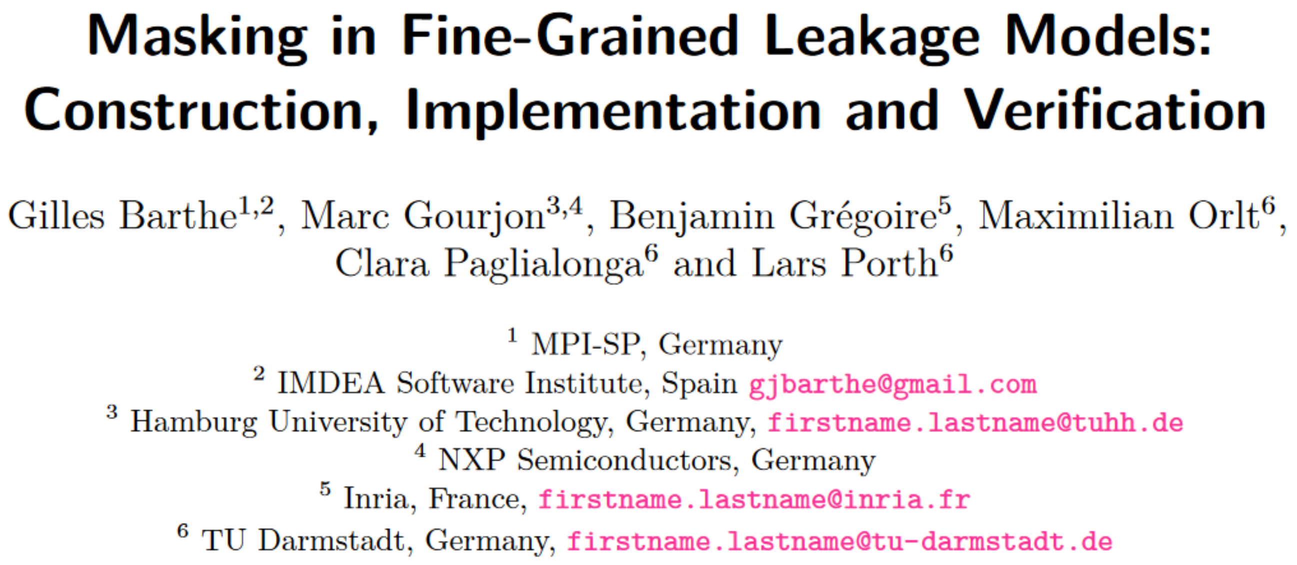
FINE-GRAINED LEAKAGE MODELS & VERIFICATION OF SOFTWARE MASKING

Marc Gourjon

VERISICC Seminar 2022

SEPTEMBER 2022

SECURE CONNECTIONS FOR A SMARTER WORLD



Masking in Fine-Grained Leakage Models: Construction, Implementation and Verification

Gilles Barthe[1,2], Marc Gourjon[3,4], Benjamin Grégoire[5], Maximilian Orlt[6], Clara Paglialonga[6] and Lars Porth[6]

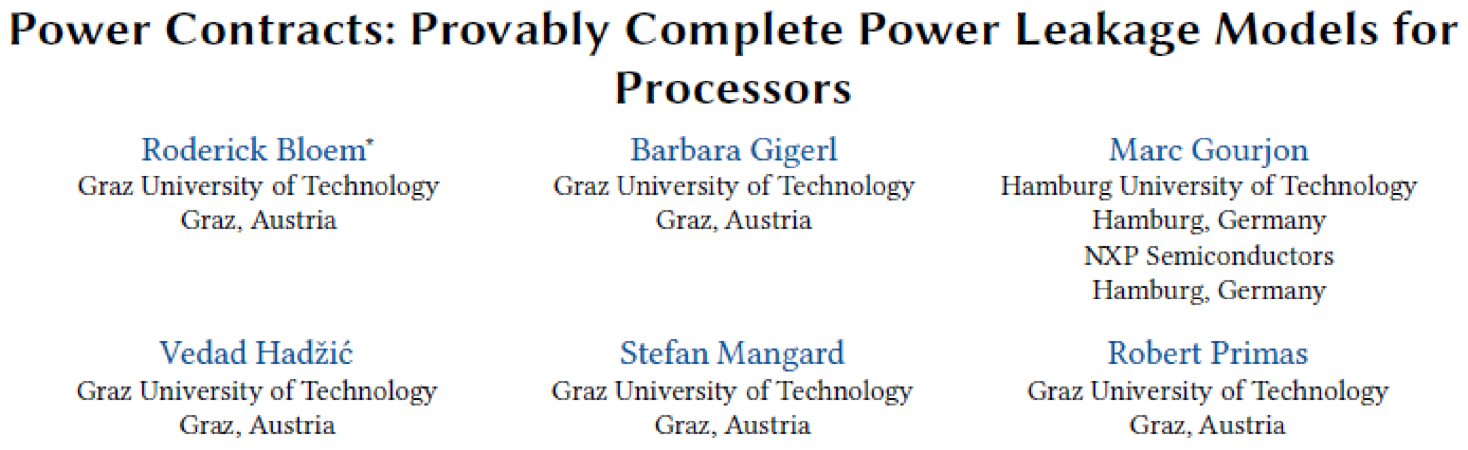
[1] MPI-SP, Germany
[2] IMDEA Software Institute, Spain gjbarthe@gmail.com
[3] Hamburg University of Technology, Germany, firstname.lastname@tuhh.de
[4] NXP Semiconductors, Germany
[5] Inria, France, firstname.lastname@inria.fr
[6] TU Darmstadt, Germany, firstname.lastname@tu-darmstadt.de

scVerif: https://github.com/scverif/scverif

Gadgets: https://github.com/scverif/gadgets

Contract: https://eprint.iacr.org/2022/565.pdf

Kyber: https://eprint.iacr.org/2021/483.pdf



Power Contracts: Provably Complete Power Leakage Models for Processors

Roderick Bloem*
Graz University of Technology
Graz, Austria

Barbara Gigerl
Graz University of Technology
Graz, Austria

Marc Gourjon
Hamburg University of Technology
Hamburg, Germany
NXP Semiconductors
Hamburg, Germany

Vedad Hadžić
Graz University of Technology
Graz, Austria

Stefan Mangard
Graz University of Technology
Graz, Austria

Robert Primas
Graz University of Technology
Graz, Austria

These works received funding from the Federal Ministry of Education and Research (BMBF) as part of the VE-Jupiter project (grant number 16ME0231K).

MODEL-BASED SECURITY ASSESSMENT

masked_program.s

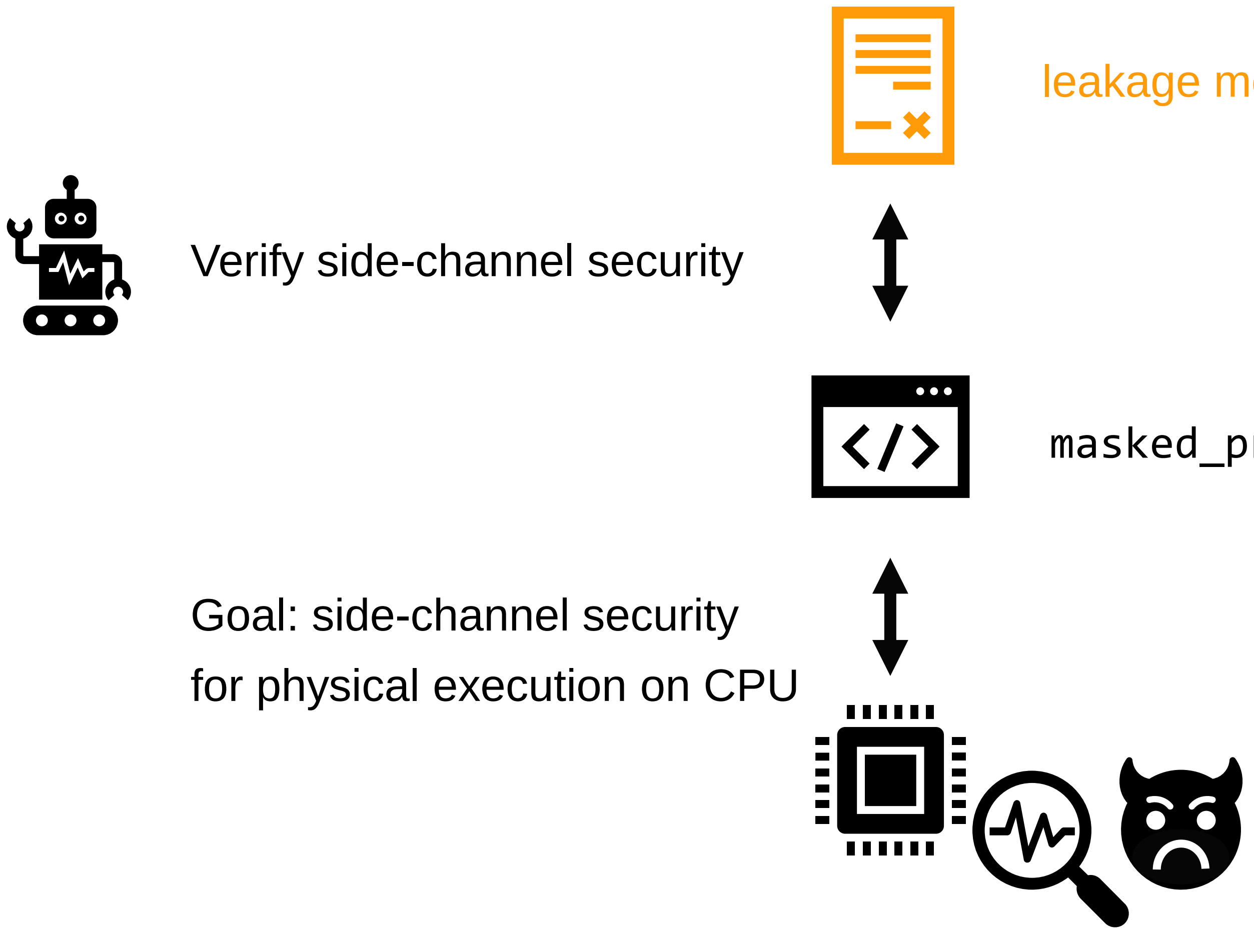
MODEL-BASED SECURITY ASSESSMENT

leakage model

Verify side-channel security

`masked_program.s`

MODEL-BASED SECURITY ASSESSMENT



leakage model

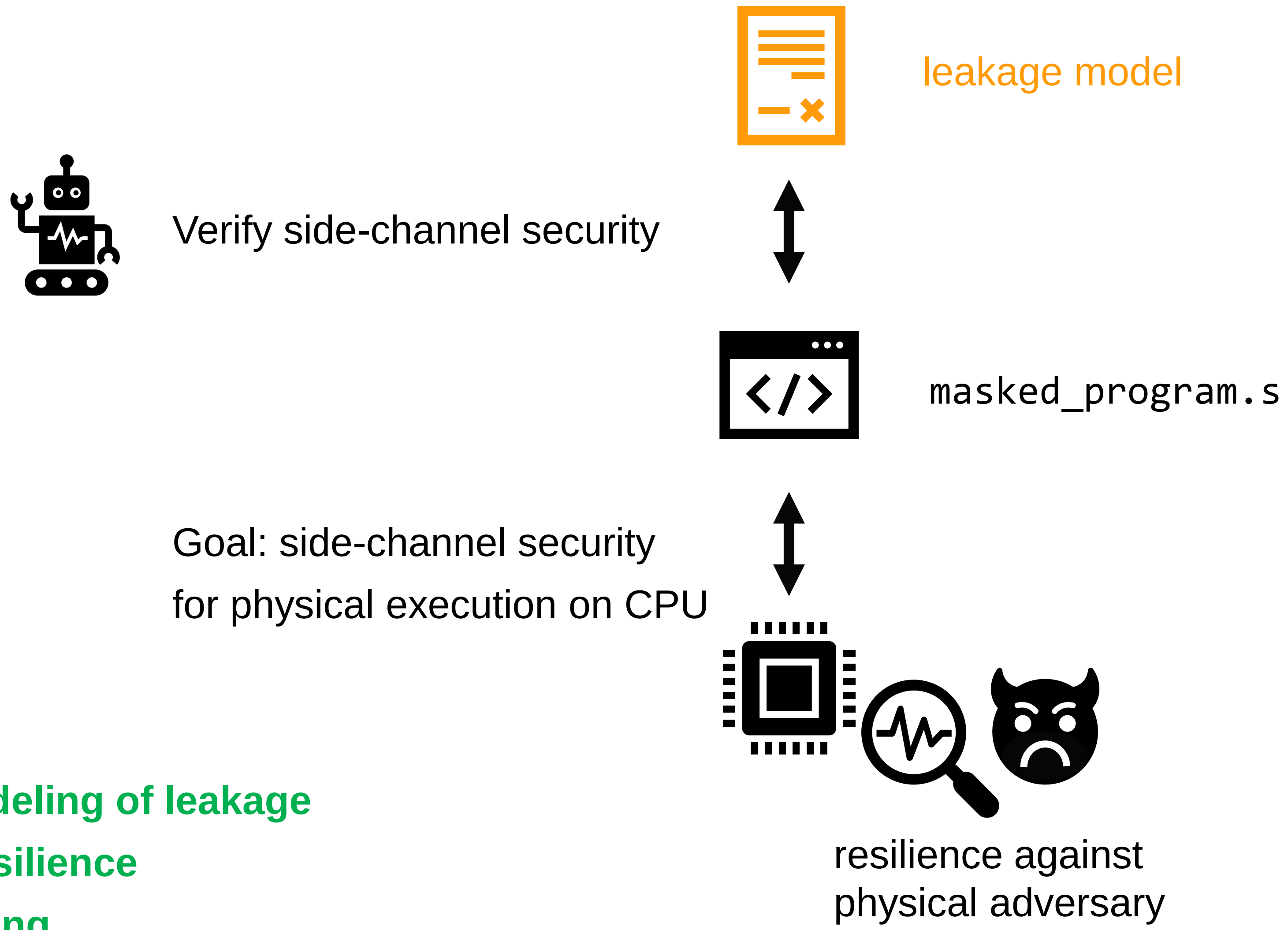
Verify side-channel security

masked_program.s

Goal: side-channel security for physical execution on CPU

resilience against physical adversary

MODEL-BASED SECURITY ASSESSMENT

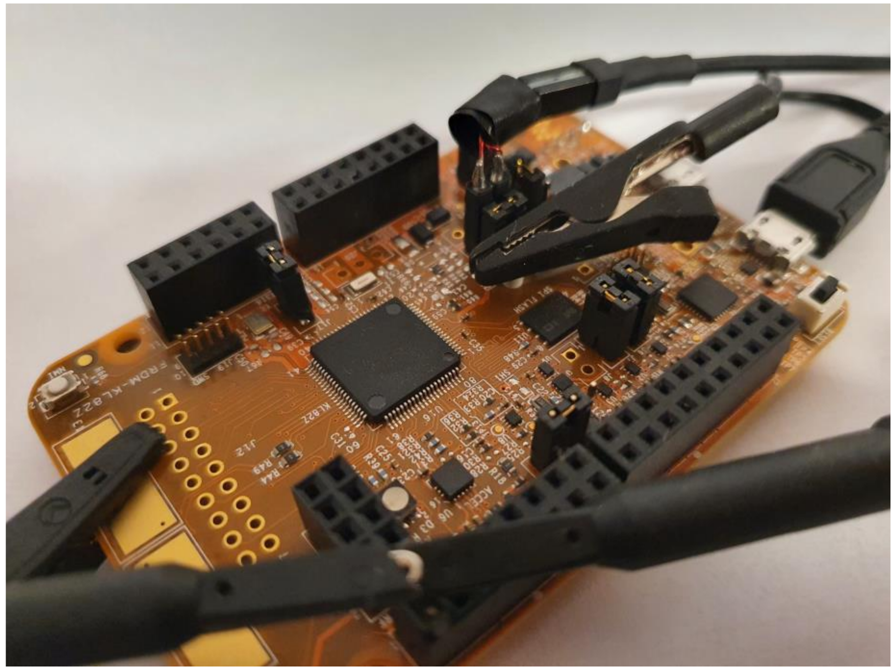

leakage model
Verify side-channel security
masked_program.s
Goal: side-channel security
for physical execution on CPU
resilience against
physical adversary

In this talk

- **precise modeling of leakage**
- **verifying resilience**
- **a bit of tooling**

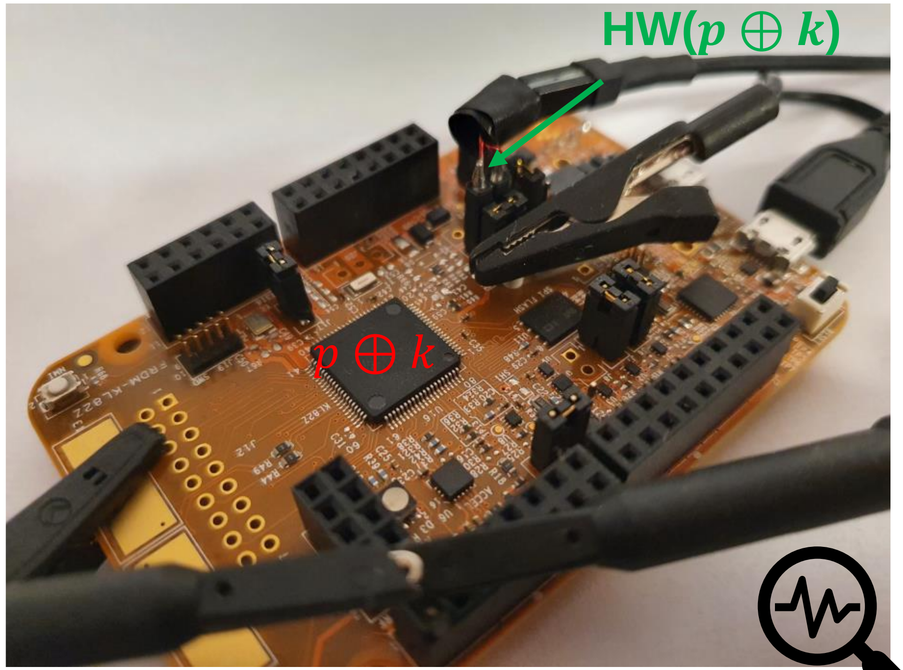
POWER SIDE-CHANNELS

- Physical computation on CPU
  - Electrical charge flowing
  - Charge = State = bit = {1,0} = key?



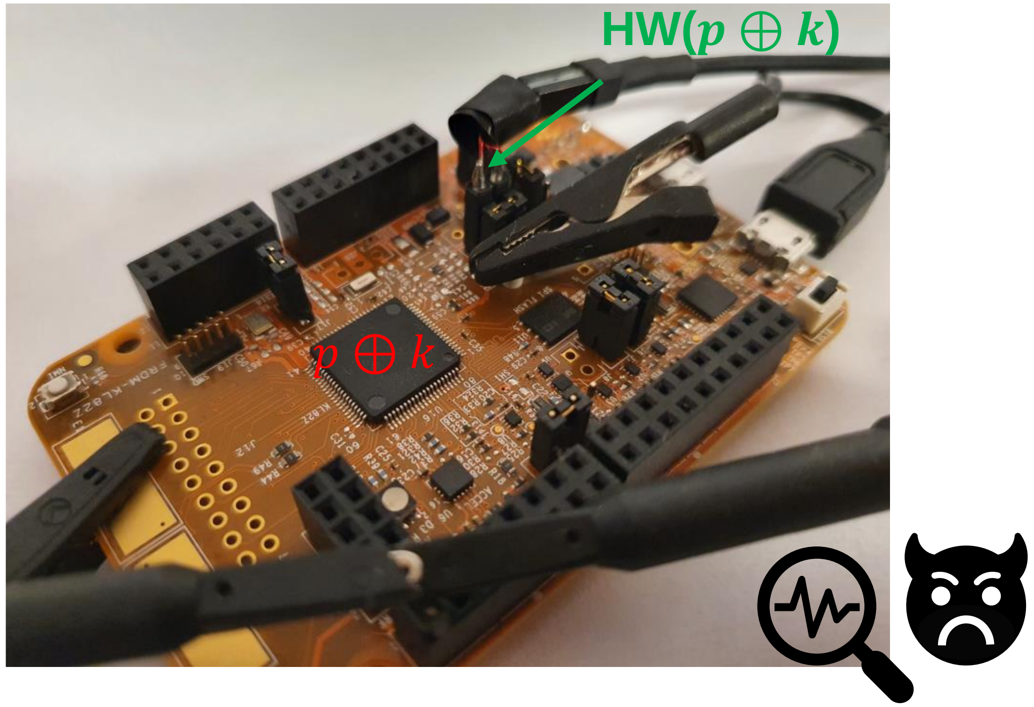
POWER SIDE-CHANNELS

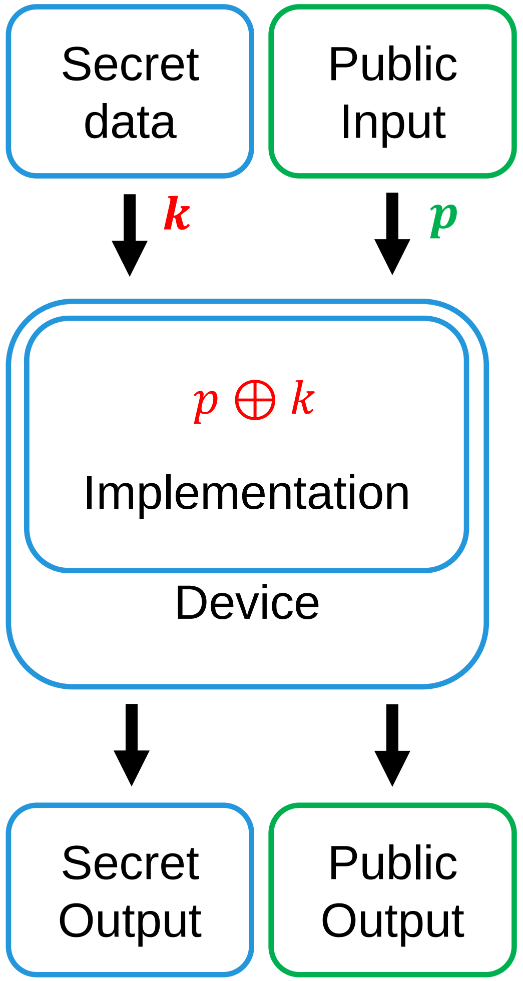
- Physical computation on CPU
  - Electrical charge flowing
  - Charge = State = bit = {1,0} = key?



POWER SIDE-CHANNELS

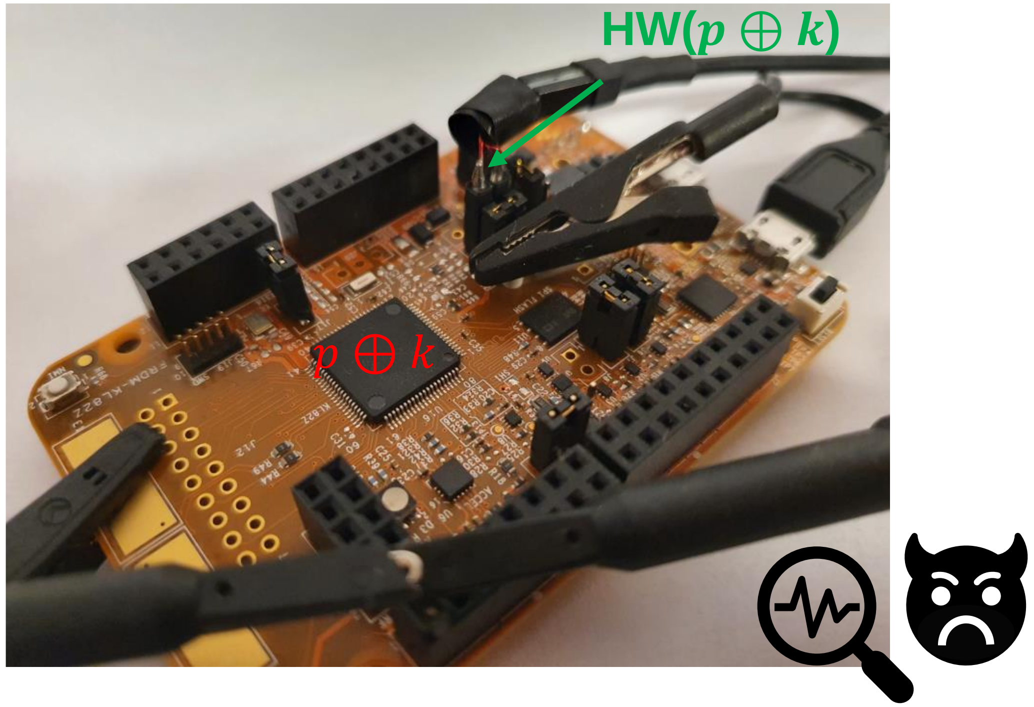
- Physical computation on CPU
  - Electrical charge flowing
  - Charge = State = bit = {1,0} = key?

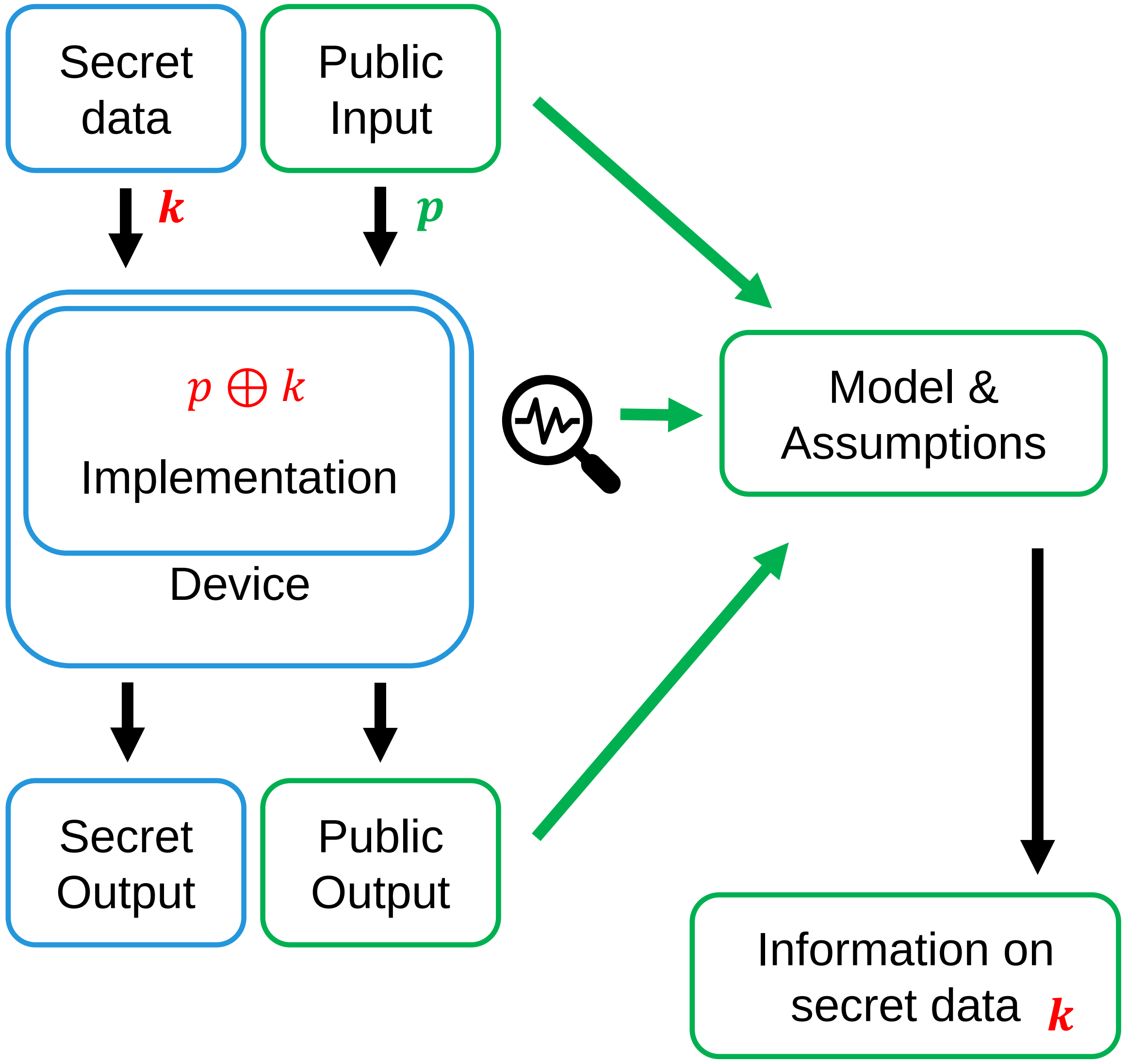




POWER SIDE-CHANNELS

- Physical computation on CPU
  - Electrical charge flowing
  - Charge = State = bit = {1,0} = key?

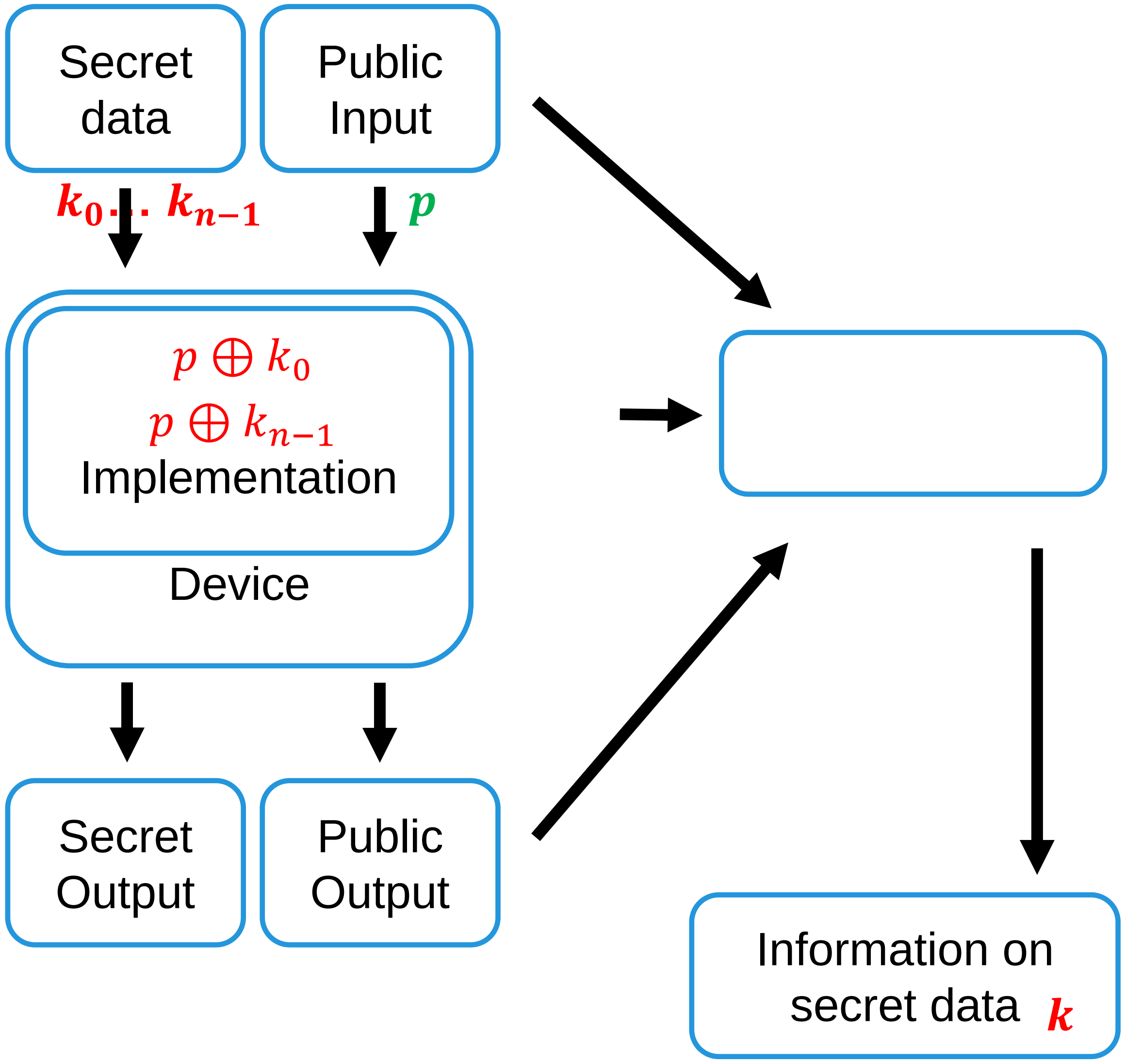




MASKING

- Split all secret (dependent) data
  - $k = k_0 \oplus k_1 \oplus \cdots \oplus k_{n-1}$
  - Information theoretical security guarantee
    - Prove no information on any secret key can be retrieved under certain assumption

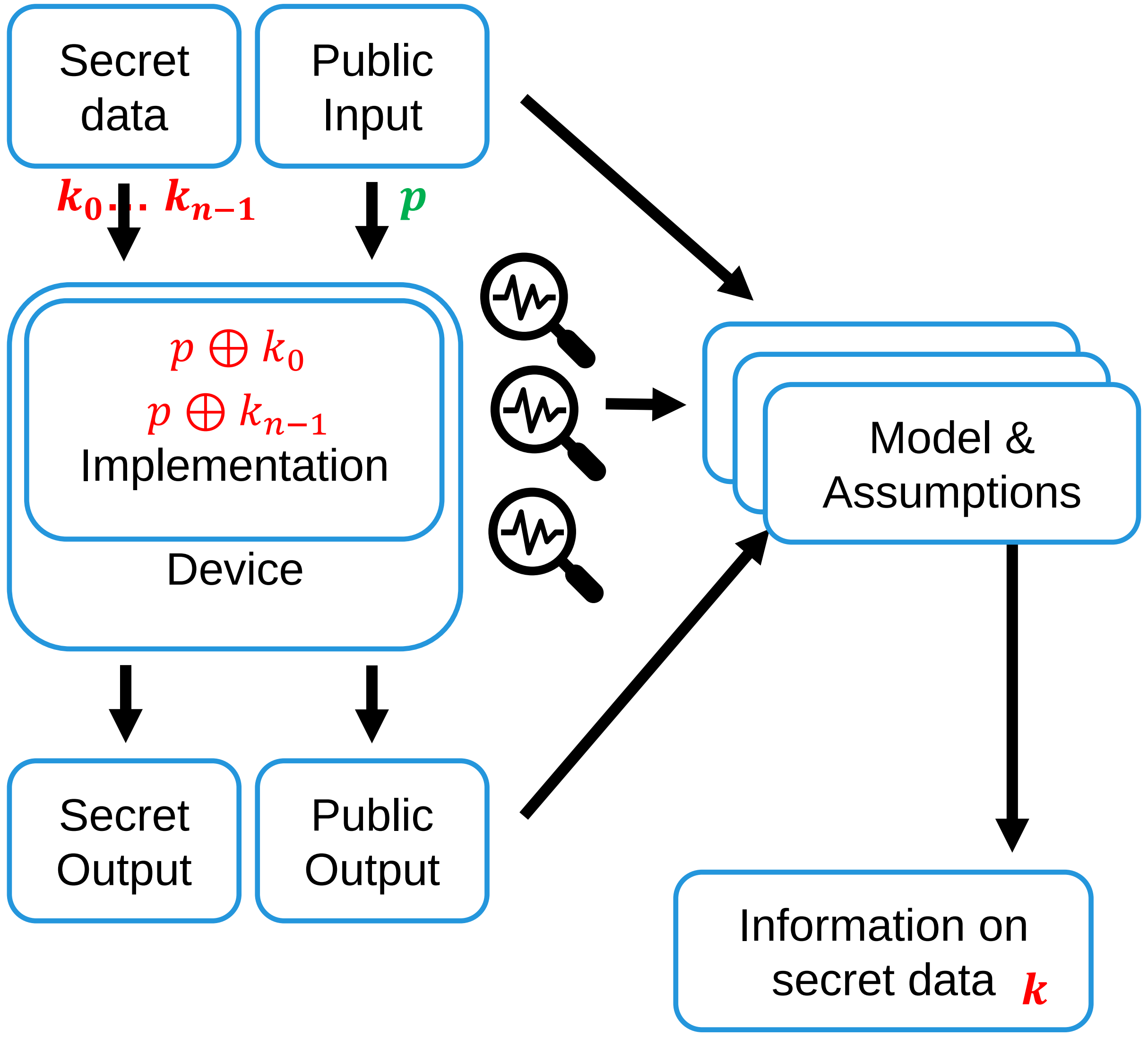
    → Adversaries must recover at least $\mathrm{d} < n$ shares



MASKING

- Split all secret (dependent) data
  - $k = k_0 \oplus k_1 \oplus \cdots \oplus k_{n-1}$
  - Information theoretical security guarantee
    - Prove no information on any secret key can be retrieved under certain assumption

  → Adversaries must recover at least $\mathrm{d} < n$ shares



$n > d$

MASKED GADGETS & PROVABLE SECURITY (1)

masked AND $z = x \wedge y$
Inputs: $(x_0, x_1, \dots, x_{n-1}), (y_0, y_1, \dots, y_{n-1})$

```
For $i = 0$ to $n-1$
        $\boldsymbol{z_i \leftarrow x_i \wedge y_i}$
For $i = 0$ to $n-1$
    For $j = i+1$ to $n-1$
        $\boldsymbol{r \leftarrow \{0,1\}}$
        $\boldsymbol{r' \leftarrow \left(r \oplus (x_i \wedge y_j)\right) \oplus (x_j \wedge y_i)}$
        $\boldsymbol{z_i \leftarrow z_i \oplus r}$
        $\boldsymbol{z_j \leftarrow z_j \oplus r'}$
Return $(z_0, z_1, \dots, z_{n-1})$
```

Proven secure!

dth -Order probing security

Any **set of d *observations*** an attacker could make must be independent of secrets => must perform at least a d+1 order attack

$n > d$

MASKED GADGETS & PROVABLE SECURITY (1)

masked AND $z = x \wedge y$

Inputs: $(x_0, x_1, \dots, x_{n-1})$, $(y_0, y_1, \dots, y_{n-1})$

```
For $i$ = 0 to $n - 1$
        $\boldsymbol{z_i} \leftarrow \boxed{\boldsymbol{x_i \wedge y_i}}$
For $i$ = 0 to $n - 1$
    For $j$ = $i + 1$ to $n - 1$
        $\boldsymbol{r} \leftarrow \boxed{\{\boldsymbol{0, 1}\}}$
        $\boldsymbol{r'} \leftarrow \boxed{\left(\boldsymbol{r} \oplus (\boldsymbol{x_i} \wedge \boldsymbol{y_j})\right) \oplus (\boldsymbol{x_j} \wedge \boldsymbol{y_i})}$
        $\boldsymbol{z_i} \leftarrow \boxed{\boldsymbol{z_i} \oplus \boldsymbol{r}}$
        $\boldsymbol{z_j} \leftarrow \boxed{\boldsymbol{z_j} \oplus \boldsymbol{r'}}$
Return $(z_0, z_1, \dots, z_{n-1})$
```

Proven secure!

dth -Order probing security

Any **set of** d ***observations*** an attacker could make must be independent of secrets => must perform at least a d+1 order attack

MASKED GADGETS & PROVABLE SECURITY (1)

$n > d$

masked AND $z = x \wedge y$
Inputs: $(x_0, x_1, \dots, x_{n-1}), (y_0, y_1, \dots, y_{n-1})$

Proven secure!

```
For $i = 0$ to $n-1$
        $z_i \leftarrow x_i \wedge y_i$
For $i = 0$ to $n-1$
    For $j = i+1$ to $n-1$
        $r \leftarrow \{0,1\}$
        $r' \leftarrow (r \oplus (x_i \wedge y_j)) \oplus (x_j \wedge y_i)$
        $z_i \leftarrow z_i \oplus r$
        $z_j \leftarrow z_j \oplus r'$
Return $(z_0, z_1, \dots, z_{n-1})$
```

dth -Order probing security

Any **set of d *observations*** an attacker could make must be independent of secrets => must perform at least a d+1 order attack

$n > d$

MASKED GADGETS & PROVABLE SECURITY (1)

masked AND $z = x \wedge y$

Inputs: $(x_0, x_1, \dots, x_{n-1}), (y_0, y_1, \dots, y_{n-1})$

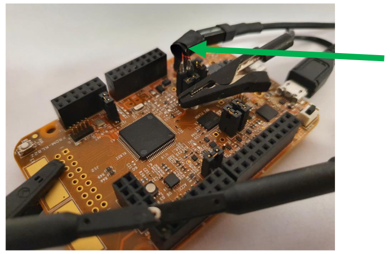
```
For $i = 0$ to $n-1$
        $\boldsymbol{z_i} \leftarrow \boxed{\boldsymbol{x_i \wedge y_i}}$
For $i = 0$ to $n-1$
    For $j = i+1$ to $n-1$
        $\boldsymbol{r} \leftarrow \boxed{\{0,1\}}$
        $\boldsymbol{r'} \leftarrow \boxed{\boxed{(\boldsymbol{r} \oplus \boxed{(\boldsymbol{x_i \wedge y_j})})} \oplus \boxed{(\boldsymbol{x_j \wedge y_i})}}$
        $\boldsymbol{z_i} \leftarrow \boxed{\boldsymbol{z_i \oplus r}}$
        $\boldsymbol{z_j} \leftarrow \boxed{\boldsymbol{z_j \oplus r'}}$
Return $(z_0, z_1, \dots, z_{n-1})$
```

Proven secure!

dth -Order probing security

Any **set of d *observations*** an attacker could make must be independent of secrets => must perform at least a d+1 order attack

MASKED GADGETS & PROVABLE SECURITY (2)



masked AND $z = x \wedge y$
Inputs: $(x_0, x_1, \dots, x_{n-1}), (y_0, y_1, \dots, y_{n-1})$

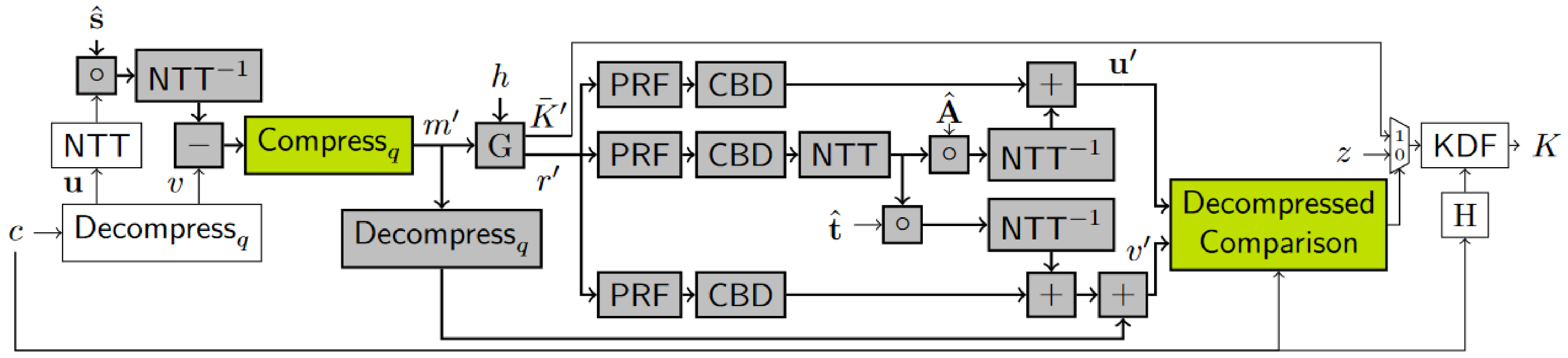
```
For $i = 0$ to $n-1$
        $\boldsymbol{z_i} \leftarrow \boldsymbol{x_i \wedge y_i}$
For $i = 0$ to $n-1$
    For $j = i+1$ to $n-1$
        $\boldsymbol{r} \leftarrow \{0,1\}$
        $\boldsymbol{r'} \leftarrow \boldsymbol{(r \oplus (x_i \wedge y_j)) \oplus (x_j \wedge y_i)}$
        $\boldsymbol{z_i} \leftarrow \boldsymbol{z_i \oplus r}$
        $\boldsymbol{z_j} \leftarrow \boldsymbol{z_j \oplus r'}$
Return $(z_0, z_1, \dots, z_{n-1})$
```

Leakage model := What is observable via side-channel

- Computational / value leakage
- Different & more observations possible in practice

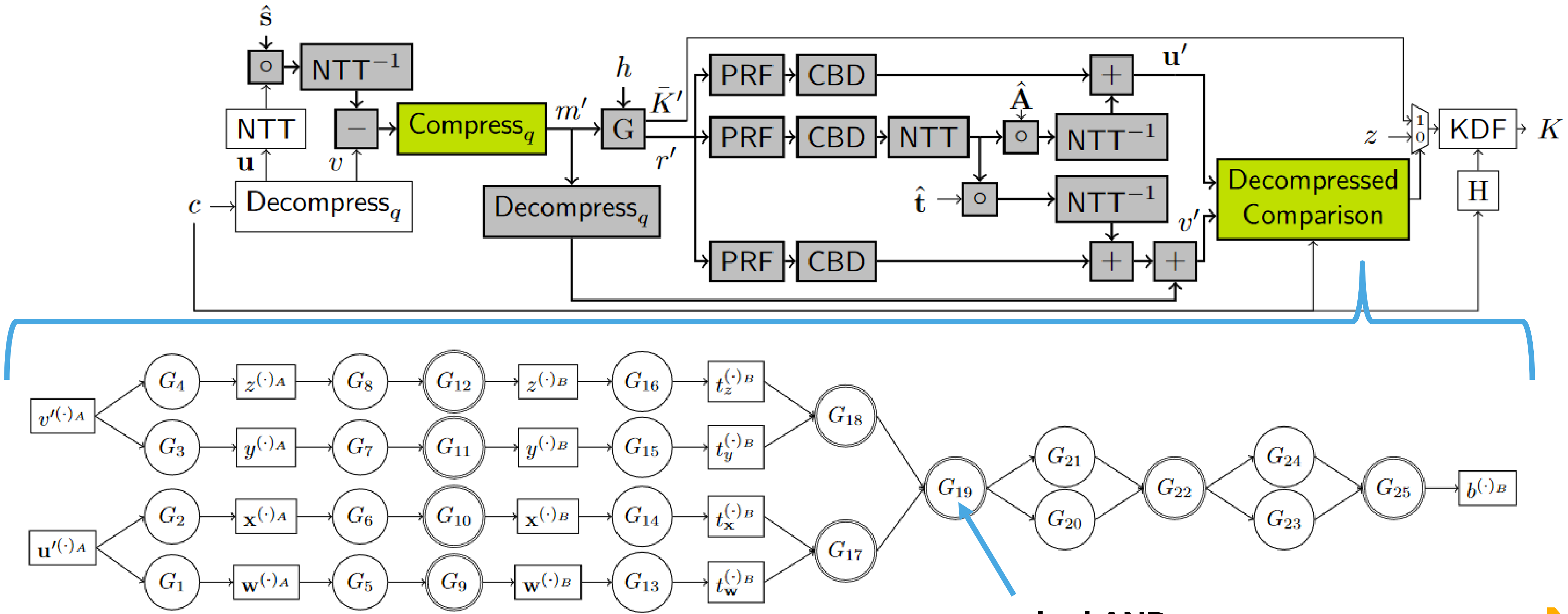
MASKING IN REAL APPLICATIONS

- Application to entire ciphers (e.g., Kyber)



MASKING IN REAL APPLICATIONS

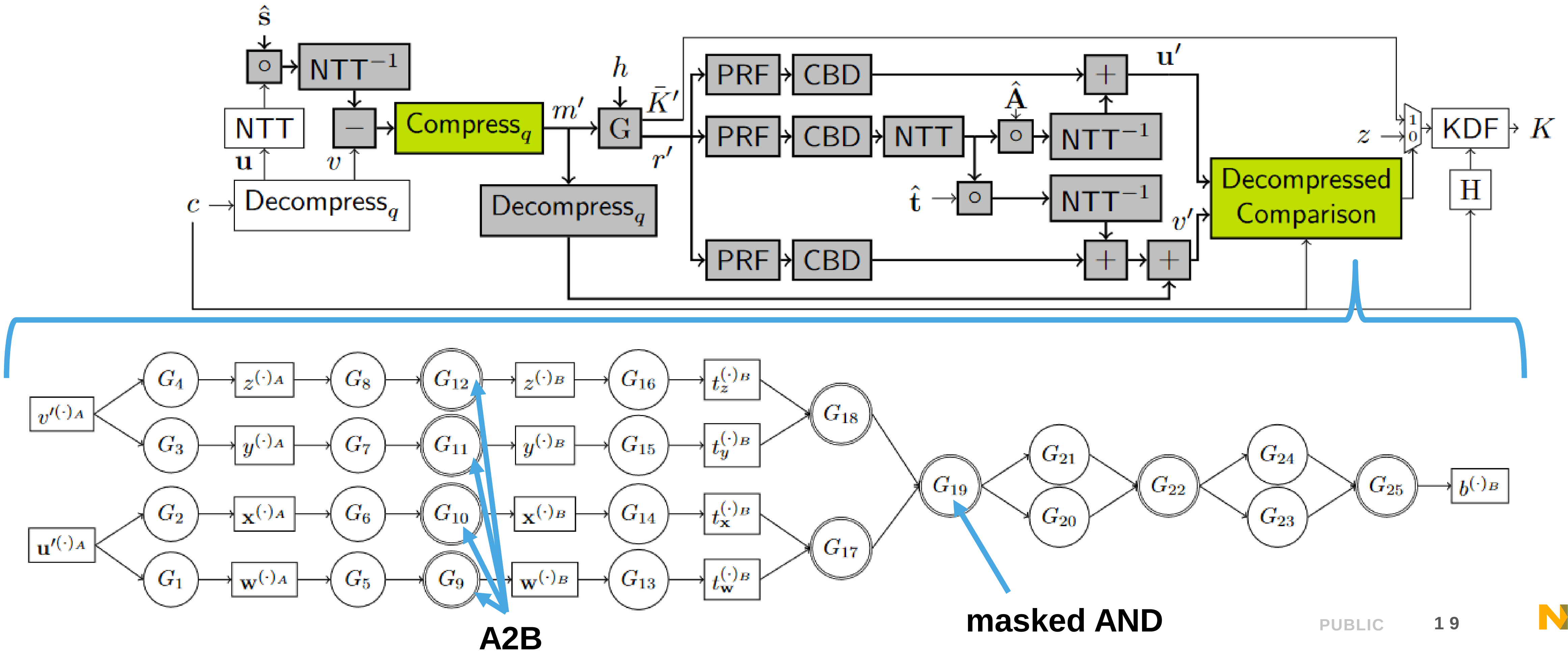
- Application to entire ciphers (e.g., Kyber)



masked AND

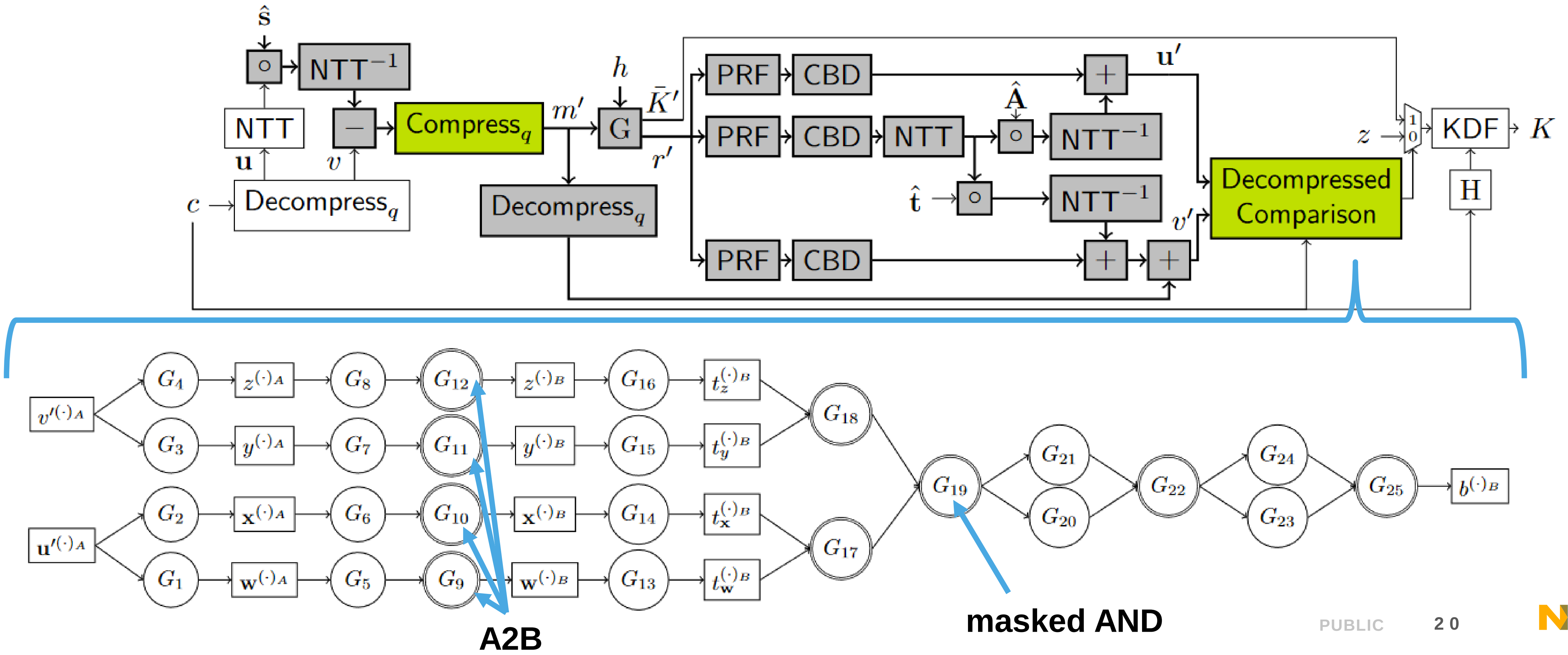
MASKING IN REAL APPLICATIONS

- Application to entire ciphers (e.g., Kyber)



MASKING IN REAL APPLICATIONS

- Application to entire ciphers (e.g., Kyber)
- Hand-crafted compositions, specialized algorithms for efficient gadgets



IMPLEMENTATION DIFFICULTIES

- Compilers will break security

  → Assembly programming

IMPLEMENTATION DIFFICULTIES

- Compilers will break security
  - → Assembly programming

```
Beware of link time optimizations
```

IMPLEMENTATION DIFFICULTIES

- Compilers will break security

  → Assembly programming

`Beware of link time optimizations`

- Functional correctness
- Adhere to observables intermediates in security proof
- Adhere to proven composition
- Randomness (re-use)

IMPLEMENTATION DIFFICULTIES

- Compilers will break security

  → Assembly programming

`Beware of link time optimizations`

- Functional correctness
- Adhere to observables intermediates in security proof
- Adhere to proven composition
- Randomness (re-use)

$$\boldsymbol{r' \leftarrow r \oplus \left((x_i \wedge y_j) \oplus (x_j \wedge y_i)\right)}\ \textbf{?}$$

IMPLEMENTATION DIFFICULTIES

- Compilers will break security
  → Assembly programming

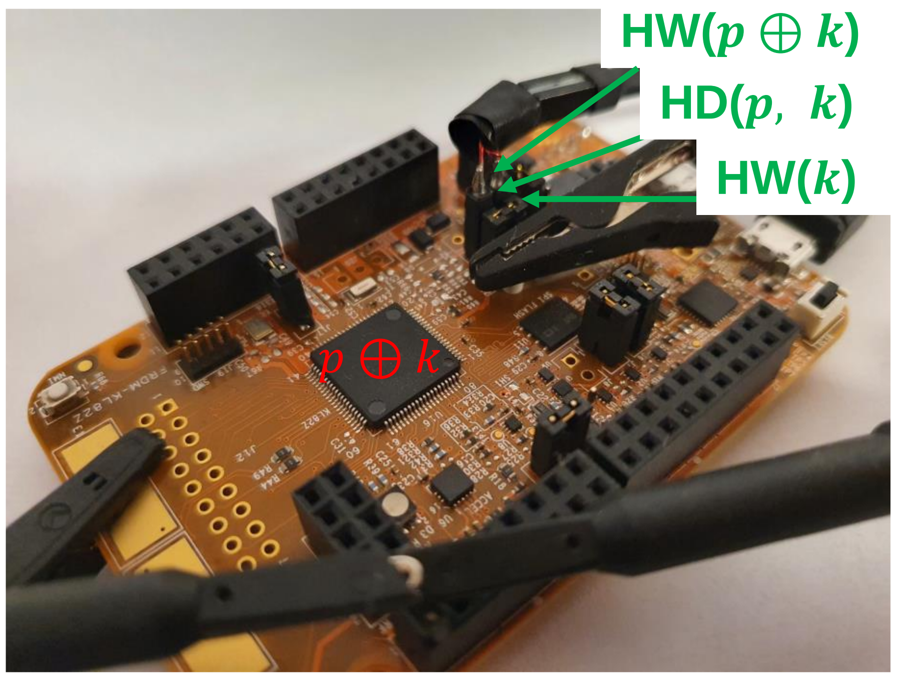
`Beware of link time optimizations`

- Functional correctness
- Adhere to observables intermediates in security proof
- Adhere to proven composition
- Randomness (re-use)

$$r' \leftarrow r \oplus \left((x_i \wedge y_j) \oplus (x_j \wedge y_i) \right) ?$$

- Device-specific leakage

MORE REALISTIC POWER LEAKAGE

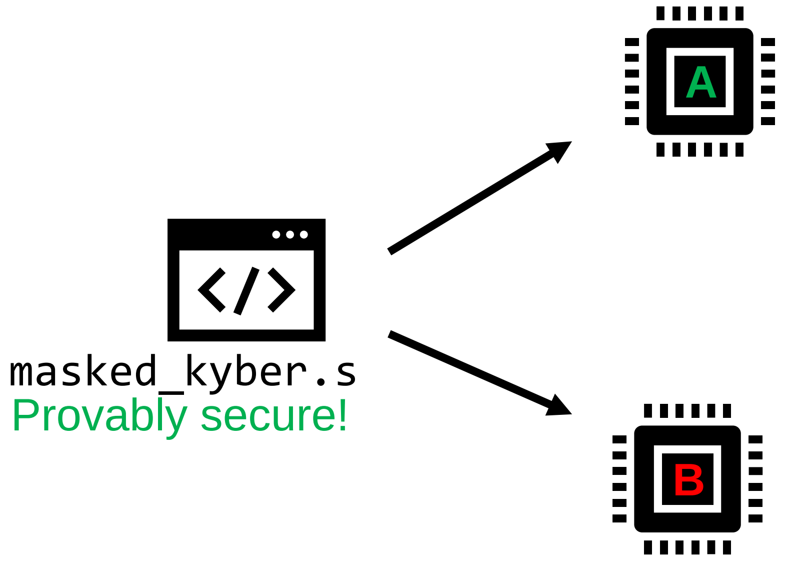


- Side-Channel = physical phenomenon
  - Not just computation leakage
  - Much more observations

- Gap in provable security
  - Any leakage observable in practice which is not captured by a proof of security

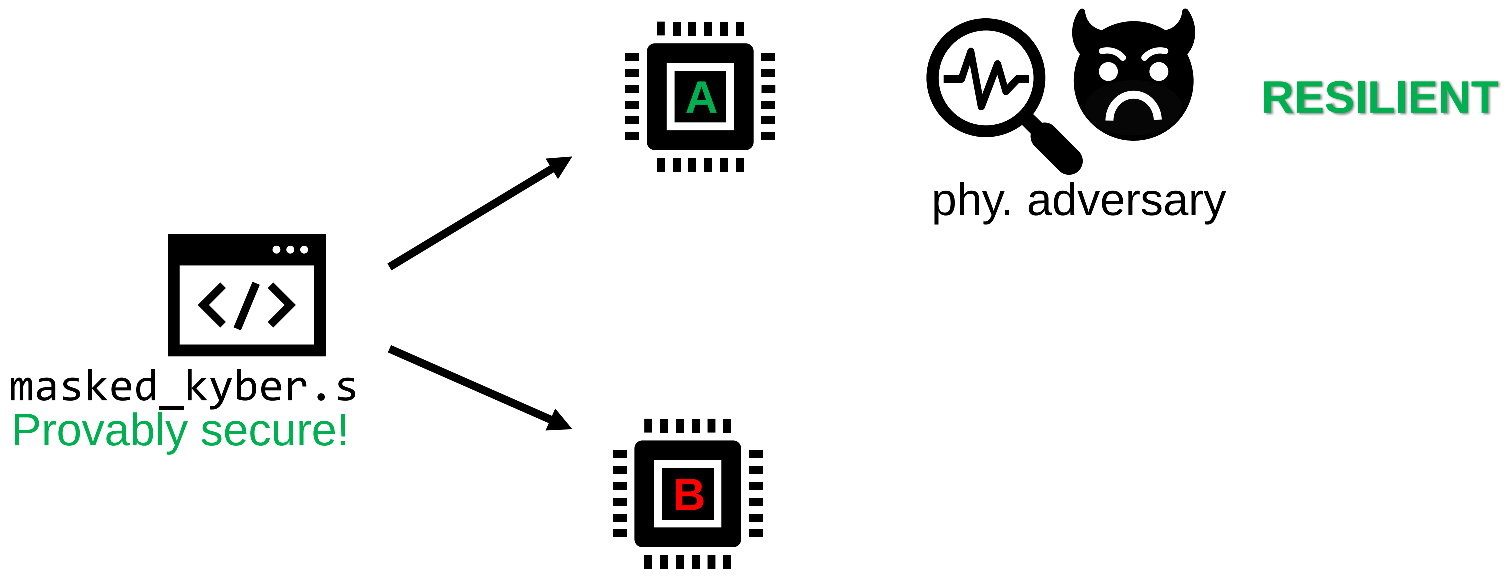
THE GAP BETWEEN PROVABLE RESILIENCE & PHYSICAL SIDE-CHANNELS
DEVICE-SPECIFIC LEAKAGE

Problem: Same program has different



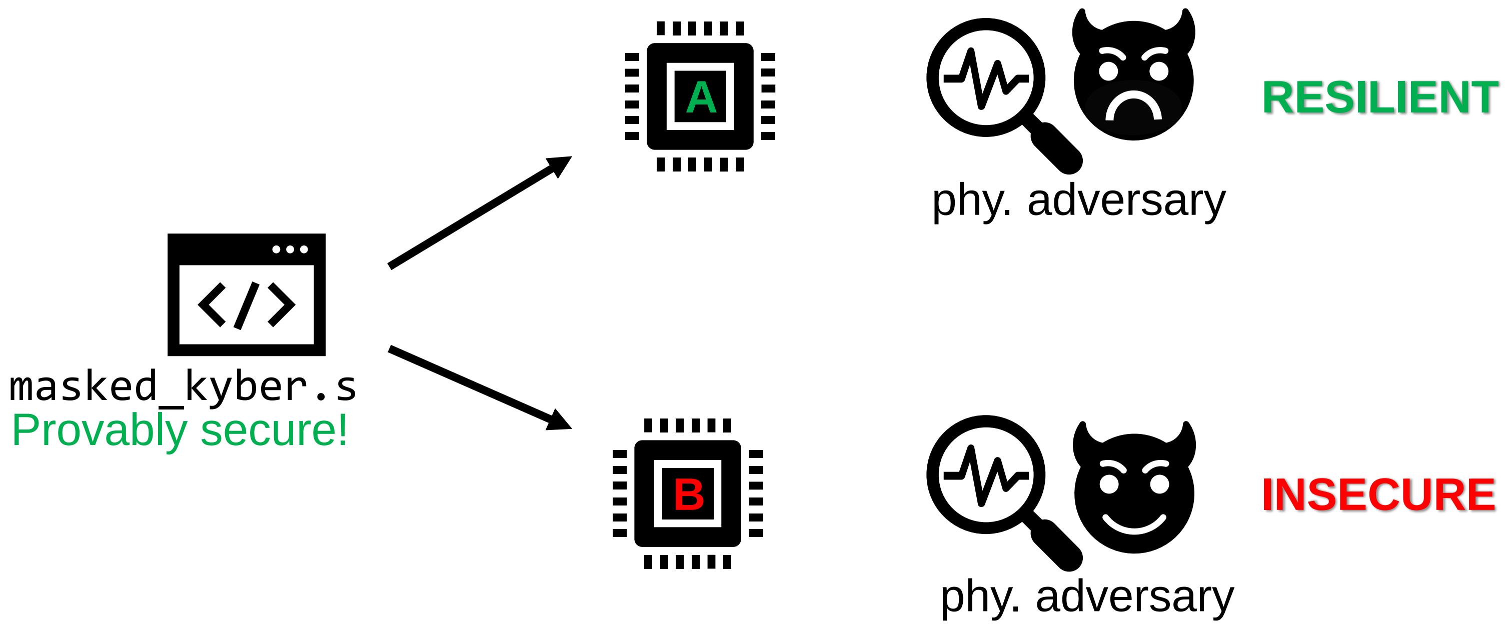
THE GAP BETWEEN PROVABLE RESILIENCE & PHYSICAL SIDE-CHANNELS
DEVICE-SPECIFIC LEAKAGE

Problem: Same program has different



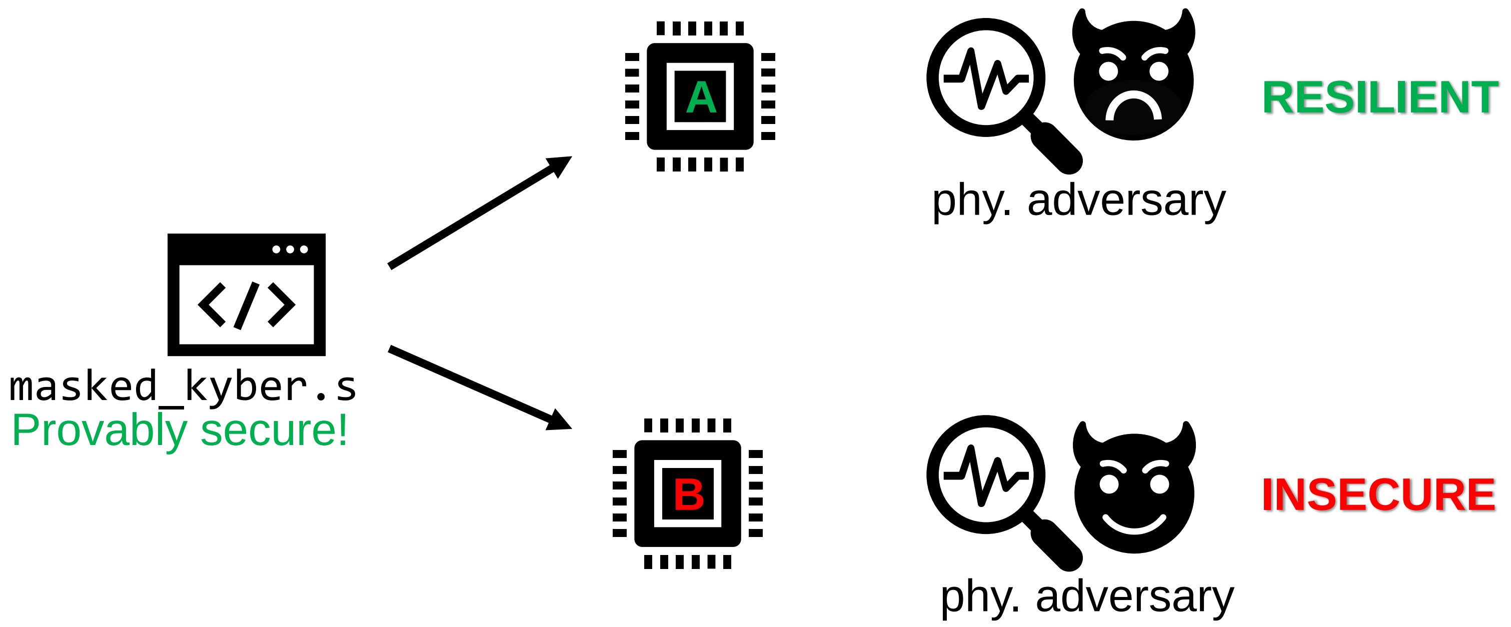
THE GAP BETWEEN PROVABLE RESILIENCE & PHYSICAL SIDE-CHANNELS
DEVICE-SPECIFIC LEAKAGE

Problem: Same program has different



THE GAP BETWEEN PROVABLE RESILIENCE & PHYSICAL SIDE-CHANNELS
DEVICE-SPECIFIC LEAKAGE

Problem: Same program has different

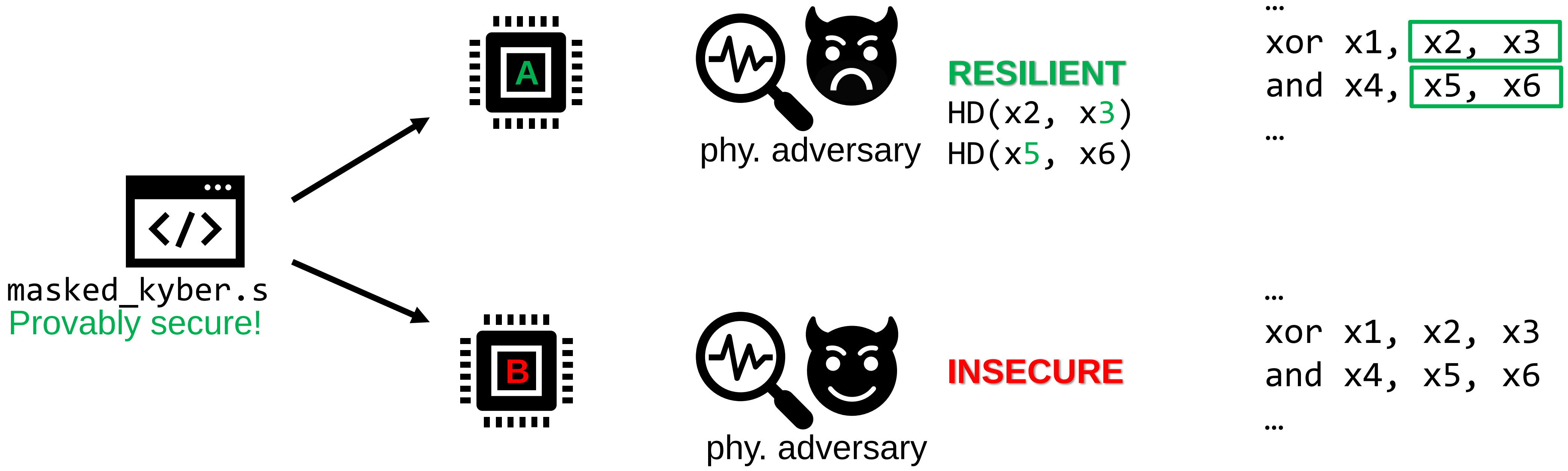


```
…
xor x1, x2, x3
and x4, x5, x6
…
```

```
…
xor x1, x2, x3
and x4, x5, x6
…
```

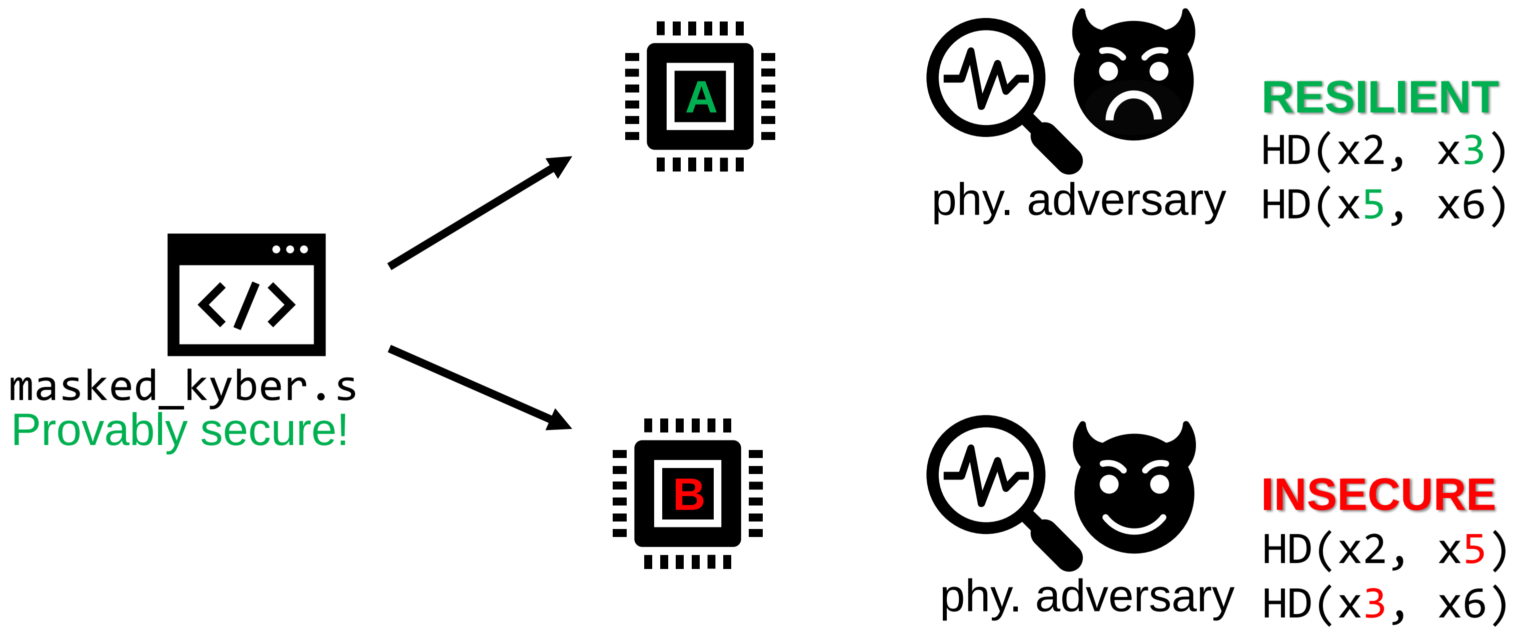
THE GAP BETWEEN PROVABLE RESILIENCE & PHYSICAL SIDE-CHANNELS
DEVICE-SPECIFIC LEAKAGE

Problem: Same program has different



THE GAP BETWEEN PROVABLE RESILIENCE & PHYSICAL SIDE-CHANNELS DEVICE-SPECIFIC LEAKAGE

Problem: Same program has different

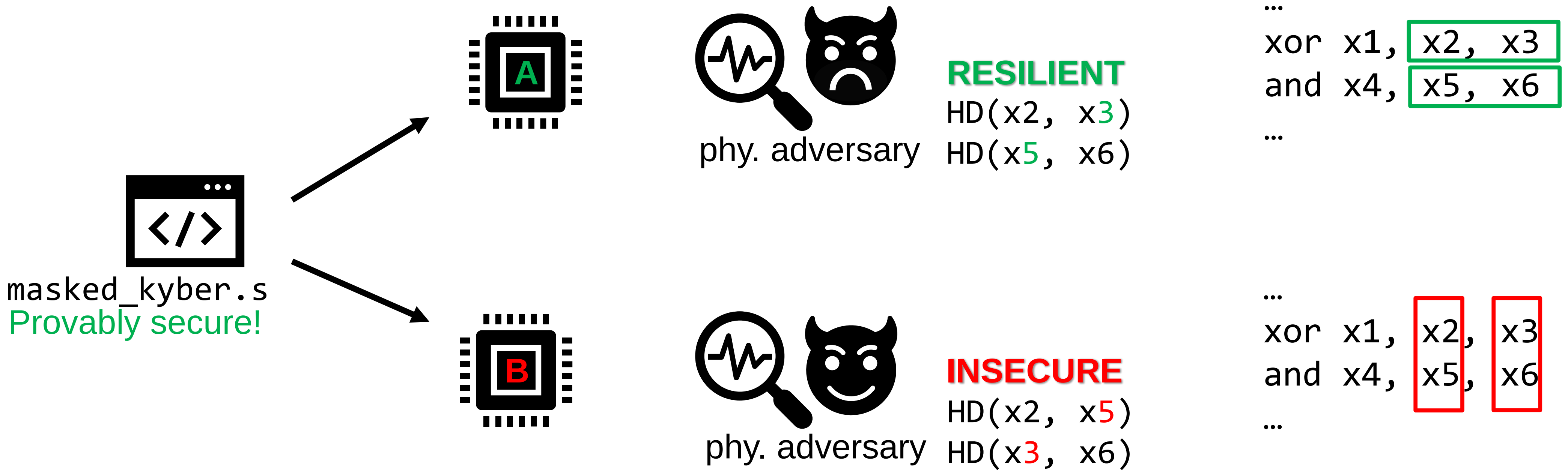


```
…
xor x1, x2, x3
and x4, x5, x6
…
```

```
…
xor x1, x2, x3
and x4, x5, x6
…
```

THE GAP BETWEEN PROVABLE RESILIENCE & PHYSICAL SIDE-CHANNELS
DEVICE-SPECIFIC LEAKAGE

Problem: Same program has different microarchitecture



```
masked_kyber.s
```

```
HD(x2, x3)
HD(x5, x6)
```

```
…
xor x1, x2, x3
and x4, x5, x6
…
```

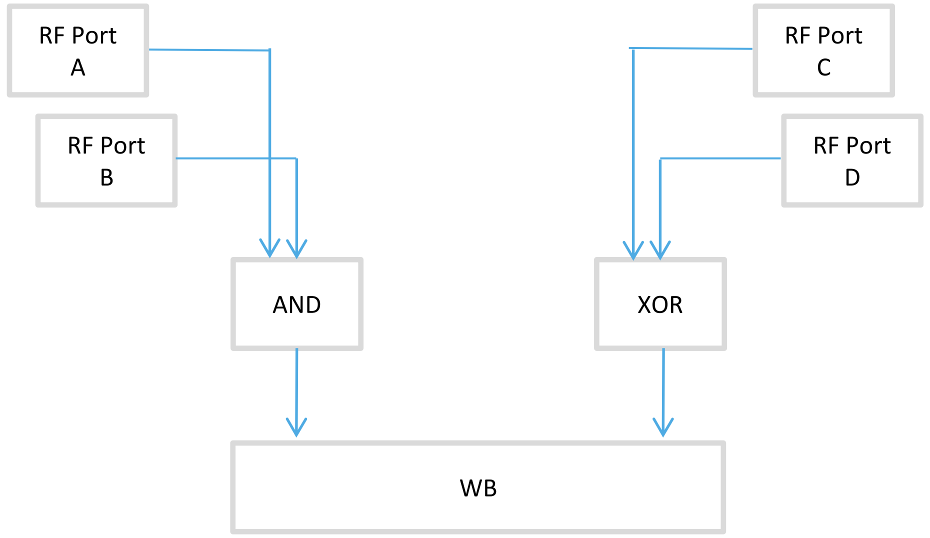
```
HD(x2, x5)
HD(x3, x6)
```

```
…
xor x1, x2, x3
and x4, x5, x6
…
```

Cause: Processor’s implementation → microarchitecture

DEVICE-SPECIFIC LEAKAGE (2) MICROARCHITECTURE

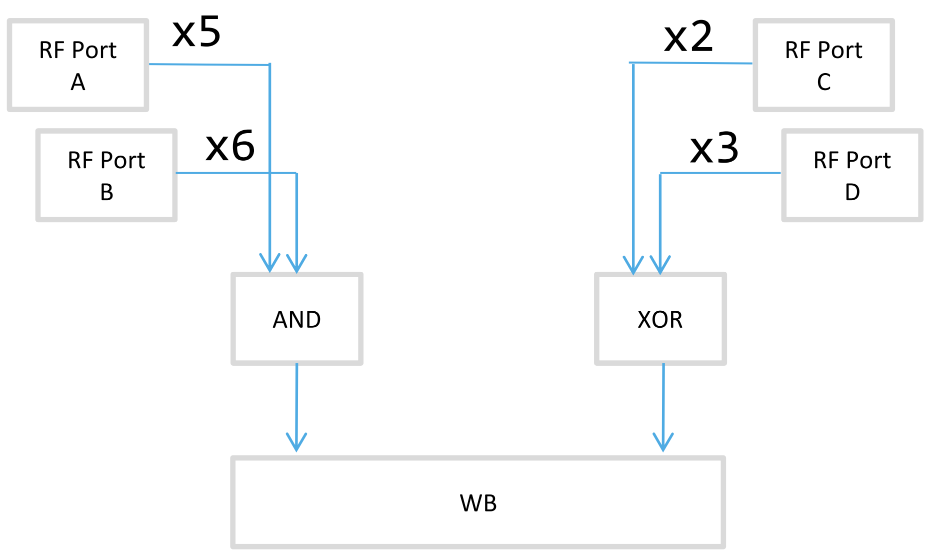
```
…
xor x1, x2, x3
and x4, x5, x6
…
```



Composable Masking Schemes in the Presence of Physical Defaults & the Robust Probing Model. Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, François-Xavier Standaert. CHES 2018.

DEVICE-SPECIFIC LEAKAGE (2)
MICROARCHITECTURE

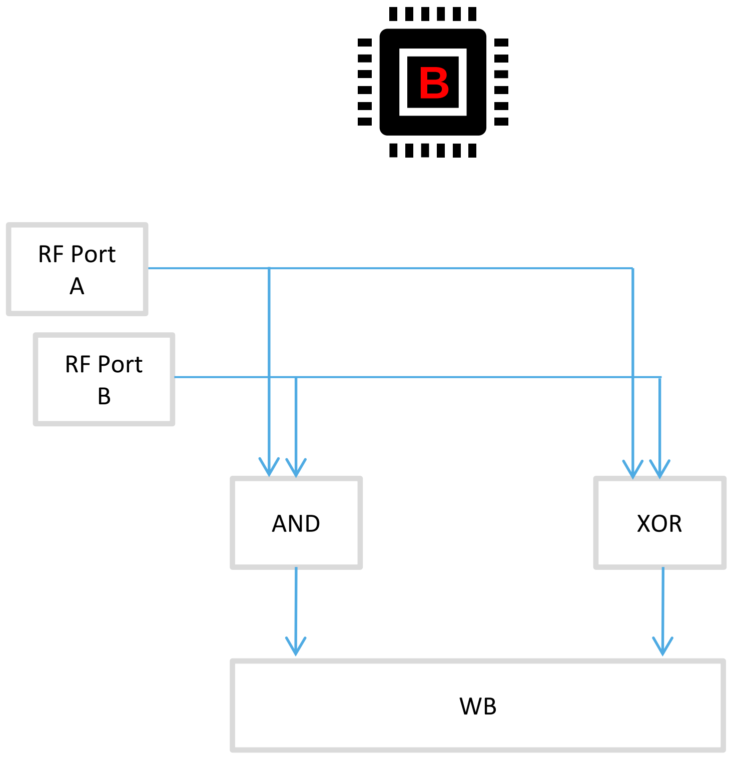
```
...
xor x1, x2, x3
and x4, x5, x6
...
```

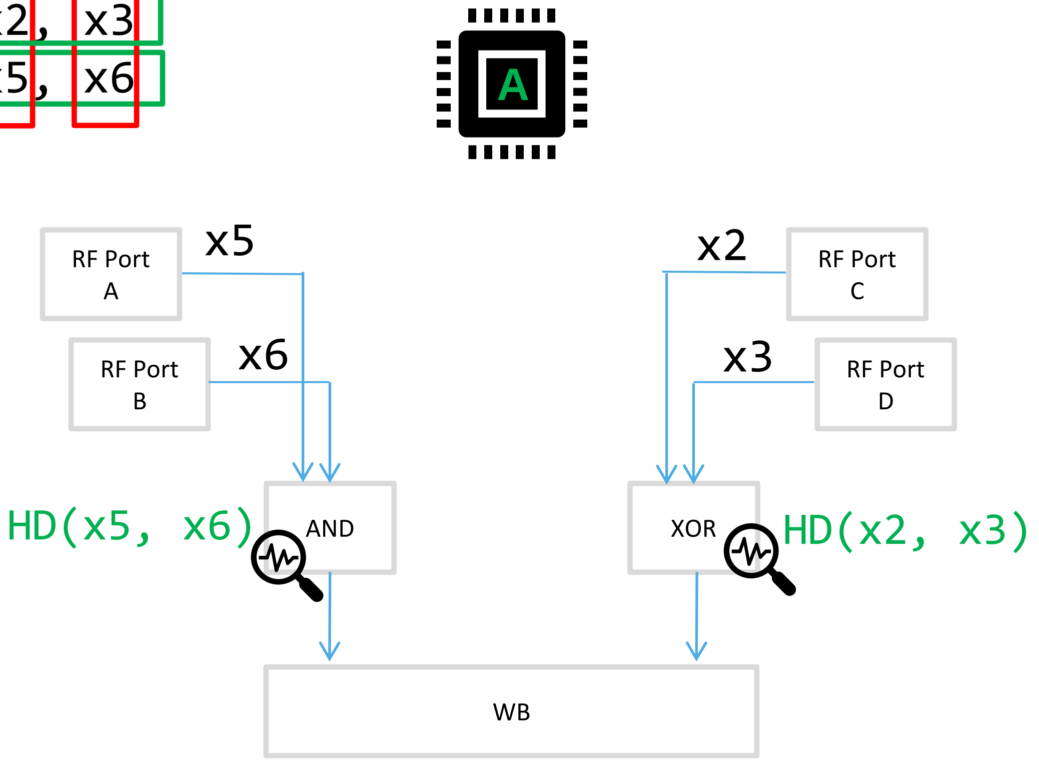


Composable Masking Schemes in the Presence of Physical Defaults & the Robust Probing Model. Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, François-Xavier Standaert. CHES 2018.

DEVICE-SPECIFIC LEAKAGE (2)
MICROARCHITECTURE

```
…
xor x1, x2, x3
and x4, x5, x6
…
```

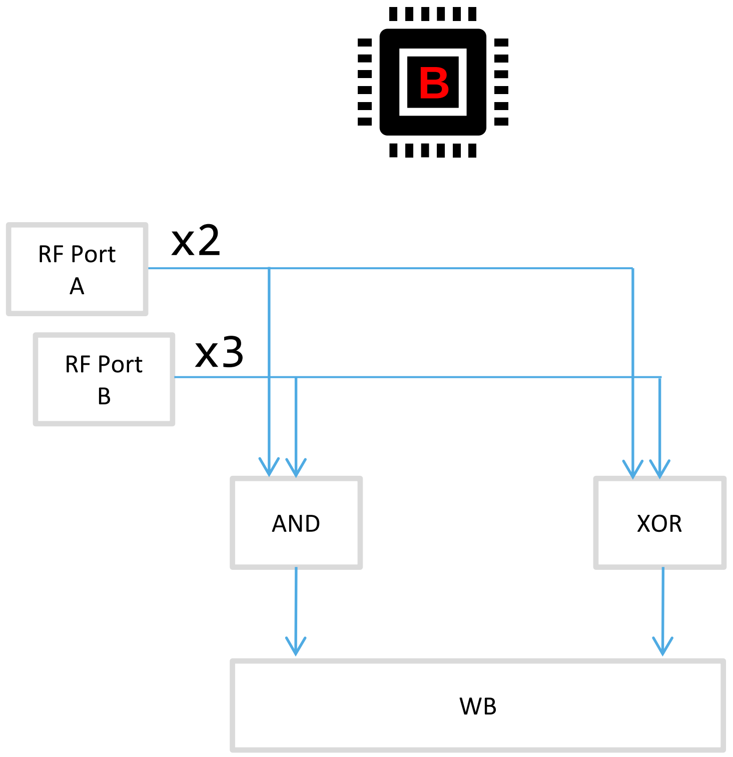


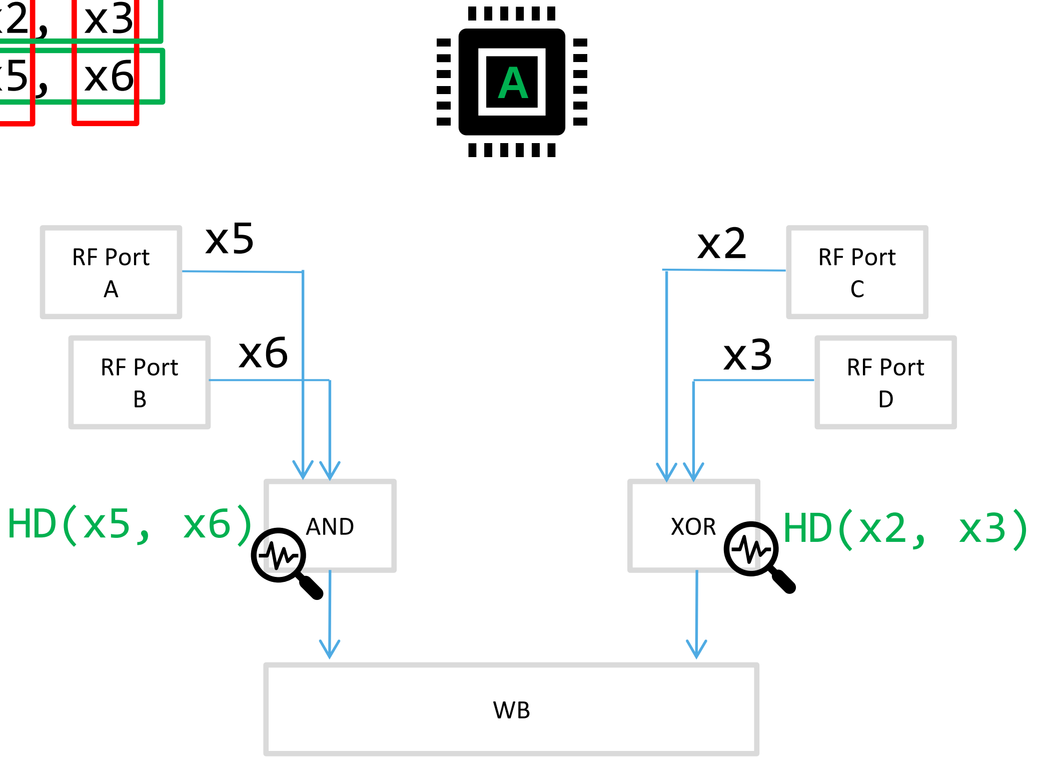


Composable Masking Schemes in the Presence of Physical Defaults & the Robust Probing Model. Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, François-Xavier Standaert. CHES 2018.

DEVICE-SPECIFIC LEAKAGE (2)
MICROARCHITECTURE

```
…
xor x1, x2, x3
and x4, x5, x6
…
```

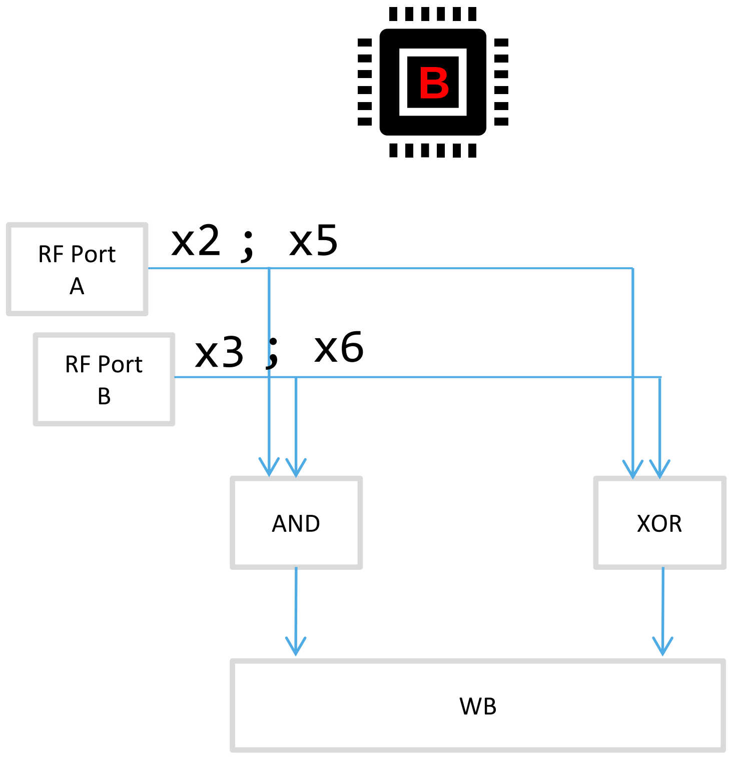


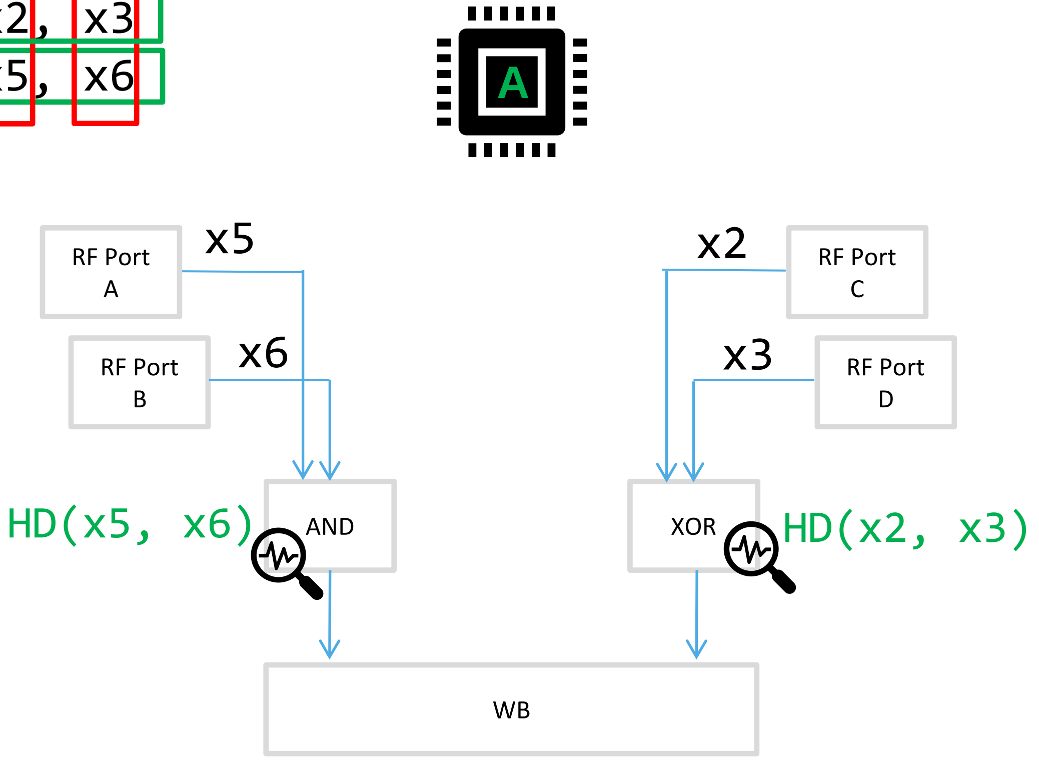


Composable Masking Schemes in the Presence of Physical Defaults & the Robust Probing Model. Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, François-Xavier Standaert. CHES 2018.

DEVICE-SPECIFIC LEAKAGE (2)
MICROARCHITECTURE

```
…
xor x1, x2, x3
and x4, x5, x6
…
```

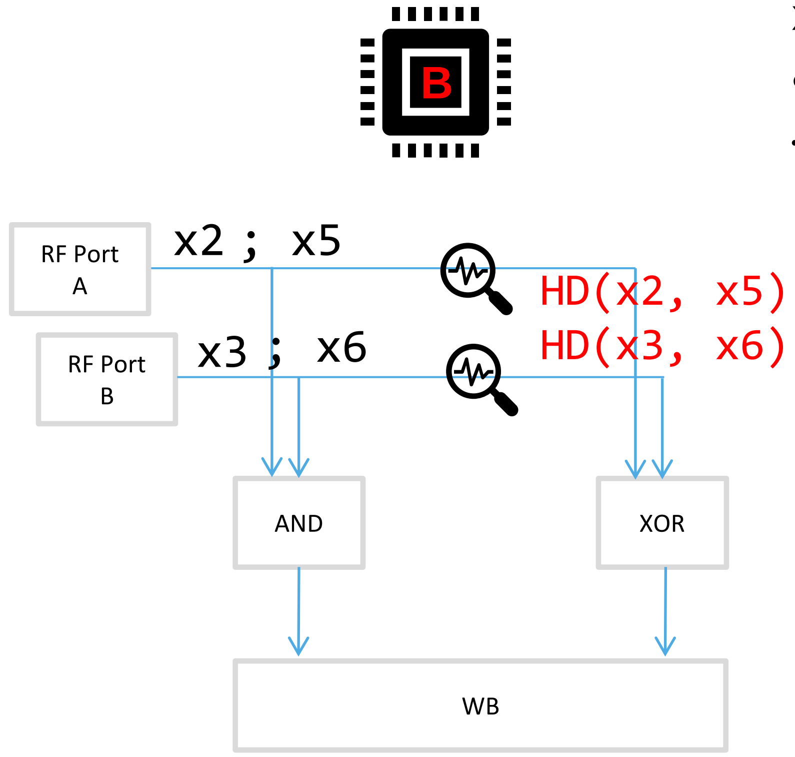


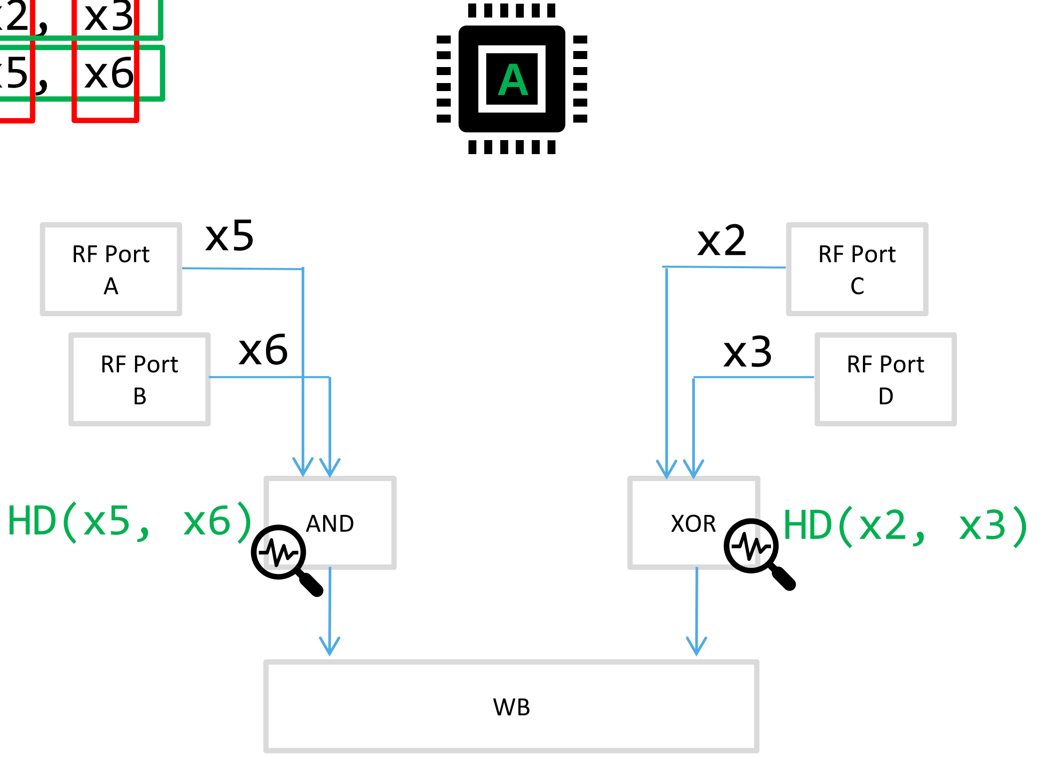


Composable Masking Schemes in the Presence of Physical Defaults & the Robust Probing Model. Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, François-Xavier Standaert. CHES 2018.

DEVICE-SPECIFIC LEAKAGE (2) MICROARCHITECTURE

```
…
xor x1, x2, x3
and x4, x5, x6
…
```

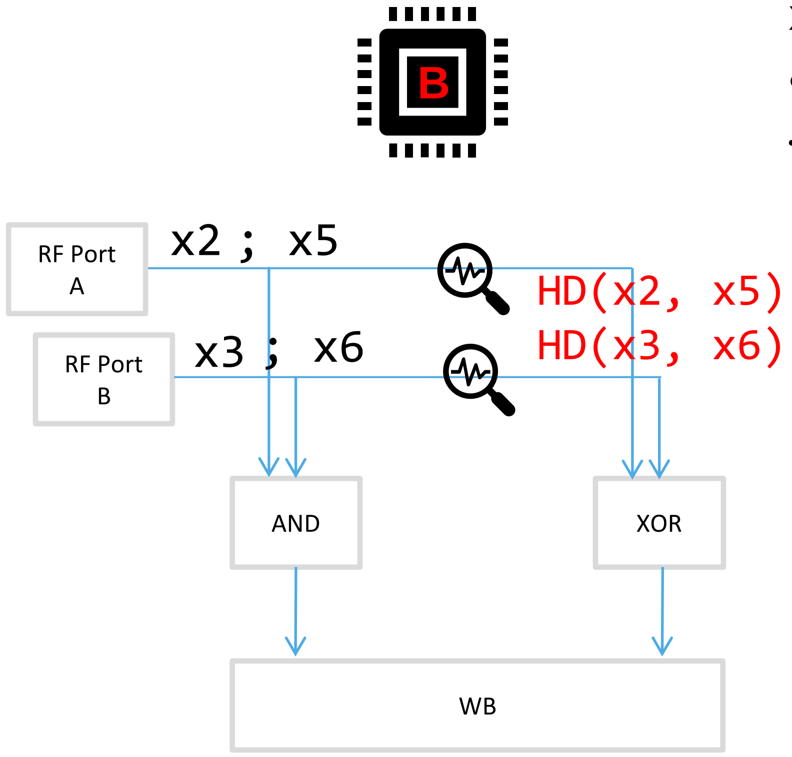


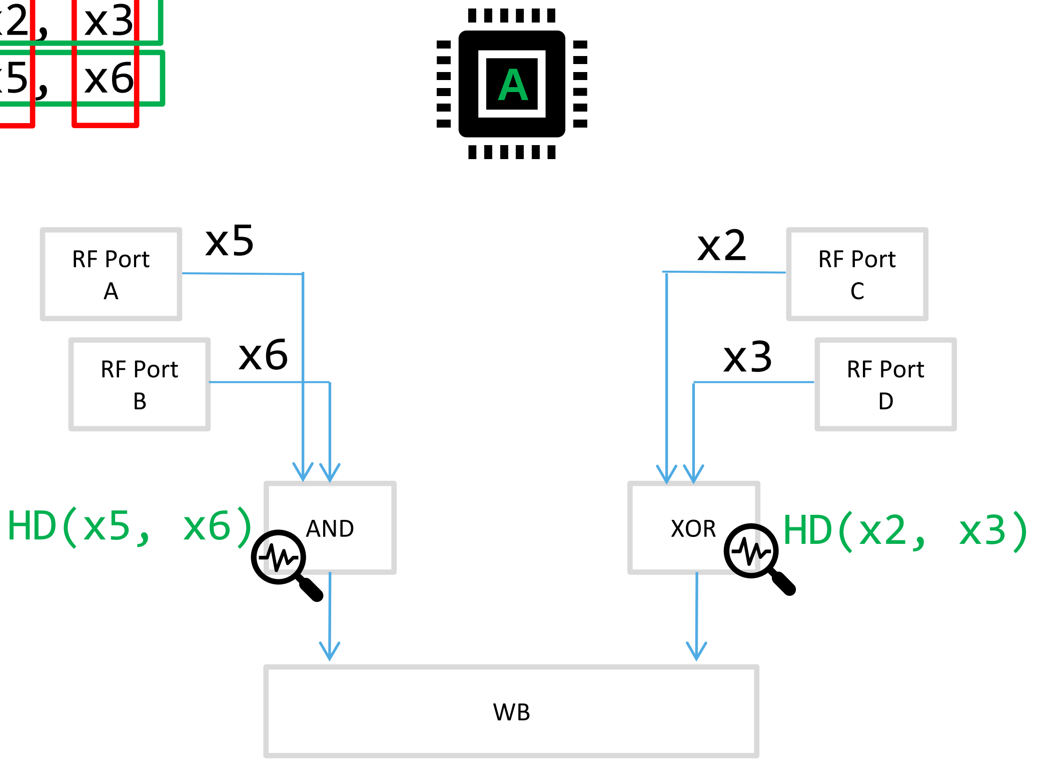


Composable Masking Schemes in the Presence of Physical Defaults & the Robust Probing Model. Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, François-Xavier Standaert. CHES 2018.

DEVICE-SPECIFIC LEAKAGE (2) MICROARCHITECTURE

```
…
xor x1, x2, x3
and x4, x5, x6
…
```

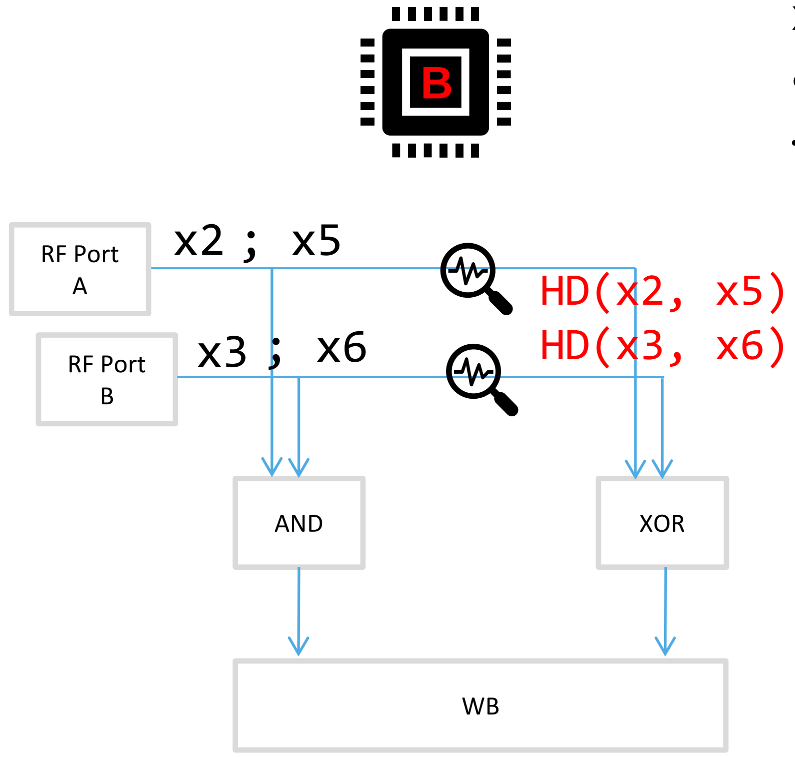


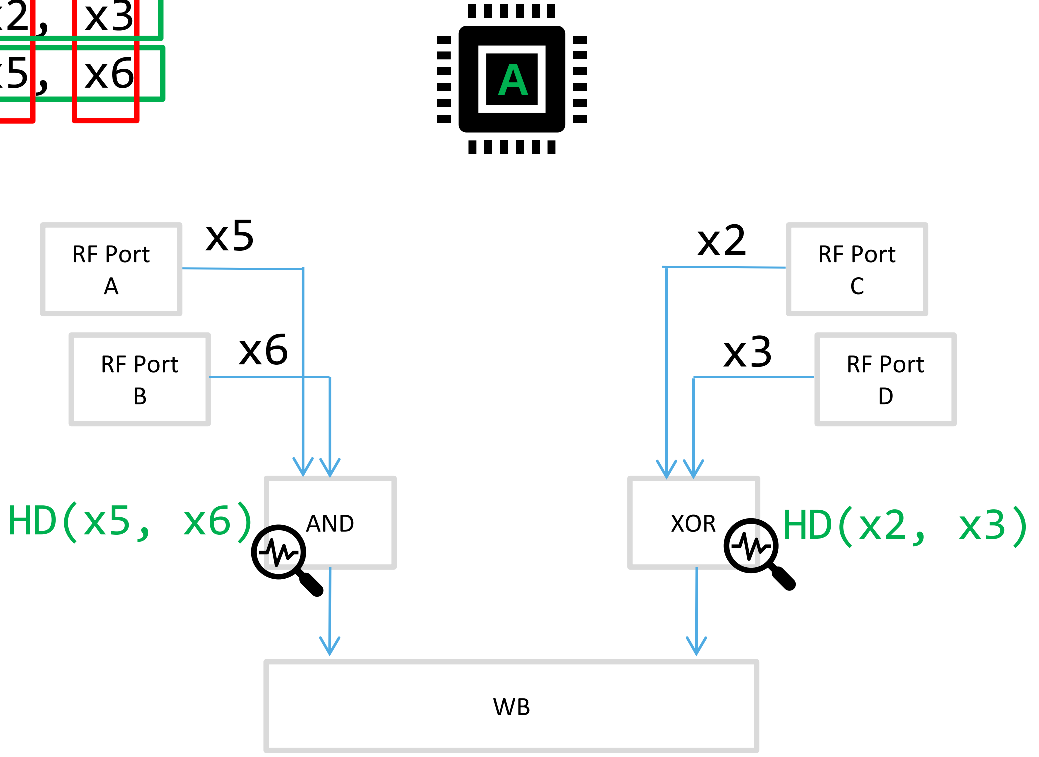


```
xor rD, rN, rM
        leak HD(rN, rM)

and rD, rN, rM
        leak HD(rN, rM)
```

DEVICE-SPECIFIC LEAKAGE (2)
MICROARCHITECTURE

```
…
xor x1, x2, x3
and x4, x5, x6
…
```





```
xor rD, rN, rM
        leak HD(rN, previous(rN))
        leak HD(rM, previous(rM))

and rD, rN, rM
        leak HD(rN, previous(rN))
        leak HD(rM, previous(rM))
```

```
xor rD, rN, rM
        leak HD(rN, rM)

and rD, rN, rM
        leak HD(rN, rM)
```

EXPLICIT LEAKAGE

- `leak( HD(value_1,  value_2) );`

- `leak(` $\underbrace{\texttt{value\_1, value\_2}}_{\texttt{Any function of terms in leak}}$ `);`

- Remainder of DSL does not expose leakage
- May effect efficiency of hardened implementations

EXPLICIT LEAKAGE

- `leak( HD(value_1,  value_2) );`

`HD(value_1,  value_2)`

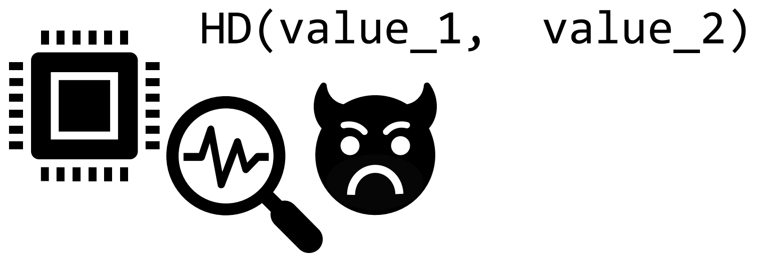
- `leak(    value_1,  value_2  );`

$\underbrace{\texttt{value\_1, value\_2}}_{\texttt{Any function of terms in leak}}$

- Remainder of DSL does not expose leakage
- May effect efficiency of hardened implementations

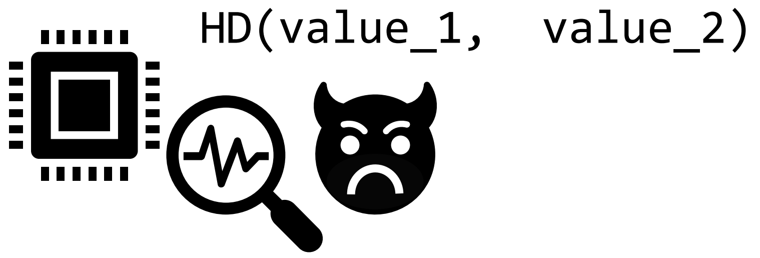
EXPLICIT LEAKAGE

- `leak( HD(value_1,  value_2) );`



- `leak(    value_1,  value_2  );`

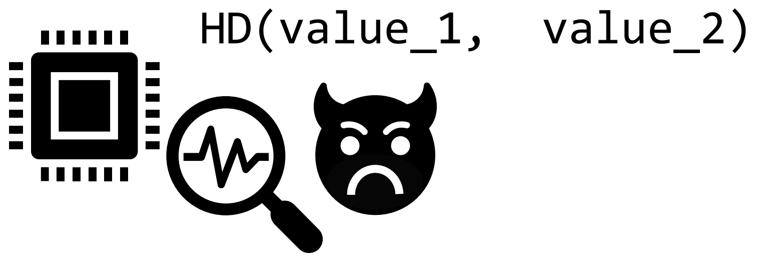
$\underbrace{\texttt{value\_1, value\_2}}_{\texttt{Any function of terms in leak}}$



- Remainder of DSL does not expose leakage
- May effect efficiency of hardened implementations

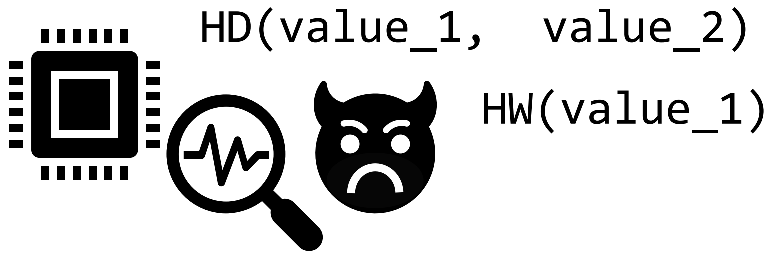
EXPLICIT LEAKAGE

- `leak( HD(value_1,  value_2) );`



- `leak(    value_1,  value_2  );`

$\underbrace{\texttt{value\_1, value\_2}}_{\texttt{Any function of terms in leak}}$



- Remainder of DSL does not expose leakage
- May effect efficiency of hardened implementations

EXPLICIT LEAKAGE

- `leak( HD(value_1,  value_2) );`

HD(value_1, value_2)

- `leak(    value_1,  value_2  );`

Any function of terms in leak

HD(value_1, value_2)

HW(value_1)

HD(MSB(value_1),
  LSB(value_2))

- Remainder of DSL does not expose leakage
- May effect efficiency of hardened implementations

EXPLICIT LEAKAGE

- `leak( HD(value_1,  value_2) );`

```
HD(value_1,  value_2)
```

- `leak(    value_1,  value_2  );`

```
Any function of terms in leak
```

```
HD(value_1,  value_2)

HW(value_1)

HD(MSB(value_1),
   LSB(value_2))

HD((value_1 >> 24),
   (value_2 & 0xFF))
```

- Remainder of DSL does not expose leakage
- May effect efficiency of hardened implementations

MODELING LEAKAGE IN GENOA (1)

- Formal leakage model in GENOA := SAIL DSL [1] + `leak` [2]

```
// see license in Listing L
// execute a XOR instruction
execute (XOR(rs2, rs1, rd)) = {
  let rs1_val = X(rs1);                 // read register rs1
  let rs2_val = X(rs2);                 // read register rs2

  let result = rs1_val ^ rs2_val;      // compute XOR operation





  X(rd) = result;                       // write result to rd
  return RETIRE_SUCCESS
}
```

[2]: **Masking in Fine-Grained Leakage Models: Construction, Implementation and Verification.** Gilles Barthe, Marc Gourjon, Benjamin Grégoire, Maximilian Orlt, Clara Paglialonga, Lars Porth. CHES 2021.

[1]: **ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS.** Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. POPL 2019.

MODELING LEAKAGE IN GENOA (1)

- Formal leakage model in GENOA := SAIL DSL [1] + leak [2]

```
// see license in Listing L
// execute a XOR instruction
execute (XOR(rs2, rs1, rd)) = {
  let rs1_val = X(rs1);             // read register rs1
  let rs2_val = X(rs2);             // read register rs2

  let result = rs1_val ^ rs2_val;   // compute XOR operation

  leak(HD(X(rs1), X(rs2)));         // leakage between operands



  X(rd) = result;                   // write result to rd
  return RETIRE_SUCCESS
}
```

[2]: **Masking in Fine-Grained Leakage Models: Construction, Implementation and Verification.** Gilles Barthe, Marc Gourjon, Benjamin Grégoire, Maximilian Orlt, Clara Paglialonga, Lars Porth. CHES 2021.

[1]: **ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS.** Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. POPL 2019.

MODELING LEAKAGE IN GENOA (1)

- Formal leakage model in GENOA := SAIL DSL [1] + leak [2]

```
// see license in Listing L
// execute a XOR instruction
execute (XOR(rs2, rs1, rd)) = {
  let rs1_val = X(rs1);                 // read register rs1
  let rs2_val = X(rs2);                 // read register rs2

  let result = rs1_val ^ rs2_val;      // compute XOR operation

  leak(HD(X(rs1), X(rs2)));            // leakage between operands
  leak(   X(rs1), X(rs2) );            // leakage between operands


  X(rd) = result;                       // write result to rd
  return RETIRE_SUCCESS
}
```

[2]: **Masking in Fine-Grained Leakage Models: Construction, Implementation and Verification.** Gilles Barthe, Marc Gourjon, Benjamin Grégoire, Maximilian Orlt, Clara Paglialonga, Lars Porth. CHES 2021.

[1]: **ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS.** Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. POPL 2019.

MODELING LEAKAGE IN GENOA (1)

- Formal leakage model in GENOA := SAIL DSL [1] + leak [2]

```
// see license in Listing L
// execute a XOR instruction
execute (XOR(rs2, rs1, rd)) = {
  let rs1_val = X(rs1);                 // read register rs1
  let rs2_val = X(rs2);                 // read register rs2

  let result = rs1_val ^ rs2_val;       // compute XOR operation

  leak(HD(X(rs1), X(rs2)));             // leakage between operands
  leak(   X(rs1), X(rs2) );             // leakage between operands
  leak(   X(rd),  result );             // transition leakage, e.g., HD

  X(rd) = result;                       // write result to rd
  return RETIRE_SUCCESS
}
```

[2]: **Masking in Fine-Grained Leakage Models: Construction, Implementation and Verification.** Gilles Barthe, Marc Gourjon, Benjamin Grégoire, Maximilian Orlt, Clara Paglialonga, Lars Porth. CHES 2021.

[1]: **ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS.** Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. POPL 2019.

MODELING LEAKAGE IN GENOA (1)

- Formal leakage model in GENOA := SAIL DSL [1] + leak [2]

```
// see license in Listing L
// execute a XOR instruction
execute (XOR(rs2, rs1, rd)) = {
  let rs1_val = X(rs1);             // read register rs1
  let rs2_val = X(rs2);             // read register rs2

  let result = rs1_val ^ rs2_val;   // compute XOR operation

  leak(HD(X(rs1), X(rs2)));         // leakage between operands
  leak(   X(rs1), X(rs2) );         // leakage between operands
  leak(   X(rd),  result );         // transition leakage, e.g., HD

  X(rd) = result;                   // write result to rd
  return RETIRE_SUCCESS
}
```

[2]: **Masking in Fine-Grained Leakage Models: Construction, Implementation and Verification.** Gilles Barthe, Marc Gourjon, Benjamin Grégoire, Maximilian Orlt, Clara Paglialonga, Lars Porth. CHES 2021.

[1]: **ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS.** Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. POPL 2019.

MODELING LEAKAGE IN GENOA (2)
LEAKAGE STATE

```
…
xor x1, x2, x3

and x4, x5, x6

…
```

```
// see license in Listing L
// execute a XOR instruction, similar for AND
execute (XOR(rs2, rs1, rd)) = {
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);

  let result = rs1_val ^ rs2_val;







  X(rd) = result;
  return RETIRE_SUCCESS
}
```

MODELING LEAKAGE IN GENOA (2)
LEAKAGE STATE

```
…
xor x1, x2, x3

and x4, x5, x6

…
```

```
// see license in Listing L
// execute a XOR instruction, similar for AND
execute (XOR(rs2, rs1, rd)) = {
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);

  let result = rs1_val ^ rs2_val;




  rf_pA = rs1_val; // leakage state to remember operand 1


  X(rd) = result;
  return RETIRE_SUCCESS
}
```

B

MODELING LEAKAGE IN GENOA (2)
LEAKAGE STATE

```
…
xor x1, x2, x3

and x4, x5, x6

…
```

```
// see license in Listing L
// execute a XOR instruction, similar for AND
execute (XOR(rs2, rs1, rd)) = {
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);

  let result = rs1_val ^ rs2_val;

  leak(   X(rs1), rf_pA); // leak of rs1 & previous rs1


  rf_pA = rs1_val; // leakage state to remember operand 1


  X(rd) = result;
  return RETIRE_SUCCESS
}
```

MODELING LEAKAGE IN GENOA (2)
LEAKAGE STATE

```
…
xor x1, x2, x3
        rf_pA = x2
and x4, x5, x6
        leak (x5, rf_pA)
…
```

```
// see license in Listing L
// execute a XOR instruction, similar for AND
execute (XOR(rs2, rs1, rd)) = {
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);

  let result = rs1_val ^ rs2_val;

  leak(   X(rs1), rf_pA); // leak of rs1 & previous rs1


  rf_pA = rs1_val; // leakage state to remember operand 1


  X(rd) = result;
  return RETIRE_SUCCESS
}
```

MODELING LEAKAGE IN GENOA (2)
LEAKAGE STATE

```
…
xor x1, x2, x3
        rf_pA = x2
and x4, x5, x6
        leak (x5, rf_pA)
…
```

```
// see license in Listing L
// execute a XOR instruction, similar for AND
execute (XOR(rs2, rs1, rd)) = {
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);

  let result = rs1_val ^ rs2_val;

  leak(   X(rs1), rf_pA); // leak of rs1 & previous rs1
  leak(   X(rs2), rf_pB);

  rf_pA = rs1_val; // leakage state to remember operand 1
  rf_pB = rs2_val;

  X(rd) = result;
  return RETIRE_SUCCESS
}
```

MODELING LEAKAGE IN GENOA (2)
LEAKAGE STATE

```
…
xor x1, x2, x3
        rf_pA = x2
and x4, x5, x6
        leak (x5, rf_pA)
…
```

B

```
// see license in Listing L
// execute a XOR instruction, similar for AND
execute (XOR(rs2, rs1, rd)) = {
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);

  let result = rs1_val ^ rs2_val;

  leak(   X(rs1), rf_pA); // leak of rs1 & previous rs1
  leak(   X(rs2), rf_pB);

  rf_pA = rs1_val; // leakage state to remember operand 1
  rf_pB = rs2_val;

  X(rd) = result;
  return RETIRE_SUCCESS
}
```

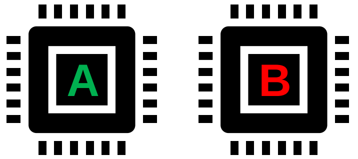
A

```
// see license in Listing L
// execute a XOR instruction
execute (XOR(rs2, rs1, rd)) = {
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);

  let result = rs1_val ^ rs2_val;

  leak(   X(rs1), X(rs2) );




  X(rd) = result;
  return RETIRE_SUCCESS
}
```

MODELING LEAKAGE IN GENOA (3)
CONTRACT

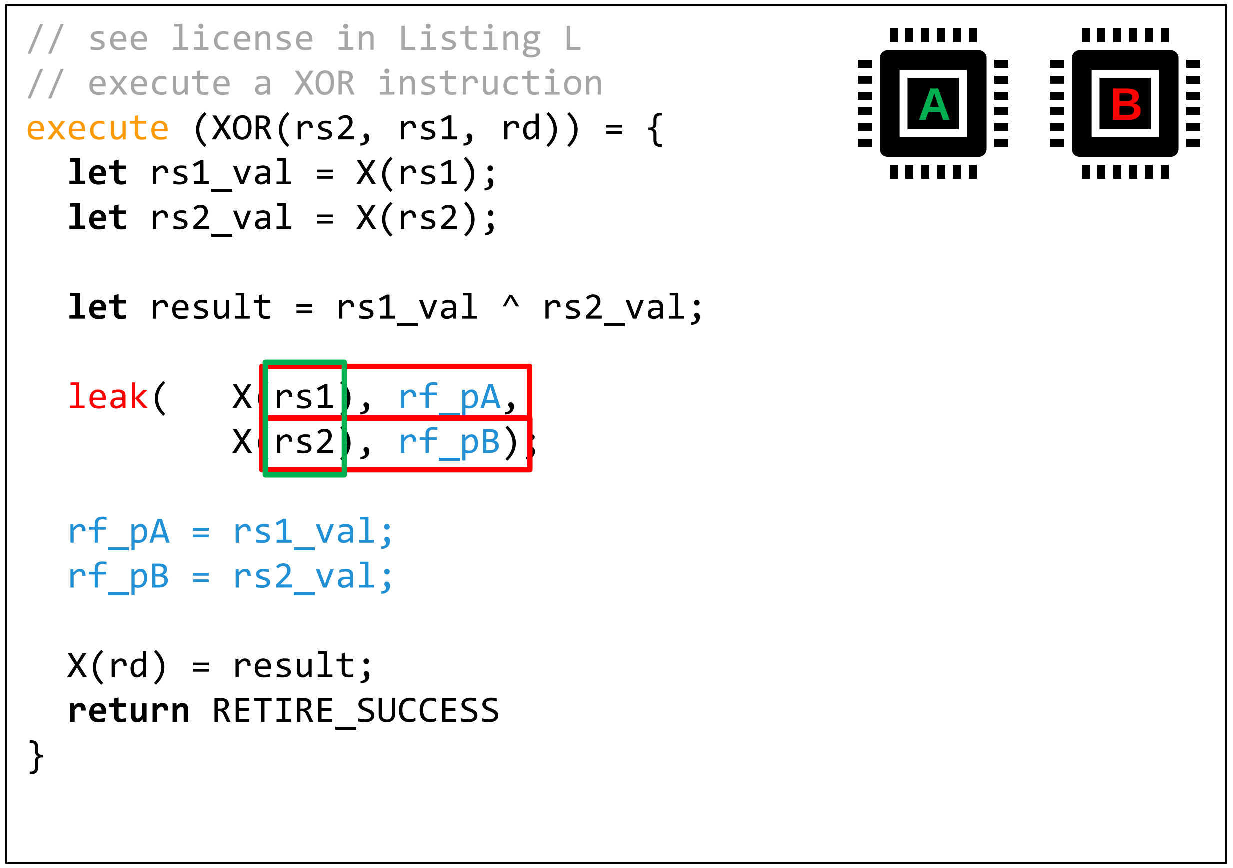
- One **contract** for many processors

```
// see license in Listing L
// execute a XOR instruction
execute (XOR(rs2, rs1, rd)) = {
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);

  let result = rs1_val ^ rs2_val;

  leak(   X(rs1), rf_pA,
          X(rs2), rf_pB);

  rf_pA = rs1_val;
  rf_pB = rs2_val;

  X(rd) = result;
  return RETIRE_SUCCESS
}
```

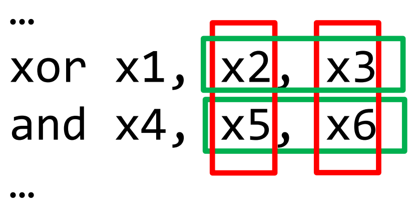


MODELING LEAKAGE IN GENOA (3) CONTRACT

- One **contract** for many processors



```
// see license in Listing L
// execute a XOR instruction
execute (XOR(rs2, rs1, rd)) = {
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);

  let result = rs1_val ^ rs2_val;

  leak(    X(rs1), rf_pA,
           X(rs2), rf_pB);

  rf_pA = rs1_val;
  rf_pB = rs2_val;

  X(rd) = result;
  return RETIRE_SUCCESS
}
```



```
…
xor x1, x2, x3
and x4, x5, x6
…
```

MODELING LEAKAGE IN GENOA (3)
CONTRACT

- One **contract** for many processors

```
// see license in Listing L
// execute a XOR instruction
execute (XOR(rs2, rs1, rd)) = {
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);

  let result = rs1_val ^ rs2_val;

  leak(    X(rs1), rf_pA,
           X(rs2), rf_pB);

  rf_pA = rs1_val;
  rf_pB = rs2_val;

  X(rd) = result;
  return RETIRE_SUCCESS
}
```

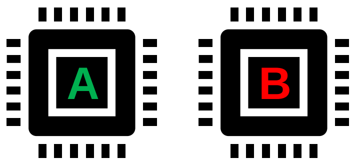
```
…
xor x1, x2, x3
and x4, x5, x6
…
```

```
…
xor x1, x2, x3
xor x0, x0, x0
and x4, x5, x6
…
```

MODELING LEAKAGE IN GENOA (3)
CONTRACT

- One **contract** for many processors

```
// see license in Listing L
// execute a XOR instruction
execute (XOR(rs2, rs1, rd)) = {
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);

  let result = rs1_val ^ rs2_val;

  leak(    X(rs1), rf_pA,
           X(rs2), rf_pB);

  rf_pA = rs1_val;
  rf_pB = rs2_val;

  X(rd) = result;
  return RETIRE_SUCCESS
}
```



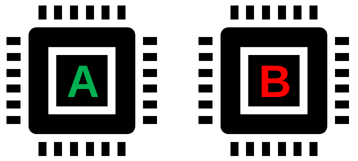
```
…
xor x1, x2, x3
and x4, x5, x6
…
```

```
…
xor x1, x2, x3
xor x0, x0, x0
and x4, x5, x6
…
```

MODELING LEAKAGE IN GENOA (3)
CONTRACT

- One **contract** for many processors

```
// see license in Listing L
// execute a XOR instruction
execute (XOR(rs2, rs1, rd)) = {
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);

  let result = rs1_val ^ rs2_val;

  leak(    X(rs1), rf_pA,
           X(rs2), rf_pB);

  rf_pA = rs1_val;
  rf_pB = rs2_val;

  X(rd) = result;
  return RETIRE_SUCCESS
}
```



```
…
xor x1, x2, x3
and x4, x5, x6
…
```

```
…
xor x1, x2, x3
xor x0, x0, x0
and x4, x5, x6
…
```

GENOA POWER CONTRACT

- Contract enables to execute entire programs symbolically
- See License in Listing L

```
// execute a decoded instruction
function clause execute (RTYPE(rs2, rs1, rd, op)) = {
  let rs1_val = X(rs1);                   // read register rs1
  let rs2_val = X(rs2);

  common_leakage(rs1_val, rs2_val);

  let result = match op {                 // match-case
    RISCV_ADD  => rs1_val + rs2_val,      // compute ADD operation
           ...
    RISCV_AND  => rs1_val & rs2_val,
  };

  overwrite_leakage(rd, result);

  X(rd) = result;                         // write result to rd
  return RETIRE_SUCCESS
}
```

GENOA POWER CONTRACT

- Contract enables to execute entire programs symbolically
- See License in Listing L

```
// decode or encode an ADD instruction
// add rd rs1 rs2 ==> RTYPE(rs2, rs1, rd, RISCV_ADD)
mapping clause encdec = RTYPE(rs2, rs1, rd, RISCV_ADD)
  <-> 0b0000000 @ rs2 @ rs1 @ 0b000 @ rd @ 0b0110011

// execute a decoded instruction
function clause execute (RTYPE(rs2, rs1, rd, op)) = {
  let rs1_val = X(rs1);                   // read register rs1
  let rs2_val = X(rs2);

  common_leakage(rs1_val, rs2_val);

  let result = match op {                 // match-case
    RISCV_ADD  => rs1_val + rs2_val,      // compute ADD operation
         ...
    RISCV_AND  => rs1_val & rs2_val,
  };

  overwrite_leakage(rd, result);

  X(rd) = result;                         // write result to rd
  return RETIRE_SUCCESS
}
```

GENOA POWER CONTRACT

- Contract enables to execute entire programs symbolically
- See License in Listing L

```
// decode or encode an ADD instruction
// add rd rs1 rs2 ==> RTYPE(rs2, rs1, rd, RISCV_ADD)
mapping clause encdec = RTYPE(rs2, rs1, rd, RISCV_ADD)
  <-> 0b0000000 @ rs2 @ rs1 @ 0b000 @ rd @ 0b0110011

// execute a decoded instruction
function clause execute (RTYPE(rs2, rs1, rd, op)) = {
  let rs1_val = X(rs1);                    // read register rs1
  let rs2_val = X(rs2);

  common_leakage(rs1_val, rs2_val);

  let result = match op {                  // match-case
    RISCV_ADD  => rs1_val + rs2_val,       // compute ADD operation
          ...
    RISCV_AND  => rs1_val & rs2_val,
  };

  overwrite_leakage(rd, result);

  X(rd) = result;                          // write result to rd
  return RETIRE_SUCCESS
}
```

```
function common_leakage(rs1_val, rs2_val) = {
  leak(rs1_val, rs2_val, rf_pA, rf_pB,
  mem_last_addr, mem_last_read);
  rf_pA = rs1_val;
  rf_pB = rs2_val;   /* update read ports      */
  mem_last_read = 0; /* clear data memory port */
}
```

GENOA POWER CONTRACT

- Contract enables to execute entire programs symbolically
- See License in Listing L

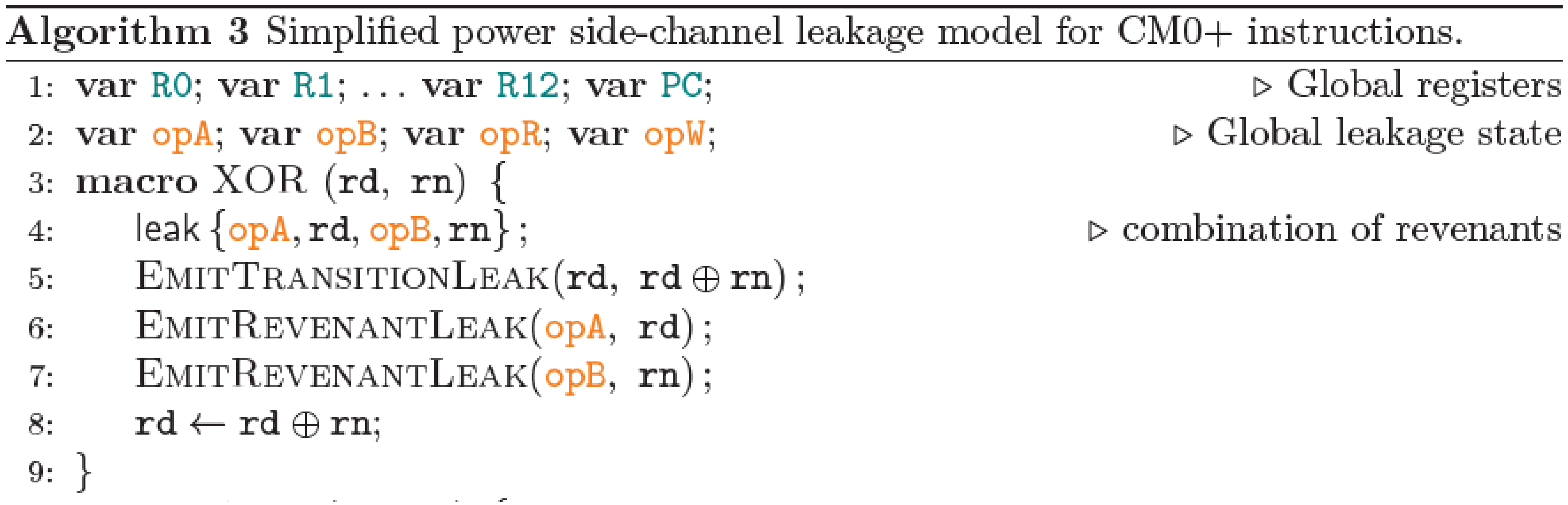
```
function step_ibex (op : bits(32)) -> bool = {
  nextPC = PC + 4;

  let instruction = encdec(op);
  let ret = execute(instruction);

  let success : bool =
    match ret {
      RETIRE_SUCCESS => true,
      RETIRE_FAIL => false
    };
  tick_pc();
  return success
}

function common_leakage(rs1_val, rs2_val) = {
  leak(rs1_val, rs2_val, rf_pA, rf_pB,
  mem_last_addr, mem_last_read);
  rf_pA = rs1_val;
  rf_pB = rs2_val;   /* update read ports      */
  mem_last_read = 0; /* clear data memory port */
}
```

```
// decode or encode an ADD instruction
// add rd rs1 rs2 ==> RTYPE(rs2, rs1, rd, RISCV_ADD)
mapping clause encdec = RTYPE(rs2, rs1, rd, RISCV_ADD)
  <-> 0b0000000 @ rs2 @ rs1 @ 0b000 @ rd @ 0b0110011

// execute a decoded instruction
function clause execute (RTYPE(rs2, rs1, rd, op)) = {
  let rs1_val = X(rs1);                 // read register rs1
  let rs2_val = X(rs2);

  common_leakage(rs1_val, rs2_val);

  let result = match op {               // match-case
    RISCV_ADD  => rs1_val + rs2_val,  // compute ADD operation
          ...
    RISCV_AND  => rs1_val & rs2_val,
  };

  overwrite_leakage(rd, result);

  X(rd) = result;                       // write result to rd
  return RETIRE_SUCCESS
}
```

GENOA VS. IL

- Models for scVerif written in IL
  - scVerif does not (yet) support Genoa

- Important differences
  - No bitvectors
  - Hardcoded assembly frontend
    - No decoding of opcodes
    - No step function
    - Symbolic addresses



Algorithm 3 Simplified power side-channel leakage model for CM0+ instructions.

```
var R0; var R1; ... var R12; var PC;                ▷ Global registers
var opA; var opB; var opR; var opW;                 ▷ Global leakage state
macro XOR (rd, rn) {
    leak {opA, rd, opB, rn} ;                       ▷ combination of revenants
    EmitTransitionLeak(rd, rd ⊕ rn) ;
    EmitRevenantLeak(opA, rd) ;
    EmitRevenantLeak(opB, rn) ;
    rd ← rd ⊕ rn;
}
```

MODEL-BASED SECURITY ASSESSMENT

masked_program.s

MODEL-BASED SECURITY ASSESSMENT

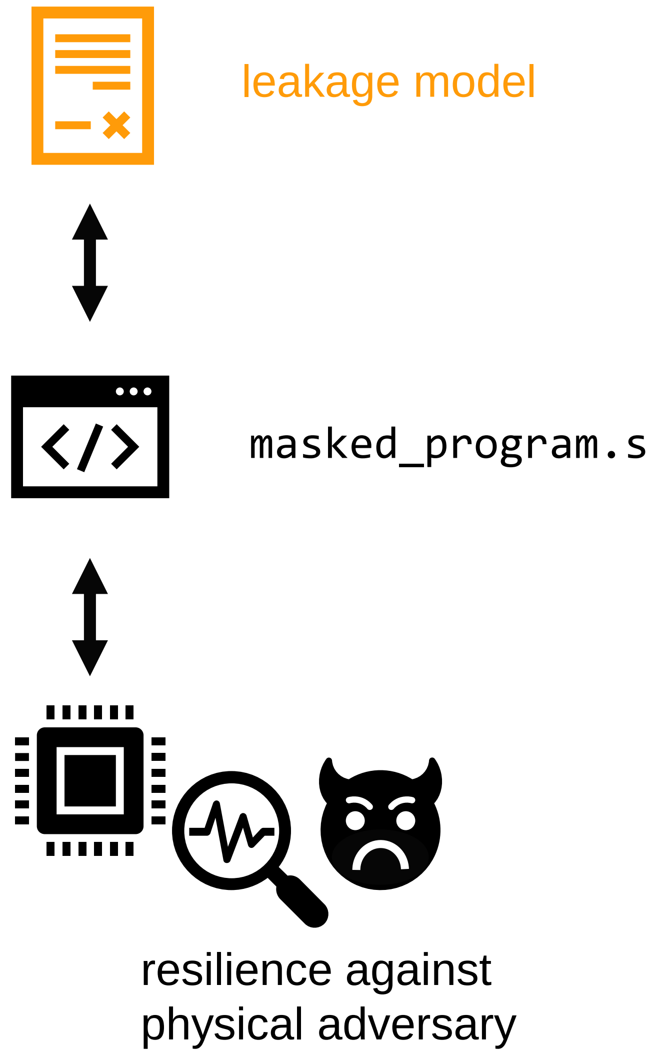
leakage model

Verify side-channel security

`masked_program.s`

MODEL-BASED SECURITY ASSESSMENT

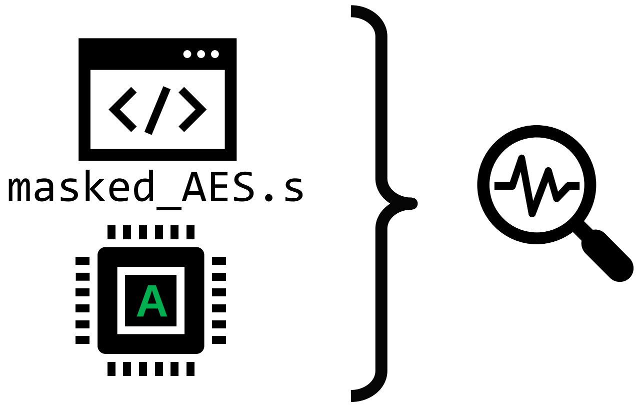
Verify side-channel security



Goal: side-channel security for physical execution on CPU

COMPLETENESS OF LEAKAGE MODELS
EMPIRICAL APPROACH

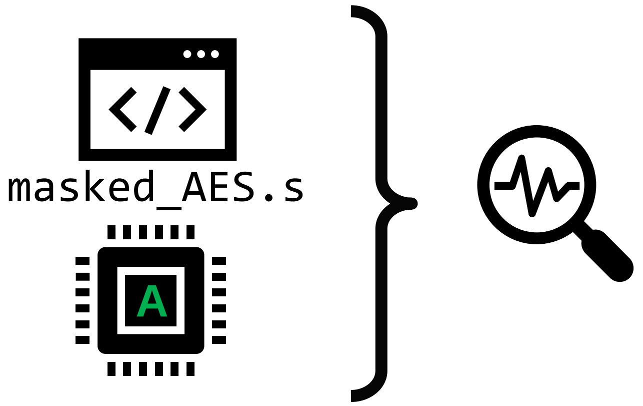
- Write test, measure, eat, sleep, repeat



```
…
xor x1, x2, x3
and x4, x5, x6
…
```

COMPLETENESS OF LEAKAGE MODELS
EMPIRICAL APPROACH

- Write test, measure, eat, sleep, repeat

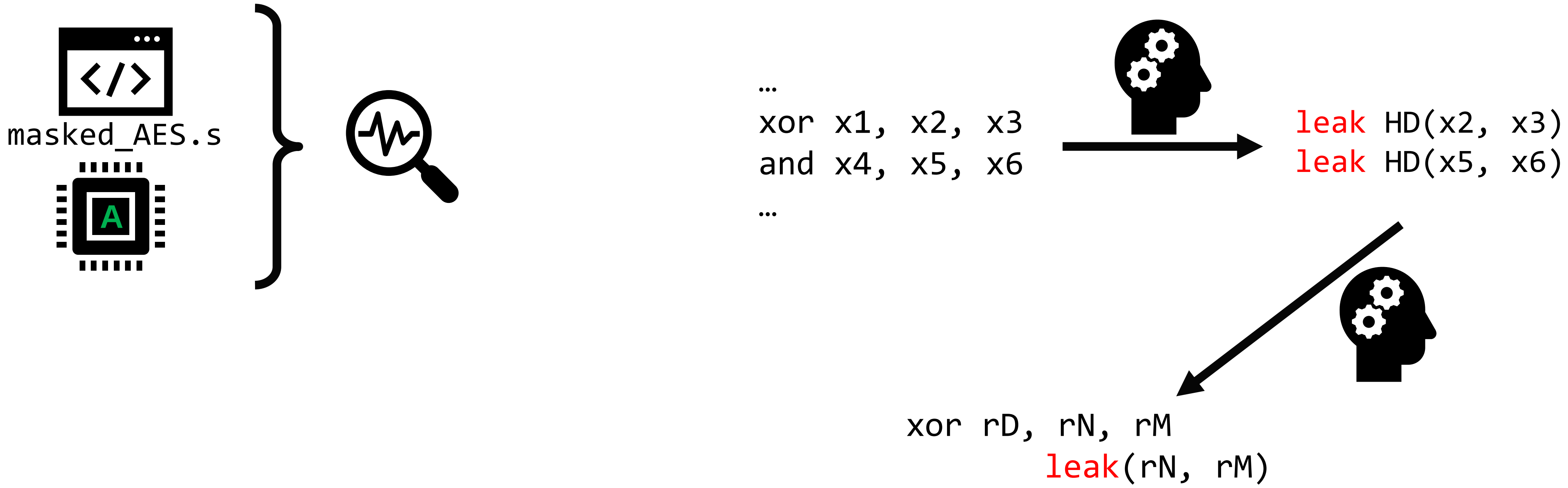


```
…
xor x1, x2, x3
and x4, x5, x6
…
```

```
leak HD(x2, x3)
leak HD(x5, x6)
```

COMPLETENESS OF LEAKAGE MODELS
EMPIRICAL APPROACH

- Write test, measure, eat, sleep, repeat



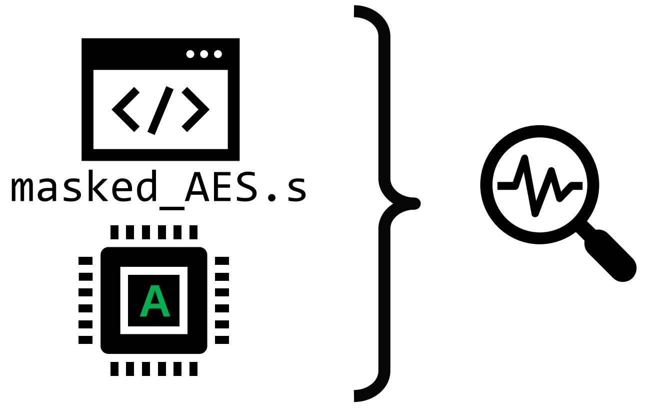
```
…
xor x1, x2, x3
and x4, x5, x6
…
```

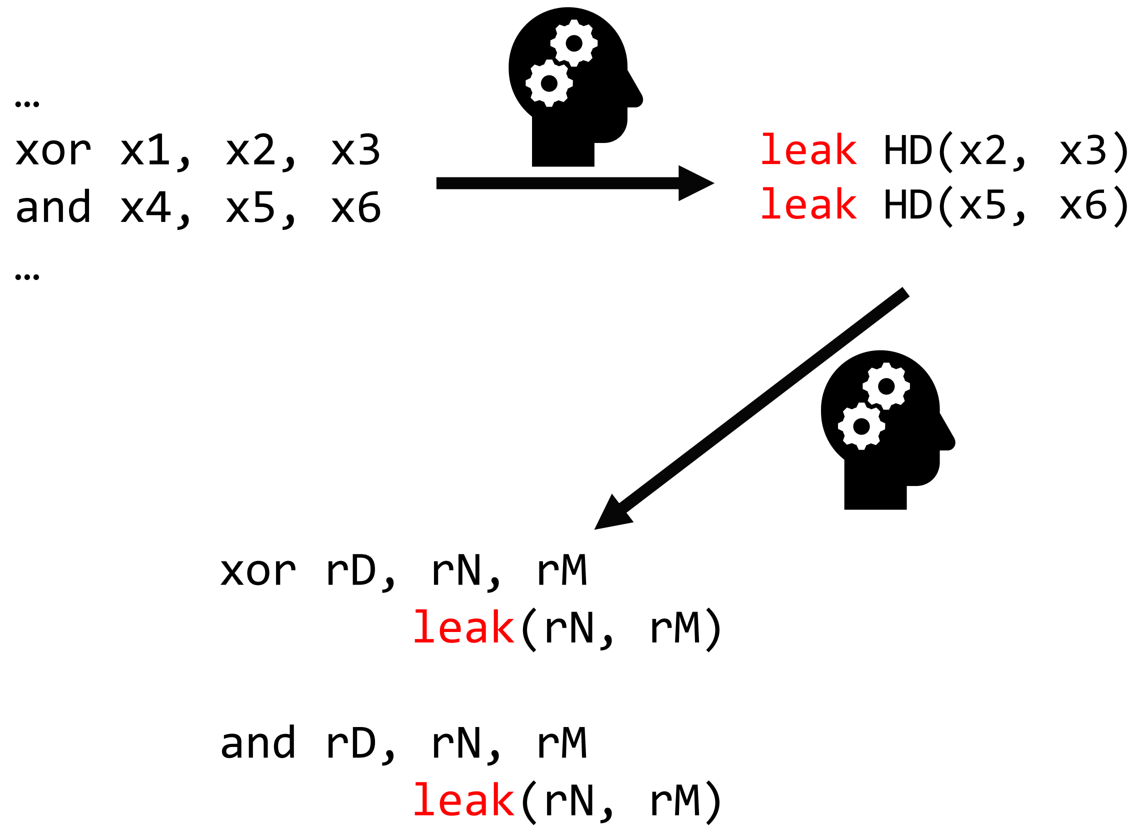
```
leak HD(x2, x3)
leak HD(x5, x6)
```

```
xor rD, rN, rM
        leak(rN, rM)
```

COMPLETENESS OF LEAKAGE MODELS
EMPIRICAL APPROACH

- Write test, measure, eat, sleep, repeat





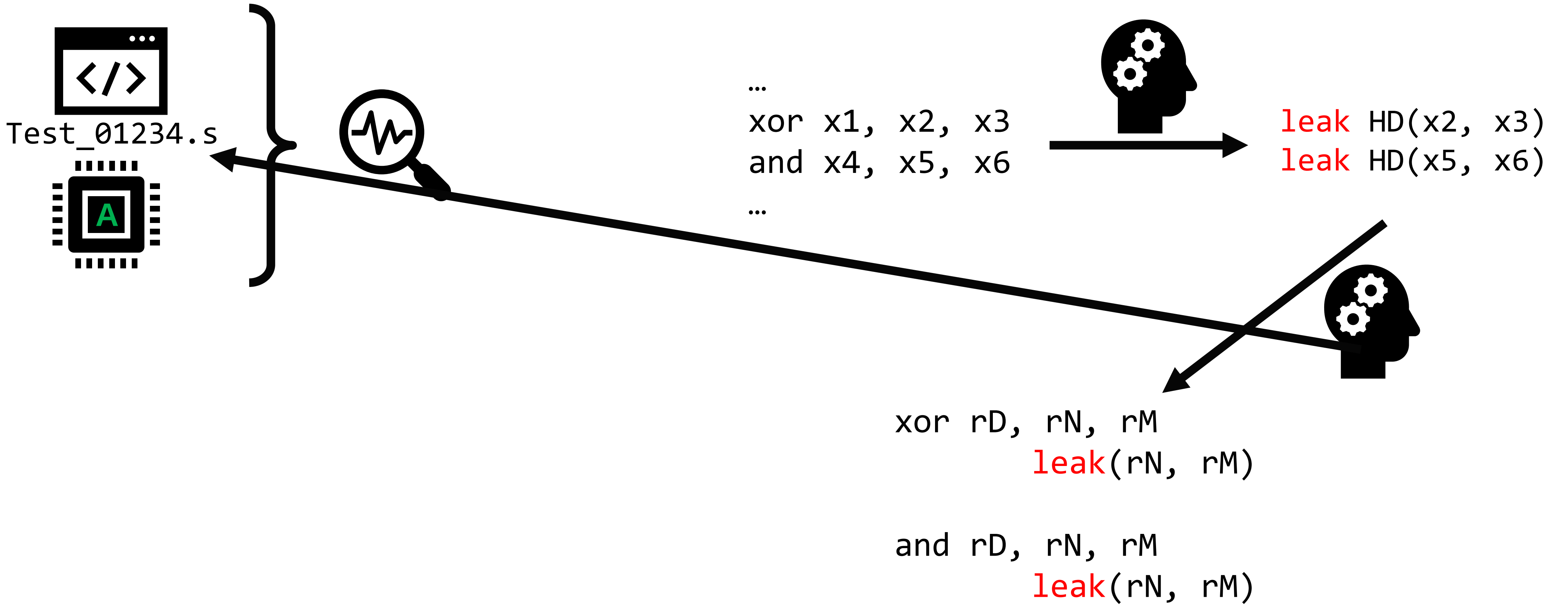
```
…
xor x1, x2, x3
and x4, x5, x6
…
```

```
leak HD(x2, x3)
leak HD(x5, x6)
```

```
xor rD, rN, rM
        leak(rN, rM)

and rD, rN, rM
        leak(rN, rM)
```

COMPLETENESS OF LEAKAGE MODELS
EMPIRICAL APPROACH

- Write test, measure, eat, sleep, repeat



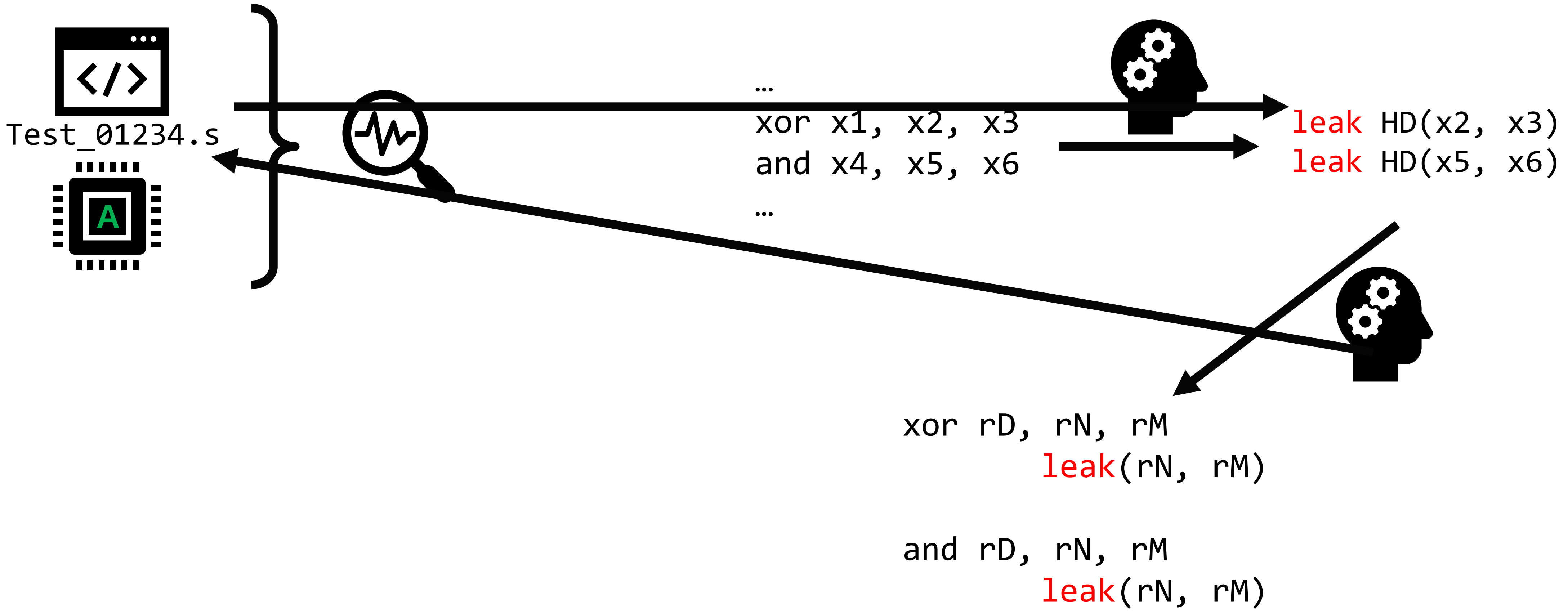
```
…
xor x1, x2, x3
and x4, x5, x6
…
```

```
leak HD(x2, x3)
leak HD(x5, x6)
```

```
xor rD, rN, rM
        leak(rN, rM)

and rD, rN, rM
        leak(rN, rM)
```

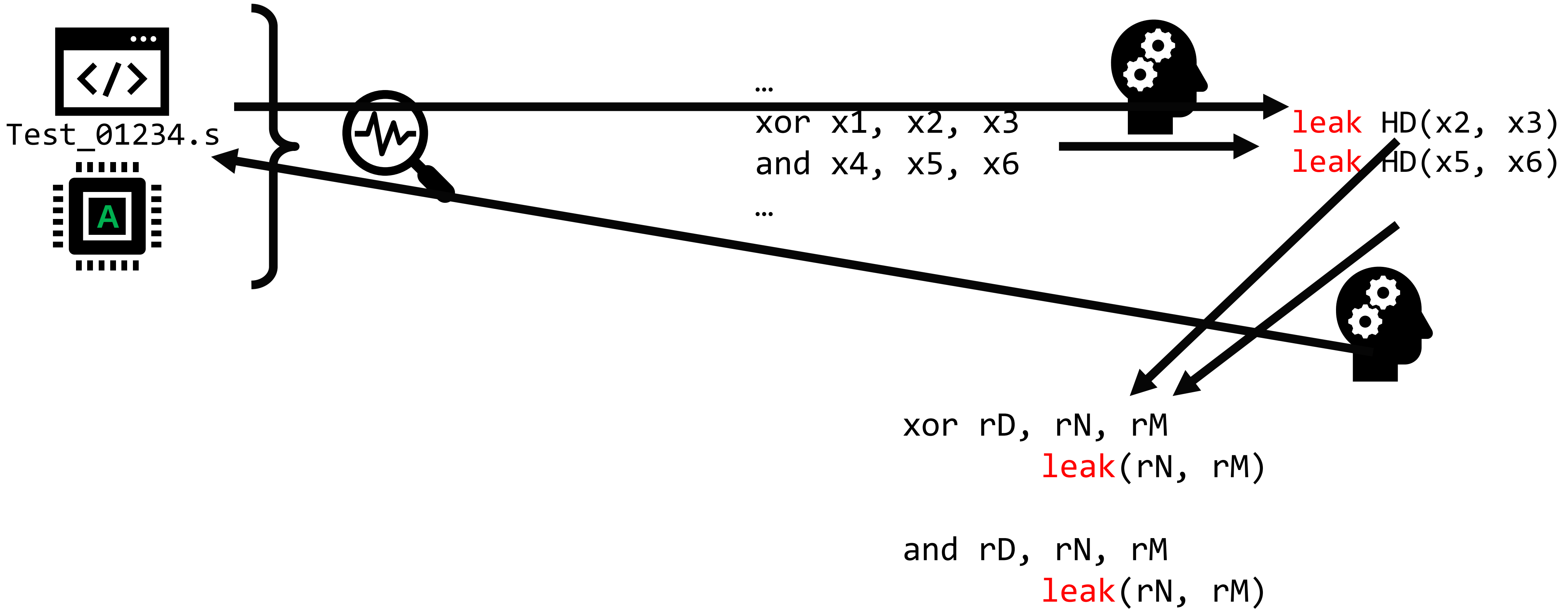
COMPLETENESS OF LEAKAGE MODELS
EMPIRICAL APPROACH

- Write test, measure, eat, sleep, repeat



COMPLETENESS OF LEAKAGE MODELS
EMPIRICAL APPROACH

- Write test, measure, eat, sleep, repeat



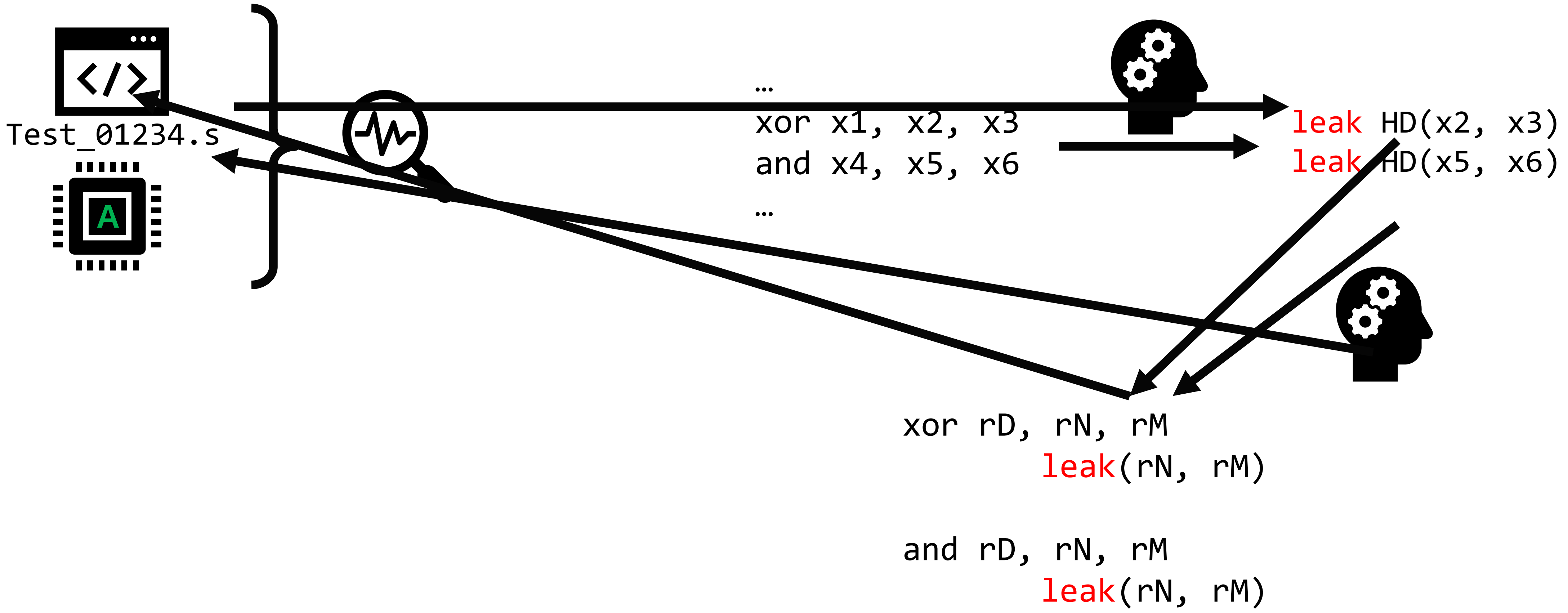
```
…
xor x1, x2, x3
and x4, x5, x6
…
```

```
leak HD(x2, x3)
leak HD(x5, x6)
```

```
xor rD, rN, rM
        leak(rN, rM)

and rD, rN, rM
        leak(rN, rM)
```

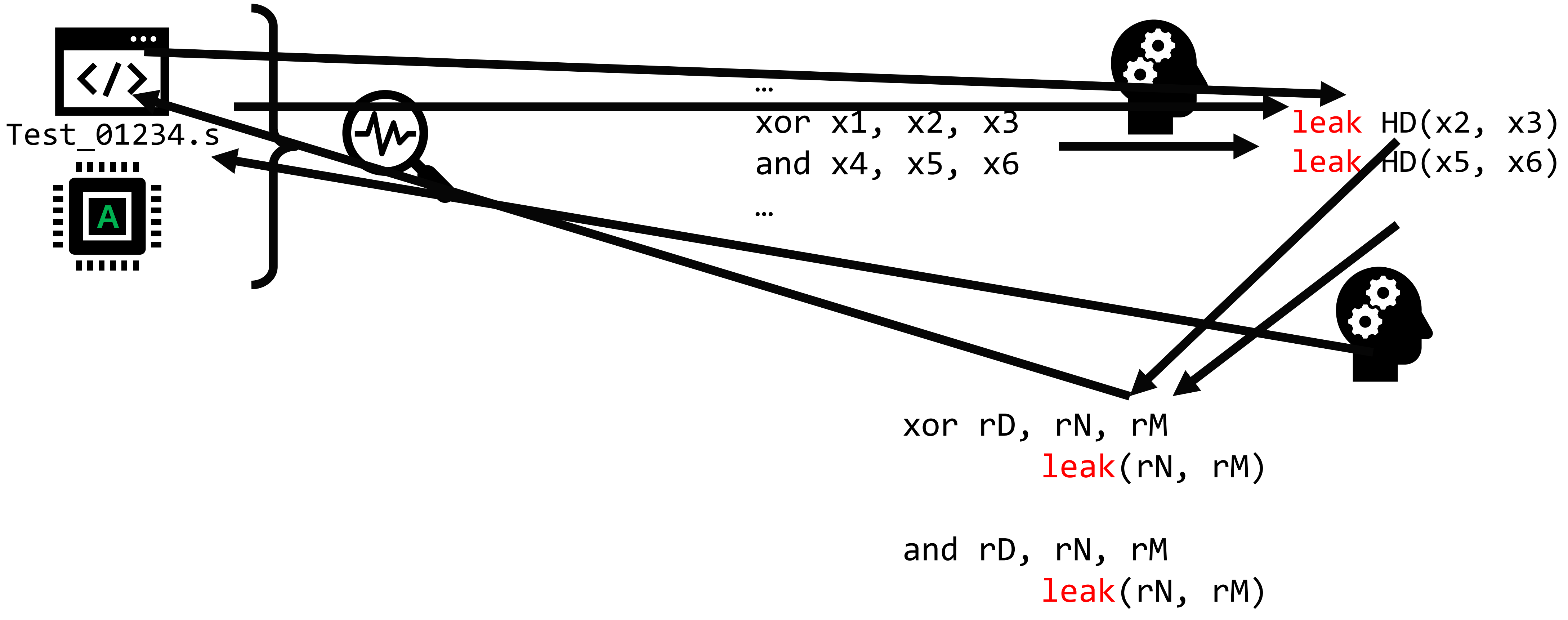
COMPLETENESS OF LEAKAGE MODELS
EMPIRICAL APPROACH

- Write test, measure, eat, sleep, repeat



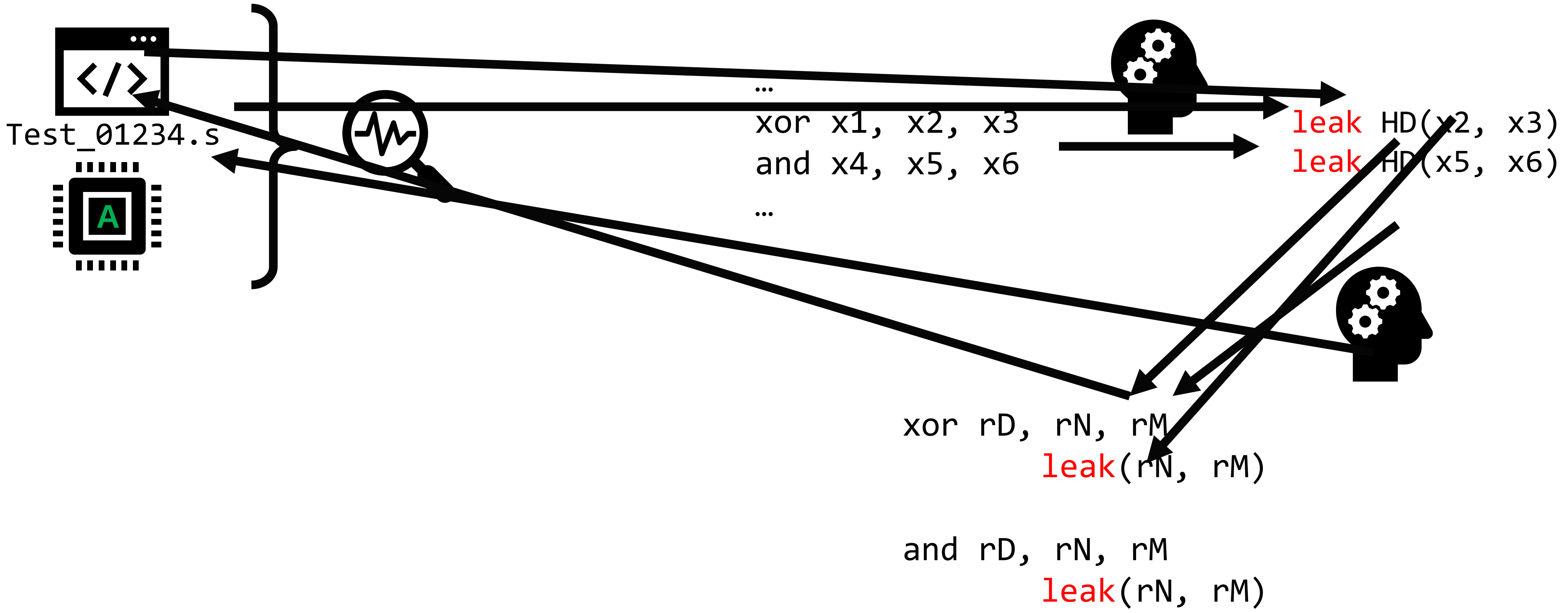
COMPLETENESS OF LEAKAGE MODELS
EMPIRICAL APPROACH

- Write test, measure, eat, sleep, repeat



COMPLETENESS OF LEAKAGE MODELS
EMPIRICAL APPROACH

- Write test, measure, eat, sleep, repeat



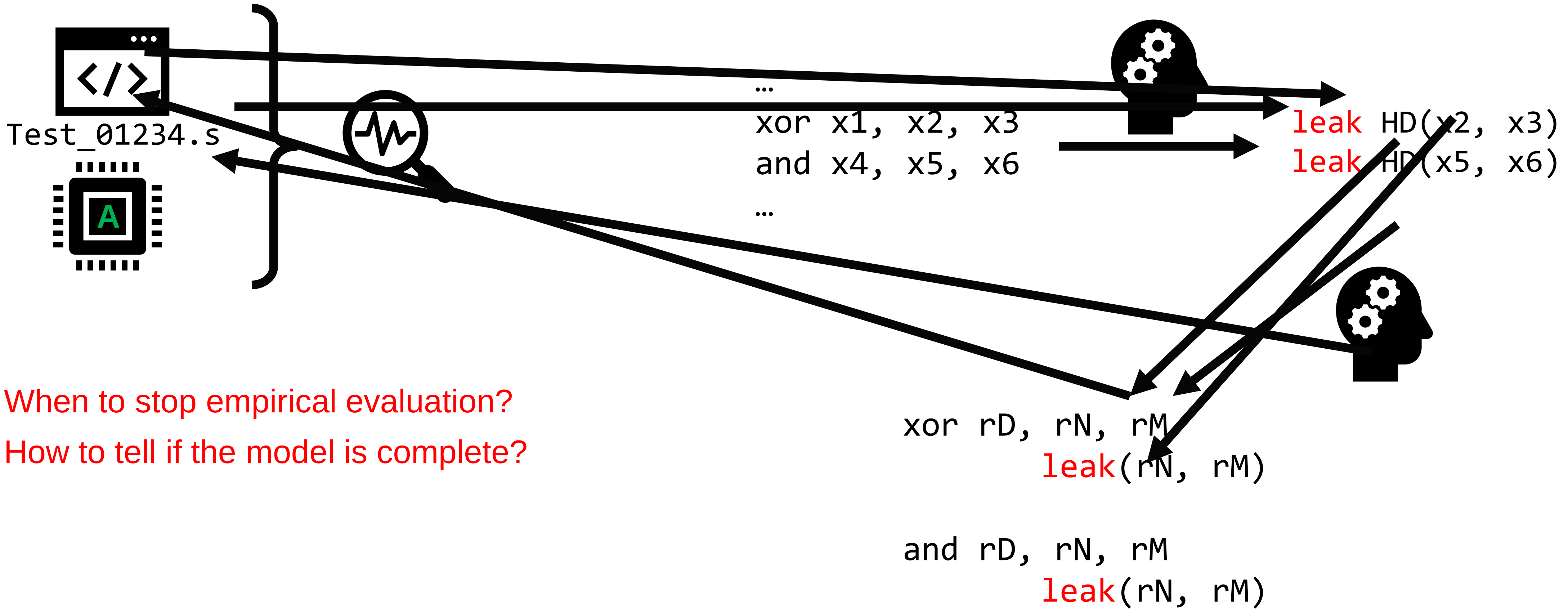
```
…
xor x1, x2, x3
and x4, x5, x6
…
```

```
leak HD(x2, x3)
leak HD(x5, x6)
```

```
xor rD, rN, rM
        leak(rN, rM)

and rD, rN, rM
        leak(rN, rM)
```

COMPLETENESS OF LEAKAGE MODELS
EMPIRICAL APPROACH

- Write test, measure, eat, sleep, repeat



```
…
xor x1, x2, x3
and x4, x5, x6
…
```

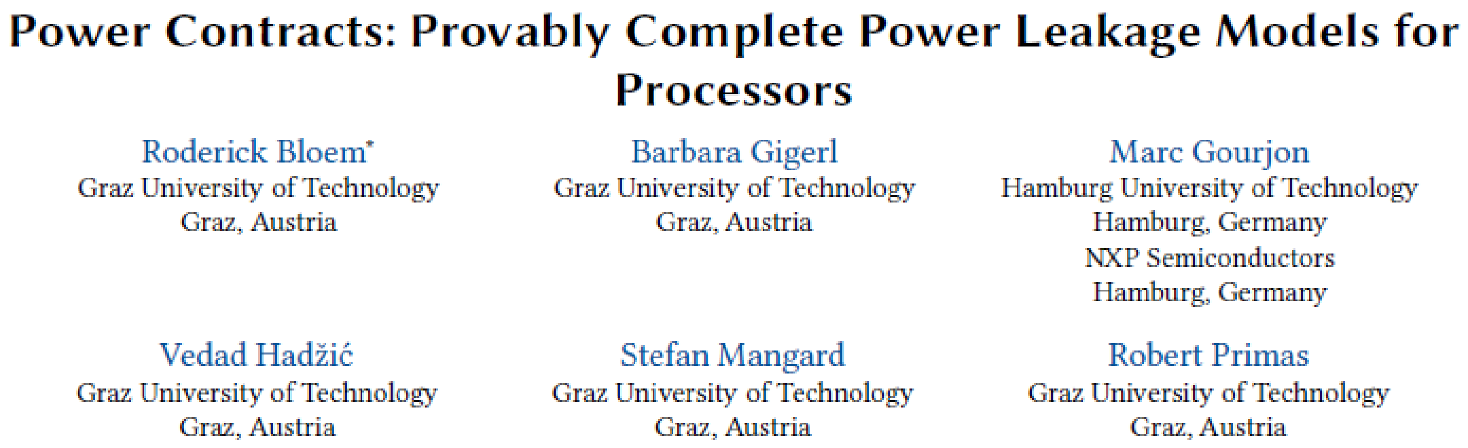
```
leak HD(x2, x3)
leak HD(x5, x6)
```

```
xor rD, rN, rM
        leak(rN, rM)

and rD, rN, rM
        leak(rN, rM)
```

When to stop empirical evaluation?

How to tell if the model is complete?

PROVABLY COMPLETE LEAKAGE MODELS



Power Contracts: Provably Complete Power Leakage Models for Processors

Roderick Bloem*
Graz University of Technology
Graz, Austria

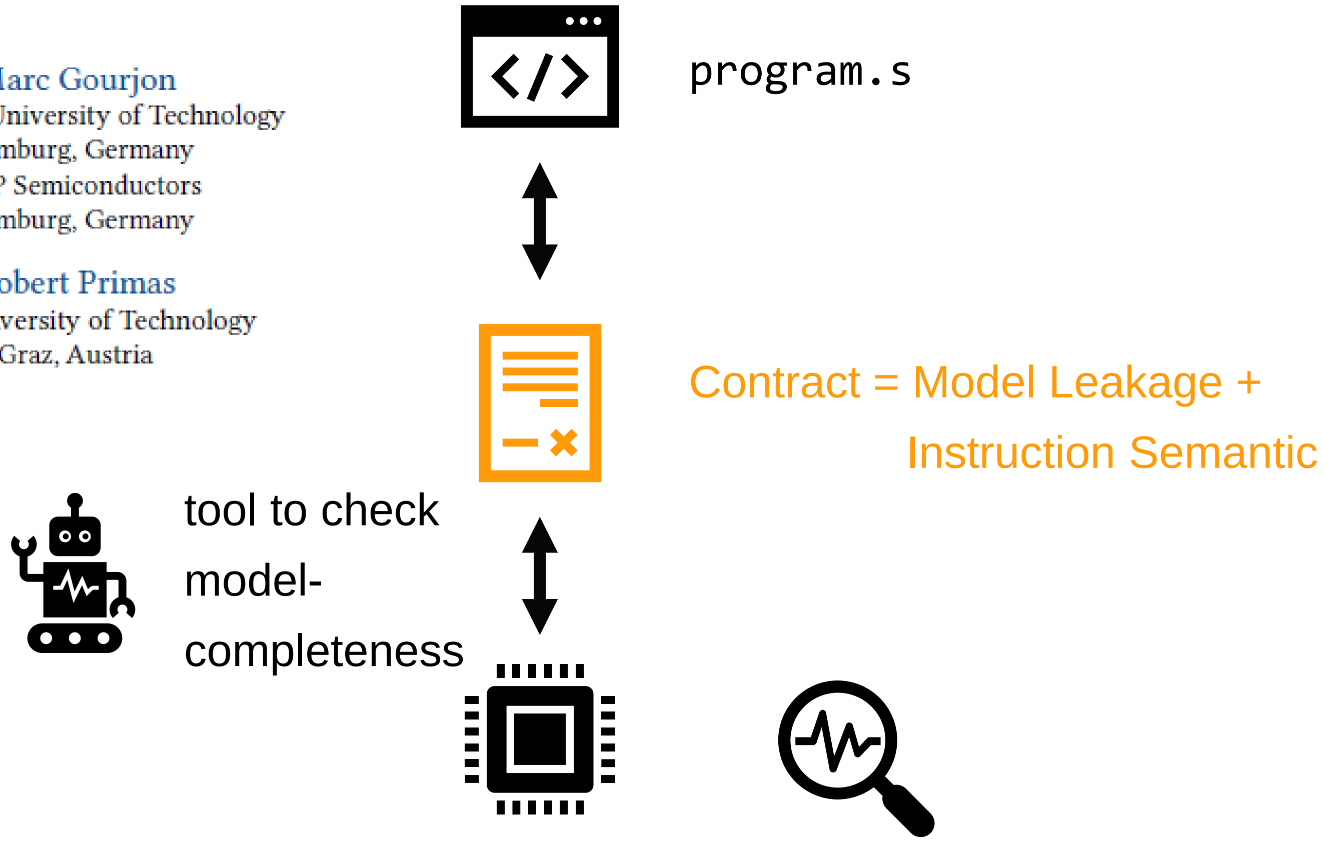
Barbara Gigerl
Graz University of Technology
Graz, Austria

Marc Gourjon
Hamburg University of Technology
Hamburg, Germany
NXP Semiconductors
Hamburg, Germany

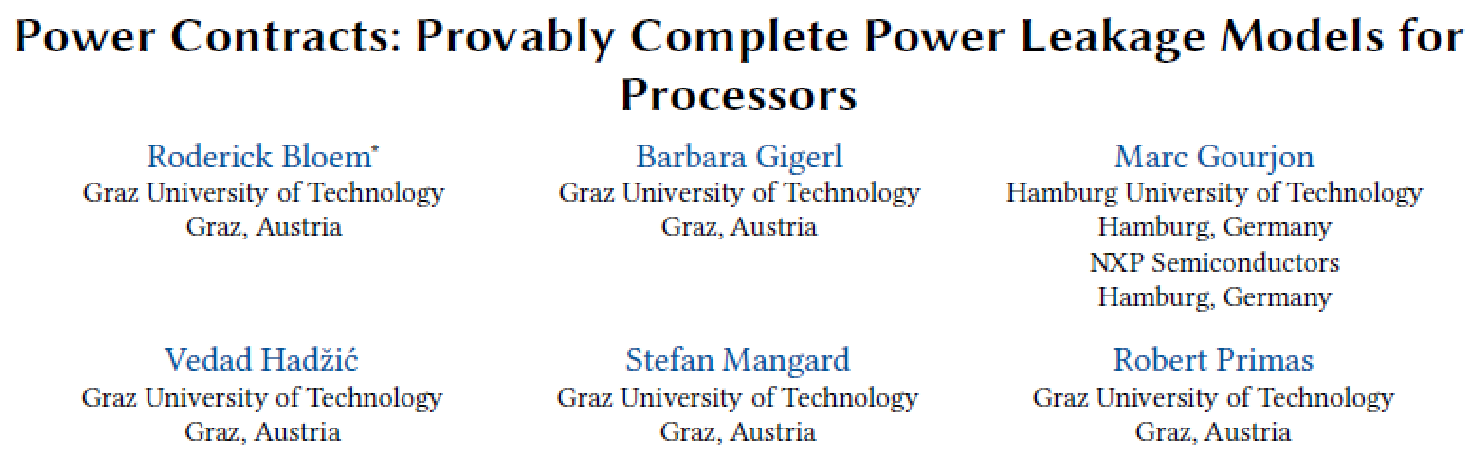
Vedad Hadžić
Graz University of Technology
Graz, Austria

Stefan Mangard
Graz University of Technology
Graz, Austria

Robert Primas
Graz University of Technology
Graz, Austria


program.s
Contract = Model Leakage + Instruction Semantic
tool to check model-completeness

PROVABLY COMPLETE LEAKAGE MODELS



Power Contracts: Provably Complete Power Leakage Models for Processors

Roderick Bloem*
Graz University of Technology
Graz, Austria

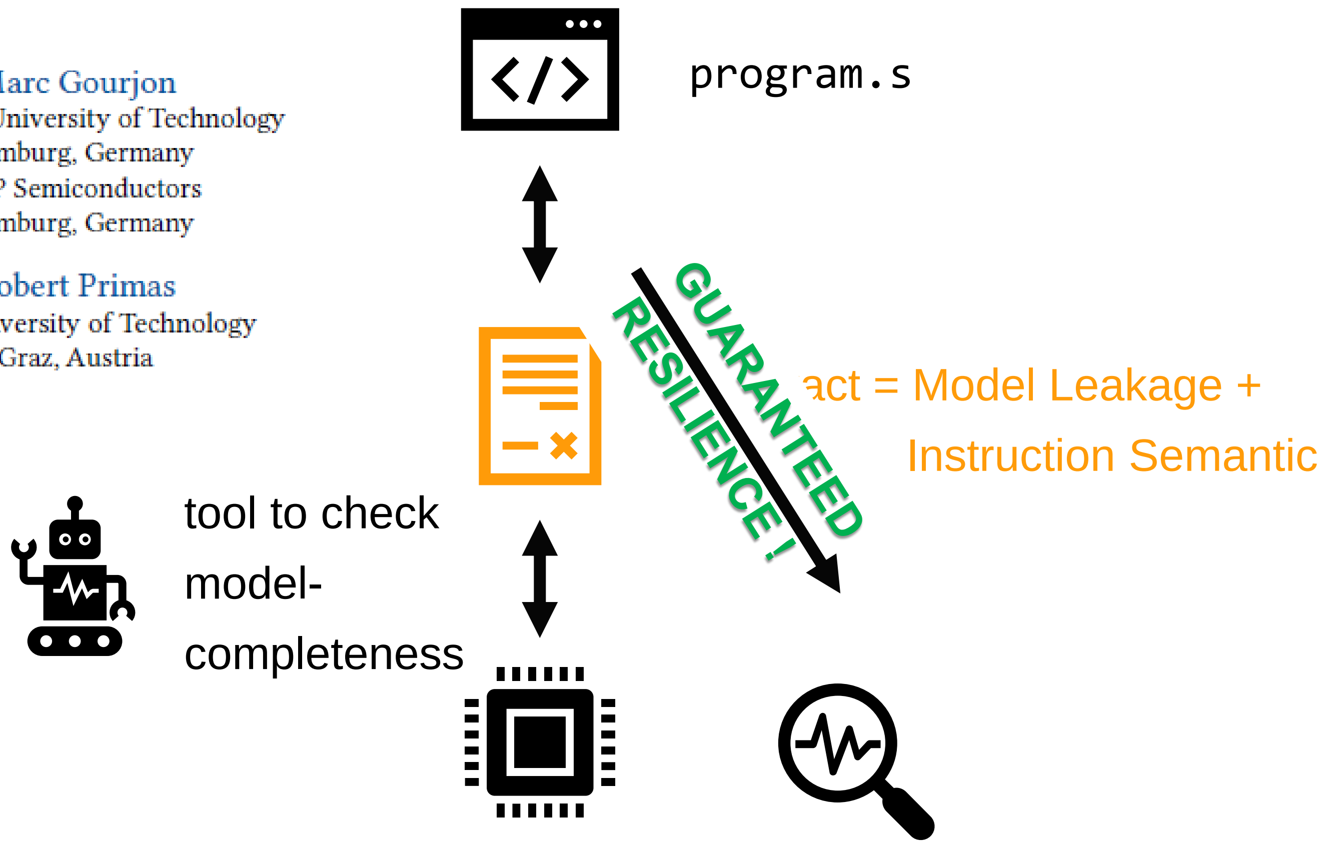
Barbara Gigerl
Graz University of Technology
Graz, Austria

Marc Gourjon
Hamburg University of Technology
Hamburg, Germany
NXP Semiconductors
Hamburg, Germany

Vedad Hadžić
Graz University of Technology
Graz, Austria

Stefan Mangard
Graz University of Technology
Graz, Austria

Robert Primas
Graz University of Technology
Graz, Austria



VERIFYING COMPLETENESS IN A NUTSHELL (1)
HW COMPLIANCE

- E2E security reduction based on ability to model any HW probe from modeled leakage in the contract

$$[\![P]\!]^{c}\left(\sigma_0^{c}\right)$$

$$[\![P]\!]^{h}\left(\sigma_0^{h}\right)$$

Definition 6 (Compliance: $[\![\cdot]\!]^h \vdash_{\mathcal{M}} [\![\cdot]\!]^c$).

VERIFYING COMPLETENESS IN A NUTSHELL (1)
HW COMPLIANCE

- E2E security reduction based on ability to model any HW probe from modeled leakage in the contract

$$[\![\mathrm{P}]\!]^c\left(\sigma_0^c\right): \quad \sigma_0^c \xrightharpoonup{\mathcal{L}_0^c} \sigma_1^c \xrightharpoonup{\mathcal{L}_1^c} \cdots \xrightharpoonup{\mathcal{L}_{n-1}^c} \sigma_n^c$$

$$[\![\mathrm{P}]\!]^h\left(\sigma_0^h\right): \quad \sigma_0^h \xrightharpoondown{\mathcal{L}_0^h} \sigma_1^h \xrightharpoondown{\mathcal{L}_1^h} \cdots \xrightharpoondown{\mathcal{L}_{m-1}^h} \sigma_m^h$$

$\uparrow$

starting state

DEFINITION 6 (COMPLIANCE: $[\![\cdot]\!]^h \vdash_{\mathcal{M}} [\![\cdot]\!]^c$).

VERIFYING COMPLETENESS IN A NUTSHELL (1)
HW COMPLIANCE

- E2E security reduction based on ability to model any HW probe from modeled leakage in the contract

$$[\![\mathrm{P}]\!]^c\left(\sigma_0^c\right): \qquad \sigma_0^c \xrightarrow{\mathcal{L}_0^c} \sigma_1^c \xrightarrow{\mathcal{L}_1^c} \ldots \xrightarrow{\mathcal{L}_{n-1}^c} \sigma_n^c$$

$$[\![\mathrm{P}]\!]^h\left(\sigma_0^h\right): \qquad \sigma_0^h \xrightarrow{\mathcal{L}_0^h} \sigma_1^h \xrightarrow{\mathcal{L}_1^h} \ldots \xrightarrow{\mathcal{L}_{m-1}^h} \sigma_m^h$$

$$\begin{tikzcd} \sigma_0^h & & & & \sigma_m^h \\ \text{starting state} \arrow[u] & & & & \text{final state} \arrow[u] \end{tikzcd}$$

Definition 6 (Compliance: $[\![\cdot]\!]^h \vdash_{\mathcal{M}} [\![\cdot]\!]^c$).

VERIFYING COMPLETENESS IN A NUTSHELL (1)
HW COMPLIANCE

- E2E security reduction based on ability to model any HW probe from modeled leakage in the contract

$$\begin{tikzcd}
[\![\mathrm{P}]\!]^c\left(\sigma_0^c\right): & \sigma_0^c \arrow[r, harpoon, "\mathcal{L}_0^c"] & \sigma_1^c \arrow[r, harpoon, "\mathcal{L}_1^c"] & \ldots \arrow[r, harpoon, "\mathcal{L}_{n-1}^c"] & \sigma_n^c
\end{tikzcd}$$

$$\begin{tikzcd}
[\![\mathrm{P}]\!]^h\left(\sigma_0^h\right): & \sigma_0^h \arrow[r, harpoon', "\mathcal{L}_0^h"] & \sigma_1^h \arrow[r, harpoon', "\mathcal{L}_1^h"] & \ldots \arrow[r, harpoon', "\mathcal{L}_{m-1}^h"] & \sigma_m^h
\end{tikzcd}$$

Actual leakage in HW

starting state

final state

DEFINITION 6 (COMPLIANCE: $[\![\cdot]\!]^h \vdash_{\mathcal{M}} [\![\cdot]\!]^c$).

VERIFYING COMPLETENESS IN A NUTSHELL (1)
HW COMPLIANCE

- E2E security reduction based on ability to model any HW probe from modeled leakage in the contract

Modeled leakage in Contract

$$\begin{tikzcd} {[\![\mathrm{P}]\!]^c\left(\sigma_0^c\right):} & \sigma_0^c \arrow[r, harpoon, "\mathcal{L}_0^c"] & \sigma_1^c \arrow[r, harpoon, "\mathcal{L}_1^c"] & \dots \arrow[r, harpoon, "\mathcal{L}_{n-1}^c"] & \sigma_n^c \end{tikzcd}$$

$$\begin{tikzcd} {[\![\mathrm{P}]\!]^h\left(\sigma_0^h\right):} & \sigma_0^h \arrow[r, harpoon, "\mathcal{L}_0^h"] & \sigma_1^h \arrow[r, harpoon, "\mathcal{L}_1^h"] & \dots \arrow[r, harpoon, "\mathcal{L}_{m-1}^h"] & \sigma_m^h \end{tikzcd}$$

starting state

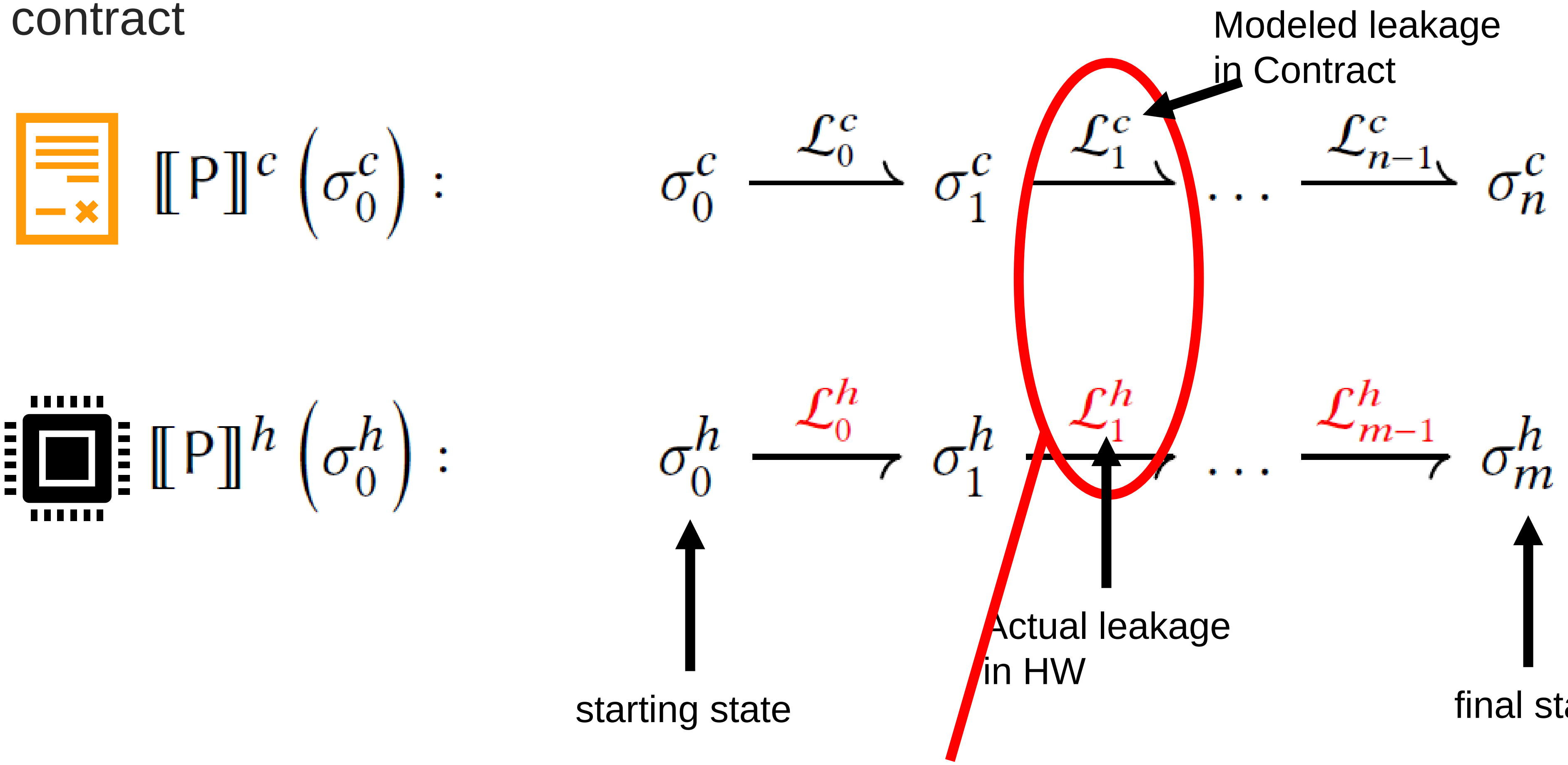
Actual leakage in HW

final state

Definition 6 (Compliance: $[\![\cdot]\!]^h \vdash_{\mathcal{M}} [\![\cdot]\!]^c$).

VERIFYING COMPLETENESS IN A NUTSHELL (1)
HW COMPLIANCE

- E2E security reduction based on ability to model any HW probe from modeled leakage in the contract



$$\llbracket P \rrbracket^c \left(\sigma_0^c\right) : \quad \sigma_0^c \xrightarrow{\mathcal{L}_0^c} \sigma_1^c \xrightarrow{\mathcal{L}_1^c} \dots \xrightarrow{\mathcal{L}_{n-1}^c} \sigma_n^c$$

$$\llbracket P \rrbracket^h \left(\sigma_0^h\right) : \quad \sigma_0^h \xrightarrow{\mathcal{L}_0^h} \sigma_1^h \xrightarrow{\mathcal{L}_1^h} \dots \xrightarrow{\mathcal{L}_{m-1}^h} \sigma_m^h$$

Prove that HW leakage can be modeled from some `leak` statement in contract

DEFINITION 6 (COMPLIANCE: $\llbracket\cdot\rrbracket^h \vdash_{\mathcal{M}} \llbracket\cdot\rrbracket^c$).

VERIFYING COMPLETENESS IN A NUTSHELL (1)
HW COMPLIANCE

- E2E security reduction based on ability to model any HW probe from modeled leakage in the contract

$[\![\mathrm{P}]\!]^c\left(\sigma_0^c\right):$

$[\![\mathrm{P}]\!]^h\left(\sigma_0^h\right):$

$$\begin{tikzcd}
\sigma_0^c \arrow[r, "\mathcal{L}_0^c"] \arrow[d, equal, "\mathcal{M}"] & \sigma_1^c \arrow[r, "\mathcal{L}_1^c"] & \cdots \arrow[r, "\mathcal{L}_{n-1}^c"] & \sigma_n^c \arrow[d, equal, "\mathcal{M}"] \\
\sigma_0^h \arrow[r, "\mathcal{L}_0^h"] & \sigma_1^h \arrow[r, "\mathcal{L}_1^h"] & \cdots \arrow[r, "\mathcal{L}_{m-1}^h"] & \sigma_m^h
\end{tikzcd}$$

Modeled leakage in Contract

Actual leakage in HW

starting state

final state

Prove that HW leakage can be modeled from some `leak` statement in contract

DEFINITION 6 (COMPLIANCE: $[\![\cdot]\!]^h \vdash_{\mathcal{M}} [\![\cdot]\!]^c$).

VERIFYING COMPLETENESS IN A NUTSHELL (2)
CHECKING THE ABILITY TO MODEL HW LEAKAGE FROM CONTRACT LEAKS

- Is there a function f(`e1,e2`) = y such that y = λ_g for all executions of a program?

$\mathcal{L}_i^c$ `:= {…, leak( e1, e2 ), …}`

$$\mathcal{L}_j^h := \{\ldots, \lambda_g(\sigma^h_{j-1}, \sigma^h_j), \ldots\}$$

- Rationale for model reduction:
  - If I know `e1,e2` which are exposed in the contract, then
  - I can simulate the observation of leakage λ_g of gate g in HW which an adversary could make

THEOREM 2 (MODEL REDUCTION).

VERIFYING COMPLETENESS IN A NUTSHELL (2)
CHECKING THE ABILITY TO MODEL HW LEAKAGE FROM CONTRACT LEAKS

- Is there a function f(`e1,e2`) = y such that y = λ_g for all executions of a program?

$\mathcal{L}_i^c$ `:= {…, leak( e1, e2 ), …}`

$\mathcal{L}_j^h$ `:= {…,` $\lambda_g(\sigma_{j-1}^h, \sigma_j^h)$`, …}`

$\exists$ f(`e1,e2`) = λ_g ?

- Rationale for model reduction:
  - If I know `e1,e2` which are exposed in the contract, then
  - I can simulate the observation of leakage λ_g of gate g in HW which an adversary could make

THEOREM 2 (MODEL REDUCTION).

VERIFYING COMPLETENESS IN A NUTSHELL (2)
CHECKING THE ABILITY TO MODEL HW LEAKAGE FROM CONTRACT LEAKS

- Is there a function f(`e1,e2`) = y such that y = λ_g for all executions of a program?

$\mathcal{L}_i^c$ `:= {…, leak( e1, e2 ), …}`

$\mathcal{L}_j^h$ `:= {…,` $\lambda_g(\sigma_{j-1}^h, \sigma_j^h)$`, …}`

∃ f(`e1,e2`) = λ_g ?

- Rationale for model reduction:
  - If I know `e1,e2` which are exposed in the contract, then
  - I can simulate the observation of leakage λ_g of gate g in HW which an adversary could make

THEOREM 2 (MODEL REDUCTION).

Provably complete leakage models for processors

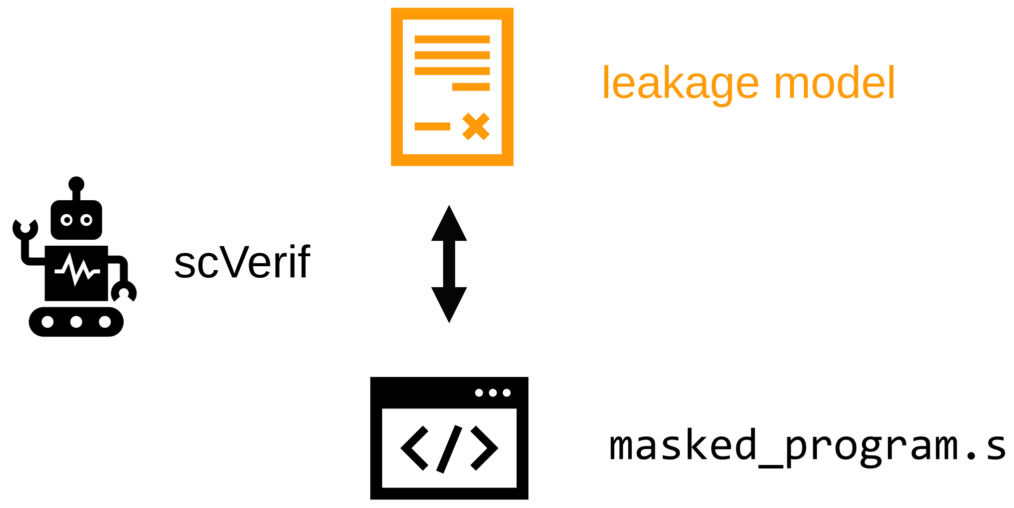
LEAKAGE MODELS FOR SECURITY RESEARCH

masked_program.s

More use-cases for fine-grained leakage model

- Foundation for power leakage emulators
- Statistical evaluation
- Masking centric
- Automated leakage mitigation
- Automated application of countermeasures

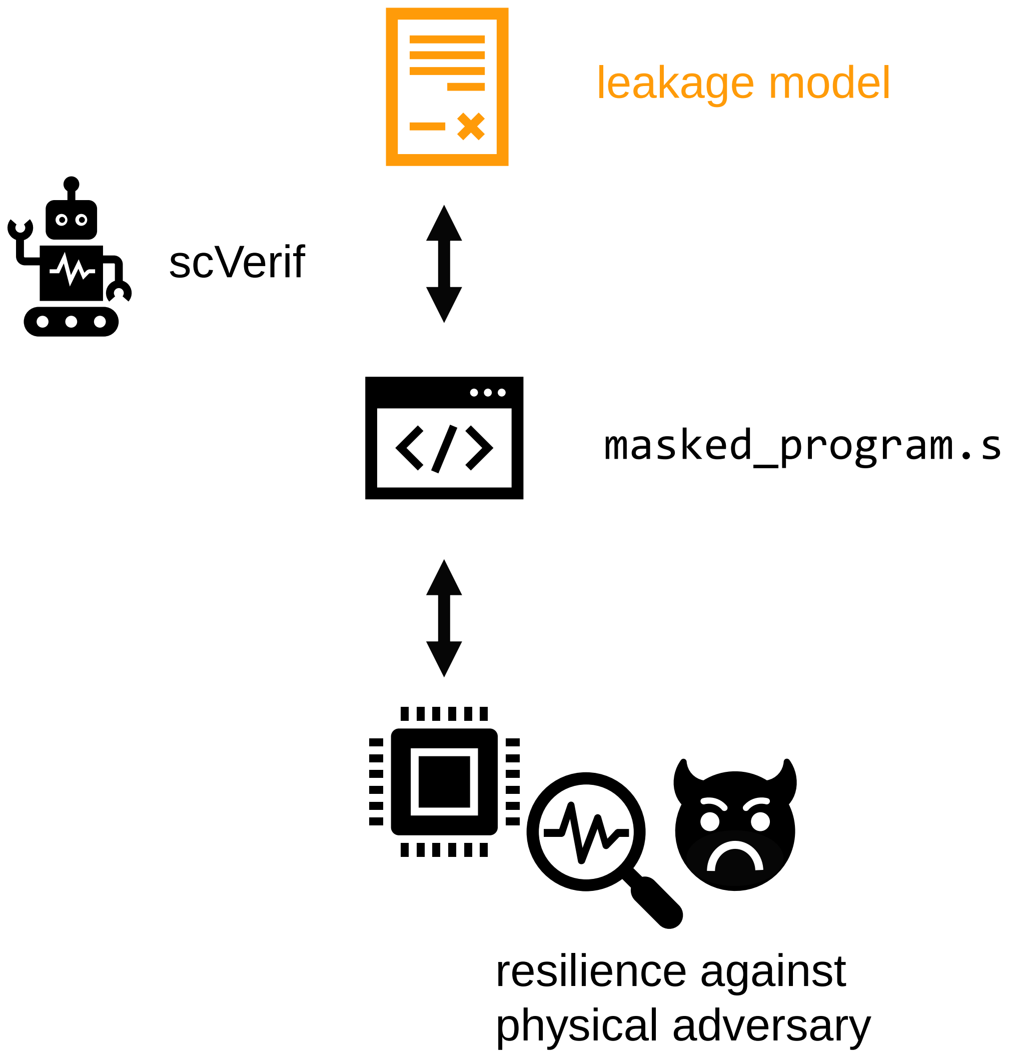
LEAKAGE MODELS FOR SECURITY RESEARCH



More use-cases for fine-grained leakage model

- Foundation for power leakage emulators
- Statistical evaluation
- Masking centric
- Automated leakage mitigation
- Automated application of countermeasures

LEAKAGE MODELS FOR SECURITY RESEARCH



More use-cases for fine-grained leakage model

- Foundation for power leakage emulators
- Statistical evaluation
- Masking centric
- Automated leakage mitigation
- Automated application of countermeasures

Verification of Resilience in Fine-Grained Models

SECURE CONNECTIONS FOR A SMARTER WORLD

MODEL-BASED SECURITY ASSESSMENT

masked_program.s

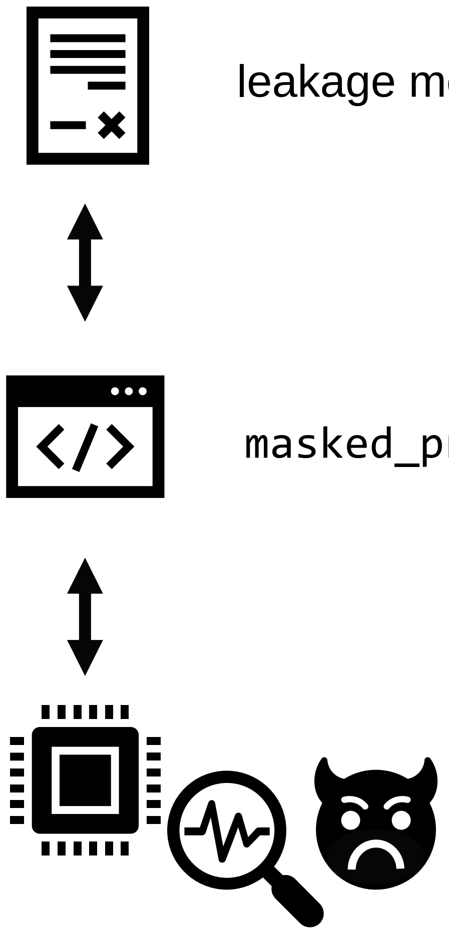
MODEL-BASED SECURITY ASSESSMENT

leakage model

Verify side-channel security

`masked_program.s`

MODEL-BASED SECURITY ASSESSMENT



leakage model

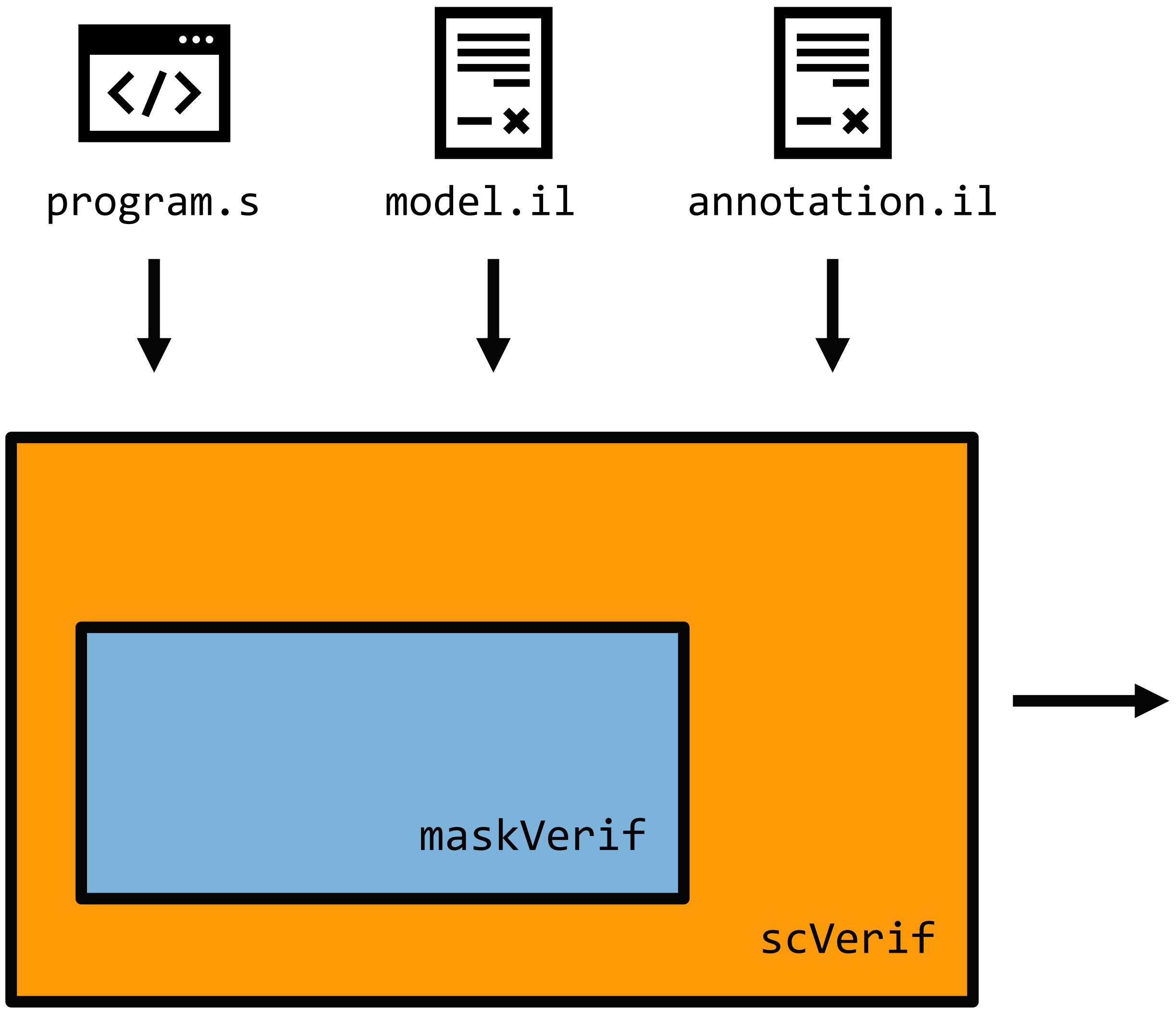
Verify side-channel security

`masked_program.s`

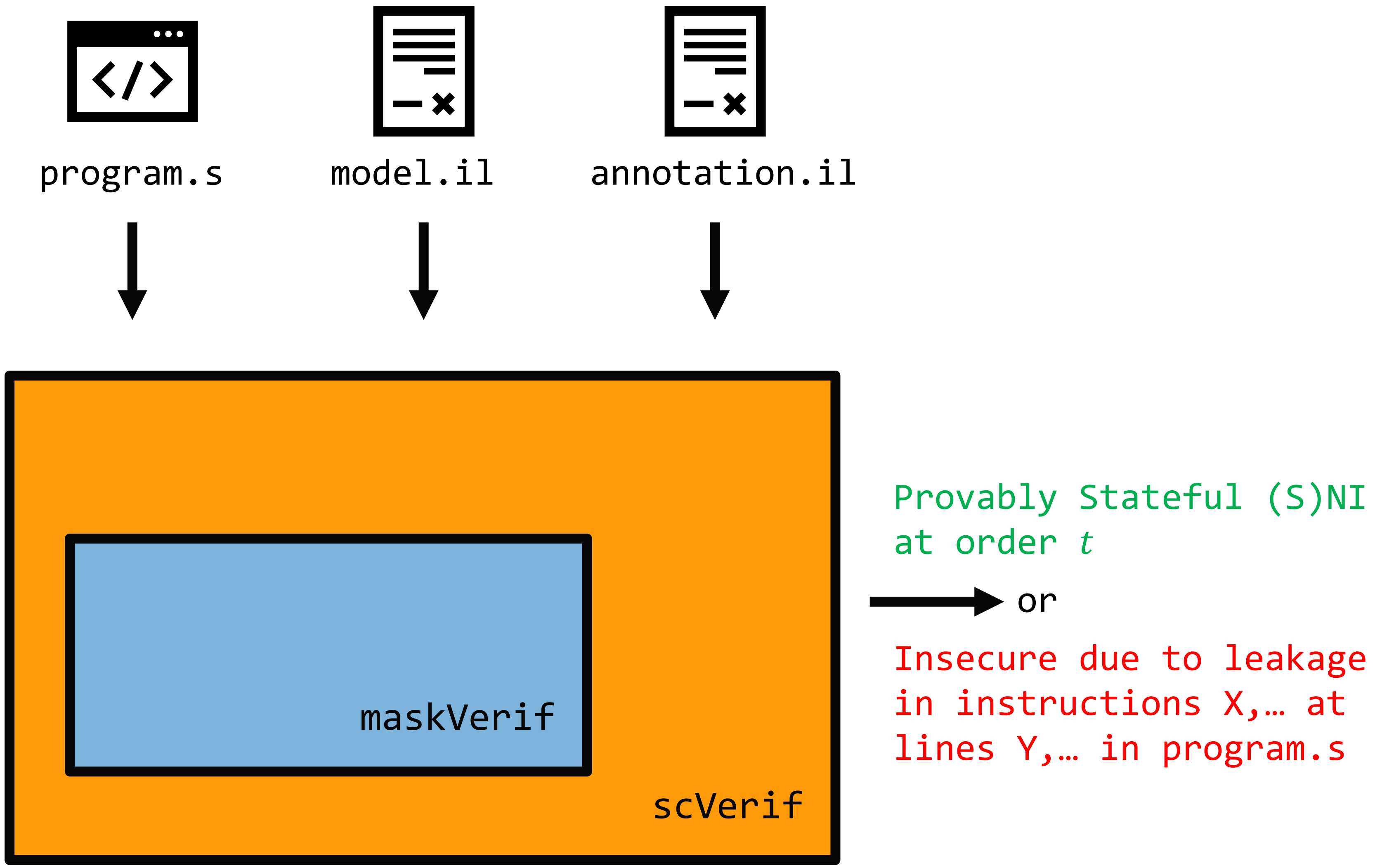
Goal: side-channel security
for physical execution on CPU

resilience against
probing adversary

SCVERIF - OVERVIEW

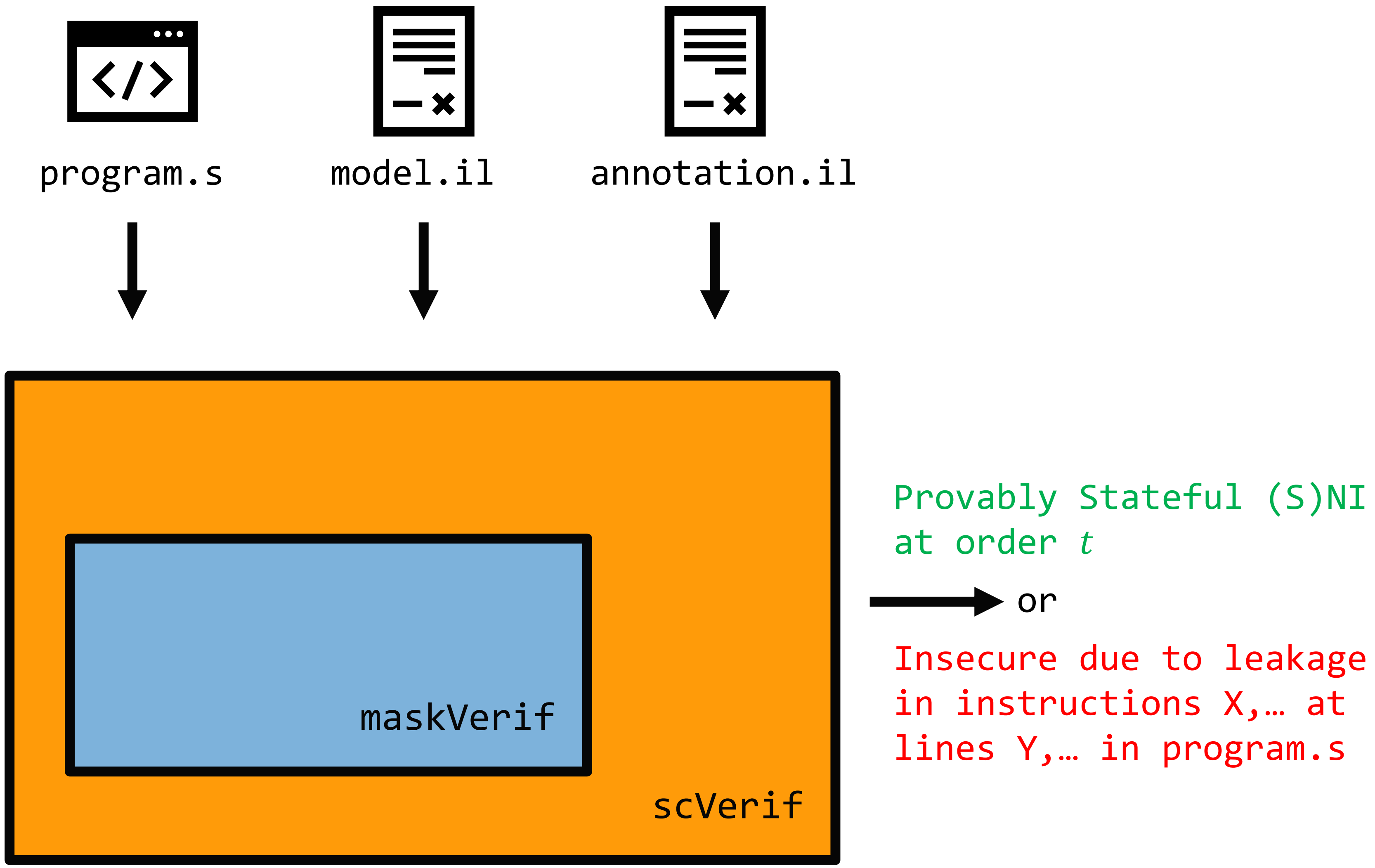


SCVERIF - OVERVIEW


program.s
model.il
annotation.il
maskVerif
scVerif
Provably Stateful (S)NI at order t
or
Insecure due to leakage in instructions X,… at lines Y,… in program.s

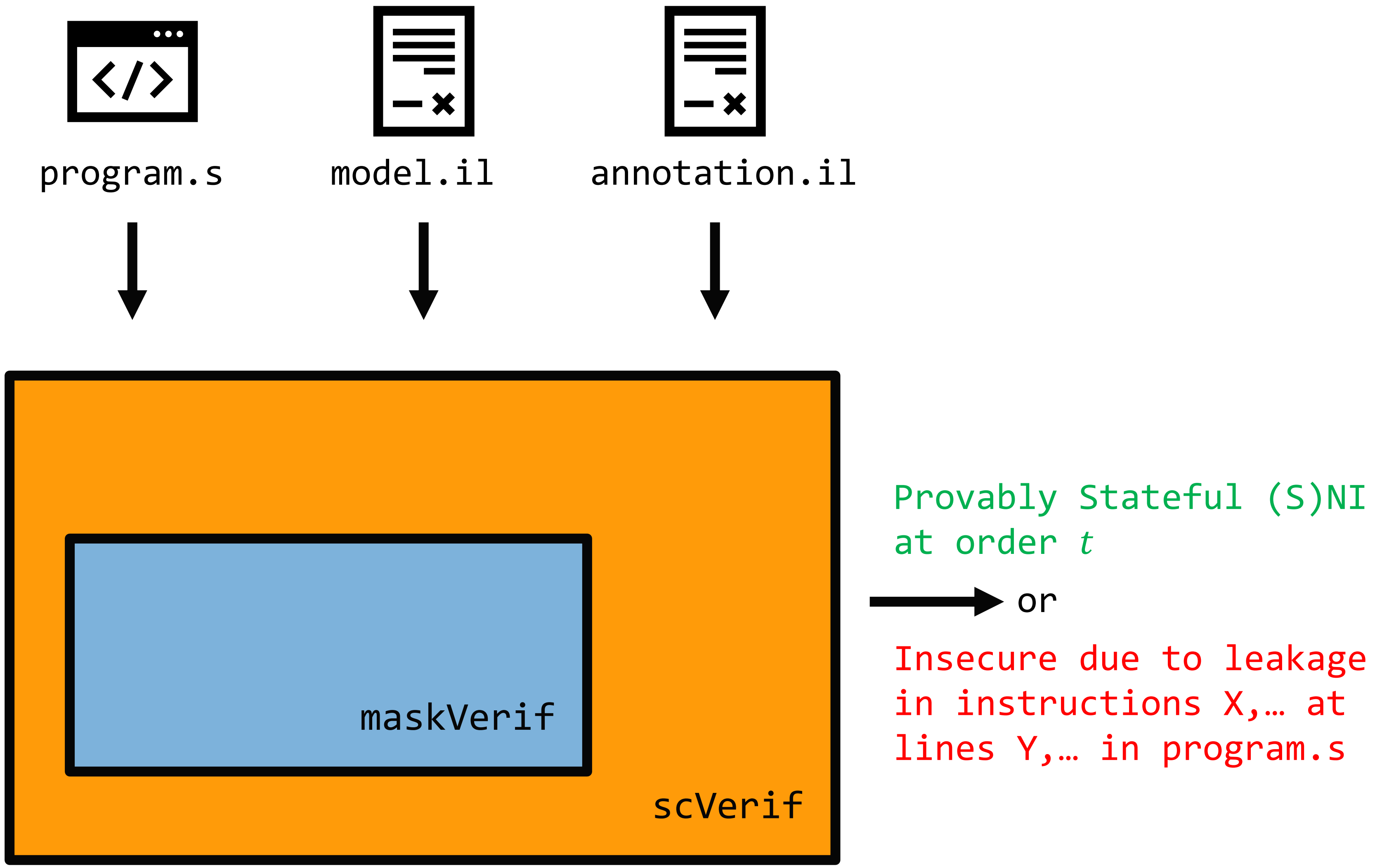
SCVERIF - OVERVIEW

- Prove security w.r.t. all device specific leakage at fixed security order


program.s
model.il
annotation.il
maskVerif
scVerif
Provably Stateful (S)NI at order *t*
or
Insecure due to leakage in instructions X,… at lines Y,… in program.s

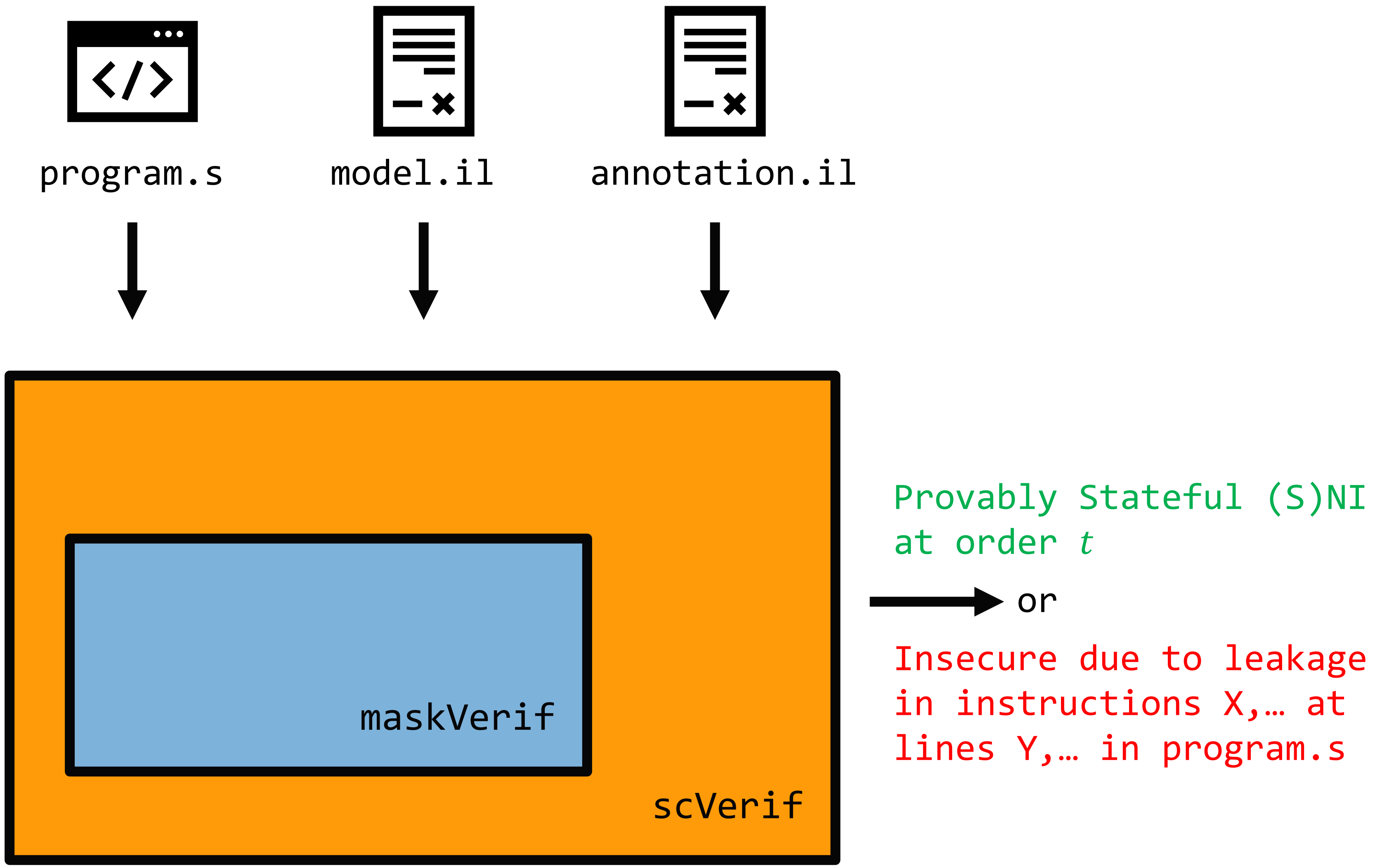
SCVERIF - OVERVIEW

- Prove security w.r.t. all device specific leakage at fixed security order
- Executable Arm / RISC-V assembly implementations


program.s
model.il
annotation.il
maskVerif
scVerif
Provably Stateful (S)NI at order *t*
or
Insecure due to leakage in instructions X,… at lines Y,… in program.s

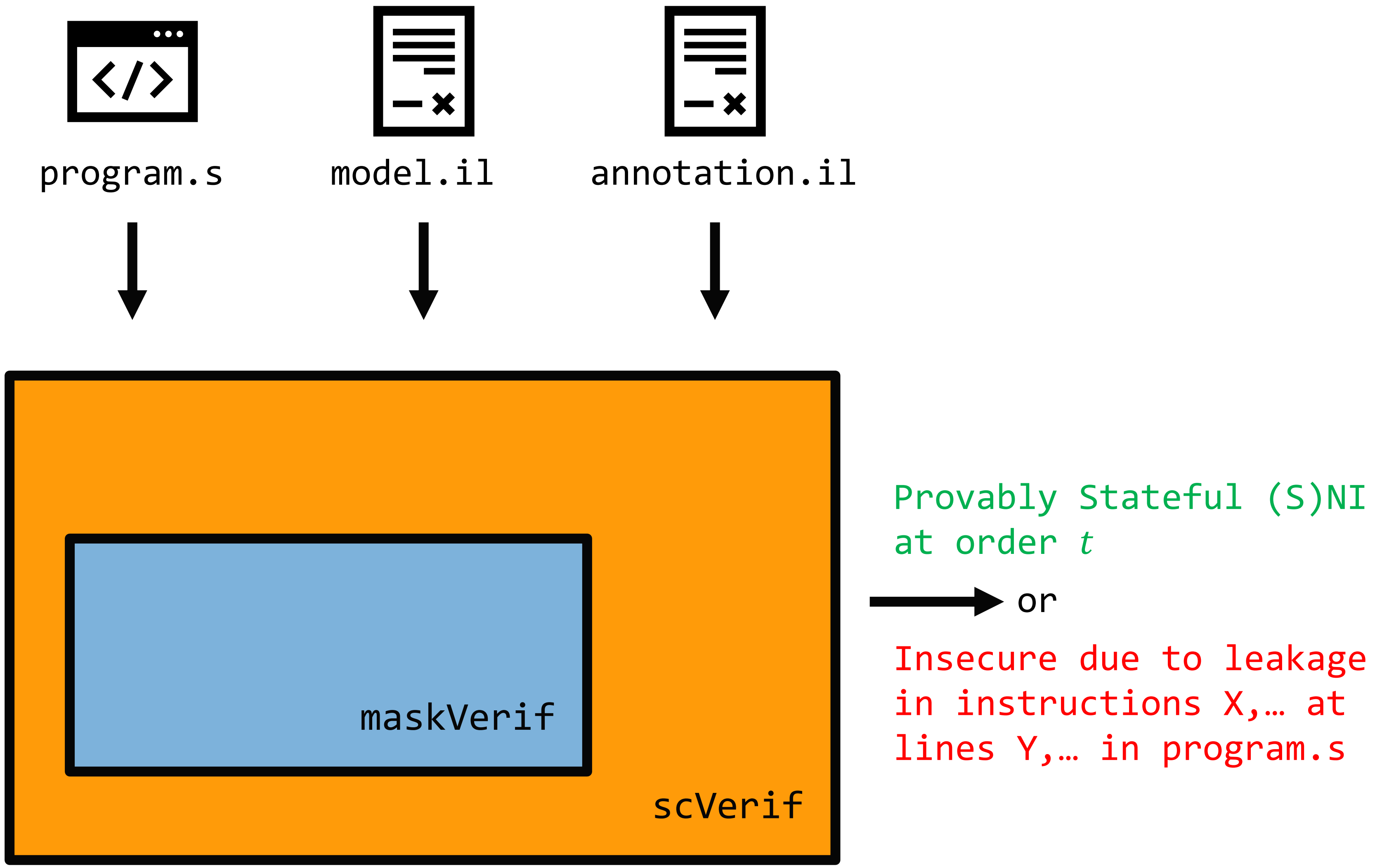
SCVERIF - OVERVIEW

- Prove security w.r.t. all device specific leakage at fixed security order
- Executable Arm / RISC-V assembly implementations
- Stateful non-interference


program.s
model.il
annotation.il
maskVerif
scVerif
Provably Stateful (S)NI at order t
or
Insecure due to leakage in instructions X,… at lines Y,… in program.s

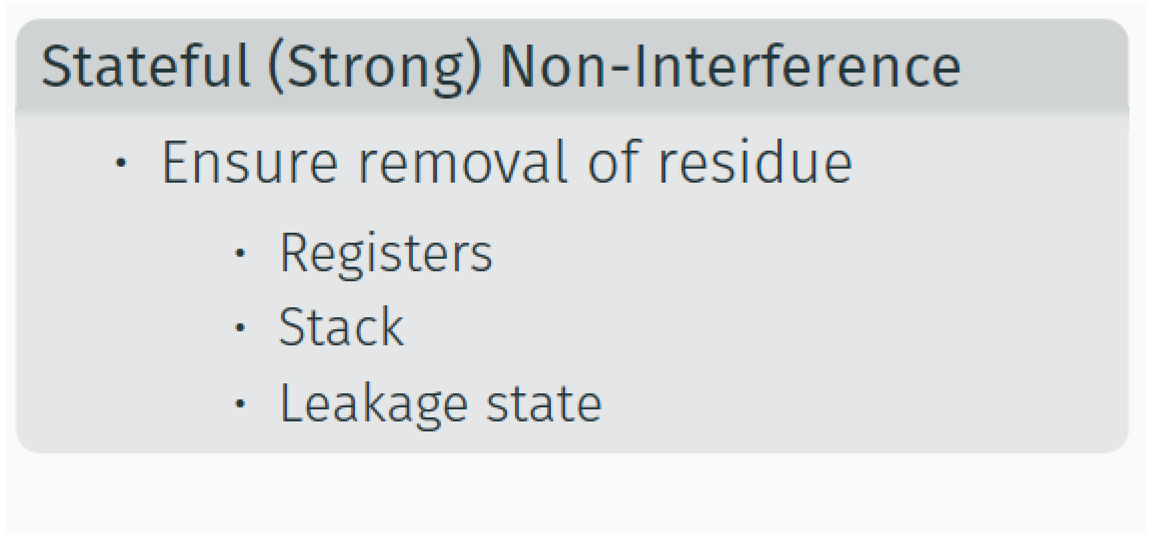
SCVERIF - OVERVIEW

- Prove security w.r.t. all device specific leakage at fixed security order
- Executable Arm / RISC-V assembly implementations
- Stateful non-interference
- PINI + probing security


program.s
model.il
annotation.il
maskVerif
scVerif
Provably Stateful (S)NI at order t
or
Insecure due to leakage in instructions X,… at lines Y,… in program.s

STATEFUL NON-INTERFERENCE

- Gadget
  - Masked secret inputs x_{in}
  - Randomness r
  - Public input state s_{in} (indep. of x_{in} and r)
  - Secret outputs y_{out}
  - Public output state s_{out}
  - Exposes internal observable leakage L



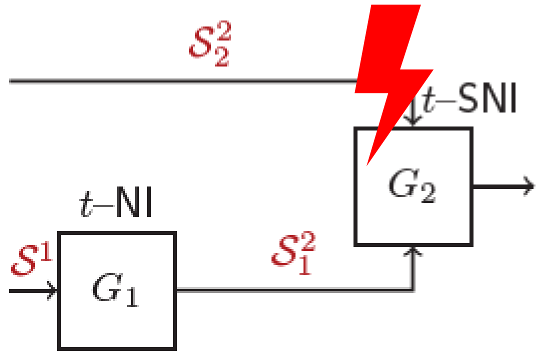
Stateful (Strong) Non-Interference

- Ensure removal of residue
  - Registers
  - Stack
  - Leakage state

- Stateful t-Strong Non-Interference
  1. Any set of t_{int} observations on internal leakage L in combination with t_{out} observations on outputs s_{out} s.t. $t_{int} + t_{out} \leq t$, combined with any number of observations on public output state s_{out} can be simulated from just t input shares and the input state s_{in}.
  2. The output state s_{out} can be simulated from only the input state s_{in}.

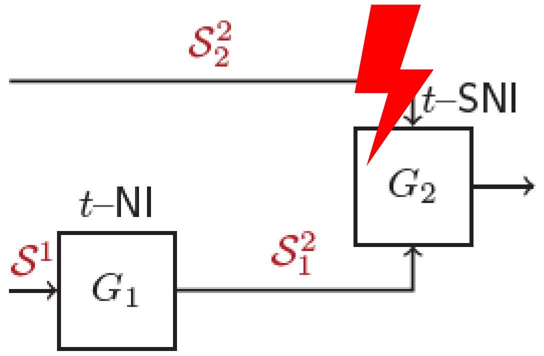
SECURE COMPOSITION WITH STATEFUL (S)NI

- Well known
  - Secure + insecure code $\not\Rightarrow$ secure
- Side-Channel
  - Resilient + Resilient $\not\Rightarrow$ Resilient
  - Observed knowledge propagates
  - Especially residue



SECURE COMPOSITION WITH STATEFUL (S)NI

- Well known
  - Secure + insecure code $\not\Rightarrow$ secure
- Side-Channel
  - Resilient + Resilient $\not\Rightarrow$ Resilient
  - Observed knowledge propagates
  - Especially residue

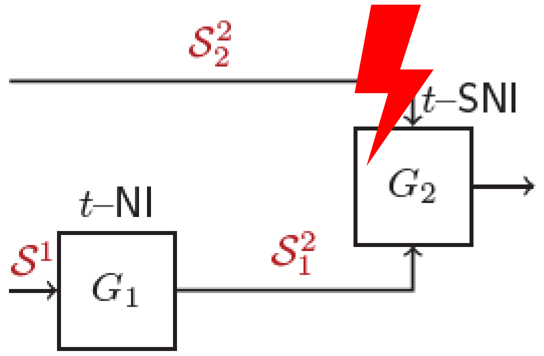


Stateful notions ensure removal of residue

→ Standard (S)NI composition rules restored in stateful setting

SECURE COMPOSITION WITH STATEFUL (S)NI

- Well known
  - Secure + insecure code $\nRightarrow$ secure
- Side-Channel
  - Resilient + Resilient $\nRightarrow$ Resilient
  - Observed knowledge propagates
  - Especially residue

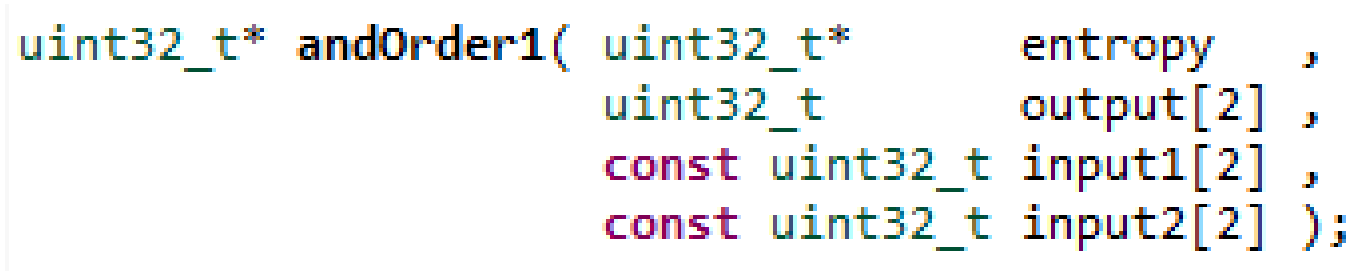


```
annotate andOrder1
  …
  output public r0
  …
  output public r7
  output public stack
```

Stateful notions ensure removal of residue

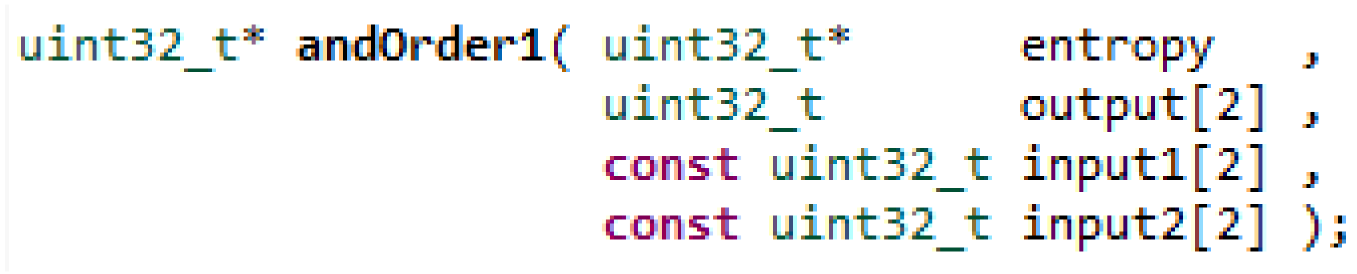
→ Standard (S)NI composition rules restored in stateful setting

ANNOTATIONS FOR CONCRETE IMPLEMENTATIONS



```
uint32_t* andOrder1( uint32_t*       entropy   ,
                     uint32_t        output[2] ,
                     const uint32_t input1[2] ,
                     const uint32_t input2[2] );
```

ANNOTATIONS FOR CONCRETE IMPLEMENTATIONS



```
uint32_t* andOrder1( uint32_t*       entropy   ,
                     uint32_t        output[2] ,
                     const uint32_t  input1[2] ,
                     const uint32_t  input2[2] );
```

Pointers to

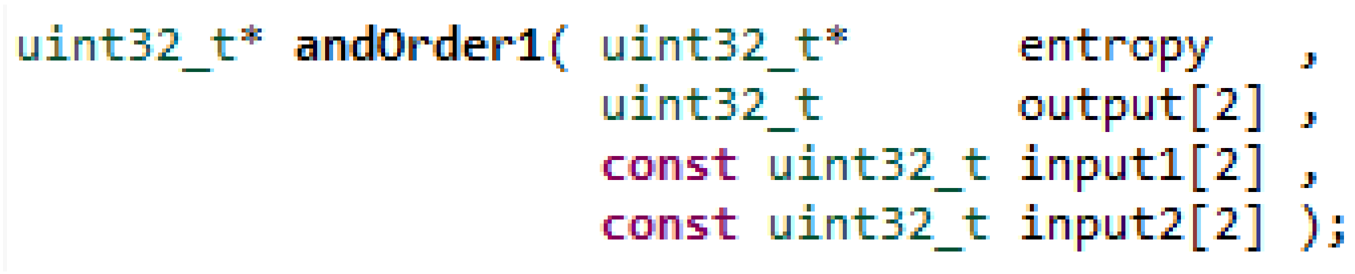
r

y_o, y_1

$x_0^{(0)}, x_1^{(0)}$

$x_0^{(1)}, x_1^{(1)}$

ANNOTATIONS FOR CONCRETE IMPLEMENTATIONS



```
uint32_t* andOrder1( uint32_t*       entropy   ,
                     uint32_t        output[2] ,
                     const uint32_t  input1[2] ,
                     const uint32_t  input2[2] );
```

Pointers to

r

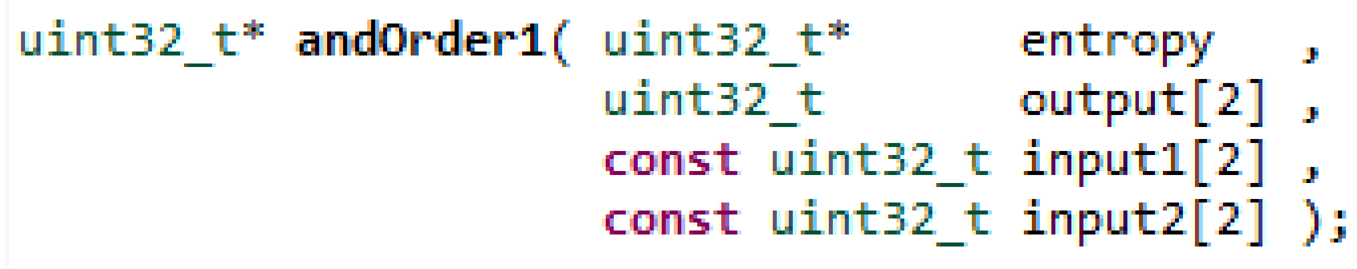
y_o, y_1

$x_0^{(0)}, x_1^{(0)}$

$x_0^{(1)}, x_1^{(1)}$

```
annotate andOrder1
  region mem w32 a[0:1]
```

ANNOTATIONS FOR CONCRETE IMPLEMENTATIONS



```
uint32_t* andOrder1( uint32_t*       entropy   ,
                     uint32_t        output[2] ,
                     const uint32_t input1[2] ,
                     const uint32_t input2[2] );
```

Pointers to

r

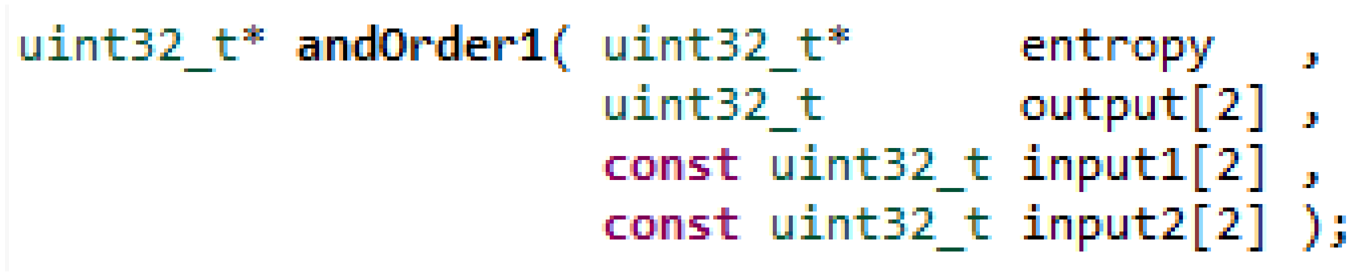
y_o, y_1

$x_0^{(0)}, x_1^{(0)}$

$x_0^{(1)}, x_1^{(1)}$

```
annotate andOrder1
  region mem w32 a[0:1]
  region mem w32 b[0:1]
  region mem w32 c[0:1]
  region mem w32 rnd[0:0]
```

ANNOTATIONS FOR CONCRETE IMPLEMENTATIONS



```
uint32_t* andOrder1( uint32_t*      entropy   ,
                     uint32_t       output[2] ,
                     const uint32_t input1[2] ,
                     const uint32_t input2[2] );
```

Pointers to

r

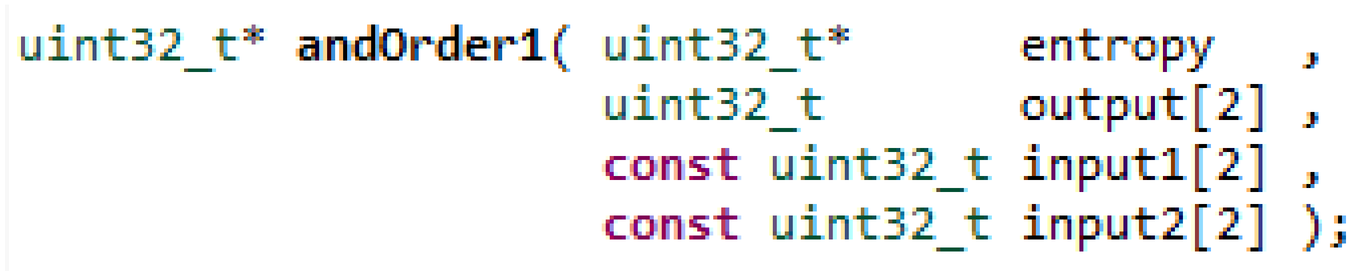
y_o, y_1

$x_0^{(0)}, x_1^{(0)}$

$x_0^{(1)}, x_1^{(1)}$

```
annotate andOrder1
  region mem w32 a[0:1]
  region mem w32 b[0:1]
  region mem w32 c[0:1]
  region mem w32 rnd[0:0]

  init r0 [rnd 0]
  init r1 [c 0]
  init r2 [a 0]
  init r3 [b 0]
```

ANNOTATIONS FOR CONCRETE IMPLEMENTATIONS



```
uint32_t* andOrder1( uint32_t*      entropy   ,
                     uint32_t       output[2] ,
                     const uint32_t input1[2] ,
                     const uint32_t input2[2] );
```

Pointers to

r

y_o, y_1

$x_0^{(0)}, x_1^{(0)}$

$x_0^{(1)}, x_1^{(1)}$

```
annotate andOrder1
  region mem w32 a[0:1]
  region mem w32 b[0:1]
  region mem w32 c[0:1]
  region mem w32 rnd[0:0]
  region mem w32 stack[-2:-1]
  init r0 [rnd 0]
  init r1 [c 0]
  init r2 [a 0]
  init r3 [b 0]
  init sp [stack 0]
  init lr exit
```

LOWERING TO MASKVERIF PARTIAL-EVALUATION

- MaskVerif
  - Provides verification algorithms for high-level algorithms
  - No addressable memory →just arrays with static indices
  - No control-flow

LOWERING TO MASKVERIF PARTIAL-EVALUATION

- MaskVerif
  - Provides verification algorithms for high-level algorithms
  - No addressable memory → just arrays with static indices
  - No control-flow
- Partial evaluation
  - Evaluate control flow
  - Resolve memory accesses

LOWERING TO MASKVERIF
PARTIAL-EVALUATION

- MaskVerif
  - Provides verification algorithms for high-level algorithms
  - No addressable memory →just arrays with static indices
  - No control-flow
- Partial evaluation
  - Evaluate control flow
  - Resolve memory accesses

```
Annotate andOrder1:
  region mem w32 rnd[0:2]
  init r0 [rnd 0]
```

LOWERING TO MASKVERIF
PARTIAL-EVALUATION

- MaskVerif
  - Provides verification algorithms for high-level algorithms
  - No addressable memory →just arrays with static indices
  - No control-flow
- Partial evaluation
  - Evaluate control flow
  - Resolve memory accesses

```
Annotate andOrder1:
  region mem w32 rnd[0:2]
  init r0 [rnd 0]
…
LDR r4, 0x04(r0)
…
```

LOWERING TO MASKVERIF PARTIAL-EVALUATION

- MaskVerif
  - Provides verification algorithms for high-level algorithms
  - No addressable memory → just arrays with static indices
  - No control-flow
- Partial evaluation
  - Evaluate control flow
  - Resolve memory accesses

```
Annotate andOrder1 rnd[1]
  region mem w32 rnd[0:2]
  init r0 [rnd 0]
…
LDR r4, 0x04(r0)
…
evaluates to: r4 ← rnd[1]
```

LOWERING TO MASKVERIF PARTIAL-EVALUATION

- MaskVerif
  - Provides verification algorithms for high-level algorithms
  - No addressable memory →just arrays with static indices
  - No control-flow
- Partial evaluation
  - Evaluate control flow
  - Resolve memory accesses

```
Annotate andOrder1 rnd[1]
  region mem w32 rnd[0:2]
  init r0 [rnd 0]
…
LDR r4, 0x04(r0)
…
evaluates to: r4 ← rnd[1]
```

$$\frac{[\![e_i]\!]^\rho_\mu = (\vartheta_i, e'_i)}{[\![o(e_1,\dots,e_n)]\!]^\rho_\mu = (\tilde{o}(\vartheta_1,\dots,\vartheta_n), o(e'_1,\dots,e'_n))} \qquad \frac{}{[\![x]\!]^\rho_\mu = (\mu(x), x)}$$

$$\frac{[\![e]\!]^\rho_\mu = (n, e')}{[\![x[e]]\!]^\rho_\mu = (\mu(x)[n], x[n])} \qquad \frac{[\![e]\!]^\rho_\mu = (\langle x, \mathrm{ofs}\rangle, e')}{[\![\langle e\rangle]\!]^\rho_\mu = (\rho(x)[\mathrm{ofs}], x[\mathrm{ofs}])}$$

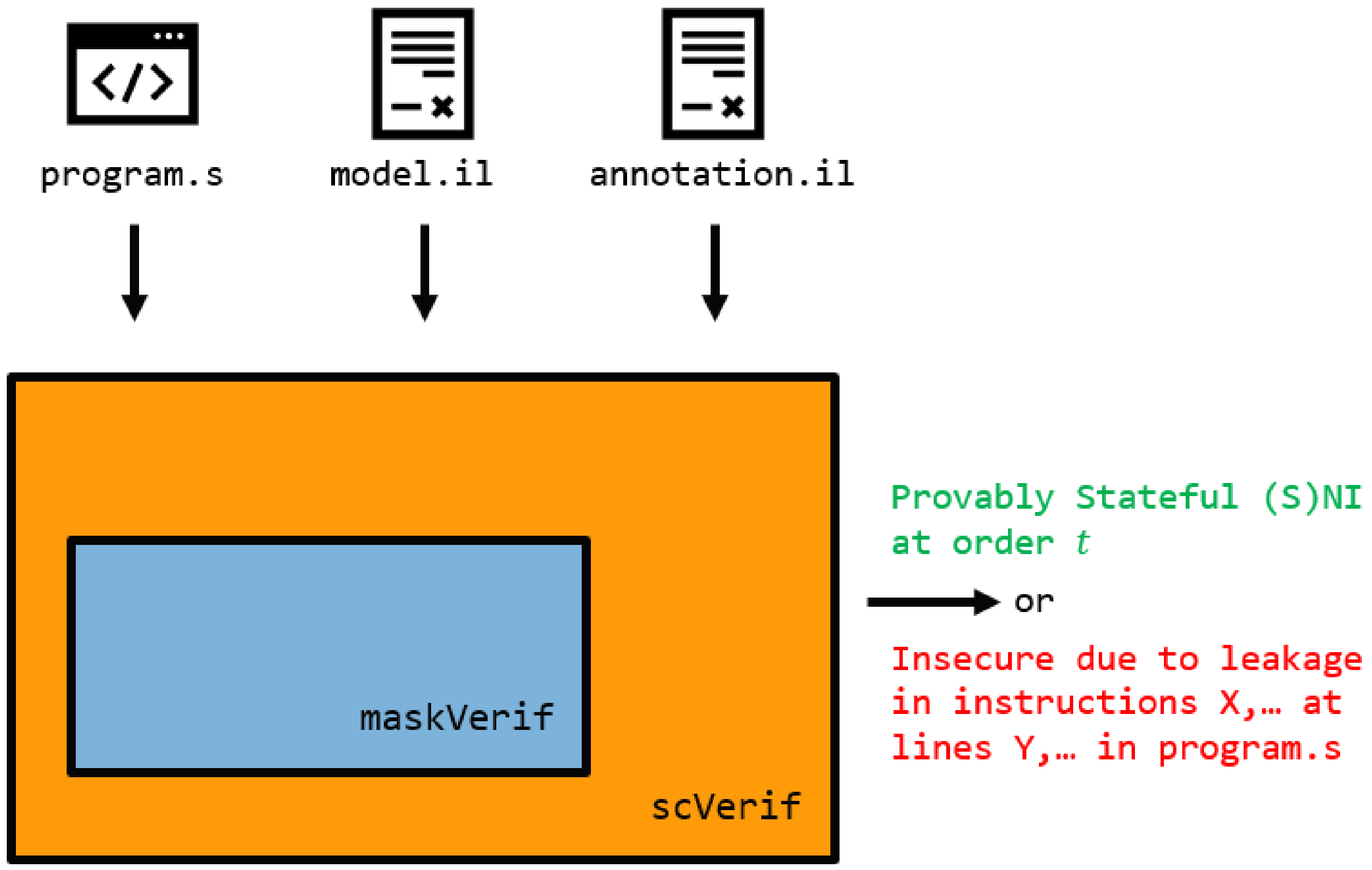
$$\frac{i = \chi \leftarrow e \quad i' = \chi' \leftarrow e' \qquad [\![\chi]\!]^\rho_\mu = (\vartheta', \chi') \quad [\![e]\!]^\rho_\mu = (\vartheta, e') \qquad (\mu, \rho)\{\chi' \leftarrow \vartheta\} = (\mu', \rho')}{\langle p, i; c, \mu, \rho, ec\rangle \rightsquigarrow \langle p, c, \mu', \rho', ec; i'\rangle}$$

$$\frac{[\![e_i]\!]^\rho_\mu = (\vartheta_i, e'_i)}{\text{leak}\ \{e_1,\dots,e_j\} \rightsquigarrow \text{leak}\ \{e'_1,\dots,e'_j\}} \qquad \frac{i = \text{goto}\ e \quad [\![e]\!]^\rho_\mu = (l, e') \quad p_l = c'}{\langle p, i; c, \mu, \rho, ec\rangle \rightsquigarrow \langle p, c', \mu, \rho, ec\rangle}$$

$$\frac{i = \text{if}\ e\ c_t\ c_f \quad [\![e]\!]^\rho_\mu = (b, e')}{\langle p, i; c, \mu, \rho, ec\rangle \rightsquigarrow \langle p, c_b; c, \mu, \rho, ec\rangle} \qquad \frac{i = \text{while}\ e\ c_w \quad i' = (\text{if}\ e\ c_w; i); c}{\langle p, i; c, \mu, \rho, ec\rangle \rightsquigarrow \langle p, i', \mu, \rho, ec\rangle}$$

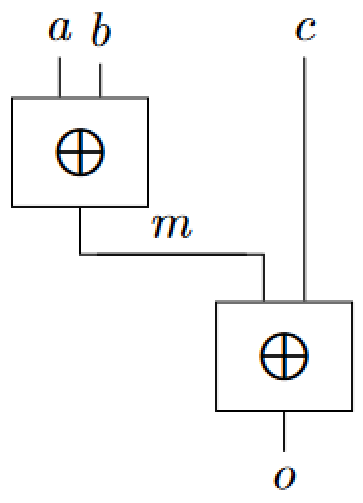
VERIFICATION FLOW OF SCVERIF

1. Represent program code using modeled instruction semantics
2. Partially evaluate using annotations
3. Verify resulting symbolic trace (representing the executed program) with maskVerif
4. Report verification result to user


program.s
model.il
annotation.il
maskVerif
scVerif
Provably Stateful (S)NI at order *t*
or
Insecure due to leakage in instructions X,… at lines Y,… in program.s

TOOL ASSISTED OPTIMIZATION STRATEGIES FOR EFFICIENT MASKING

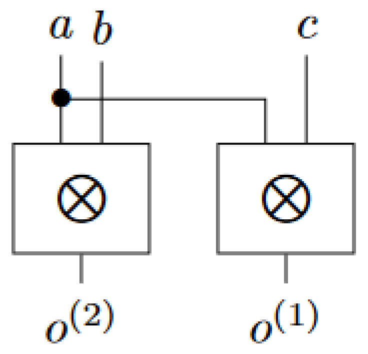
- Applied to masked Present S-box
  – speedup in dev time, speedup in exec time & program size
  + fine-tuning to device-specific leakage
  – scVerif + gadgets publicly available
- Also applied to Kyber modules
  – Very large
  – still phy. leakage free without conservative choices



- Linear compositions share-wise

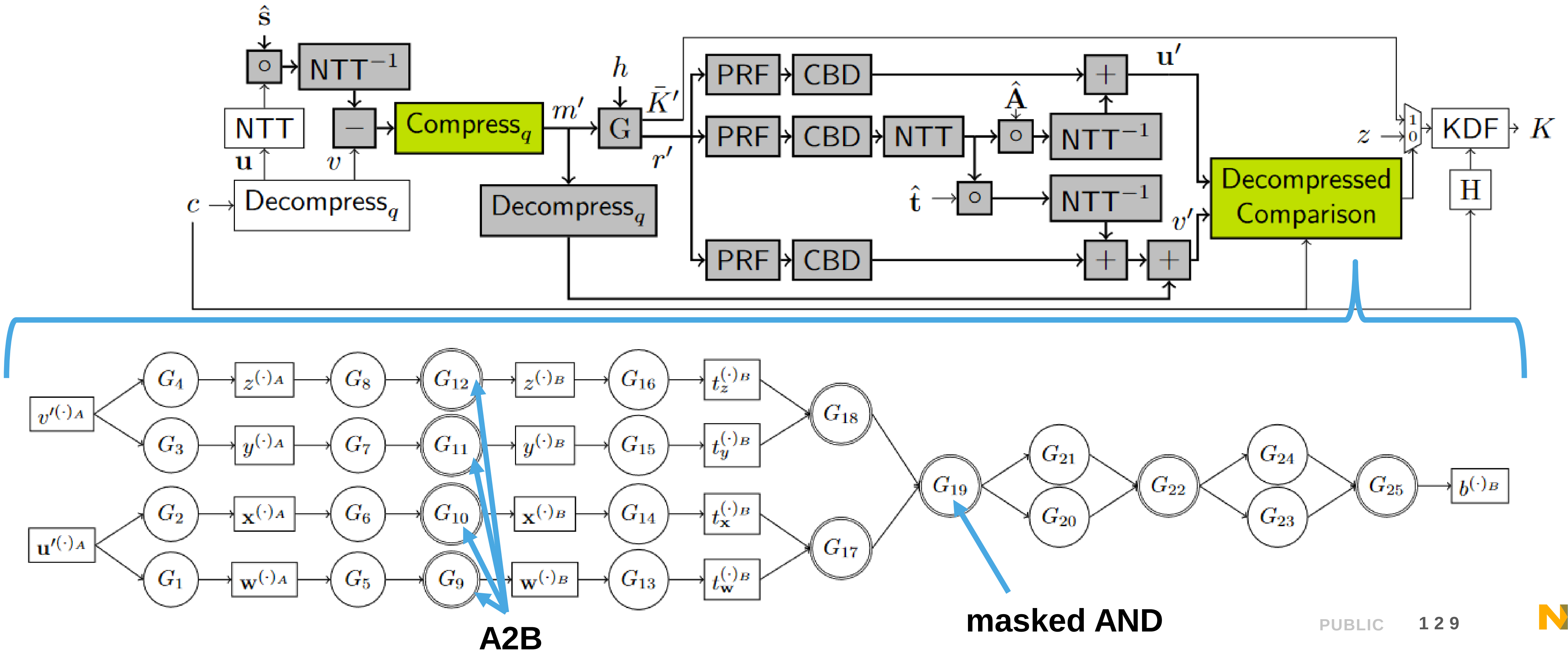
$$\hat{\mathcal{F}}(a,b,c) = \left(\left(\hat{\mathcal{F}}_i(a_i,b_i,c_i), \text{CLEAR}_i\right)_{i\in[d]}, \text{FCLEAR}\right)$$

- Merging of non-linear gadgets
  – Reduce memory access at increased complexity



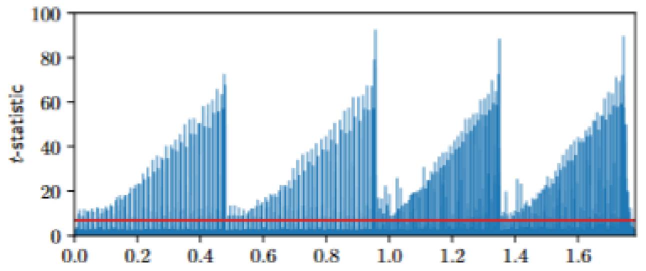
MASKING IN REAL APPLICATIONS

- Application to entire ciphers (e.g., Kyber)
- Hand-crafted composition, specialized algorithms for efficient gadgets

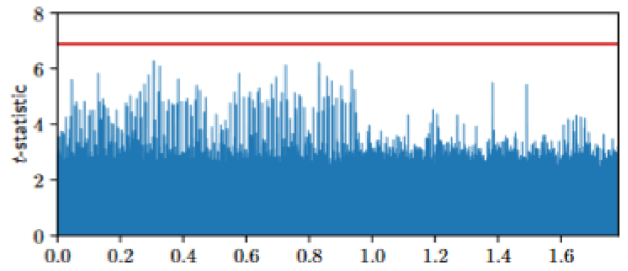


$\hat{\mathbf{s}}$
$\circ$
NTT^{-1}
NTT
$-$
Compress$_q$
m'
h
G
$\bar{K}'$
r'
$\mathbf{u}$
v
c
Decompress$_q$
Decompress$_q$
PRF
CBD
PRF
CBD
NTT
PRF
CBD
$\hat{\mathbf{A}}$
$\hat{\mathbf{t}}$
NTT^{-1}
NTT^{-1}
$+$
$\mathbf{u}'$
v'
Decompressed Comparison
z
1
0
KDF
K
H
$v'^{(\cdot)_A}$
$\mathbf{u}'^{(\cdot)_A}$
G_1 G_2 G_3 G_4 G_5 G_6 G_7 G_8 G_9 G_{10} G_{11} G_{12} G_{13} G_{14} G_{15} G_{16} G_{17} G_{18} G_{19} G_{20} G_{21} G_{22} G_{23} G_{24} G_{25}
$z^{(\cdot)_A}$ $y^{(\cdot)_A}$ $\mathbf{x}^{(\cdot)_A}$ $\mathbf{w}^{(\cdot)_A}$
$z^{(\cdot)_B}$ $y^{(\cdot)_B}$ $\mathbf{x}^{(\cdot)_B}$ $\mathbf{w}^{(\cdot)_B}$
$t_z^{(\cdot)_B}$ $t_y^{(\cdot)_B}$ $t_{\mathbf{x}}^{(\cdot)_B}$ $t_{\mathbf{w}}^{(\cdot)_B}$
$b^{(\cdot)_B}$
A2B
masked AND

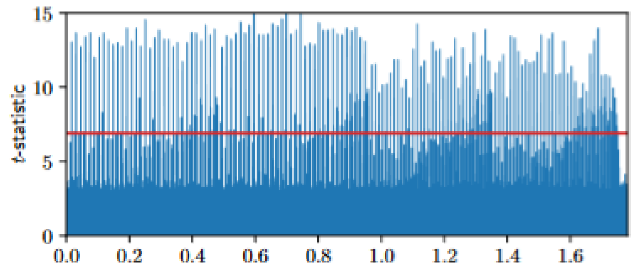
APPLICATION (2)



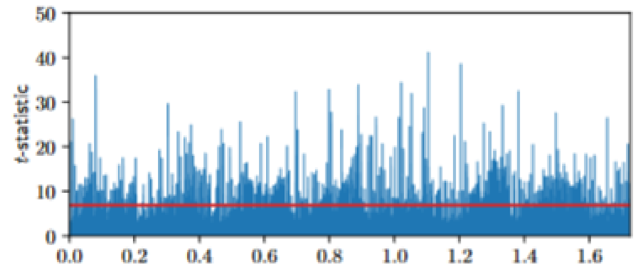
(a) 1 000 traces, RND off



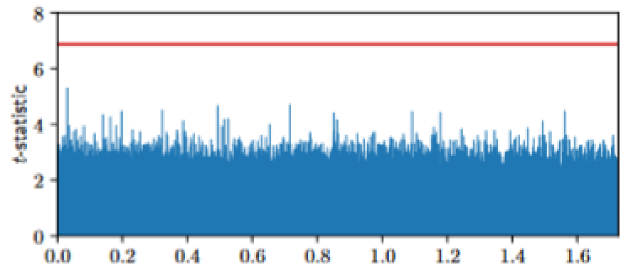
(b) 100 000 traces, RND on



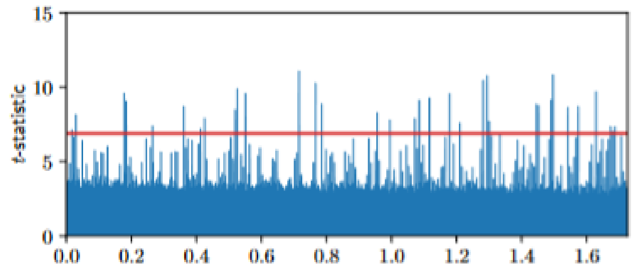
(c) 100 000 traces, 2nd order



(d) 1 000 traces, RND off



(e) 100 000 traces, RND on



(f) 100 000 traces, 2nd order

Figure 6: Results of TVLA assessment: the top row shows decompressed comparison for (a) 1st order without randomness, (b) 1st order with randomness, and (c) 2nd order with randomness, while the bottom row shows 1-bit compression for (d) 1st order without randomness, (e) 1st order with randomness, and (f) 2nd order with randomness. The x axis represents sample point index $\times 10^6$.

LIMITATIONS

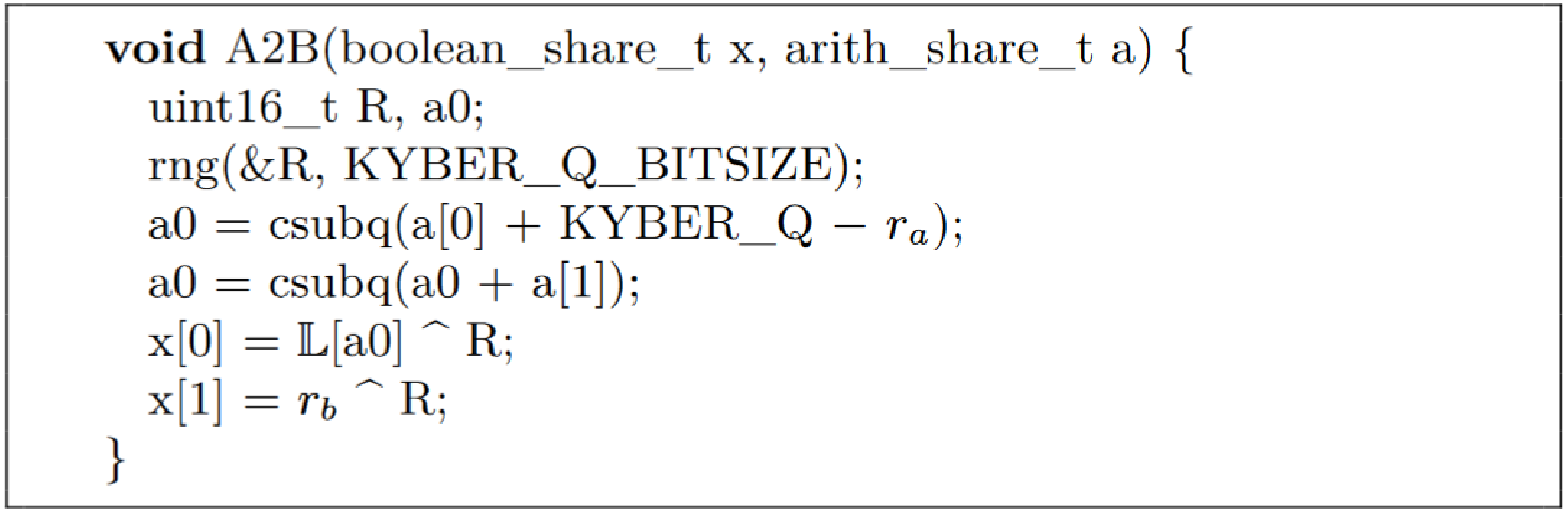
- Partial evaluation
  - `memcpy` with symbolic size
- Generic order

LIMITATIONS

- Partial evaluation
  - `memcpy` with symbolic size
- Generic order
- Secret dependent memory accesses
  - Masked table lookup ?!

LIMITATIONS

- Partial evaluation
  - `memcpy` with symbolic size
- Generic order
- Secret dependent memory accesses
  - Masked table lookup ?!



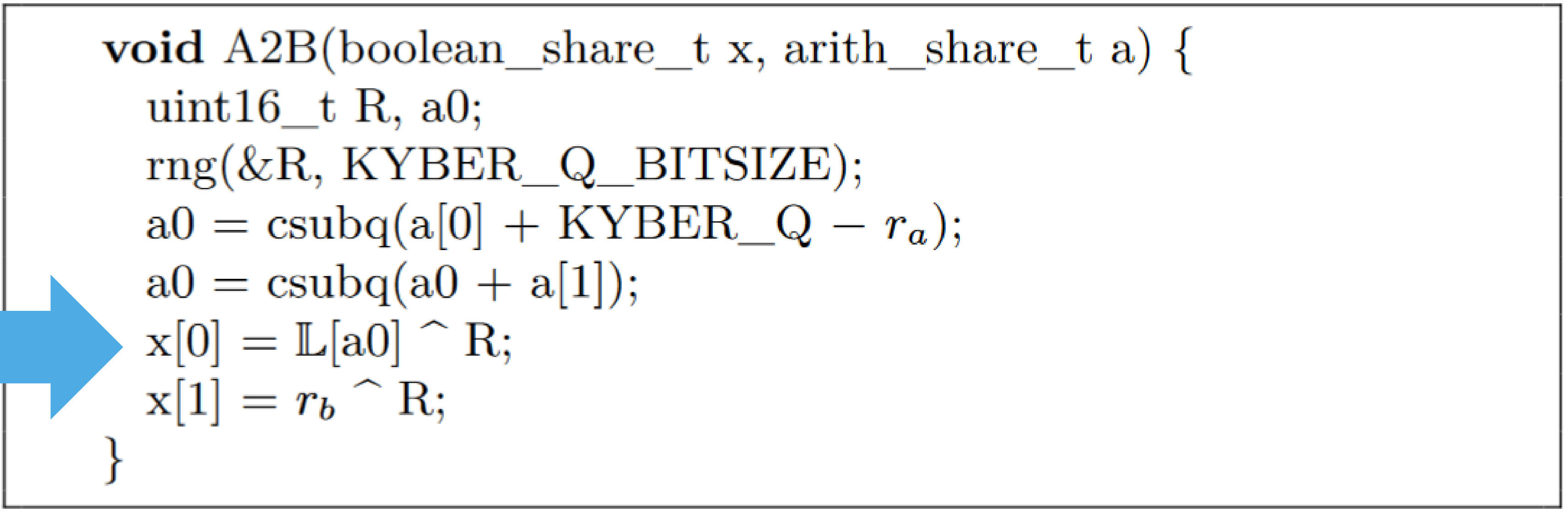
```
void A2B(boolean_share_t x, arith_share_t a) {
  uint16_t R, a0;
  rng(&R, KYBER_Q_BITSIZE);
  a0 = csubq(a[0] + KYBER_Q − r_a);
  a0 = csubq(a0 + a[1]);
  x[0] = L[a0] ^ R;
  x[1] = r_b ^ R;
}
```

Figure 5: LUT-based arithmetic to Boolean version based on [Deb12].

LIMITATIONS

- Partial evaluation
  - `memcpy` with symbolic size
- Generic order
- Secret dependent memory accesses
  - Masked table lookup ?!

```
LDR rd, a0
```



```
void A2B(boolean_share_t x, arith_share_t a) {
  uint16_t R, a0;
  rng(&R, KYBER_Q_BITSIZE);
  a0 = csubq(a[0] + KYBER_Q − r_a);
  a0 = csubq(a0 + a[1]);
  x[0] = 𝕃[a0] ^ R;
  x[1] = r_b ^ R;
}
```

Figure 5: LUT-based arithmetic to Boolean version based on [Deb12].

VERIFYING LOOK-UP-TABLES WITH SECRET DEPENDENT MEMORY ACCESS

- Replace `LDR` by virtual LUT instruction
  - Express semantic of lookup table without memory access

```
LDR rD, rIDX
  val ← mem[rIDX]
```

```
LUT rD, rIDX
  val ← f[rIDX]
```

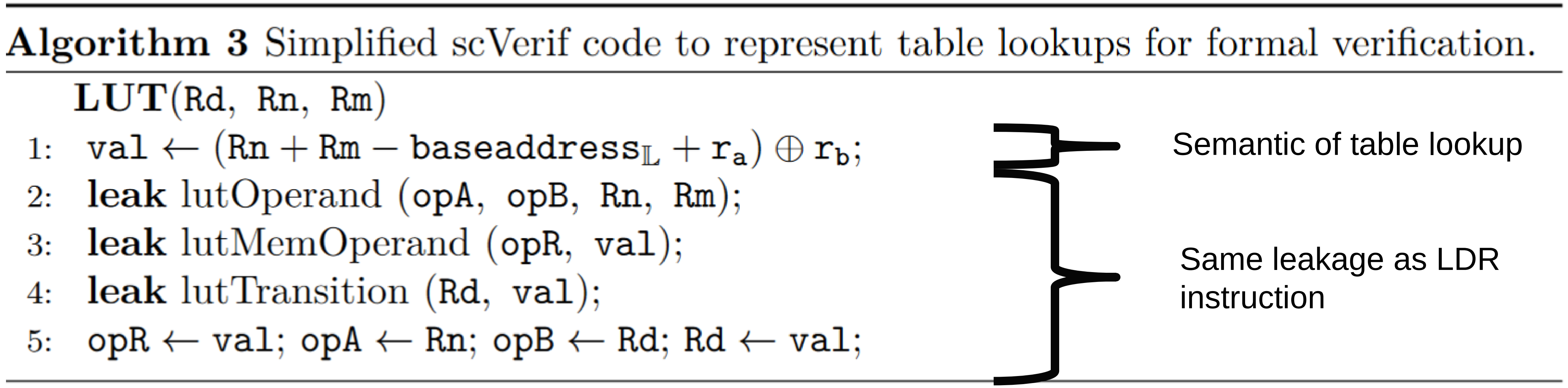
VERIFYING LOOK-UP-TABLES WITH SECRET DEPENDENT MEMORY ACCESS

- Replace `LDR` by virtual LUT instruction
  - Express semantic of lookup table without memory access

```
LDR rD, rIDX
  val ← mem[rIDX]
```

$\Rightarrow$

```
LUT rD, rIDX
  val ← f[rIDX]
```



Algorithm 3 Simplified scVerif code to represent table lookups for formal verification.

```
    **LUT**(Rd, Rn, Rm)
1:  val ← (Rn + Rm − $\texttt{baseaddress}_{\mathbb{L}}$ + $r_a$) ⊕ $r_b$;
2:  **leak** lutOperand (opA, opB, Rn, Rm);
3:  **leak** lutMemOperand (opR, val);
4:  **leak** lutTransition (Rd, val);
5:  opR ← val; opA ← Rn; opB ← Rd; Rd ← val;
```

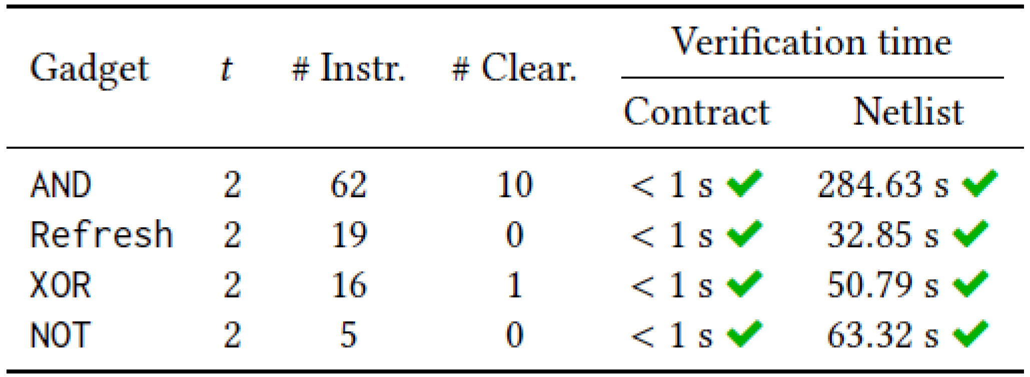
$\}$ Semantic of table lookup

$\}$ Same leakage as LDR instruction

SUMMARY

- Fine-grained models for software masking
  - Reliable & accurate
  - User-defined arbitrary leakage behavior
  - Not sacrificing efficiency

- `scVerif`
  - Fast verification
  - Accurate error reports
  - Support specialized constructions
  - Support highly-efficient masking



Gadget	t	# Instr.	# Clear.	Verification time	
				Contract	Netlist
AND	2	62	10	< 1 s ✓	284.63 s ✓
Refresh	2	19	0	< 1 s ✓	32.85 s ✓
XOR	2	16	1	< 1 s ✓	50.79 s ✓
NOT	2	5	0	< 1 s ✓	63.32 s ✓

LISTING L
LICENSE OF SHOWN CODE-SNIPPETS

SECURE CONNECTIONS
FOR A SMARTER WORLD