Problem: the cost of testing probing security increases (very) fast with circuit size and the # of shares (since ∃ many tuples)

- Solution #1: *direct verification* of (weaker) circuit properties
  - [Barthe et al., 2015/2019], [Bloem et al., 2018]
- Solution #2: *composable verification* with (stronger) properties
  - [Barthe et al., 2016] – but limited to “abstract” circuits
- Solution #3: test more specific properties [Arribas et al., 2018]

Technical challenge: scalability



***q*-probing security** [ISW, 2004]**:** any q-tuple of shares in the protected circuit is independent of any sensitive variable

Problem: the cost of testing probing security increases (very) fast with circuit size and the # of shares (since ∃ many tuples)

- Solution #1: *direct verification* of (weaker) circuit properties
  - [Barthe et al., 2015/2019], [Bloem et al., 2018]
- Solution #2: *composable verification* with (stronger) properties
  - [Barthe et al., 2016] – but limited to "abstract" circuits
- Can be complementary: use #1 for gadgets, #2 for circuits

Does it go wrong (for hardware masking)?

- State-of-the-art hardware-oriented masking schemes
  - Consolidating Masking Scheme (CMS, 2015)
  - Domain-Oriented Masking (DOM, 2016)
  - Unified Masking Approach (UMA, 2017)
  - Generic Low-Latency Masking (GLM, 2018)

Does it go wrong (for hardware masking)?

- State-of-the-art hardware-oriented masking schemes
  - Consolidating Masking Scheme (CMS, 2015)
  - Domain-Oriented Masking (DOM, 2016)
  - Unified Masking Approach (UMA, 2017)
  - Generic Low-Latency Masking (GLM, 2018)

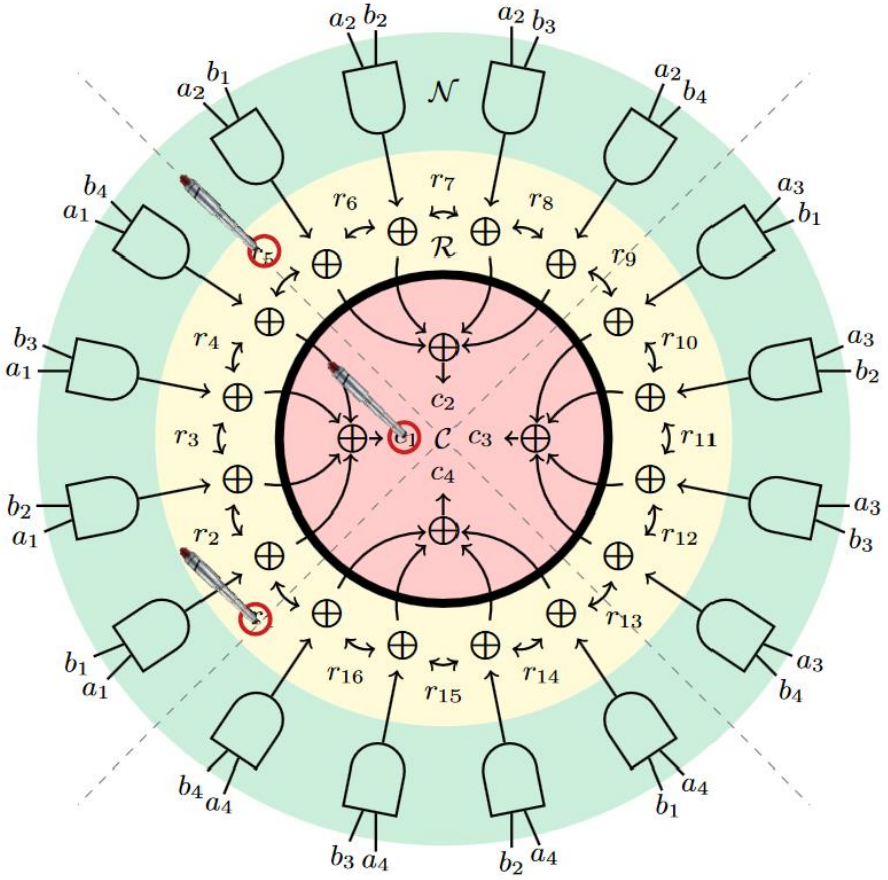
- Intuitively appealing constructions
- But no probing security proof at high orders
  - Theoretical concern or practical risk?

Does it go wrong (for hardware masking)?

- State-of-the-art hardware-oriented masking schemes
  - Consolidating Masking Scheme (CMS, 2015)
  - Domain-Oriented Masking (DOM, 2016)
  - Unified Masking Approach (UMA, 2017)
  - Generic Low-Latency Masking (GLM, 2018)
- Intuitively appealing constructions
- But no probing security proof at high orders
  - Theoretical concern or practical risk?
- [Moos et al., 2019]: all the higher-order extensions of these schemes are affected by concrete flaws
  - Next: CMS (local) and DOM (composability) examples…

Consolidating Masking Scheme

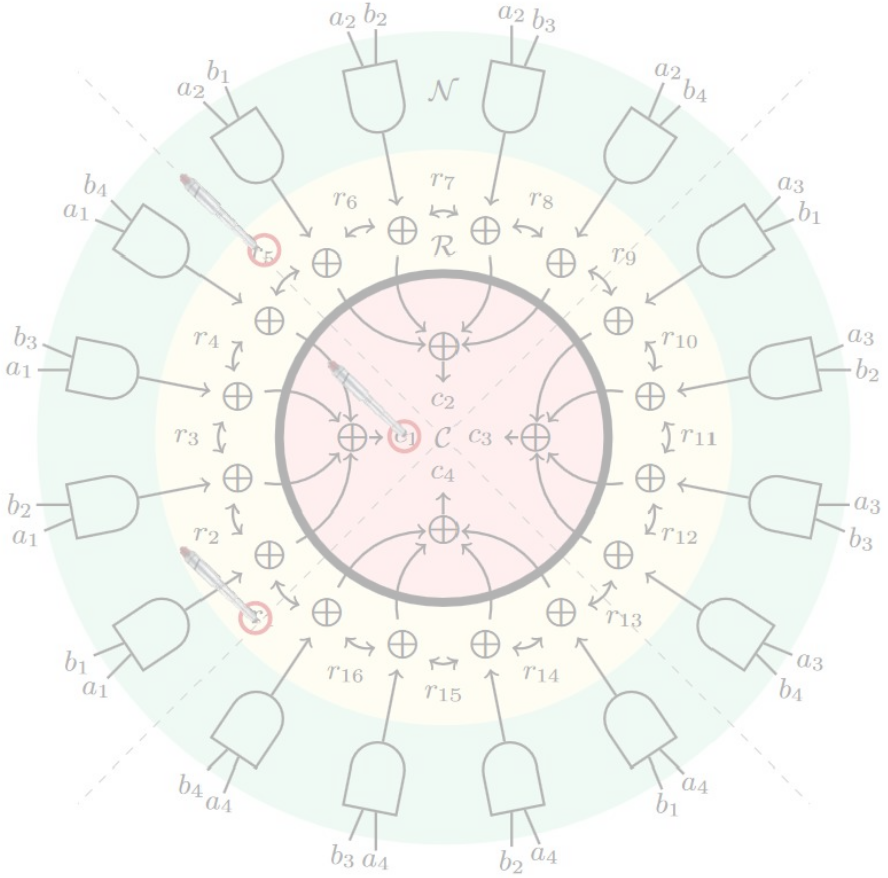
- Local flaw in the “ring refreshing” algorithm
  - Attack with 3 probes for any *d*>3 shares



Problem: most of the randomness cancels out…

Consolidating Masking Scheme

- Local flaw in the “ring refreshing” algorithm
  - Attack with 3 probes for any *d*>3 shares

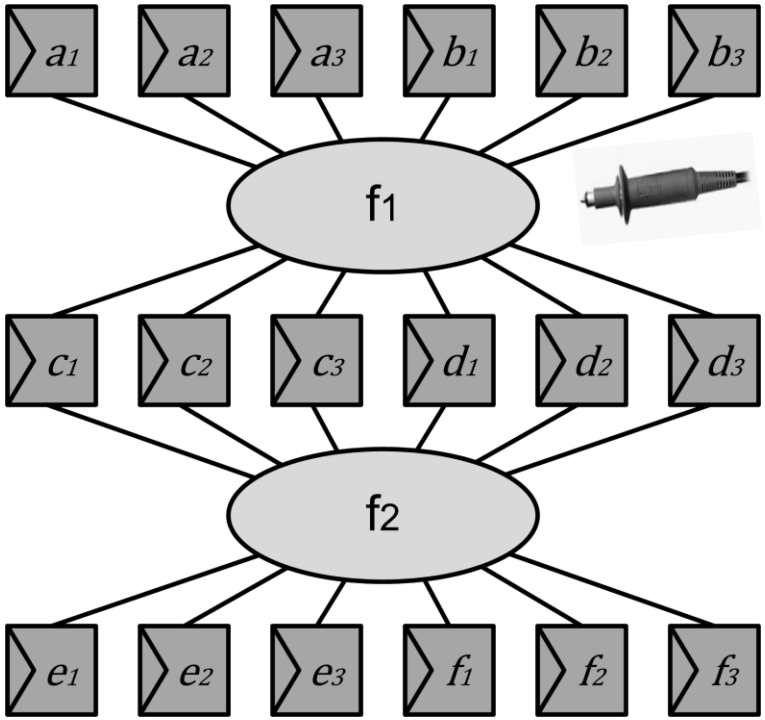


Problem: most of the randomness cancels out…

Fix proposed by De Cnudde
(⇒ CMS more similar to DOM)

Composability remains unclear

Composability requirements (example)



$q_1 + q_2 \leq q$

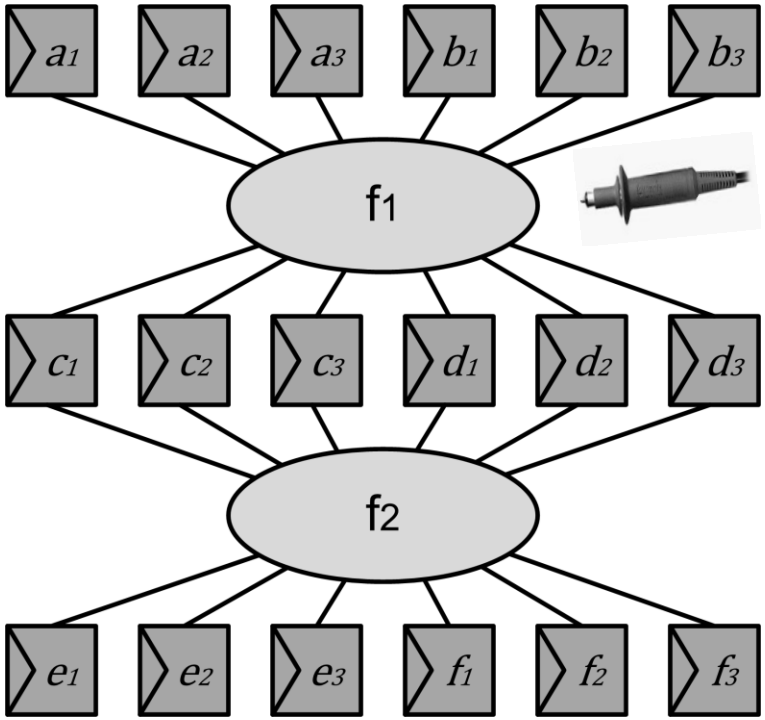
q_1 internal probes

q_2 output probes

***q*-(Strong) Non Interference** [Barthe et al., CCS 2016]**:** a circuit gadget (e.g., f1) is NI (SNI) any set of $q_1 + q_2$ probes can be simulated with at most $q_1 + q_2$ (only q_1) shares of each input

D(input shares||probes) ≈ D(input shares||simulation)

Composability requirements (example)



$q_1 + q_2 \leq q$

q_1 internal probes

q_2 output probes

Theorem [trivial composition]
≈ any composition of
q-SNI gadget is q-SNI

***q*-(Strong) Non Interference** [Barthe et al., CCS 2016]**:** a circuit gadget (e.g., f1) is NI (SNI) any set of $q_1 + q_2$ probes can be simulated with at most $q_1 + q_2$ (only q_1) shares of each input

D(input shares||probes) ≈ D(input shares||simulation)

Domain Oriented Masking

- Two algorithms: DOM-indep and DOM-dep
  - DOM-indep not sufficient to compose, e.g., $z=x \otimes x$

$$\begin{pmatrix} z_0 \\ z_1 \\ z_2 \end{pmatrix} = \begin{pmatrix} x_0 \otimes x_0 & \oplus & (x_0 \otimes x_1 \oplus r_0) & \oplus & (x_0 \otimes x_2 \oplus r_1) \\ (x_1 \otimes x_0 \oplus r_0) & \oplus & x_1 \otimes x_1 & \oplus & (x_1 \otimes x_2 \oplus r_2) \\ (x_2 \otimes x_0 \oplus r_1) & \oplus & (x_2 \otimes x_1 \oplus r_2) & \oplus & x_2 \otimes x_2 \end{pmatrix}$$

Domain Oriented Masking

- Two algorithms: DOM-indep and DOM-dep
  - DOM-indep not sufficient to compose, e.g., $z=x \otimes x$

$$\begin{pmatrix} z_0 \\ z_1 \\ z_2 \end{pmatrix} = \begin{pmatrix} x_0 \otimes x_0 & \oplus & (x_0 \otimes x_1 \oplus r_0) & \oplus & (x_0 \otimes x_2 \oplus r_1) \\ (x_1 \otimes x_0 \oplus r_0) & \oplus & x_1 \otimes x_1 & \oplus & (x_1 \otimes x_2 \oplus r_2) \\ (x_2 \otimes x_0 \oplus r_1) & \oplus & (x_2 \otimes x_1 \oplus r_2) & \oplus & x_2 \otimes x_2 \end{pmatrix}$$

⇒ DOM-dep critical to compose but broken (& no fix)

Domain Oriented Masking

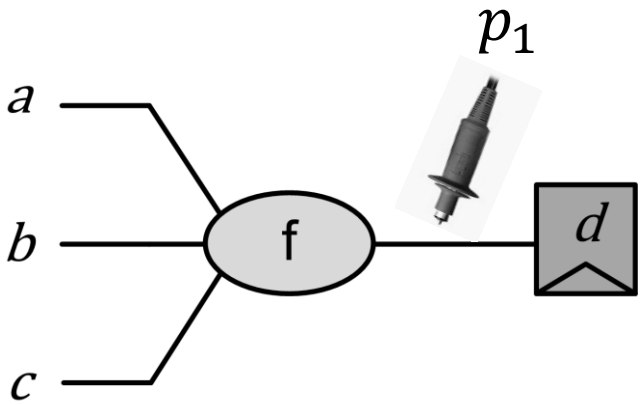
- Two algorithms: DOM-indep and DOM-dep
  - DOM-indep not sufficient to compose, e.g., $z=x \otimes x$

$$\begin{pmatrix} z_0 \\ z_1 \\ z_2 \end{pmatrix} = \begin{pmatrix} x_0 \otimes x_0 & \oplus & (x_0 \otimes x_1 \oplus r_0) & \oplus & (x_0 \otimes x_2 \oplus r_1) \\ (x_1 \otimes x_0 \oplus r_0) & \oplus & x_1 \otimes x_1 & \oplus & (x_1 \otimes x_2 \oplus r_2) \\ (x_2 \otimes x_0 \oplus r_1) & \oplus & (x_2 \otimes x_1 \oplus r_2) & \oplus & x_2 \otimes x_2 \end{pmatrix}$$

  $\Rightarrow$ DOM-dep critical to compose but broken (& no fix)

- SOTA (2018): $\exists$ composable masking schemes that ignore physical defaults such as glitches & hardware-oriented masking schemes that mitigate glitches but are at best probing secure (so not provably composable)

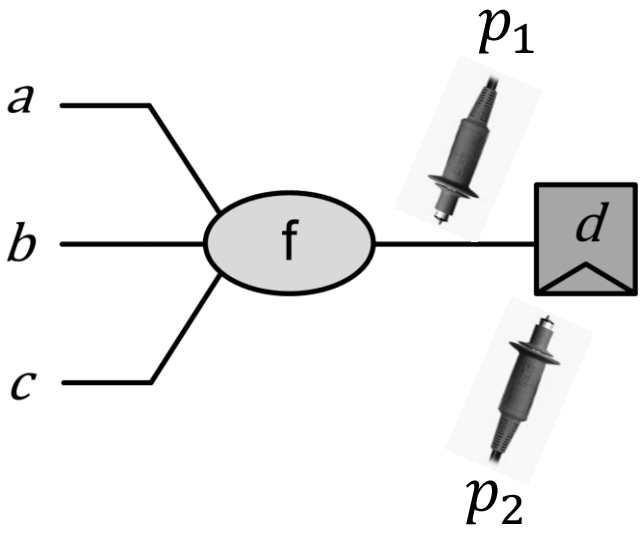
(Refined) model and security definition



Glitch-extended probes: probing any output of a combinatorial circuit allows the adversary to observe all the circuit inputs

Example: p_1 gives a, b and c

(Refined) model and security definition

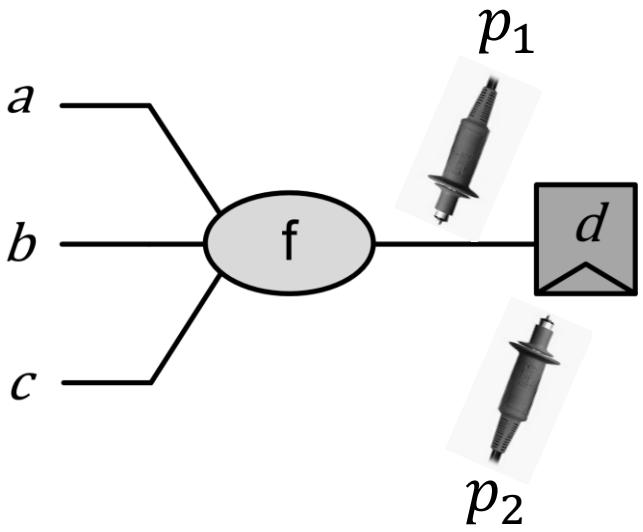


Glitch-extended probes: probing any output of a combinatorial circuit allows the adversary to observe all the circuit inputs

Example: p_1 gives a, b and c

(SNI-related) clarification: the adversary can also probe the stable register output d so both p_1 and p_2 appear in proofs

(Refined) model and security definition



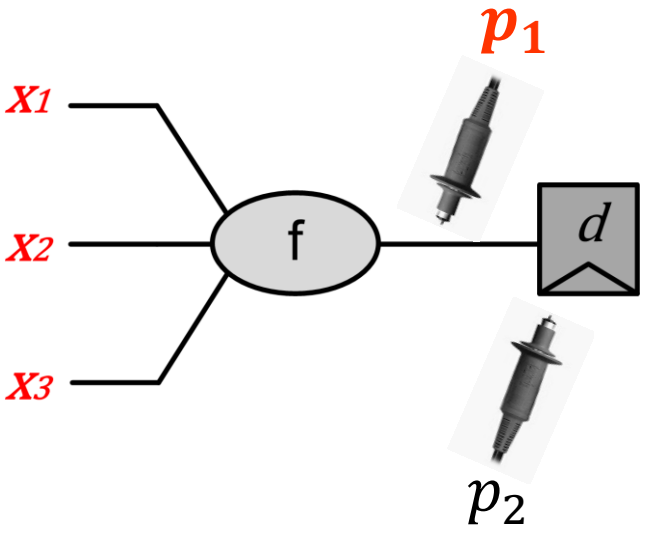
Glitch-extended probes: probing any output of a combinatorial circuit allows the adversary to observe all the circuit inputs

Example: p_1 gives a, b and c

(SNI-related) clarification: the adversary can also probe the stable register output d so both p_1 and p_2 appear in proofs

Definition: a gadget is **glitch-robust q-SNI** if it is q-SNI in the "glitch-extended" probing model

(Refined) model and security definition



Glitch-extended probes: probing any output of a combinatorial circuit allows the adversary to observe all the circuit inputs

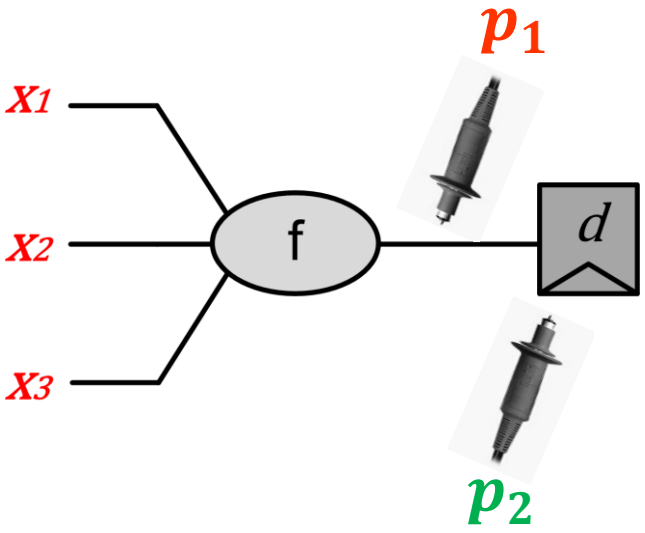
Example: p_1 gives a, b and c

(SNI-related) clarification: the adversary can also probe the stable register output d so both p_1 and p_2 appear in proofs

Definition: a gadget is **glitch-robust q-SNI** if it is q-SNI in the "glitch-extended" probing model

⇒ Shares' fan in of secure gadgets should be minimum

(Refined) model and security definition



Glitch-extended probes: probing any output of a combinatorial circuit allows the adversary to observe all the circuit inputs

Example: p_1 gives a, b and c

(SNI-related) clarification: the adversary can also probe the stable register output d so both p_1 and p_2 appear in proofs

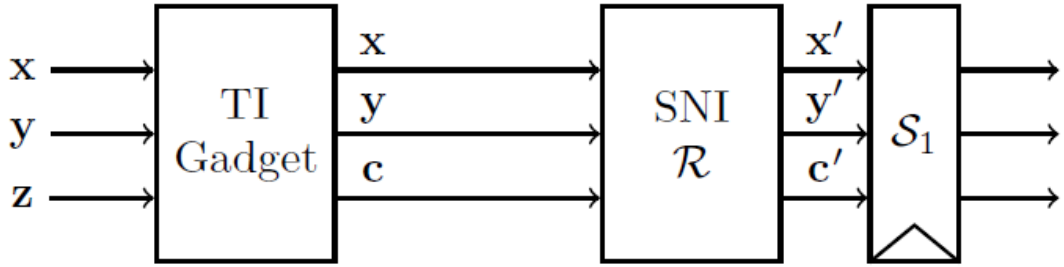
Definition: a gadget is **glitch-robust q-SNI** if it is q-SNI in the "glitch-extended" probing model

⇒ Shares' fan in of secure gadgets should be minimum

⇒ Output probes (excluded in the SNI count) must be stable

Note: the problem must be solved jointly

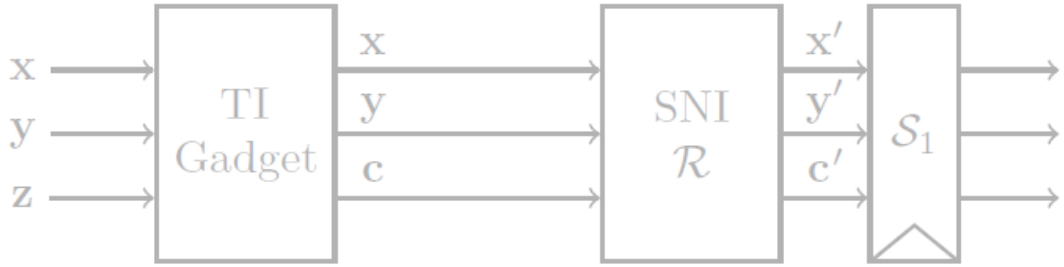
- TI gadget + SNI refresh + register: robust against glitches & composable without glitches (not both)



  - Extended probe on c' reveals all $\mathcal{R}$'s randomness

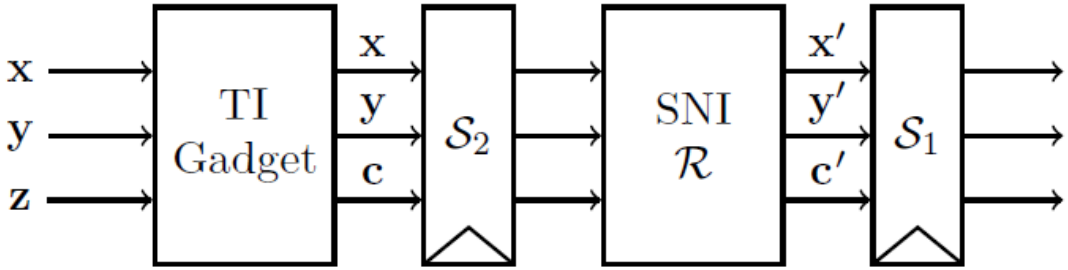
Note: the problem must be solved jointly

- TI gadget + SNI refresh + register: robust against glitches & composable without glitches (not both)

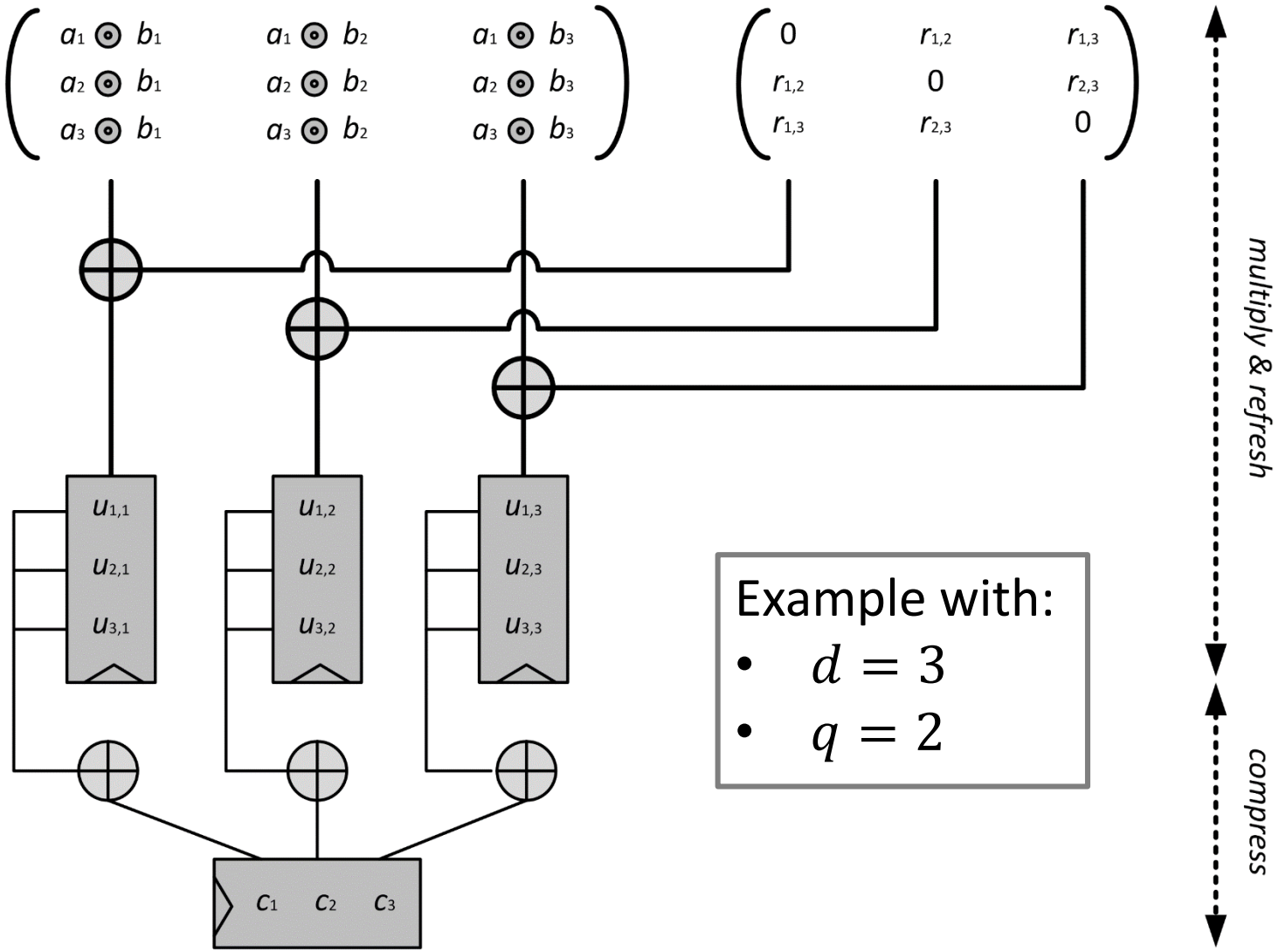


  - Extended probe on c' reveals all $\mathcal{R}$'s randomness

- Adding a register does not help (just probe c)



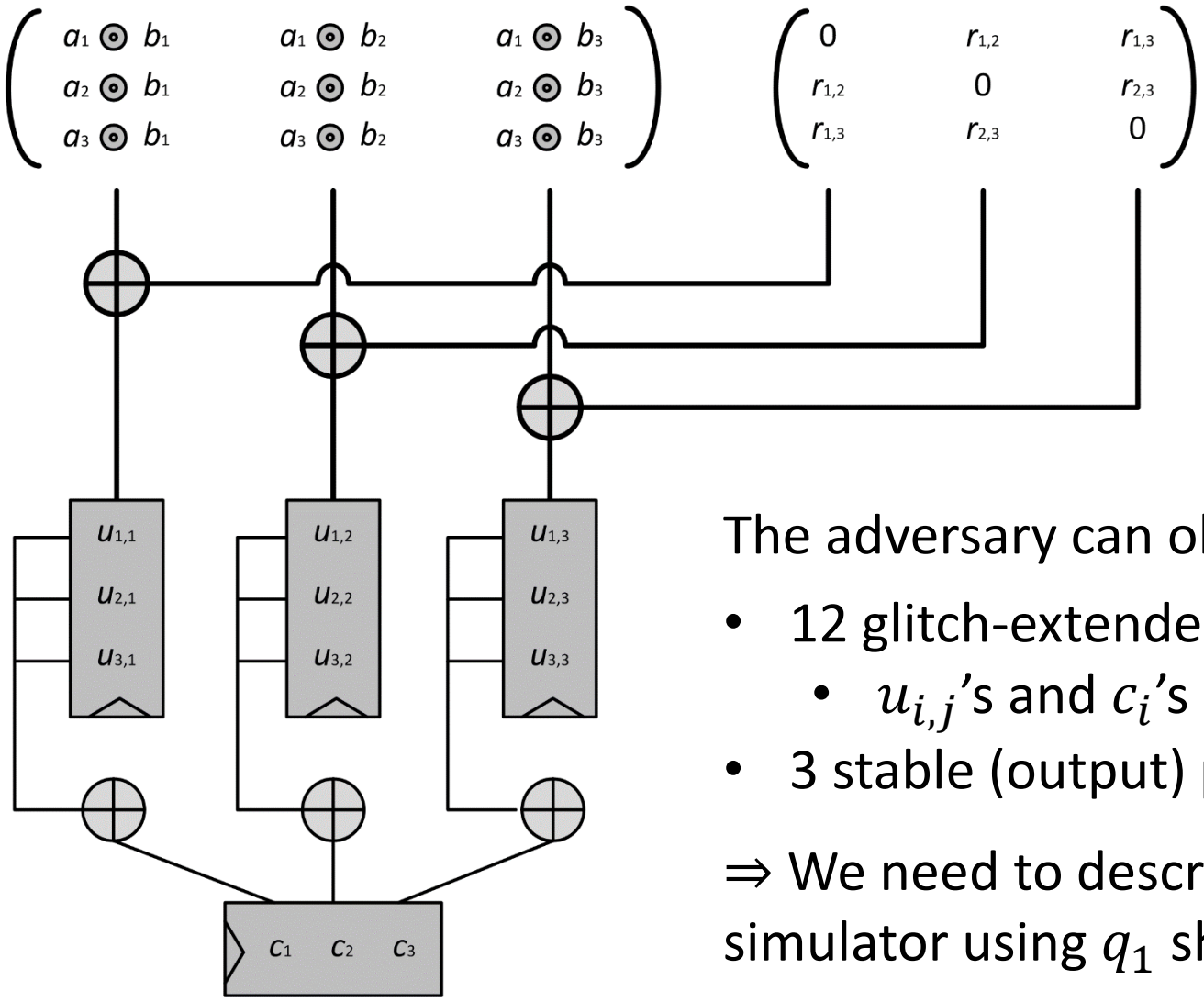
ISW mult. is glitch-robust q-SNI in 2 cycles



Example with:

- $d = 3$
- $q = 2$

ISW mult. is glitch-robust q-SNI in 2 cycles

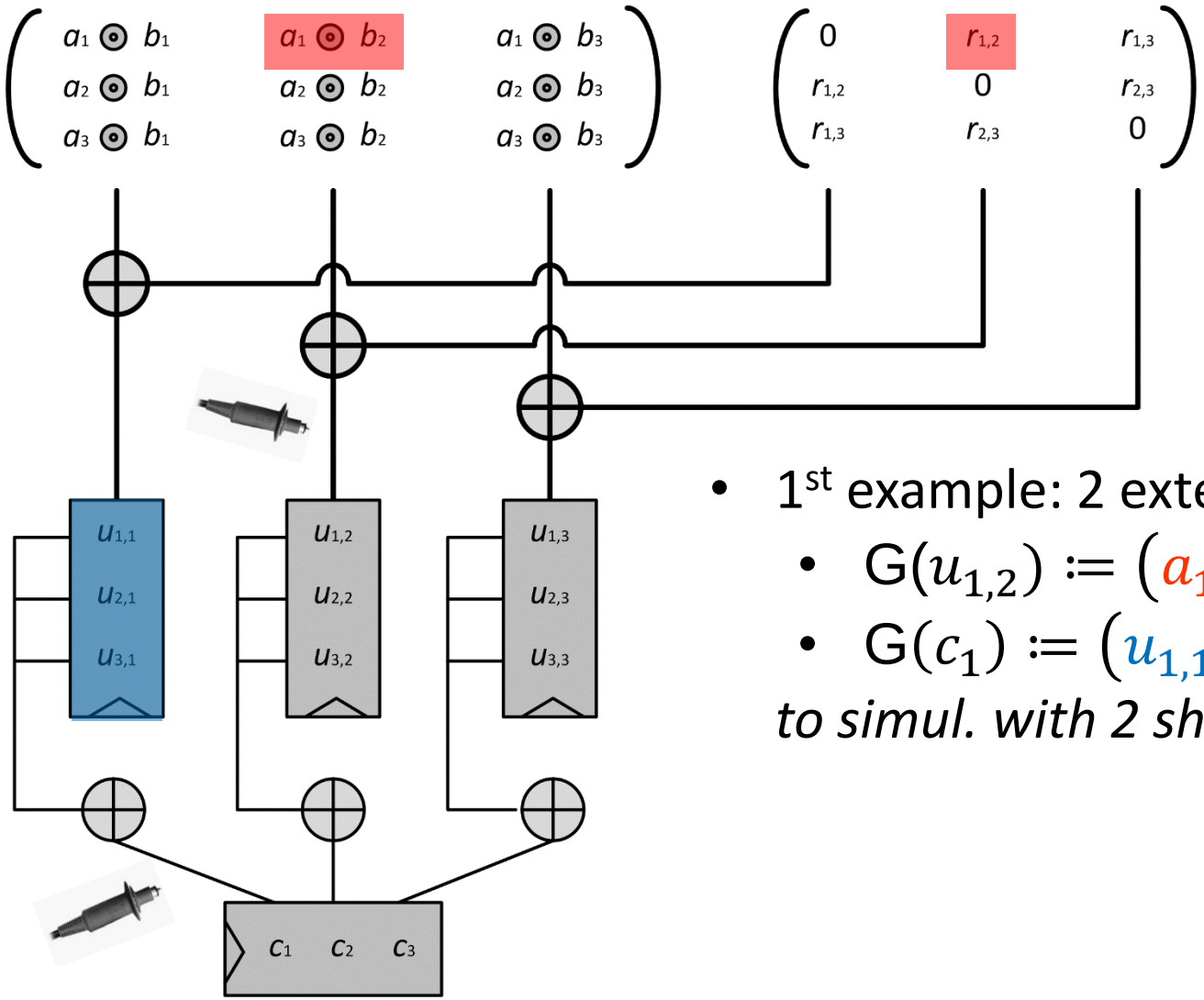


The adversary can observe:

- 12 glitch-extended probes
  - $u_{i,j}$'s and c_i's
- 3 stable (output) probes c_i's

⇒ We need to describe a simulator using q_1 shares/input

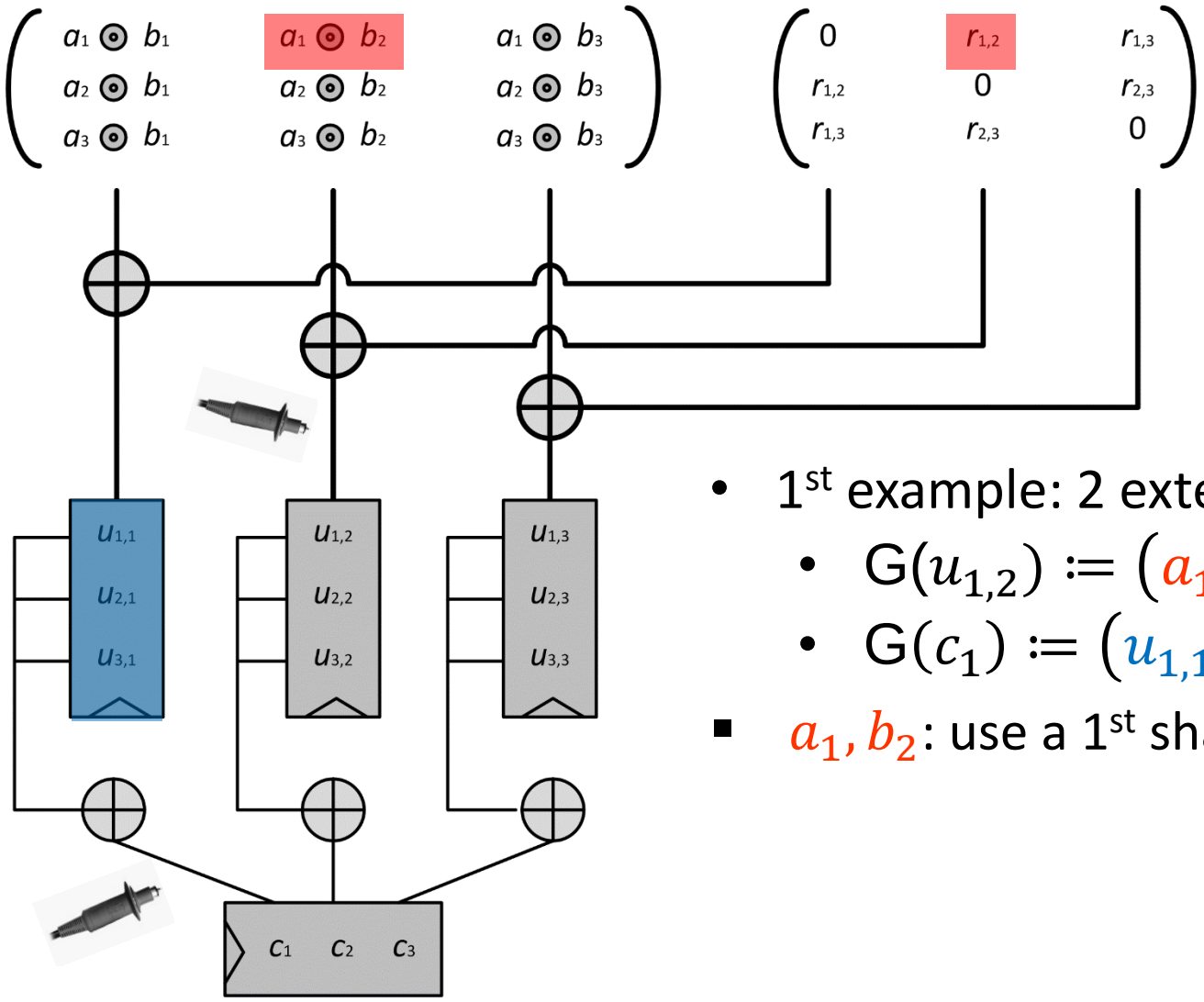
ISW mult. is glitch-robust q-SNI in 2 cycles



- 1st example: 2 extended probes
  - $\mathrm{G}(u_{1,2}) := (a_1, b_2, r_{1,2})$
  - $\mathrm{G}(c_1) := (u_{1,1}, u_{2,1}, u_{3,1})$

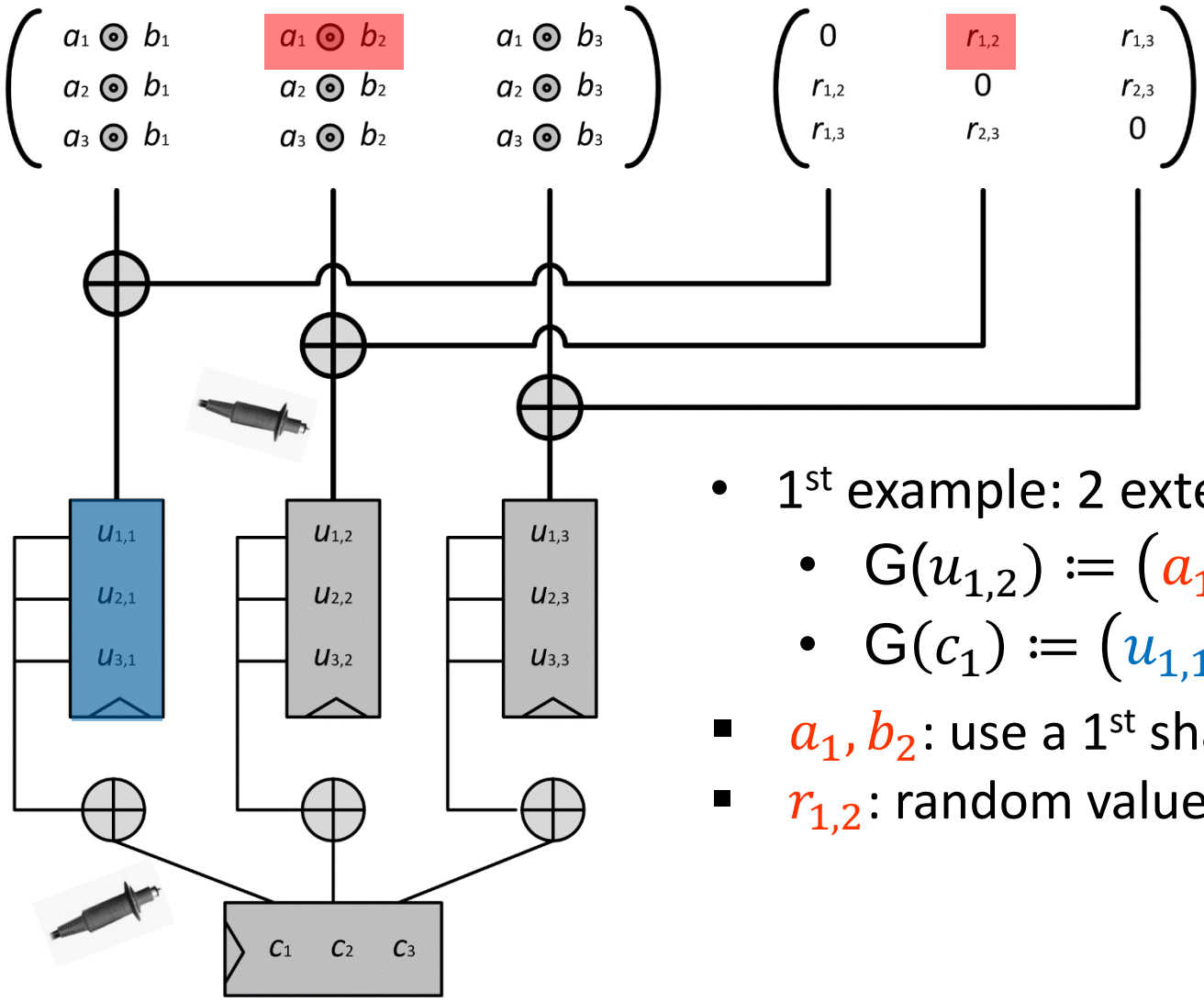
  to simul. with 2 shares/input

ISW mult. is glitch-robust q-SNI in 2 cycles



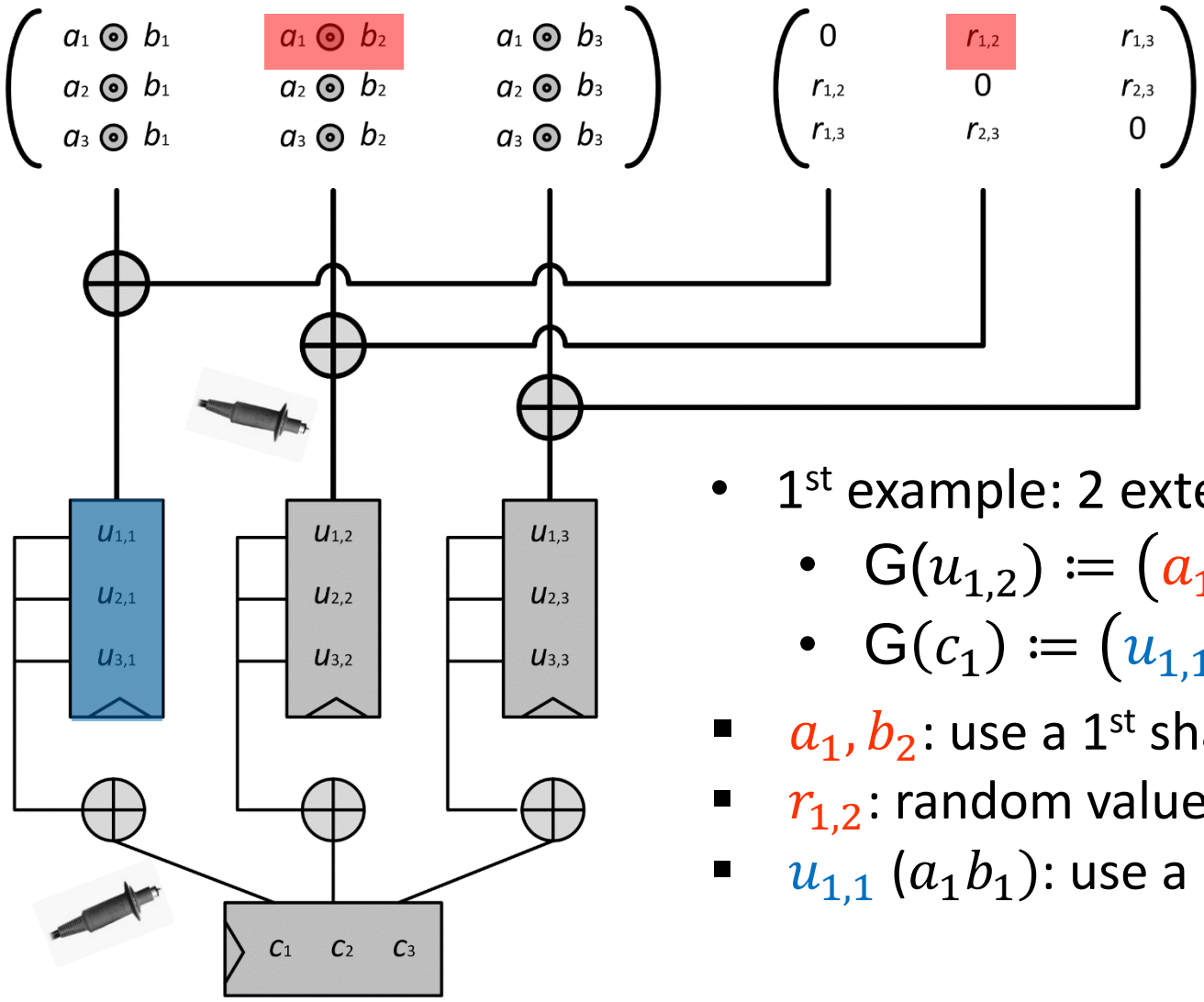
- 1st example: 2 extended probes
  - $\mathrm{G}(u_{1,2}) := (a_1, b_2, r_{1,2})$
  - $\mathrm{G}(c_1) := (u_{1,1}, u_{2,1}, u_{3,1})$
- a_1, b_2: use a 1st share of a, b

ISW mult. is glitch-robust q-SNI in 2 cycles



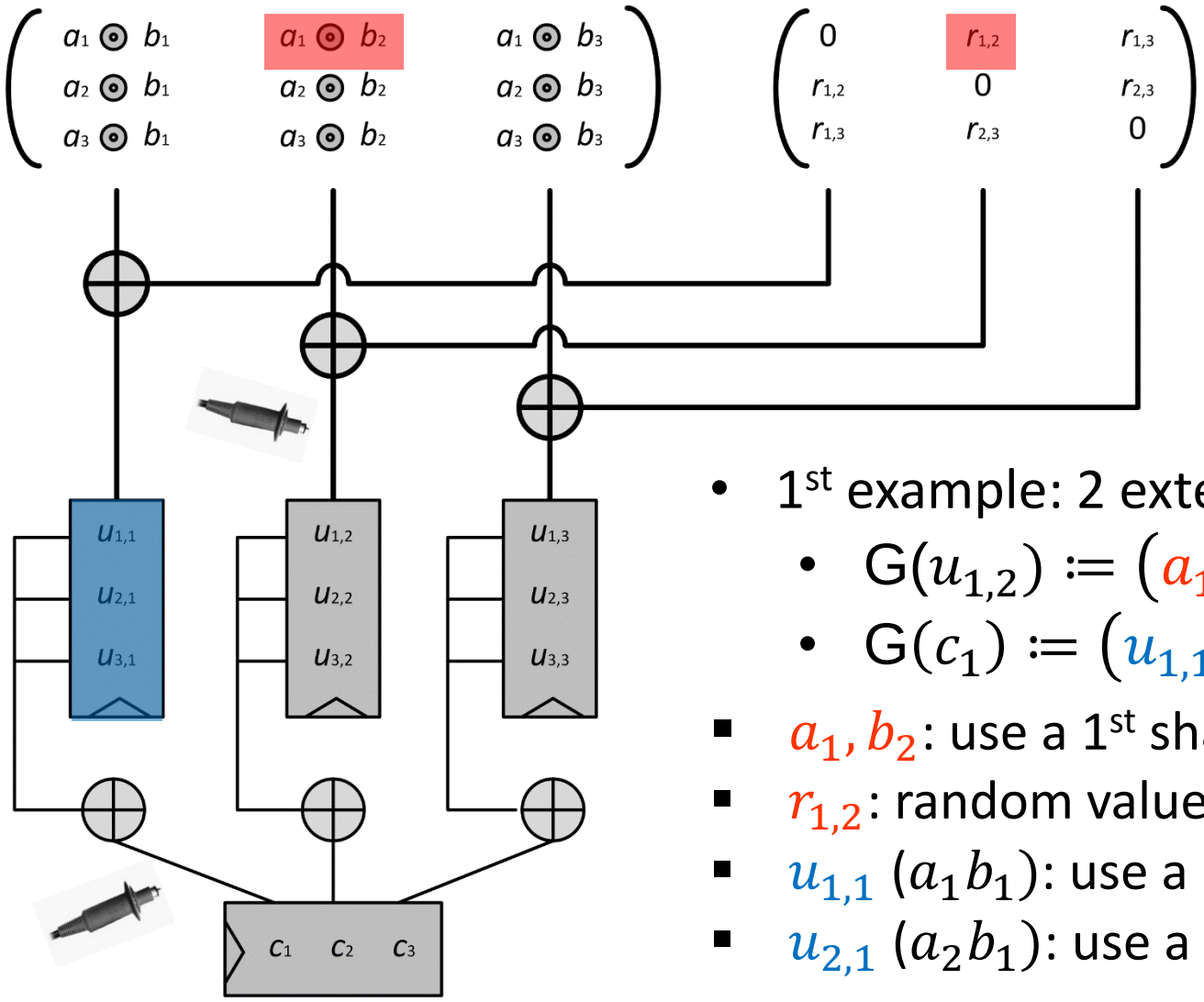
- 1st example: 2 extended probes
  - $\mathrm{G}(u_{1,2}) := (a_1, b_2, r_{1,2})$
  - $\mathrm{G}(c_1) := (u_{1,1}, u_{2,1}, u_{3,1})$
- a_1, b_2: use a 1st share of a, b
- $r_{1,2}$: random value

ISW mult. is glitch-robust q-SNI in 2 cycles



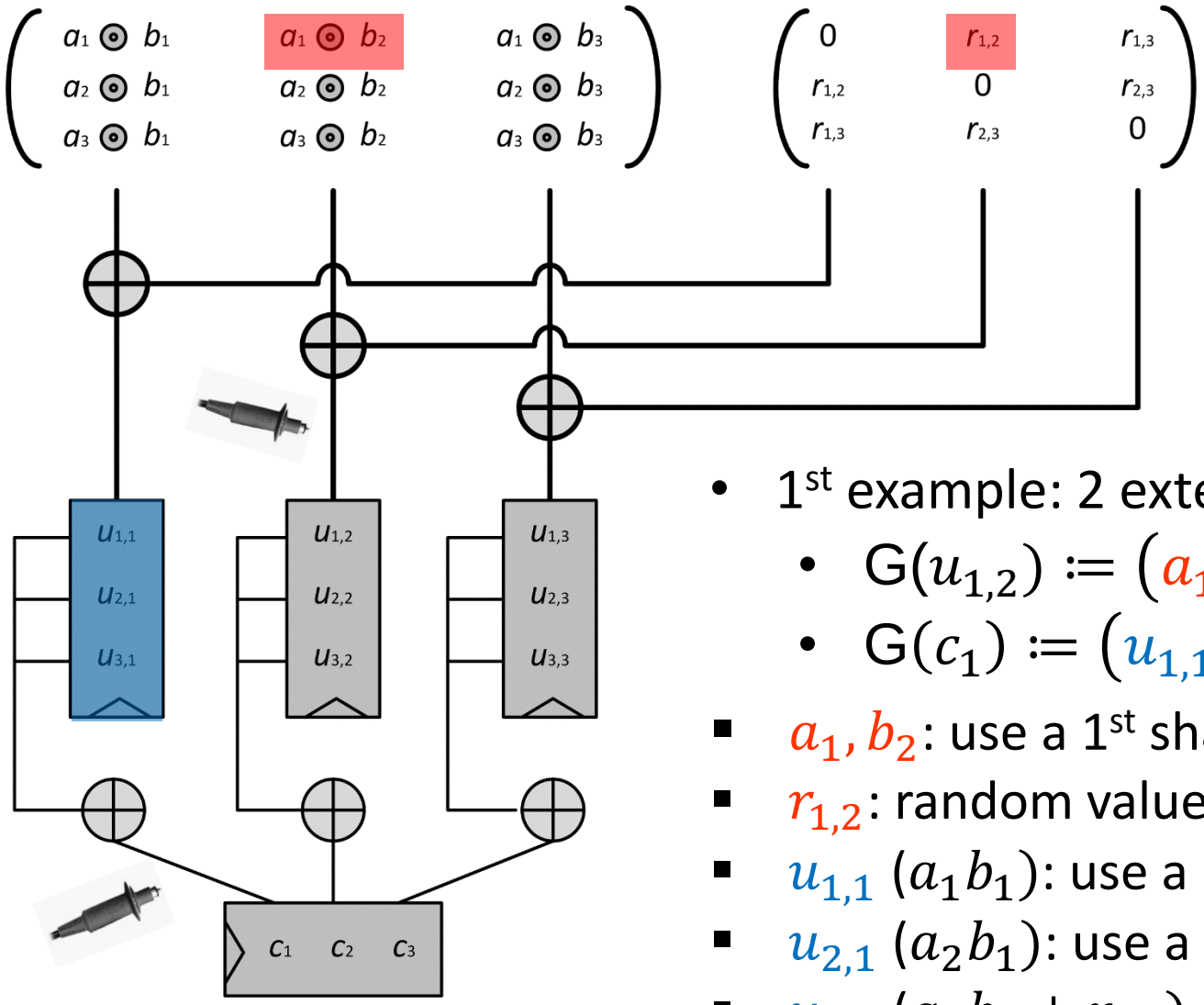
- 1st example: 2 extended probes
  - $\mathrm{G}(u_{1,2}) := (a_1, b_2, r_{1,2})$
  - $\mathrm{G}(c_1) := (u_{1,1}, u_{2,1}, u_{3,1})$
- a_1, b_2: use a 1st share of a, b
- $r_{1,2}$: random value
- $u_{1,1}$ $(a_1 b_1)$: use a 2nd share of b

ISW mult. is glitch-robust q-SNI in 2 cycles



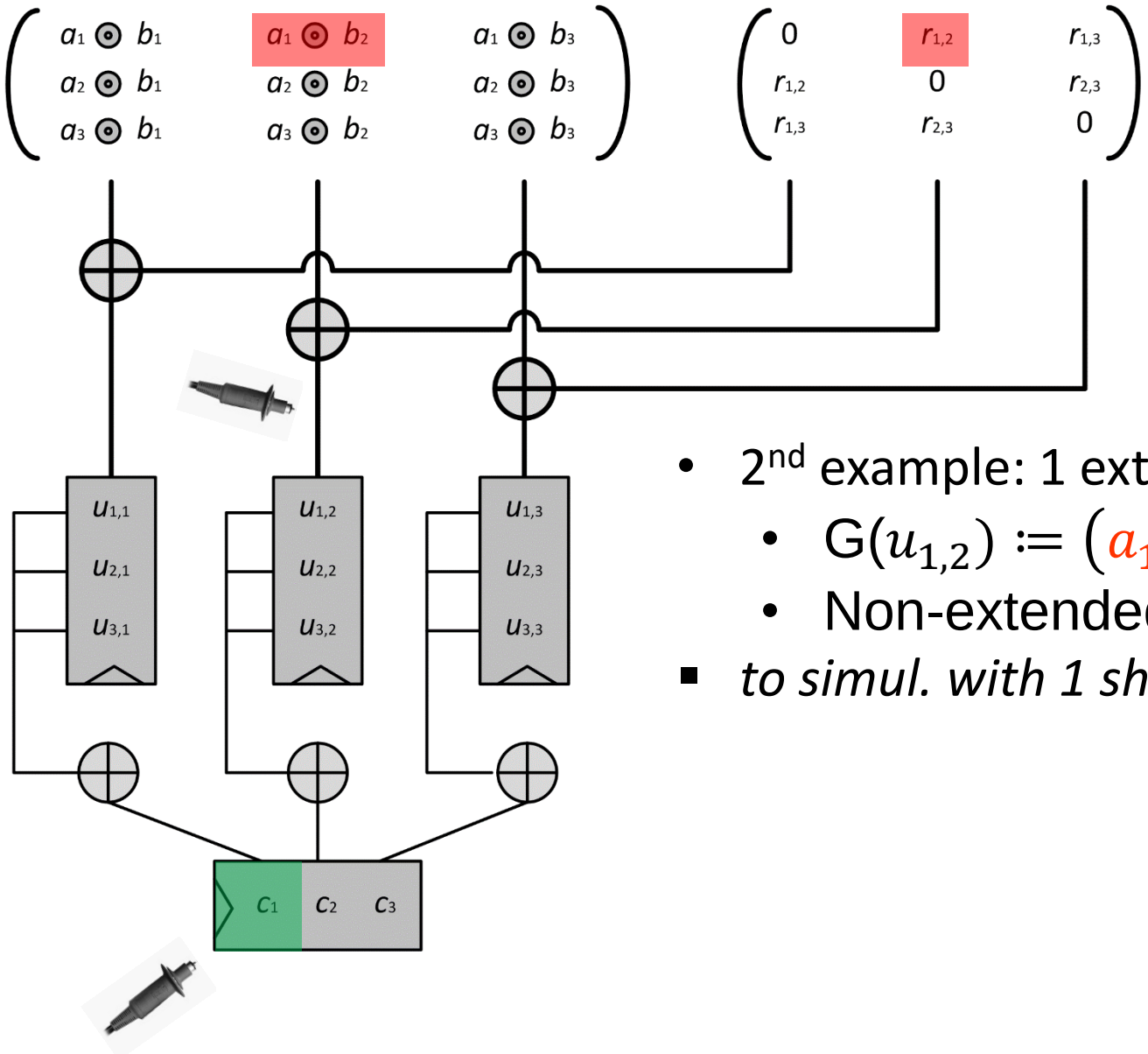
- 1st example: 2 extended probes
  - $G(u_{1,2}) := (a_1, b_2, r_{1,2})$
  - $G(c_1) := (u_{1,1}, u_{2,1}, u_{3,1})$
- a_1, b_2: use a 1st share of a, b
- $r_{1,2}$: random value
- $u_{1,1}$ $(a_1 b_1)$: use a 2nd share of b
- $u_{2,1}$ $(a_2 b_1)$: use a 2nd share of a

ISW mult. is glitch-robust q-SNI in 2 cycles



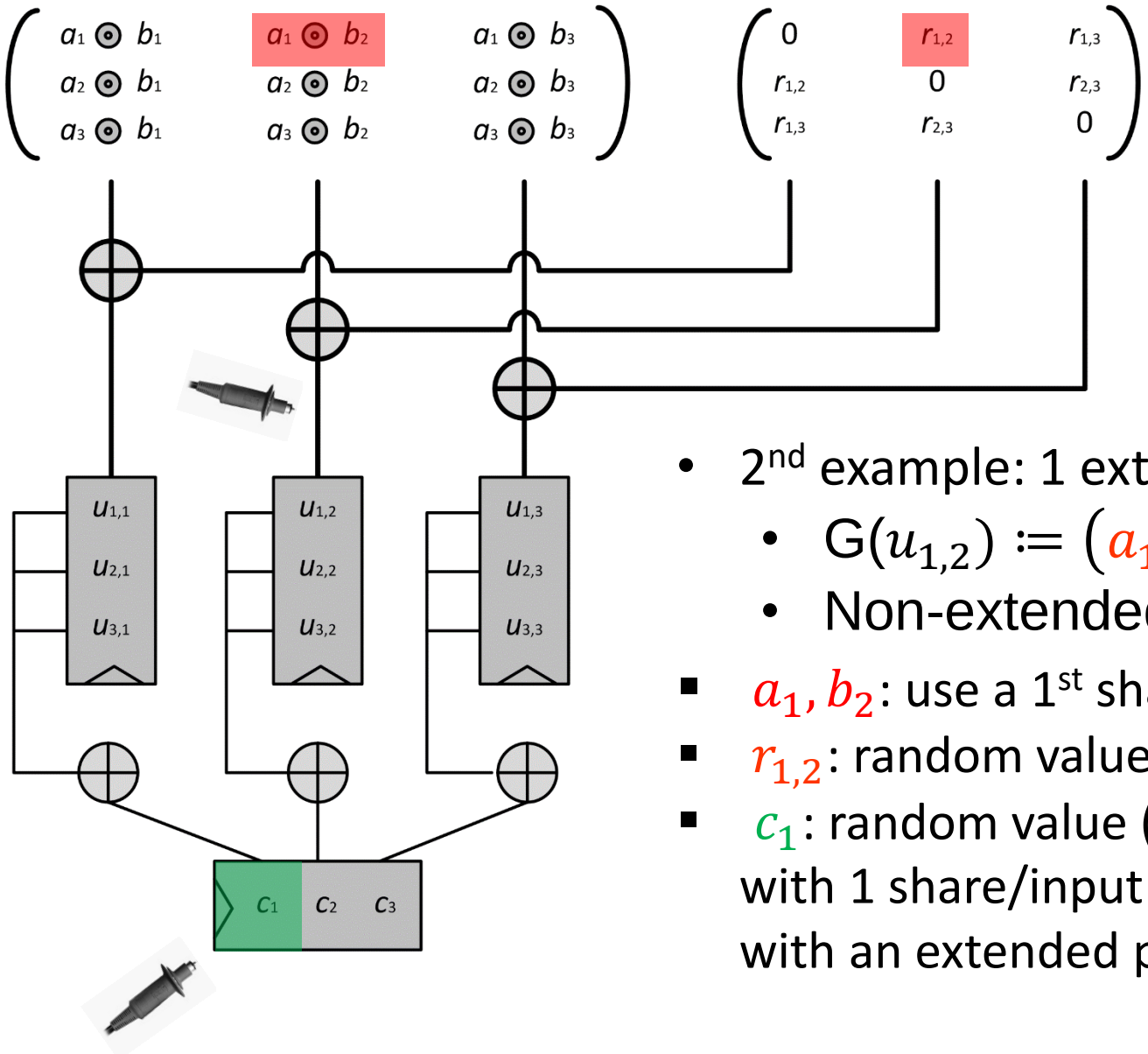
- 1st example: 2 extended probes
  - $\mathrm{G}(u_{1,2}) := (a_1, b_2, r_{1,2})$
  - $\mathrm{G}(c_1) := (u_{1,1}, u_{2,1}, u_{3,1})$
- a_1, b_2: use a 1st share of a, b
- $r_{1,2}$: random value
- $u_{1,1}$ $(a_1 b_1)$: use a 2nd share of b
- $u_{2,1}$ $(a_2 b_1)$: use a 2nd share of a
- $u_{3,1}$ $(a_3 b_1 + r_{1,3})$: random value

ISW mult. is glitch-robust q-SNI in 2 cycles



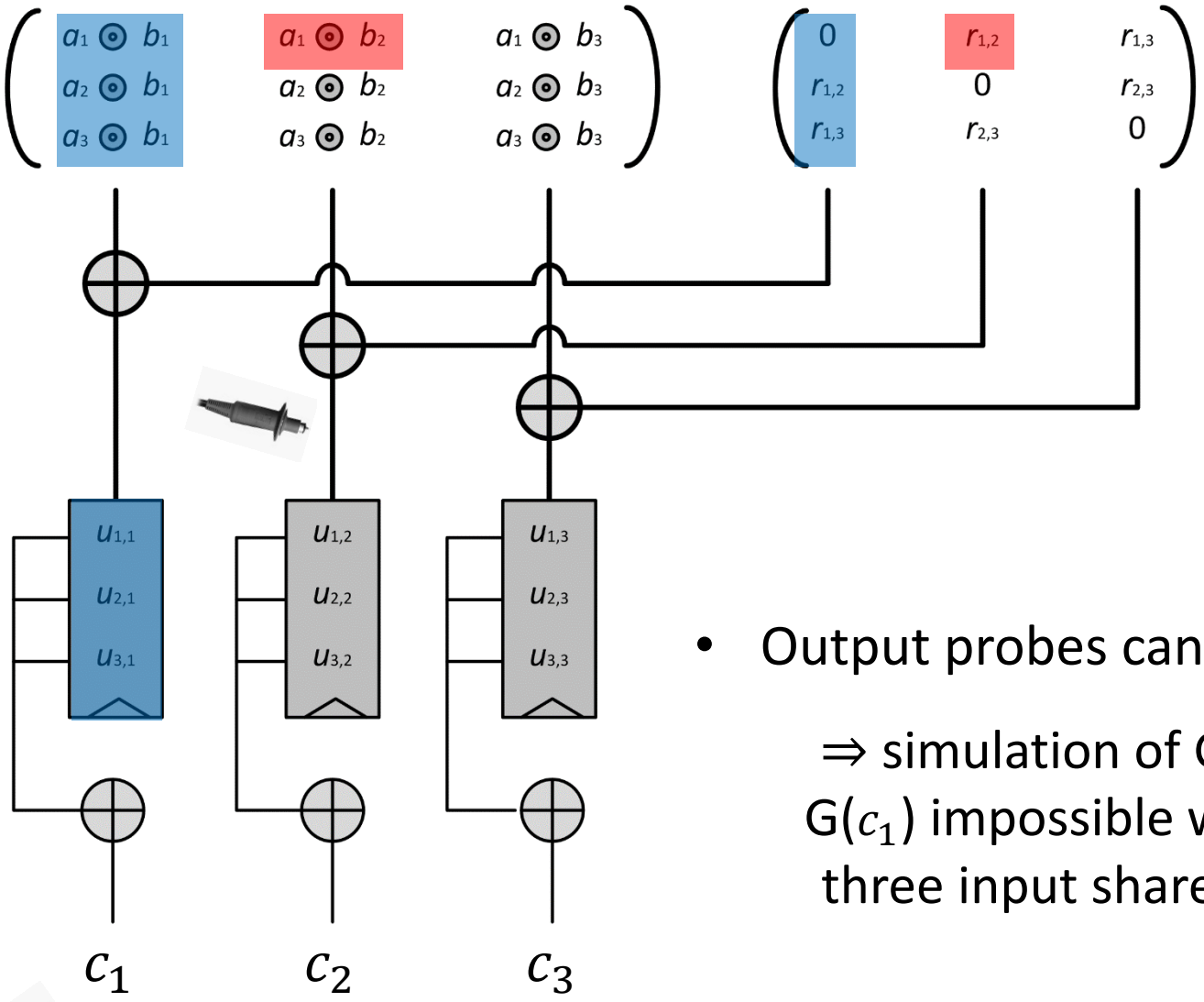
- 2nd example: 1 extended probe
  - G($u_{1,2}$) := ($a_1, b_2, r_{1,2}$)
  - Non-extended c_1
- *to simul. with 1 share/input*

ISW mult. is glitch-robust q-SNI in 2 cycles



- 2nd example: 1 extended probe
  - $G(u_{1,2}) := (a_1, b_2, r_{1,2})$
  - Non-extended c_1
- a_1, b_2: use a 1st share of a, b
- $r_{1,2}$: random value
- c_1: random value (simulation with 1 share/input impossible with an extended probe on c_1)

DOM-indep is glitch-robust q-NI in 1 cycle



- Output probes can be extended

  $\Rightarrow$ simulation of G($u_{1,2}$) and G(c_1) impossible without the three input shares of a & b

Approaches to composition

- [Faust et al., 2018]: glitch-robust SNI mult. in 2 cycles
- DOM-indep: glitch-robust NI mult. in 1 cycle
  - What can we construct based on that?

Approaches to composition

- [Faust et al., 2018]: glitch-robust SNI mult. in 2 cycles
- DOM-indep: glitch-robust NI mult. in 1 cycle
  - What can we construct based on that?

- [Cassiers et al., 2019] proof that any compositional strategy that is correct in the standard probing model remains valid in the robust probing model

Approaches to composition

- [Faust et al., 2018]: glitch-robust SNI mult. in 2 cycles
- DOM-indep: glitch-robust NI mult. in 1 cycle
  - What can we construct based on that?
- [Cassiers et al., 2019] proof that any compositional strategy that is correct in the standard probing model remains valid in the robust probing model

⇒ Both *trivial composition* (e.g., using only SNI gadgets) or *optimized composition* (e.g., combining NI/SNI multiplications with SNI refreshes) can work

≈ tradeoff between verification complexity and performance

Approaches to composition

- [Faust et al., 2018]: glitch-robust SNI mult. in 2 cycles
- DOM-indep: glitch-robust NI mult. in 1 cycle
  - What can we construct based on that?
- [Cassiers et al., 2019] proof that any compositional strategy that is correct in the standard probing model remains valid in the robust probing model

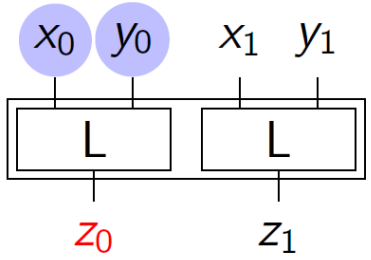
⇒ Both *trivial composition* (e.g., using only SNI gadgets) or *optimized composition* (e.g., combining NI/SNI multiplications with SNI refreshes) can work

≈ tradeoff between verification complexity and performance

- Next: focus on trivial composition (natural first step & instrumental in our tool for full verification)

Improving trivial composition

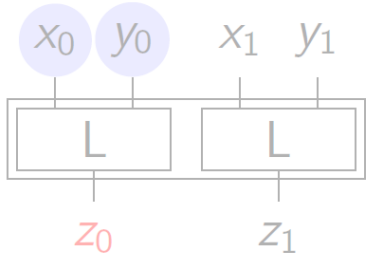
- Linear gadgets enable share isolation



⇒ Informally we expect trivial composition for free

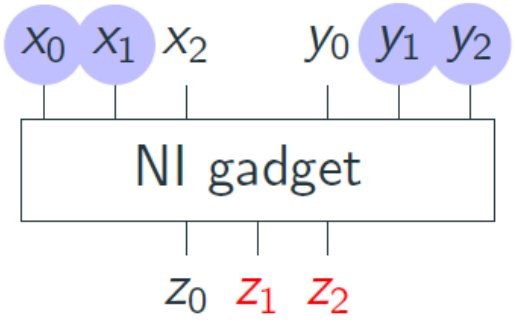
Improving trivial composition

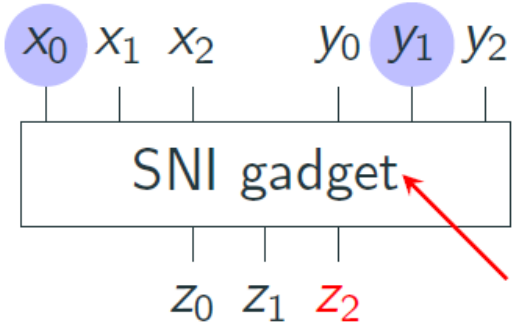
- Linear gadgets enable share isolation



⇒ Informally we expect trivial composition for free

- But “share-by-share” linear gadgets are only NI

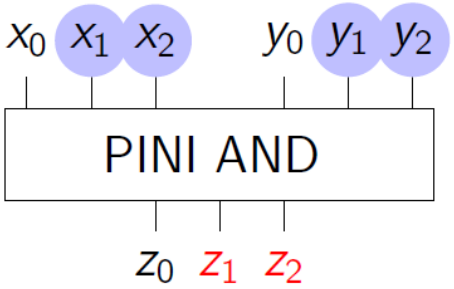




⇒ Trivial SNI composition must refresh linear gadgets

Probe Isolating Non-Interference (PINI)

- [Cassiers & Standaert, 2018]: gadgets should behave (w.r.t. simulatability) as if shares were isolated

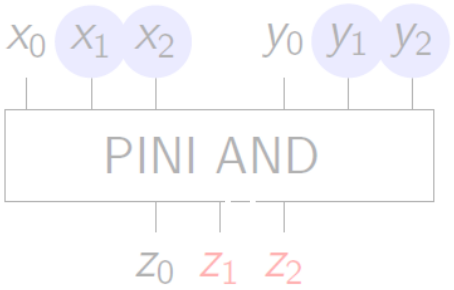


$\Rightarrow$ “share-by-share” linear gadgets are PINI
(formalizes the idea of circuit share in DOM/TIs)

- Theorem: any combination of q-PINI gadgets is q-PINI

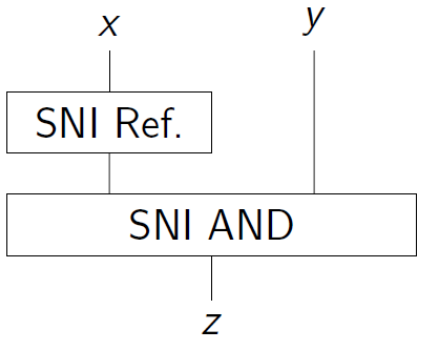
Probe Isolating Non-Interference (PINI)

- [Cassiers & Standaert, 2018]: gadgets should behave (w.r.t. simulatability) as if shares were isolated



⇒ "share-by-share" linear gadgets are PINI (formalizes the idea of circuit share in DOM/TIs)

- Theorem: any combination of q-PINI gadgets is q-PINI
- Used to prove a strategy by [Goudarzi & Rivain, 2017]
- But can lead to much more…



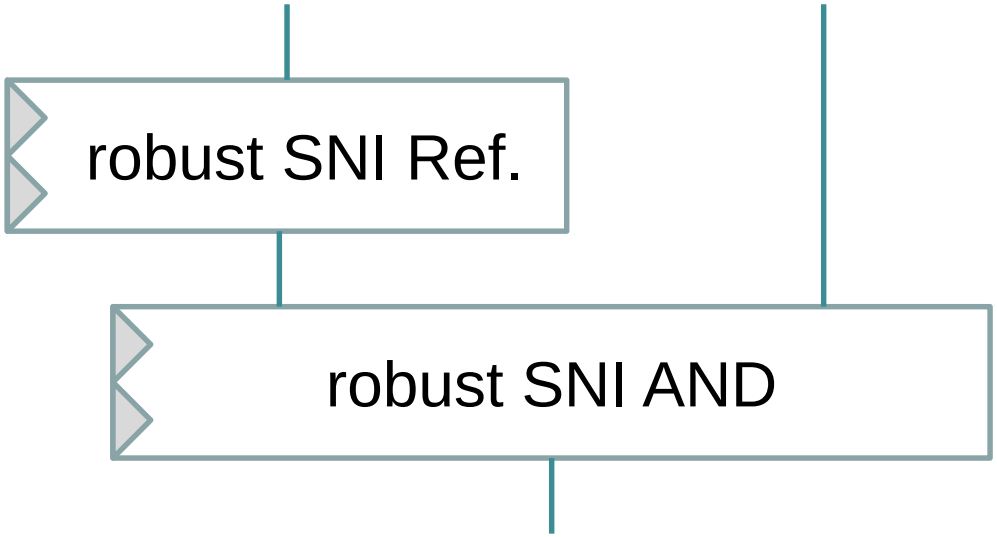
is PINI

Hardware Private Circuits

- (∃ more efficient PINI multiplications in software)

Hardware Private Circuits

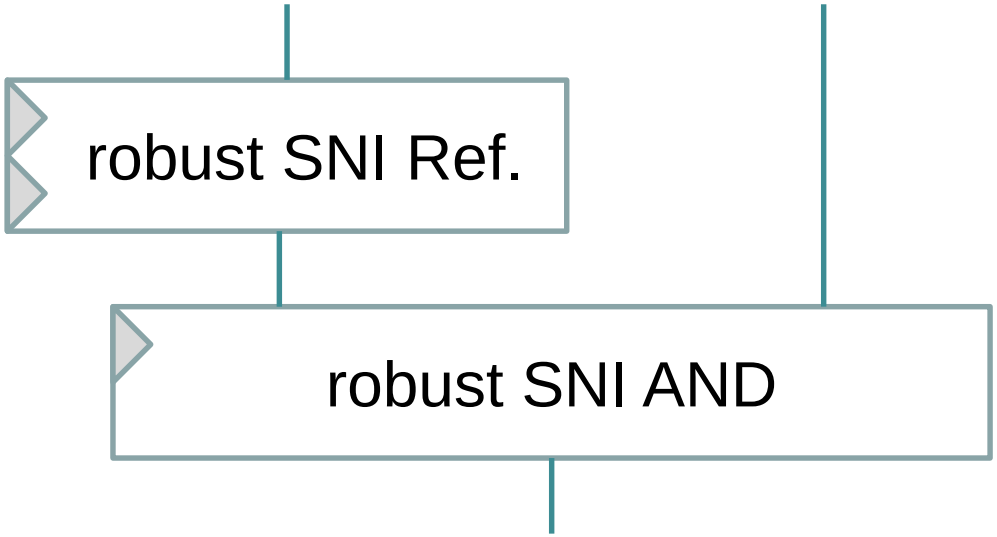
- (∃ more efficient PINI multiplications in software)
- Significantly improves trivial composition in hardware



[Faust et al., 2018]:
SNI-based PINI mult.
in 4 cycles

Hardware Private Circuits

- (∃ more efficient PINI multiplications in software)
- Significantly improves trivial composition in hardware

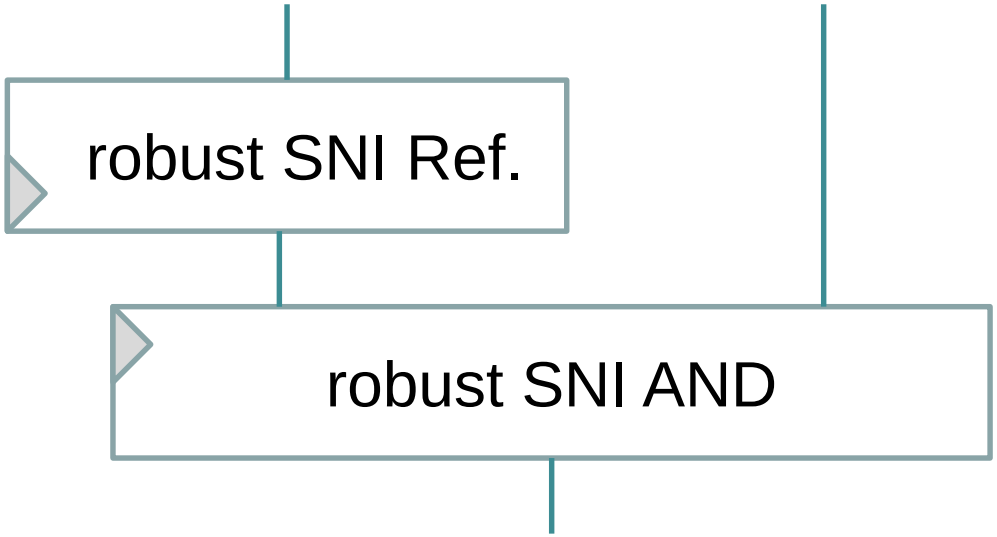


[Faust et al., 2018]: SNI-based PINI mult. in 4 cycles

[Cassiers et al., 2019]: PINI maintained without output register (…remember this would not work with SNI)

Hardware Private Circuits

- (∃ more efficient PINI multiplications in software)
- Significantly improves trivial composition in hardware

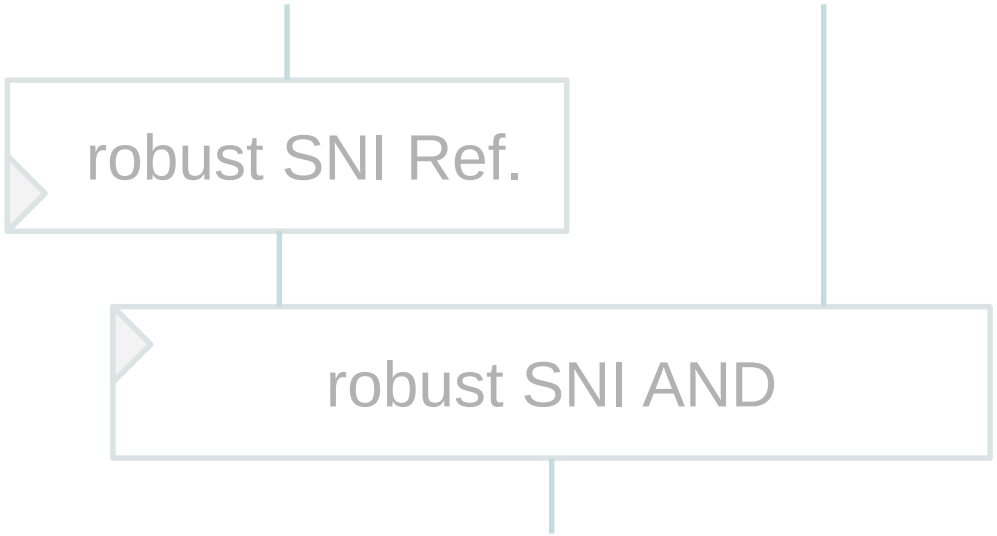


[Faust et al., 2018]: SNI-based PINI mult. in 4 cycles

[Cassiers et al., 2019]: PINI maintained without output register & refresh randomness can be accumulated off-path

Hardware Private Circuits

- (∃ more efficient PINI multiplications in software)
- Significantly improves trivial composition in hardware



[Faust et al., 2018]:
SNI-based PINI mult.
in 4 cycles

[Cassiers et al., 2019]: PINI maintained without output register & refresh randomness can be accumulated off-path

≈ optimization of [Faust et al., 2018] or fix of DOM

Other PINI advantages

- First efficient glitch-resistant masking scheme that is provably composable at arbitrary orders
  - Further improvements with optimized composition are an interesting open problem
  - But overheads compared to 1-cycle DOM limited
    - Especially for some S-box structures that can take advantage of the input refreshing asymmetry

Other PINI advantages

- First efficient glitch-resistant masking scheme that is provably composable at arbitrary orders
  - Further improvements with optimized composition are an interesting open problem
  - But overheads compared to 1-cycle DOM limited
    - Especially for some S-box structures that can take advantage of the input refreshing asymmetry

- Instrumental in the design of *full verification* tool
  - Composable verification like [Barthe et al., 2016] that applies to synthetized VHDL code rather than abstract (e.g., glitch-free) circuit descriptions
  - Also captures transitions (thanks to isolation)?

State-of-the-art tools (roughly)

	abstract	concrete
direct	Barthe et al. (Eurocrypt 2015)	REBECCA (Eurocrypt 2018) maskVerif (ESORICS 2019)
comp.-based	maskComp. (ACM CCS 2016) Tight Private Circuits (Asiacrypt 2018)	fullVerif (*new*)

State-of-the-art tools (roughly)

	abstract	concrete
direct	Barthe et al. (Eurocrypt 2015)	REBECCA (Eurocrypt 2018) maskVerif (ESORICS 2019)
comp.-based	maskComp. (ACM CCS 2016) Tight Private Circuits (Asiacrypt 2018)	fullVerif (*new*)

- ∃ other approaches (e.g., spanning multiples cells like the one by Eldib et al., or aiming at different, more specific, goals like the one of Arribas et al.)

State-of-the-art tools (roughly)

	abstract	concrete
direct	Barthe et al. (Eurocrypt 2015)	REBECCA (Eurocrypt 2018) maskVerif (ESORICS 2019)
comp.-based	maskComp. (ACM CCS 2016) Tight Private Circuits (Asiacrypt 2018)	fullVerif (*new*)

- ∃ other approaches (e.g., spanning multiples cells like the one by Eldib et al., or aiming at different, more specific, goals like the one of Arribas et al.)
- Next, first full verification tool that applies to synthetized HDL code and captures all physical defaults that can be naturally modeled with probes (i.e., transitions & glitches)

Hardware composition verification tool

Trivial composition makes it simple for the designer:

"Just connect PINI gadgets together."

Do you really want to write a tool to check that all gadgets are PINI ?

```
for gadget in gadgets:
  assert gadget.is_pini(); // Uses maskVerif
```

Done ?

A masked Verilog block cipher implementation

Code:
- ~30 files
- ~4k LoC

Parmeters:
- d = 2,...,16
- roll_sb = 0, ..., 5
- roll_lb = 0, 1, 2

15*6*3 = 270 parameter sets

Complex code:
- FSM
- loops
- procedurally generated code
- pipelining
- thousands of gadget instances
- - ...

Example LoC:

```
rinrfrs1_chunk[Nrndrfrs1_each-1+ii*Nrndrfrs1_each -: Nrndrfrs1_each]
<=  {rinrfrs1[(Nrndrfrs1_each/4)*4-1+(ii+8)*Nrndrfrs1_each -:
(Nrndrfrs1_each/4)],{(Nrndrfrs1_each/4){1'b0}}, rinrfrs1[(Nrndrfrs1_each/4)*2-
1+(ii+8)*Nrndrfrs1_each -: (Nrndrfrs1_each/4)],{(Nrndrfrs1_each/4){1'b0}}};
```

Is this thing (glitch,transition)-robust t-probing secure ?

What could go wrong ?

- Bad randomness input to gadgets
- Re-order of wires in a sharing
- Mix clock signals
- Mix valid and invalid data
- Output data at the right cycle
- Keep state around after computation is over
- …

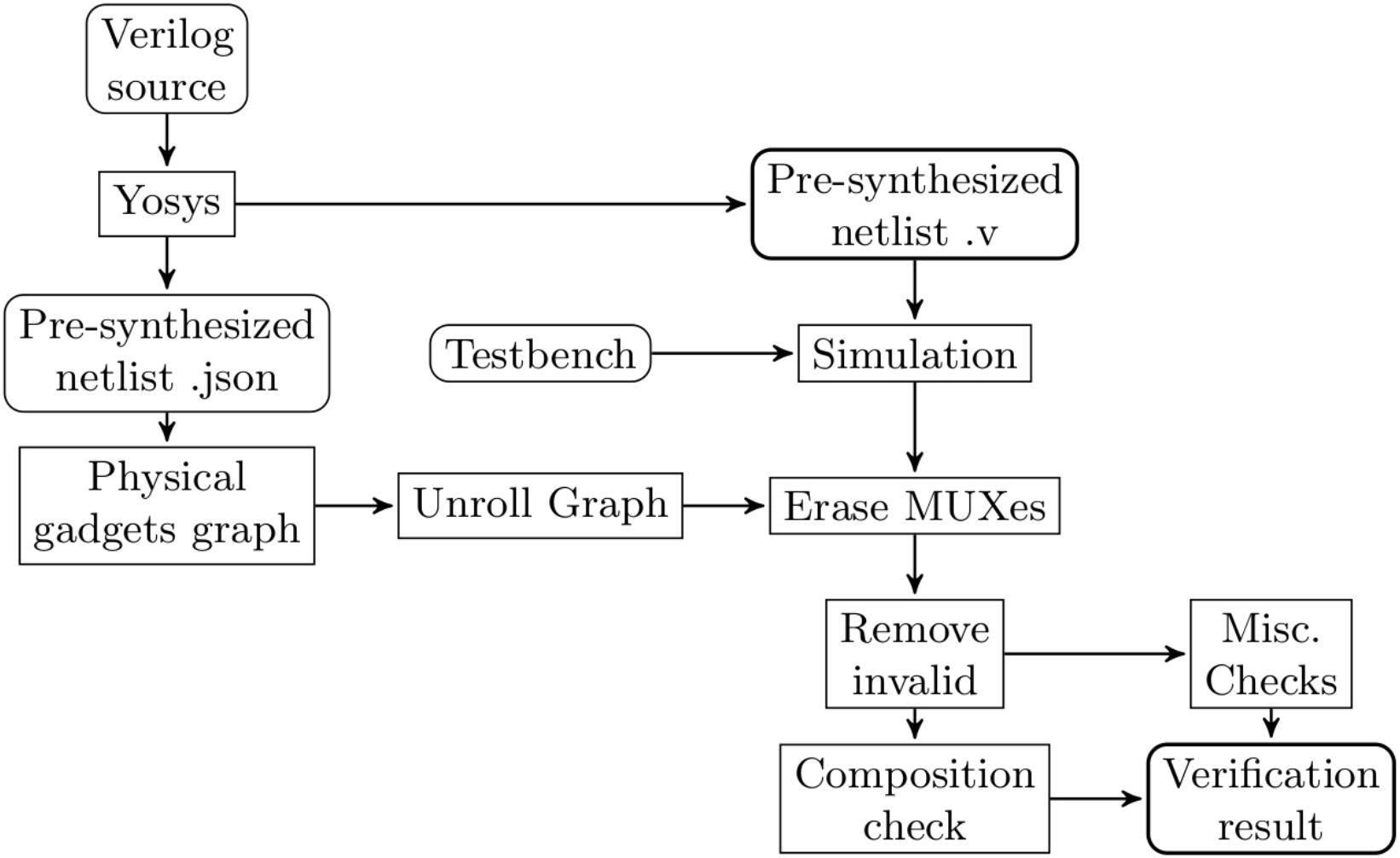
Code written by a side-channel expert hardware designer.

And no:

- Use non-PINI gadgets

(Experiment might be biased.)

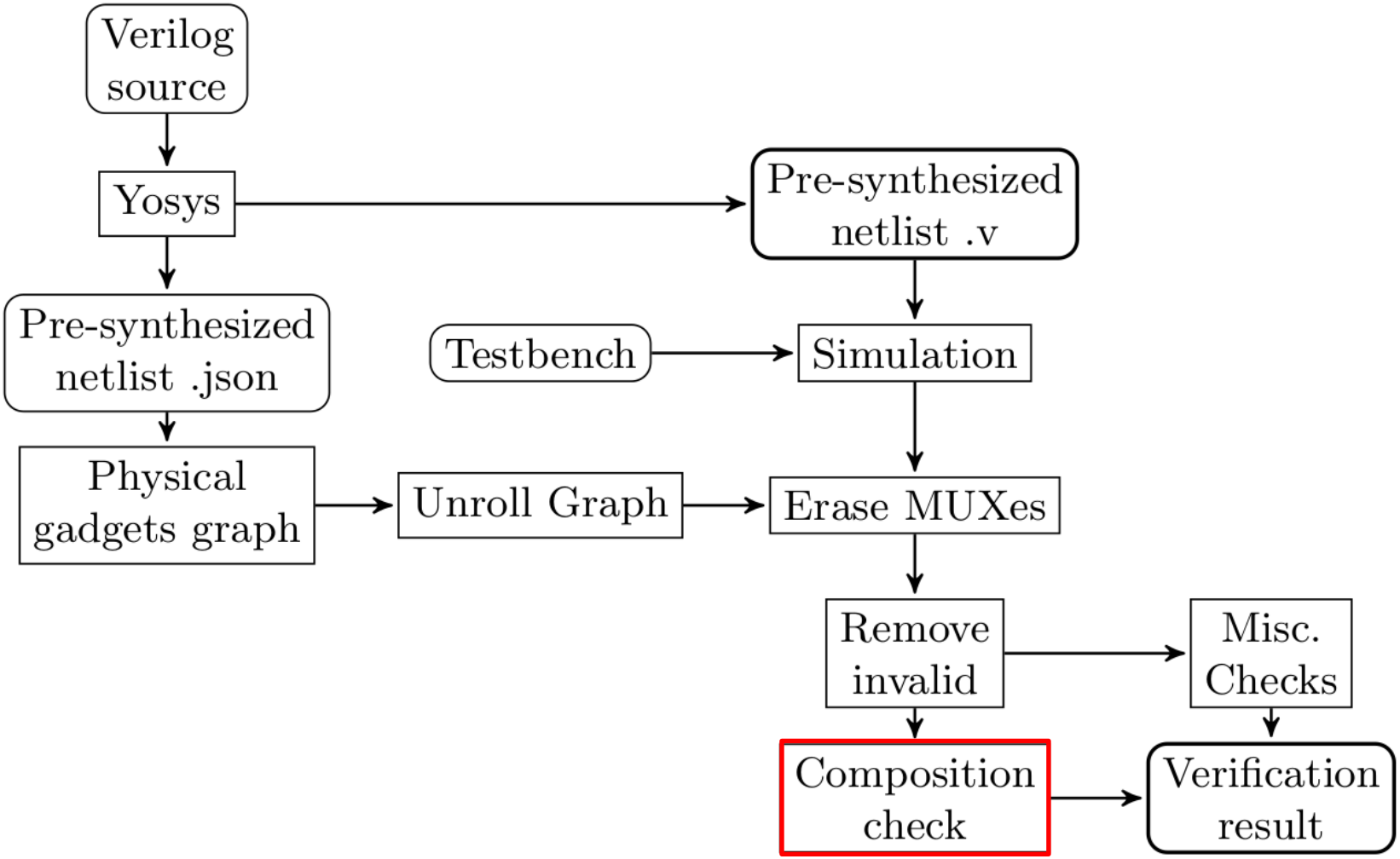
Tool workflow



- 2.5kLoC
- <10s runtime

Flexible: it is easy to implement other strategies.

Tool workflow



- 2.5kLoC
- <10s to check

Checking other strategies: 1 box change!

Source annotations

```
1  (* psim_prop = "PINI", psim_strat = "assumed", psim_order=d *)
2  module and_pini #(parameter d=2) (ina, inb, rnd, clk, out);
3
4  `include "ref_rnd.inc"
5  localparam n_rnd_mul = d*(d-1)/2;
6  localparam n_rnd = and_pini_nrnd;
7
8  (* psim_type = "sharing", psim_latency = 1 *) input  [d-1:0] ina;
9  (* psim_type = "sharing", psim_latency = 0 *) input  [d-1:0] inb;
10 (* psim_type = "sharing", psim_latency = 2 *) output [d-1:0] out;
11 (* psim_type = "clock" *) input clk;
12 (*
13     psim_type = "random", psim_count=4,
14     psim_rnd_lat_0=1, psim_rnd_count_0=and_pini_lat_1,
15     psim_rnd_lat_1=0, psim_rnd_count_1=and_pini_lat_0,
16     psim_rnd_lat_2 = -1, psim_rnd_count_2=and_pini_lat_m1,
17     psim_rnd_lat_3=-2, psim_rnd_count_3=and_pini_lat_m2
18 *) input [n_rnd-1:0] rnd;
19
20 wire [n_rnd_mul-1:0] rnd_mul = rnd[n_rnd-1:ref_n_rnd];
21 wire [ref_n_rnd-1:0] rnd_ref = rnd[ref_n_rnd-1:0];
22
23 wire [d-1:0] inb_ref;
24 MSKref #(.d(d)) rfrsh (.in(inb), .clk(clk), .out(inb_ref), .rnd(rnd_ref));
25 MSKand #(.d(d)) mul (.ina(ina), .inb(inb_ref), .clk(clk), .rnd(rnd_mul), .out(out));
26
27 endmodule
```

Source annotations: composite gadget

```
1  (* psim_prop = "PINI", psim_strat = "composite", psim_order=d *)
2  module MSKspook_sbox #(parameter d=4) (in, rnd_ref, rnd_mul, clk, out);
3
4  `include "spook_sbox_rnd.inc"
5
6  (* psim_type = "sharing", psim_latency = 0, psim_count=spook_sbox_nbits*)
7  input  [d*spook_sbox_nbits-1:0] in;
8  (* psim_type = "sharing", psim_latency = spook_sbox_lat, psim_count=spook_sbox_nbits *)
9  output [d*spook_sbox_nbits-1:0] out;
10 (* psim_type = "clock" *) input clk;
11 (*
12  psim_type = "random", psim_count=6,
13  psim_rnd_lat_0=0, psim_rnd_count_0=2*ref_lat_0,
14  psim_rnd_lat_1=-1, psim_rnd_count_1=2*ref_lat_m1,
15  psim_rnd_lat_2=-2, psim_rnd_count_2=2*ref_lat_m2,
16  psim_rnd_lat_3=1, psim_rnd_count_3=2*ref_lat_0,
17  psim_rnd_lat_4=0, psim_rnd_count_4=2*ref_lat_m1,
18  psim_rnd_lat_5=-1, psim_rnd_count_5=2*ref_lat_m2
19 *) input [4*ref_n_rnd-1:0] rnd_ref;
20 (*
21  psim_type = "random", psim_count=2,
22  psim_rnd_lat_0=0, psim_rnd_count_0=2*and_pini_lat_1,
23  psim_rnd_lat_1=1, psim_rnd_count_1=2*and_pini_lat_1
24 *) input [4*and_pini_lat_1-1:0] rnd_mul;
25
26 MSKreg #(d) reg1 (clk, 1'b0, in[d+d*(3)-1 -: d], x0F);
27 MSKreg #(d) reg2 (clk, 1'b0, in[d+d*(2)-1 -: d], x1F);
28 [...]
```

Source annotations: flatten (for the lazy)

```
1  (* psim_prop = "PINI", psim_strat = "composite", psim_order=d *)
2  module MSKsbox_unit # (
3      parameter d = 2,
4      parameter PDSBOX = 0,
5      parameter Nbits = 128,
6      // Generation params (DO NOT TOUCH)
7      localparam AM_BUND_cols = 2**PDSBOX,
8      localparam SIZE_BUND_cols = d*Nbits/AM_BUND_cols,
9      localparam AM_cols = 32/AM_BUND_cols
10 ) (cols, rnd, clk, cols_post_sb);
11 `include "spook_sbox_rnd.inc"
12 input [SIZE_BUND_cols-1:0] cols;
13 input [spook_sbox_rnd*(SIZE_BUND_cols/(4*d))-1:0] rnd;
14 input clk;
15 output [SIZE_BUND_cols-1:0] cols_post_sb;
16 genvar i;
17 for(i=0;i<AM_cols;i=i+1) begin: sb
18     MSKspook_sbox #(.d(d)) sbi (
19         .in(cols[(i+1)*4*d-1:i*4*d]),
20         .rnd_ref(rnd[ (i+1)*4*ref_n_rnd-1   +(32/(2**PDSBOX))*and_pini_lat_1    : i*
21         .rnd_mul(rnd[ (i+1)*4*and_pini_lat_1 -1                                 : i*
22         .clk(clk),
23         .out(cols_post_sb[(i+1)*4*d-1:i*4*d])
24     );
25 end
26 endmodule
```

THANKS

http://perso.uclouvain.be/fstandae/