Formal Verification of Masked Implementations

Sonia Belaïd    Benjamin Grégoire

CHES 2018 - Tutorial
September 9th 2018
1. Side-Channel Attacks and Masking

2. Formal Tools for Verification at Fixed Order

3. Formal Tools for Verification of Generic Implementations
1. Side-Channel Attacks and Masking

2. Formal Tools for Verification at Fixed Order

3. Formal Tools for Verification of Generic Implementations
Cryptanalysis

- Black-box cryptanalysis
- Side-channel analysis

Alice $k \rightarrow m \rightarrow ENC \rightarrow c \rightarrow Bob$

$c = 011100110101010110001010$

Bob $k \rightarrow c \rightarrow DEC \rightarrow m$
Cryptanalysis

→ Black-box cryptanalysis: \( A \leftarrow (m, c) \)

→ Side-Channel Analysis

Alice \[ k \]

\[ m \rightarrow \text{ENC} \rightarrow c \]

Bob \[ k \]

\[ c = 011100110101010110001010 \rightarrow \]

\[ c \rightarrow \text{DEC} \rightarrow m \]
Cryptanalysis

- Black-box cryptanalysis
- Side-Channel Analysis: $\mathcal{A} \leftarrow (m, c, \mathcal{L})$

![Diagram showing encryption and decryption process]

$\mathcal{A} \leftarrow (m, c, \mathcal{L})$

$\leftrightarrow c = 011100110101010110001010$

Alice

$\rightarrow ENC \rightarrow c$

Bob

$\rightarrow DEC \rightarrow m$

$L$

$c$
Cryptanalysis

→ Black-box cryptanalysis

→ Side-Channel Analysis: $\mathcal{A} \leftarrow (m, c, \mathcal{L})$

Alice $\xrightarrow{k} m \rightarrow \text{ENC} \rightarrow c$

$\mathcal{L}$

Bob $\xleftarrow{k} c \rightarrow \text{DEC} \rightarrow m$

$c = 011100110101010110001010$
Cryptanalysis

→ Black-box cryptanalysis

→ Side-Channel Analysis: $\mathcal{A} \leftarrow (m, c, \mathcal{L})$

$\begin{align*}
\text{Alice} & \quad c = 011100110101010110001010 \\
\begin{array}{c}
m \\
\downarrow \text{ENC} \\
k
\end{array} & \quad c \\
\text{Bob} & \quad c = 011100110101010110001010 \\
\begin{array}{c}
\downarrow \text{DEC} \\
k
\end{array} & \quad m
\end{align*}$
Cryptanalysis

- Black-box cryptanalysis
- Side-Channel Analysis: \( A \leftarrow (m, c, \mathcal{L}) \)

Alice \( k \)
\( m \rightarrow \text{ENC} \rightarrow c \)

Bob \( k \)
\( c \rightarrow \text{DEC} \rightarrow m \)

\( c = 011100110101010110001010 \)
Cryptanalysis

- Black-box cryptanalysis
- Side-Channel Analysis: $A \leftarrow (m, c, \mathcal{L})$

Alice $\xrightarrow{k} m \rightarrow \text{ENC} \rightarrow c$  $\xleftarrow{c=011100110101010110001010}$  Bob $\xrightarrow{k} c \rightarrow \text{DEC} \rightarrow m$

\[ L \]
Example of SPA

Algorithm 1 Example

for $i = 1$ to $n$ do
  if $\text{key}[i] = 0$ then
    do treatment 0
  else
    do treatment 1
  end if
end for

SPA: one single trace to recover the secret key

<table>
<thead>
<tr>
<th>treatment 0</th>
<th>treatment 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

secret = 1011100101001
Example of DPA

DPA: several traces to recover the secret key
How to thwart SCA?

Issue: leakage $\mathcal{L}$ is key-dependent

$$v_0 \leftarrow v \oplus (\bigoplus_{1 \leq i \leq t} v_i)$$

$$v_1 \leftarrow \ldots$$

$$v_t \leftarrow \ldots$$

Any $t$-uple of $v_i$ is independent from $v_0$.
How to thwart SCA?

Issue: leakage $\mathcal{L}$ is key-dependent

Idea of masking: make leakage $\mathcal{L}$ random

Sensitive value: $v = f(m, k)$

$v_0 \leftarrow v \oplus \left( \bigoplus_{1 \leq i \leq t} v_i \right)$

$v_1 \leftarrow \$

$\ldots$

$v_t \leftarrow \$

$\Rightarrow$ any $t$-uple of $v_i$ is independent from $v$
Masked Implementations

- Linear functions: apply the function to each share

\[ v \oplus w \rightarrow (v_0 \oplus w_0, v_1 \oplus w_1, \ldots, v_t \oplus w_t) \]
Masked Implementations

- **Linear functions**: apply the function to each share
  \[ v \oplus w \rightarrow (v_0 \oplus w_0, v_1 \oplus w_1, \ldots, v_t \oplus w_t) \]

- **Non-linear functions**: much more complex
  \[
  \begin{align*}
  \forall 0 \leq i < j \leq t - 1, & \quad r_{i,j} \leftarrow $
  \\
  \forall 0 \leq i < j \leq t - 1, & \quad r_{j,i} \leftarrow (r_{i,j} \oplus v_i w_j) \oplus v_j w_i \\
  \forall 0 \leq i \leq d - 1, & \quad c_i \leftarrow v_i w_i \oplus \sum_{j \neq i} r_{i,j} \\
  vw & \rightarrow (c_0, c_1, \ldots, c_t)
  \end{align*}
  \]
Leakage Models

- **Probing model** by Ishai, Sahai, and Wagner (Crypto 2003)
  - a circuit is $t$-probing secure iff any set composed of the **exact values** of at most $t$ intermediate variables is independent from the secret

- **Noisy leakage model** by Chari, Jutla, Rao, and Rohatgi (Crypto 1999) then Rivain and Prouff (EC 2013)
  - a circuit is secure in the noisy leakage model iff the adversary cannot recover information on the secret from the noisy values of all the intermediate variables

- **Reduction** by Duc, Dziembowski, and Faust (EC 2014)
  - $t$-probing security $\implies$ security in the noisy leakage model for some level of noise
Leakage Models

- **Probing model** by Ishai, Sahai, and Wagner (Crypto 2003)
  - a circuit is $t$-probing secure iff any set composed of the exact values of at most $t$ intermediate variables is independent from the secret

- **Noisy leakage model** by Chari, Jutla, Rao, and Rohatgi (Crypto 1999) then Rivain and Prouff (EC 2013)
  - a circuit is secure in the noisy leakage model iff the adversary cannot recover information on the secret from the noisy values of all the intermediate variables
Leakage Models

- **Probing model** by Ishai, Sahai, and Wagner (Crypto 2003)
  - a circuit is $t$-probing secure iff any set composed of the exact values of at most $t$ intermediate variables is independent from the secret

- **Noisy leakage model** by Chari, Jutla, Rao, and Rohatgi (Crypto 1999) then Rivain and Prouff (EC 2013)
  - a circuit is secure in the noisy leakage model iff the adversary cannot recover information on the secret from the noisy values of all the intermediate variables

- **Reduction** by Duc, Dziembowskii, and Faust (EC 2014)
  - $t$-probing security $\Rightarrow$ security in the noisy leakage model for some level of noise
How to Verify Probing Security?

- variables: secret, shares, constant
- masking order $t = 3$

```plaintext
function Ex-t3($x_0, x_1, x_2, x_3, c$):
(* $x_0, x_1, x_2 = \$ * )
(* $x_3 = x + x_0 + x_1 + x_2 * )

$\begin{align*}
    r_0 & \leftarrow \$ \\
    r_1 & \leftarrow \$ \\
    y_0 & \leftarrow x_0 + r_0 \\
    y_1 & \leftarrow x_3 + r_1 \\
    t_1 & \leftarrow x_1 + r_0 \\
    t_2 & \leftarrow (x_1 + r_0) + x_2 \\
    y_2 & \leftarrow (x_1 + r_0 + x_2) + r_1 \\
    y_3 & \leftarrow c + r_1
\end{align*}$

return($y_0, y_1, y_2, y_3$)
```
How to Verify Probing Security?

- Variables: secret, shares, constant
- Masking order $t = 3$

```plaintext
function Ex-t3(x_0, x_1, x_2, x_3, c):
    (* $x_0, x_1, x_2 = \$ * )
    (* $x_3 = x + x_0 + x_1 + x_2 * )

    r_0 ← \$
    r_1 ← \$
    y_0 ← x_0 + r_0
    y_1 ← x_3 + r_1
    t_1 ← x_1 + r_0
    t_2 ← (x_1 + r_0) + x_2
    y_2 ← (x_1 + r_0 + x_2) + r_1
    y_3 ← c + r_1
    return(y_0, y_1, y_2, y_3)
```

Independent from the secret?
How to Verify Probing Security?

- variables: secret, shares, constant
- masking order $t = 3$

```plaintext
function Ex-t3(x_0, x_1, x_2, x_3, c):

(* $x_0, x_1, x_2 = \$ (*)
(* $x_3 = x + x_0 + x_1 + x_2 (*)

r_0 \leftarrow \$

r_1 \leftarrow \$

y_0 \leftarrow x_0 + r_0

y_1 \leftarrow x_3 + r_1

t_1 \leftarrow x_1 + r_0

t_2 \leftarrow (x_1 + r_0) + x_2

y_2 \leftarrow (x_1 + r_0 + x_2) + r_1

y_3 \leftarrow c + r_1

return(y_0, y_1, y_2, y_3)
```

independent from the secret?
Non-Interference (NI)

- $t$-NI $\Rightarrow$ $t$-probing secure
- A circuit is $t$-NI iff any set of $t$ intermediate variables can be perfectly simulated with at most $t$ shares of each input

```plaintext
function Ex-t3(x_0, x_1, x_2, x_3, c):
(* x_0, x_1, x_2 = $ *)
(* x_3 = x + x_0 + x_1 + x_2 *)

r_0 ← $

r_1 ← $

y_0 ← x_0 + r_0
y_1 ← x_3 + r_1

\[ t_1 \] ← x_1 + r_0

\[ t_2 \] ← (x_1 + r_0) + x_2

\[ y_2 \] ← (x_1 + r_0 + x_2) + r_1

y_3 ← c + r_1

return(y_0, y_1, y_2, y_3)
```

can be simulated with $x_0$ and $x_1$
Non-Interference (NI)

- \( t\text{-NI} \Rightarrow t\text{-probing secure} \)
- A circuit is \( t\text{-NI} \) iff any set of \( t \) intermediate variables can be perfectly simulated with at most \( t \) shares of each input

\[
\begin{align*}
\begin{array}{c}
x_0, x_1, x_2, x_3 \\
y_0, y_1, y_2, y_3 \\
\text{Ex-t3}
\end{array}
\end{align*}
\]

\( (= x + x_0 + x_1 + x_2) \)

3 observations
1. Side-Channel Attacks and Masking

2. Formal Tools for Verification at Fixed Order

3. Formal Tools for Verification of Generic Implementations
State-Of-The-Art

- several tools were built to formally verify security of first-order implementations $t = 1$
- then a sequence of work tackled higher-order implementations $t \leq 5$
  - `maskVerif` from Barthe et al.: first tool to achieve verification at high orders
  - `CheckMasks` from Coron: improvements in terms of efficiency
  - Bloem et al.’s tool: treatment of glitches attacks
State-Of-The-Art

- several tools were built to formally verify security of first-order implementations $t = 1$
- then a sequence of work tackled higher-order implementations $t \leq 5$
  - *maskVerif* from Barthe et al.: first tool to achieve verification at high orders
  - *CheckMasks* from Coron: improvements in terms of efficiency
  - Bloem et al.’s tool: treatment of glitches attacks
maskVerif

- **input:**
  - pseudo-code of a masked implementation
  - order $t$

- **output:**
  - formal proof of $t$-probing security (or NI, SNI)
  - potential flaws

---

Checking probabilistic independence

Problem: Check if a program expression $e$ is probabilistic independent from a secret $s$

Example: $e = (s \oplus r_1) \cdot (r_1 \oplus r_2)$

First solution:
- for each value of $s$ computes the associate distribution of $e$
- if all the resulting distribution are equals then $e$ is independent of $s$

\[
\begin{align*}
&\begin{cases}
  r_1 & r_2 & e \\
  0 & 0 & 0 \\
  0 & 1 & 0 \\
  1 & 0 & 1 \\
  1 & 1 & 0 \\
\end{cases} \\
&s = 0 \\
&\begin{cases}
  r_1 & r_2 & e \\
  0 & 0 & 0 \\
  0 & 1 & 1 \\
  1 & 0 & 0 \\
  1 & 1 & 0 \\
\end{cases} \\
&s = 1
\end{align*}
\]
Checking probabilistic independence

Problem: Check if a program expression $e$ is probabilistic independent from a secret $s$
Example: $e = (s \oplus r_1) \cdot (r_1 \oplus r_2)$

First solution:
- for each value of $s$ computes the associate distribution of $e$
- if all the resulting distribution are equals then $e$ is independent of $s$

- Complete
- Exponential in the number of secret and random values
Checking probabilistic independence

Second solution, using simple rules:

- Rule 1: If \( e \) does not use \( s \) then it is independent

- Rule 2: If \( e \) can be written as \( C[f \oplus r] \) and \( r \) does not occur in \( C \) and \( f \) then it is sufficient to test the independence of \( C[r] \)

- Rule 3: If Rules 1 and 2 do not apply then use the first solution (when possible)

Problem: finding occurrence of Rule 2 is relatively costly
Checking probabilistic independence

Second solution, using simple rules:

- Rule 1: If \( e \) does not use \( s \) then it is independent.
- Rule 2: If \( e \) can be written as \( C[f \oplus r] \) and \( r \) does not occur in \( C \) and \( f \) then it is sufficient to test the independence of \( C[r] \).

The distribution of \( f \oplus r \) is equal to the distribution of \( r \).
Checking probabilistic independence

Second solution, using simple rules:

- Rule 1: If $e$ does not use $s$ then it is independent
- Rule 2: If $e$ can be written as $C[f \oplus r]$ and $r$ does not occur in $C$ and $f$ then it is sufficient to test the independence of $C[r]$
- Rule 3: If Rules 1 and 2 do not apply then use the first solution (when possible)
Second solution, using simple rules:

- **Rule 1:** If $e$ does not use $s$ then it is independent.
- **Rule 2:** If $e$ can be written as $C[f \oplus r]$ and $r$ does not occur in $C$ and $f$ then it is sufficient to test the independence of $C[r]$.
- **Rule 3:** If Rules 1 and 2 do not apply then use the first solution (when possible).

**Problem:** finding occurrence of Rule 2 is relatively costly.
Independence: dag representation

\[(s \oplus r_1) \cdot (r_1 \oplus r_2)\]
Independence: dag representation

\[(s \oplus r_1) \cdot r_2\]

Diagram:

- Node labeled \(s\)
- Node labeled \(r_1\)
- Node labeled \(r_2\)
- Node labeled \(\oplus\)
- Node labeled \(\cdot\)
Independence: dag representation

\[ r_1 \cdot r_2 \]

Independent from the secret
First order Dom AND : NI

\[
\begin{align*}
\{a_0\} & \rightarrow \{a_0 \otimes b_0\} & \{a_0 \otimes b_1 \oplus r\} \\
\{a_0\} & \rightarrow \{a_1 \otimes b_0\} & \{a_0 \otimes b_1 \oplus a_0 \otimes b_1 \oplus r\} \\
\{b_0\} & \rightarrow \{b_0 \otimes b_1\} & \{a_1 \otimes b_1 \oplus a_0 \otimes b_1 \oplus r\} \\
\{b_0\} & \rightarrow \{b_1 \otimes b_0\} & \{a_1 \otimes b_0 \oplus a_1 \otimes b_0 \oplus r\} \\
\{b_1\} & \rightarrow \{b_1 \otimes b_1\} & \{a_1 \otimes b_1 \oplus a_0 \otimes b_1 \oplus r\} \\
\{b_1\} & \rightarrow \{b_0 \otimes b_0\} & \{a_0 \otimes b_0 \oplus a_1 \otimes b_0 \oplus r\} \\
\{r\} & \rightarrow \{a_0 \otimes b_1 \oplus r\} & \{a_0 \otimes b_0 \oplus r\} \\
\{r\} & \rightarrow \{a_1 \otimes b_1 \oplus r\} & \{a_0 \otimes b_0 \oplus a_1 \otimes b_0 \oplus r\} \\
\end{align*}
\]
Verification of first order masking is just a linear iteration of the previous algorithm (one call for each program point) 100 checks for a program of 100 lines
Extension to All Possible Sets

- Verification of first order masking is just a linear iteration of the previous algorithm (one call for each program point)
  100 checks for a program of 100 lines

- For second order masking:
  forall pair of program point, the corresponding pair of expressions is independent from the secrets
  4,950 checks for a program of 100 lines
Extension to All Possible Sets

- Verification of first order masking is just a linear iteration of the previous algorithm (one call for each program point)
  100 checks for a program of 100 lines

- For second order masking:
  forall pair of program point, the corresponding pair of expressions is independent from the secrets
  4,950 checks for a program of 100 lines

- For $t$-order masking:
  forall $t$-tuple of program point, the corresponding $t$-tuple of expressions is independent from the secrets
  \[ \binom{N}{t} \] where $N$ is the number program points
  3,921,225 for a program of 100 lines and 4 observations
Extension to All Possible Sets

Idea: if $e_1, \ldots, e_p$ is independent from the secrets then all subtuples are independent from the secrets.
Extension to All Possible Sets

Idea: if \( e_1, \ldots, e_p \) is independent from the secrets then all subtuples are independent from the secrets.
Extension to All Possible Sets

Idea: if $e_1, \ldots, e_p$ is independent from the secrets then all subtuples are independent from the secrets.

1. select $X = (t \text{ variables})$ and prove its independence
Extension to All Possible Sets

Idea: if $e_1, \ldots, e_p$ is independent from the secrets then all subtuples are independent from the secrets.

1. select $X = (t$ variables) and prove its independence
2. extend $X$ to $\hat{X}$ with more observations but still independence
Extension to All Possible Sets

Idea: if $e_1, \ldots, e_p$ is independent from the secrets then all subtuples are independent from the secrets.

1. select $X = (t$ variables$)$ and prove its independence
2. extend $X$ to $\hat{X}$ with more observations but still independence
3. recursively descend in set $C(\hat{X})$
Extension to All Possible Sets

Idea: if $e_1, \ldots, e_p$ is independent from the secrets then all subtuples are independent from the secrets.

1. select $X = (t$ variables) and prove its independence
2. extend $X$ to $\hat{X}$ with more observations but still independence
3. recursively descend in set $C(\hat{X})$
4. merge $\hat{X}$ and $C(\hat{X})$ once they are processed separately.
Extension to All Possible Sets

Idea: if $e_1, \ldots, e_p$ is independent from the secrets then all subtuples are independent from the secrets.

1. select $X = (t \text{ variables})$ and prove its independence
2. extend $X$ to $\hat{X}$ with more observations but still independence
3. recursively descend in set $C(\hat{X})$
4. merge $\hat{X}$ and $C(\hat{X})$ once they are processed separately.

Finding $\hat{X}$ can be done very efficiently using a dag representation.
Benchmark

It is working for relatively small programs:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Order</th>
<th>Tuples</th>
<th>Secure</th>
<th>Verification time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Refresh</td>
<td>9</td>
<td>$2.10^{10}$</td>
<td>✓</td>
<td>2s</td>
</tr>
<tr>
<td>Refresh</td>
<td>17</td>
<td>$2.10^{20}$</td>
<td>✓</td>
<td>3d</td>
</tr>
<tr>
<td>Refresh</td>
<td>18</td>
<td>$4.10^{21}$</td>
<td>✓</td>
<td>1 month</td>
</tr>
</tbody>
</table>

But there is a problem with large programs:
- Full AES implementation at order 1
- only 4 rounds of AES at order 2
Demo

https://sites.google.com/view/maskverif/home

Demo maskVerif
Extending the model: glitches

For hardware implementation a more realistic model need to take into account glitches

Example: AND gate $A \otimes B$

Possible leaks: $A \cdot B$, $A$, $B$
First order DOM AND : NI with glitches
First order DOM AND : NI with glitches
First order DOM AND : NI with glitches
First order DOM AND : NI with glitches
First order DOM AND : NI with glitches
First order DOM AND : NI with glitches
First order DOM AND : NI with glitches
Hardware implementation

We have extended maskVerif to

- take Verilog implementation as input
- take extra information on input shares (random, shares secret, public input)
- Check the security with or without glitches
Demo Hardware

https://sites.google.com/view/maskverif/home

yosys + maskVerif
Examples (provided by Bloem et al)

<table>
<thead>
<tr>
<th>Algo</th>
<th># obs</th>
<th>probing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>wG</td>
</tr>
<tr>
<td><strong>first-order verification</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trichina AND</td>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td>ISW AND</td>
<td>1</td>
<td>13</td>
</tr>
<tr>
<td>DOM AND</td>
<td>4</td>
<td>13</td>
</tr>
<tr>
<td>DOM Keccak S-box</td>
<td>20</td>
<td>76</td>
</tr>
<tr>
<td>DOM AES S-box</td>
<td>96</td>
<td>571</td>
</tr>
<tr>
<td><strong>second-order verification</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DOM Keccak S-box</td>
<td>60</td>
<td>165</td>
</tr>
<tr>
<td><strong>third-order verification</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DOM Keccak S-box</td>
<td>100</td>
<td>290</td>
</tr>
<tr>
<td><strong>fourth-order verification</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DOM Keccak S-box</td>
<td>150</td>
<td>450</td>
</tr>
<tr>
<td><strong>fifth-order verification</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DOM Keccak S-box</td>
<td>210</td>
<td>618</td>
</tr>
</tbody>
</table>
1. Side-Channel Attacks and Masking

2. Formal Tools for Verification at Fixed Order

3. Formal Tools for Verification of Generic Implementations
Probing Model

Require: Encoding \([x]\)
Ensure: Fresh encoding \([x]\)

\[
\begin{align*}
\text{for } i = 1 \text{ to } t \text{ do} \\
\quad & r \leftarrow \$ \\
\quad & x_0 \leftarrow x_0 + r \\
\quad & x_i \leftarrow x_i + r \\
\text{end for} \\
\text{return } [x]
\end{align*}
\]
Probing Model

Require: Encoding \([x]\)
Ensure: Fresh encoding \([x]\)

\[
\text{for } i = 1 \text{ to } t \text{ do}
\]
\[
\begin{align*}
    r & \gets \$ \\
    x_0 & \gets x_0 + r \\
    x_i & \gets x_i + r
\end{align*}
\]
end for

return \([x]\)

Simulation-based proof:

- show that any set of \(t\) variables can be simulated with at most \(t\) input shares \(x_i\)
- any set of \(t\) shares \(x_i\) is independent from \(x\)
Probing Model

Require: Encoding $[x]$
Ensure: Fresh encoding $[x]$

\[
\text{for } i = 1 \text{ to } t \text{ do}
\]
\[
\begin{align*}
  r &\leftarrow \$
  x_0 &\leftarrow x_0 + r \\
  x_i &\leftarrow x_i + r
\end{align*}
\]
end for

return $[x]$

Simulation-based proof:

- show that any set of $t$ variables can be simulated with at most $t$ input shares $x_i$
- any set of $t$ shares $x_i$ is independent from $x$
- exactly relies on the notion of non interference (NI)
And then?

once done for small gadgets, how to extend it?
Until Recently

- composition probing secure for $2t + 1$ shares
- no solution for $t + 1$ shares
First Proposal

- Rivain and Prouff (CHES 2010): add refresh gadgets (NI)
- Example: AES S-box on $\mathbb{GF}(2^8)$

\[
\begin{align*}
\text{Require: } & \text{Encoding } [x] \\
\text{Ensure: } & \text{Fresh encoding } [x]
\end{align*}
\]

\[\begin{align*}
\text{for } i = 1 \text{ to } t \text{ do} & \\
& r \leftarrow \$ \\
& x_0 \leftarrow x_0 + r \\
& x_i \leftarrow x_i + r \\
\text{end for} & \\
\text{return } [x]
\end{align*}\]
First Proposal

- Rivain and Prouff (CHES 2010): add refresh gadgets (NI)
- Example: AES S-box on $\text{GF}(2^8)$

\[
\begin{align*}
&\times \quad [x] \\
&\quad \downarrow \quad [\cdot^2] \\
&\quad \downarrow \quad [\cdot^2] \\
\end{align*}

\[
\begin{align*}
&\times \quad [x] \\
&\quad \downarrow \\
&\quad \downarrow \\
&\quad \downarrow \\
&\quad \uparrow \\
\end{align*}
\]

\[
\begin{align*}
&\times \\
&\quad \downarrow \\
&\quad \downarrow \\
&\quad \downarrow \\
&\quad \uparrow \\
\end{align*}
\]

\[
\begin{align*}
\text{Require: } & \text{Encoding } [x] \\
\text{Ensure: } & \text{Fresh encoding } [x] \\
& \text{for } i = 1 \text{ to } t \text{ do} \\
& \quad r \leftarrow \$ \\
& \quad x_0 \leftarrow x_0 + r \\
& \quad x_i \leftarrow x_i + r \\
& \text{end for} \\
& \text{return } [x]
\end{align*}
\]

$\Rightarrow$ Flaw from $t = 2$ (FSE 2013: Coron, Prouff, Rivain, and Roche)
Why This Flaw?

- Rivain and Prouff (CHES 2010): add refresh gadgets (NI)
- Example: AES S-box on $\text{GF}(2^8)$

Constraint:

$$t_0 + t_1 + t_2 + t_3 \leq t$$
Why This Flaw?

- Rivain and Prouff (CHES 2010): add refresh gadgets (NI)
- Example: AES S-box on $\text{GF}(2^8)$

\[
\begin{align*}
\text{Constraint:} & \\
& t_0 + t_1 + t_2 + t_3 \leq t
\end{align*}
\]
Why This Flaw?

- Rivain and Prouff (CHES 2010): add refresh gadgets (NI)
- Example: AES S-box on $\text{GF}(2^8)$

Constraint:

$$t_0 + t_1 + t_2 + t_3 \leq t$$
Why This Flaw?

- Rivain and Prouff (CHES 2010): add refresh gadgets (NI)
- Example: AES S-box on GF($2^8$)

Constraint:

$$t_0 + t_1 + t_2 + t_3 \leq t$$
Why This Flaw?

- Rivain and Prouff (CHES 2010): add refresh gadgets (NI)
- Example: AES S-box on $\text{GF}(2^8)$

Constraint:
$$t_0 + t_1 + t_2 + t_3 \leq t$$
Why This Flaw?

- Rivain and Prouff (CHES 2010): add refresh gadgets (NI)
- Example: AES S-box on GF(2^8)

\[
\begin{align*}
    t_0 + t_3 & \leq t \\
    t_1 + t_2 + t_3 & \leq t
\end{align*}
\]
Why This Flaw?

- Rivain and Prouff (CHES 2010): add refresh gadgets (NI)
- Example: AES S-box on GF(2^8)

\[
\begin{align*}
 t_0 + t_3 \\
+ t_1 + t_2 + t_3 \{ [x] \\
\text{observations} \\
\} \\
\{ [\times] \\\n\} \\
\{ R \} \\
\{ [\cdot^2] \} \\
\text{Constraint:} \\
t_0 + t_1 + t_2 + t_3 \leq t
\end{align*}
\]
Second Proposal

- Barthe, B., Dupressoir, Fouque, Grégoire, Strub, Zucchi (CCS 2016): add stronger refresh gadgets (SNI)
- Example: AES S-box on GF($2^8$)

Formal security proof for any order $t$:

```
Require: Encoding [x]
Ensure: Fresh encoding [x]

for i = 0 to t do
  for j = i + 1 to t do
    r ← $\$ ;
    $x_i \leftarrow x_i + r$
    $x_j \leftarrow x_j + r$
  end for
end for
```

return [x]
Second Proposal

- Barthe, B., Dupressoir, Fouque, Grégoire, Strub, Zucchini (CCS 2016): add stronger refresh gadgets (SNI)
- Example: AES S-box on $\text{GF}(2^8)$

\[
\begin{aligned}
x \cdot \cdot^2 &\quad \Rightarrow \quad \text{Formal security proof for any order } t
\end{aligned}
\]

\[
\begin{aligned}
\text{Require: } & \quad \text{Encoding } [x] \\
\text{Ensure: } & \quad \text{Fresh encoding } [x] \\
\text{for } i = 0 \text{ to } t \text{ do} & \\
\quad & \text{for } j = i + 1 \text{ to } t \text{ do} & \\
\quad & \quad r \leftarrow \$ & \\
\quad & \quad x_i \leftarrow x_i + r & \\
\quad & \quad x_j \leftarrow x_j + r & \\
\quad & \text{end for} & \\
\text{end for} & \\
\text{return } [x] & \\
\end{aligned}
\]
Strong Non-Interference (SNI)

- $t$-SNI $\Rightarrow$ $t$-NI $\Rightarrow$ $t$-probing secure
- a circuit is $t$-SNI iff any set of $t$ intermediate variables, whose $t_1$ on the internal variables and $t_2$ and the outputs, can be perfectly simulated with at most $t_1$ shares of each input

```latex
\begin{align*}
\text{function } \text{Ex-t3}(x_0, x_1, x_2, x_3, c): \\
&(* x_0, x_1, x_2 = \$ *) \\
&(* x_3 = x + x_0 + x_1 + x_2 *) \\
&r_0 \leftarrow \$ \\
&r_1 \leftarrow \$ \\
&y_0 \leftarrow x_0 + r_0 \\
&y_1 \leftarrow x_3 + r_1 \\
&t_1 \leftarrow x_1 + r_0 \\
&t_2 \leftarrow (x_1 + r_0) + x_2 \\
&y_2 \leftarrow (x_1 + r_0 + x_2) + r_1 \\
&y_3 \leftarrow c + r_1 \\
&\text{return}(y_0, y_1, y_2, y_3)
\end{align*}
```

require $x_0$ and $x_1$ to be perfectly simulated $\Rightarrow$ not 3-SNI since $y_0$ is an output variable
Strong Non-Interference (SNI)

- $t$-SNI $\Rightarrow$ $t$-NI $\Rightarrow$ $t$-probing secure
- a circuit is $t$-SNI iff any set of $t$ intermediate variables, whose $t_1$ on the internal variables and $t_2$ and the outputs, can be perfectly simulated with at most $t_1$ shares of each input

![Diagram of a circuit with inputs $x_0, x_1, x_2, x_3$ and outputs $y_0, y_1, y_2, y_3$, labeled with refresh and observations. There are 2 internal observations and 1 output observation.]
Barthe, B., Dupressoir, Fouque, Grégoire, Strub, Zucchini (CCS 2016): add **stronger** refresh gadgets (SNI)

Example: AES S-box on $\text{GF}(2^8)$

Constraint:
\[
t_0 + t_1 + t_2 + t_3 \leq t
\]
Barthe, B., Dupressoir, Fouque, Grégoire, Strub, Zucchini (CCS 2016): add stronger refresh gadgets (SNI)

Example: AES S-box on $\text{GF}(2^8)$

Constraint:
$$t_0 + t_1 + t_2 + t_3 \leq t$$
Why Does It Work?

- Barthe, B., Dupressoir, Fouque, Grégoire, Strub, Zucchini (CCS 2016): add *stronger* refresh gadgets (SNI)
- Example: AES S-box on GF($2^8$)

**Constraint:**

$$t_0 + t_1 + t_2 + t_3 \leq t$$

Diagram:

- $t_0 + t_3$ observations
- $t_1$ observations
- $t_2$ internal observations
- $t_3$ output observations
Why Does It Works?

- Barthe, B., Dupressoir, Fouque, Grégoire, Strub, Zucchini (CCS 2016): add stronger refresh gadgets (SNI)
- Example: AES S-box on GF(2^8)

Constraint:
\[ t_0 + t_1 + t_2 + t_3 \leq t \]
Why Does It Work?

- Barthe, B., Dupressoir, Fouque, Grégoire, Strub, Zucchini (CCS 2016): add stronger refresh gadgets (SNI)
- Example: AES S-box on GF($2^8$)

![Diagram](attachment:image.png)

Constraint:
\[ t_0 + t_1 + t_2 + t_3 \leq t \]
Why Does It Works?

- Barthe, B., Dupressoir, Fouque, Grégoire, Strub, Zucchini (CCS 2016): add **stronger** refresh gadgets (SNI)
- Example: AES S-box on $\text{GF}(2^8)$

Constraint:

$$t_0 + t_1 + t_2 + t_3 \leq t$$
Why Does It Works?

- Barthe, B., Dupressoir, Fouque, Grégoire, Strub, Zucchini (CCS 2016): add stronger refresh gadgets (SNI)
- Example: AES S-box on $\text{GF}(2^8)$

Constraint:
$$t_0 + t_1 + t_2 + t_3 \leq t$$

Diagram:
- $t_0 + t_3$
- $+ t_1 + t_2$
- $\{ x \}$
- $[\times]$
- $[\cdot^2]$
- $R$
- $t_3$ output observations
Tool maskComp

- from $t$-NI and $t$-SNI gadgets ⇒ build a $t$-NI circuit by inserting $t$-SNI refresh gadgets at carefully chosen locations
- formally proven

Implementation in C language with no countermeasure $\rightarrow$ maskComp $\rightarrow$ $t$-NI secure implementation in C language

Demo AES S-box without refresh

https://sites.google.com/site/maskingcompiler/home

```c
bint8_t x3(bint8_t x) {
    bint8_t r, z;
    z = gf256_pow2(x);
    r = gf256_mul(x, z);
    return r;
}
```

Start type checking of x3
insert refresh 1 1
x3 : {S_34 } ->
    0_21
side
constraints LE:S_34 <= I_35
NEEDED:[{0_21}]
1 refresh inserted in x3.
1 refresh inserted.

> ./maskcomp.native -o myoutput_masked.c x3.c
Demo AES S-box with refresh

https://sites.google.com/site/maskingcompiler/home

```c
bint8_t x3(bint8_t x) {
    bint8_t r, w, z;
    z = gf256_pow2(x);
    w = bint8_refresh(x);
    r = gf256_mul(w, z);
    return r;
}
```

Start type checking of x3
x3 : {S_29 } ->
  0_21
  side
  constraints LE:S_29 <= I_30
  NEEDED:[ {0_21 }]

> ./maskcomp.native  -o myoutput_masked.c x3.c
Demo full AES

https://sites.google.com/site/maskingcompiler/home

> ./maskcomp.native -o myoutput_masked.c aes.c
Limitations of maskComp

- maskComp adds a refresh gadget to Circuit 1
- but Circuit 1 was already $t$-probing secure

Figure: Circuit 1.

Figure: Circuit 1 after maskComp.
Tool tightPROVE

- Joint work with Dahmum Goudarzi and Matthieu Rivain to appear in Asiacrypt 2018
- Apply to tight shared circuits:
  - sharewise additions,
  - ISW-multiplications,
  - ISW-refresh gadgets
- Determine exactly whether a tight shared circuit is probing secure for any order $t$
  1. Reduction to a simplified problem
  2. Resolution of the simplified problem
  3. Extension to larger circuits
Demo tightPROVE 1

\[ x_1 \]

\[ x_2 \]

\[ x \]

\[ \oplus \]

\[ \otimes \]

\[
\begin{align*}
\text{in 0} \\
\text{in 1} \\
\text{xor 0 1} \\
\text{and 0 2} \\
\text{out 3}
\end{align*}
\]

\[
\begin{align*}
\text{list_comb} &= [1, 3] \\
\text{comb} &= 1 \\
&\Rightarrow \text{NO ATTACK (G2 = G1)} \\
\text{G} &: [[(1,3)], []] \\
\text{O} &: [[3], []] \\
\end{align*}
\]

\[
\begin{align*}
\text{comb} &= 3 \\
&\Rightarrow \text{NO ATTACK (G2 = G1)} \\
\text{G} &: [[(1,3)], []] \\
\text{O} &: [[1], []] \\
\end{align*}
\]

No attack found

\[
\begin{align*}
> \text{sage verif.sage example1.circuit}
\end{align*}
\]
Demo tightPROVE 2

\[
\begin{align*}
[x_1] & \quad [x_2] & \quad [x_3] \\
(1) & \quad (2) & \quad (4) \\
\oplus & \quad \oplus & \quad \oplus \\
\otimes & \quad \otimes & \quad \otimes \\
\text{in 0} & \quad \text{in 1} & \quad \text{in 2} \\
\text{xor 0 1} & \quad \text{xor 1 2} & \quad \text{and 0 1} \\
\text{and 3 4} & \quad \text{and 2 3} & \quad \text{out 5} \\
\text{out 6} & \quad \text{out 6} & \quad \text{out 7} \\
\rightarrow & \quad & \\
\text{list_comb = [1, 3, 2, 4, 6]} & \quad & \\
\text{comb = 1} & \quad & \Rightarrow \text{NO ATTACK (G3 = G2)} \\
G: [\{(1,2)\}, \{(3,6),(3,4)\}] & \quad & O: [\{2\}, \{6, 4\}, \{\}\} \\
\Rightarrow \text{comb = 3} & \quad & \Rightarrow \text{NO ATTACK (G3 = G2)} \\
G: [\{(3,6),(3,4)\}, \{(1,2)\}] & \quad & O: [\{6, 4\}, \{2\}, \{\}\} \\
\Rightarrow \text{comb = 2} & \quad & \Rightarrow \text{ATTACK} \\
G: [\{(1,2)\}, \{(3,6),(3,4)\}] & \quad & O: [\{1\}, \{6, 4\}] \\
\Rightarrow \text{Attack found: 2 in span [1, 6, 4]}
\end{align*}
\]

\$> \text{sage verif.sage example2.circuit}\$
Demo tightPROVE 2

\[
\begin{align*}
\begin{array}{c}
[x_1] \\
(1) \\

[x_2] \\
(2) \\

[x_3] \\
(4) \\

[\oplus] \\
(3) \\

[\oplus] \\
(6) \\

[\otimes] \\

\end{array}
\end{align*}
\]

\[
\begin{align*}
in 0 \\
in 1 \\
in 2 \\
xor 0 1 \\
xor 1 2 \\
and 0 1 \\
and 3 4 \\
and 2 3 \\
out 5 \\
out 6 \\
out 7 \\
\end{align*}
\]

\[
\begin{align*}
\text{list\_comb} &= [1, 3, 2, 4, 6] \\
\text{comb} &= 1 \\
\Rightarrow &\text{ NO ATTACK (G3 = G2)} \\
G: &\left[\left[(1,2)\right], \left[(3,6),(3,4)\right], \left[\right]\right] \\
O: &\left[\left[2\right], \left[6, 4\right], \left[\right]\right]
\end{align*}
\]

\[---\]

\[
\begin{align*}
\text{comb} &= 3 \\
\Rightarrow &\text{ NO ATTACK (G3 = G2)} \\
G: &\left[\left[(3,6),(3,4)\right], \left[(1,2)\right], \left[\right]\right] \\
O: &\left[\left[6, 4\right], \left[2\right], \left[\right]\right]
\end{align*}
\]

\[---\]

\[
\begin{align*}
\text{comb} &= 2 \\
\Rightarrow &\text{ ATTACK} \\
G: &\left[\left[(1,2)\right], \left[(3,6),(3,4)\right]\right] \\
O: &\left[\left[1\right], \left[6, 4\right]\right]
\end{align*}
\]

\[---\]

\[
\text{Attack found: 2 in span [1, 6, 4]}
\]

\[
\text{> sage verif.sage example2.circuit}
\]
Conclusion

In a nutshell...

- Formal tools to verify security of masked implementations
- Trade-off between security and performances

To continue...

- Achieve better performances
- Apply such formal verifications to every circuit