

# Side-Channel Attacks : Sécurité et Attaques par Canaux Auxiliaires

## Attaques différentielles, Attaques template, Masquage et vérification

---

Quentin Meunier

2024

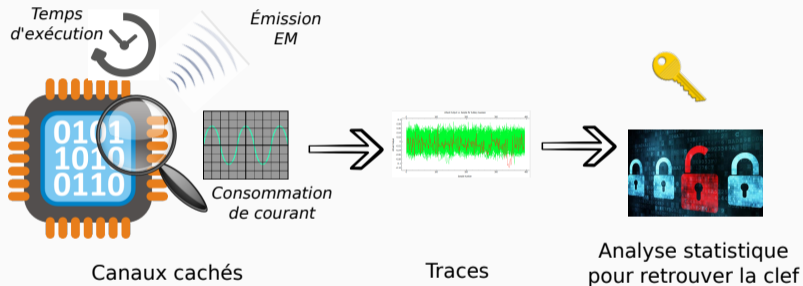
Sorbonne Université

Laboratoire d'Informatique de Paris 6

4 Place Jussieu, 75252 Paris, France



# Introduction : Side-Channel Attacks





PAYS-BAS

Actualisé 6 février 2015, 11:42

## Pas de neige sur le toit? Cultivez-vous de l'herbe?

**La police néerlandaise incite les habitants de Haarlem à observer de près les toits de leur ville. S'ils ne sont pas couverts de neige, cela pourrait indiquer la présence d'une plantation de cannabis.**



par  
ofu



0

0



Politie Basisteam Haarlem  
@POL\_Haarlem · [Follow](#)



Geen sneeuw op het dak van de bureu? Melden van vermoeden [#hennekwekerij](#) kan ook anoniem via [@M08007000](#).



3 occitanie

changer de localité



accueil



émissions



menu

Accueil > Occitanie > Pyrénées-Orientales > Perpignan

**DROGUE. Des milliers de  
plants de cannabis saisis  
dans des maisons de luxe,  
les trafiquants trahis par  
leur trop forte  
consommation d'électricité**



Ils opéraient dans des maisons de luxe de la région du Maresme, située dans la province de Barcelone. • © Mossos d'Esquadra

## Attaques

Rappel : Consommation des instructions et des données

Principe de la DPA : Differential Power Analysis

Principe de la CPA : Correlation Power Analysis

Attaque par template

## Masquage

## Attaques

Rappel : Consommation des instructions et des données

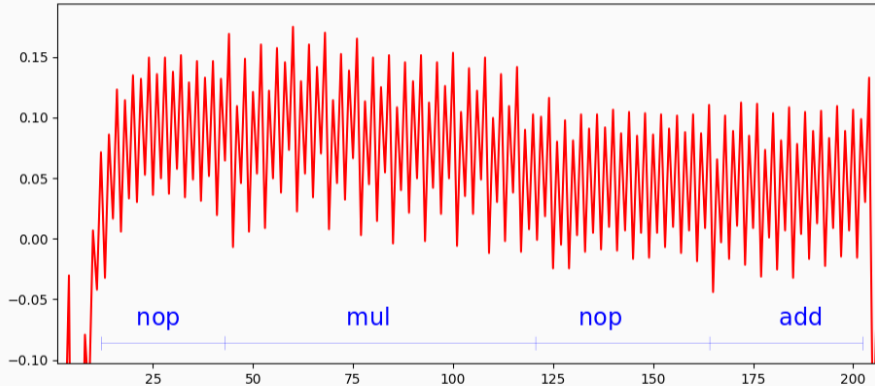
Principe de la DPA : Differential Power Analysis

Principe de la CPA : Correlation Power Analysis

Attaque par template

## Masquage

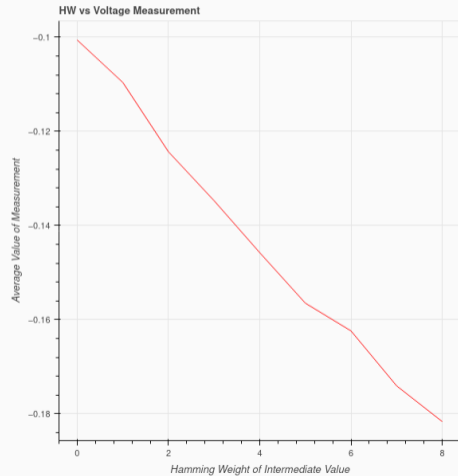
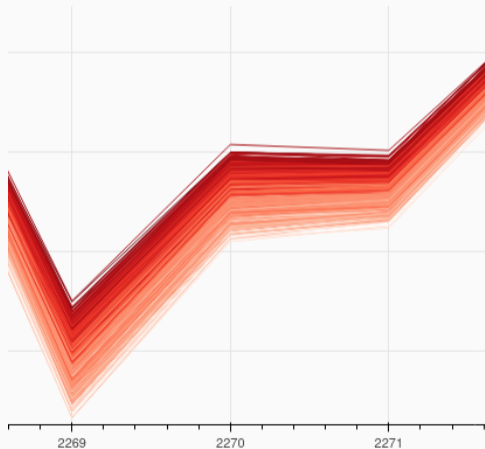
# Consommation électrique d'instructions assembleur



- Des instructions différentes consomment différemment

## Consommation des données

- **Modèle simple** : considérer le poids de Hamming des données
- Ce modèle simple marche !





## Attaques

Rappel : Consommation des instructions et des données

Principe de la DPA : Differential Power Analysis

Principe de la CPA : Correlation Power Analysis

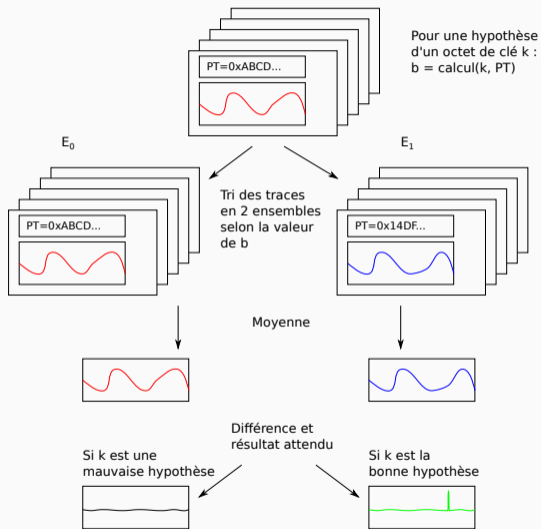
Attaque par template

## Masquage

- **Objectif** : retrouver la clé secrète d'un algorithme cryptographique à partir de la consommation d'encryptions matérielles ou logicielles
- **Hypothèse de base** : la valeur de consommation d'un calcul à un instant donné dépend de la valeur du résultat du calcul
- **Conséquence** : la valeur d'un bit intermédiaire du calcul  $b$  a une incidence sur la consommation à cet instant
- **Hypothèse supplémentaire** : La consommation d'un calcul fixé suit une loi normale à un instant donné (c'est le cas en pratique)
- Si on choisit le bit  $b$  suffisamment tôt dans le calcul, cela découple les hypothèses requises sur les différents octets de la clé  $\Rightarrow$  On s'intéresse à la première ronde (ou à la dernière en attaquant la dernière clé de ronde)
  - Dans la suite on s'intéresse à l'attaque d'un octet de la clé

- On part d'un ensemble de traces  $T$  de consommation (pour lesquelles on connaît le *plain text* PT), et pour chaque valeur possible pour l'octet de clé ( $\in [0; 255]$ ), on divise les traces en 2 ensembles  $T_0$  et  $T_1$  selon la valeur de  $b$  sous cette hypothèse de clé
- Si l'**hypothèse de clé est mauvaise** :
  - On a réparti les traces selon un critère arbitraire, qui s'apparente à un critère aléatoire du point de vue de la consommation
  - $T_0$  et  $T_1$  doivent avoir les mêmes moyennes de consommation au point d'intérêt qui correspond à l'instant où  $b$  est calculé
- Si l'**hypothèse de clé est la bonne** :
  - On a trié les traces selon un critère qui a une influence sur la consommation, cela doit donc être visible au point d'intérêt
  - $T_0$  et  $T_1$  doivent donc avoir des moyennes de consommation différentes au point d'intérêt (sinon cela veut dire que l'hypothèse de base est fausse)
- On peut donc déterminer si l'hypothèse de clé est la bonne en regardant l'écart entre les moyennes des traces  $T_0$  et  $T_1$

# Principe de la DPA : exemple sur l'AES



# Principe de la DPA : exemple sur l'AES

- Comment choisir le bit  $b$ ?
  - Le premier calcul effectué est un AddRoundKey, qui met en jeu uniquement un octet de clé avec le plain text connu
  - Donc par exemple, regarder un bit (n'importe lequel) de  $x[0][0]$  après le premier ARK pour attaquer l'octet de clé 0 : on peut penser à faire ce choix
- **Problème** : Une hypothèse de clé différente d'un bit (au sens de hamming) de la bonne clé donnera également un bon tri des traces, l'opposé de la clé également (même tri que la bonne hypothèse)
  - $(k \oplus \delta) \oplus PT = (k \oplus PT) \oplus \delta$  proche de  $k \oplus PT$  pour  $\delta$  petit
- Et comme on ne regarde qu'un bit, c'est encore pire : 1 seul découpage possible !
  - Si on regarde le bit 0, toutes les clés paires donneront le même découpage (bon ou mauvais), et toutes les clés impaires, le découpage "inverse"
- Pour ne pas avoir ces problèmes, on regarde un bit après la première SBox : tous les bits de l'hypothèse de clé entrent en compte dans le calcul et deux clés proches n'ont pas de raisons de donner des résultats proches (moyennage sur les valeurs de PT)
  - $Sbox[(k \oplus \delta) \oplus PT]$  ne "matche"  $Sbox[k \oplus PT]$  pour tous les PT que si  $\delta = 0$
- **Variante** : Au lieu de ne regarder qu'un seul bit, on regarde le poids de Hamming de la valeur intermédiaire, et on trie les traces dans T0 ou T1 selon que le HW soit inférieur ou supérieur à 4

## Exemple DPA sur l'AES en multi-bit

- On mesure les 8 valeurs de consommation suivantes en sortie de la SBox : 5.3, 5.9, 2.0, 2.2, 5.4, 0.8, 4.1, 5.2
- Pour les 8 traces, le plaintexts associés sont : 0x59, 0xF1, 0x75, 0xB7, 0x64, 0x15, 0x85, 0xC2
- L'octet de clé correct est **0x41**

|      | Octet de clé = 0x41   |    |          | Octet de clé = 0x00   |    |          |
|------|---|----|----------|---|----|----------|
| PT   | SBox[k ⊕ pt]  | HW | T0 ou T1 | SBox[k ⊕ pt]  | HW | T0 ou T1 |
| 0x59 | 0xAD  | 5  | T1       | 0xCB  | 5  | T1       |
| 0xF1 | 0xE7  | 6  | T1       | 0xA1  | 3  | T0       |
| 0x75 | 0x18  | 2  | T0       | 0x9D  | 5  | T1       |
| 0xB7 | 0x0D  | 3  | T0       | 0xA9  | 4  | -        |
| 0x64 | 0x3F  | 6  | T1       | 0x43  | 3  | T0       |
| 0x15 | 0x20  | 1  | T0       | 0x59  | 4  | -        |
| 0x85 | 0x1C  | 3  | T0       | 0x97  | 5  | T1       |
| 0xC2 | 0xEC  | 5  | T1       | 0x25  | 3  | T0       |
|      | $\overline{T0} = (2.0 + 2.2 + 0.8 + 4.1)/4 = 2.3$<br>$\overline{T1} = (5.3 + 5.9 + 5.4 + 5.2)/4 = 5.4$<br>$ \overline{T0} - \overline{T1}  = 3.1$ |    |          | $\overline{T0} = (5.9 + 5.4 + 5.2)/3 = 5.5$<br>$\overline{T1} = (5.3 + 2.0 + 4.1)/3 = 3.8$<br>$ \overline{T0} - \overline{T1}  = 1.7$ |    |          |

## Attaques

Rappel : Consommation des instructions et des données

Principe de la DPA : Differential Power Analysis

Principe de la CPA : Correlation Power Analysis

Attaque par template

## Masquage

- **Objectif** : retrouver la clé secrète d'un algorithme cryptographique à partir de la consommation d'encryptions matérielles ou logicielles
- **Hypothèse de base** : la valeur de consommation d'un calcul un instant donné dépend de la valeur du résultat du calcul
- Attaque basée sur un modèle de consommation d'énergie
- De même que pour la DPA, l'attaque cible un point particulier dans le temps au cours de l'exécution
  - La valeur intermédiaire doit dépendre de la valeur secrète et changer à chaque trace
  - $\Rightarrow$  elle doit dépendre du *plaintext*
- Pour l'AES, comme pour la DPA, on cible en général un point après la première SBox ou avant la dernière
  - Pas encore de mélange des octets de clé entre eux, permet de tester moins d'hypothèses
- Modèle le plus utilisé pour la consommation d'énergie : **poinds de Hamming** (HW) d'une valeur particulière ou **distance de Hamming** (HD) entre 2 valeurs intéressantes (par exemple stockées consécutivement dans le même registre)
- Dans la suite, on ne considère qu'un seul octet de clé



- Le modèle de consommation est corrélé avec la consommation énergétique réelle en utilisant le coefficient de corrélation empirique (dit coefficient de Pearson).
- Ce coefficient de corrélation empirique entre 2 échantillons  $x_i$  et  $y_i$  ( $1 \leq i \leq n$ ) est donné par :

$$\hat{r} = \frac{\hat{\sigma}_{X,Y}}{\hat{\sigma}_X \hat{\sigma}_Y} \quad (1)$$

avec

$$\hat{\sigma}_{X,Y} = \frac{1}{N} \sum_{i=1}^N (x_i - \bar{x})(y_i - \bar{y}) \quad (2)$$

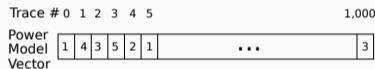
$$\hat{\sigma}_X = \sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \bar{x})^2} = \sqrt{\hat{\sigma}_{X,X}}, \text{ et } \hat{\sigma}_Y = \sqrt{\frac{1}{N} \sum_{i=1}^N (y_i - \bar{y})^2} \quad (3)$$

où  $\bar{x}$  et  $\bar{y}$  sont les moyennes empiriques de  $x_i$  et  $y_i$ .

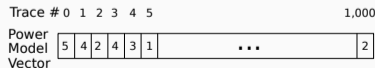
- On ne considère qu'un seul instant tempore : le moment auquel la valeur interne est calculée



HW of the internal value for each trace for a correct Key Hypothesis



HW of the (incorrect) internal value for each trace for a wrong Key Hypothesis



- Complexité totale pour tous les instants :  $\# \text{ traces} \times \# \text{ samples} \times \# \text{ key hypotheses}$

- **Cas 1** : on connaît le POI pour faire l'attaque
- Étant données  $N$  traces de consommation de  $L$  échantillons, on note  $t_n$  la valeur de consommation de la trace  $n$  au POI (avec  $1 \leq n \leq N$ ).
- Pour  $K$  clés (sous-clés) possibles (on a typiquement  $K = 256$ ), on note  $h_{n,k}$  la valeur estimée de la consommation, selon le modèle choisi, pour la trace  $n$  et l'hypothèse de clé  $k$  ( $0 \leq k < K$ )
  - **Remarque** : cette valeur dépend du *plaintext*
- On peut ensuite voir à quel point le modèle et les mesures correspondent pour chaque hypothèse de clé  $k$  en calculant :

$$r_k = \frac{\sum_{n=1}^N (h_{n,k} - \bar{h}_k)(t_n - \bar{t})}{\sqrt{\sum_{n=1}^N (h_{n,k} - \bar{h}_k)^2 \sum_{n=1}^N (t_n - \bar{t})^2}} \quad (4)$$

où  $\bar{h}_k$  et  $\bar{t}$  sont respectivement les valeurs moyennes de la consommation modélisée et de la consommation mesurée.

- Puis, en prenant le maximum des  $r_k$  pour toutes les valeurs de  $k$ , on peut en déduire quelle hypothèse de clé est la plus probable

- **Cas 2** : on ne connaît pas le POI pour faire l'attaque
- Étant données  $N$  traces de consommation de  $L$  échantillons, on note  $t_{n,i}$  la valeur de consommation au point  $i$  de la trace  $n$  (avec  $1 \leq n \leq N, 1 \leq i \leq L$ ).
- Pour  $K$  clés (sous-clés) possibles (on a typiquement  $K = 256$ ), on note  $h_{n,k}$  la valeur estimée de la consommation, selon le modèle choisi, pour la trace  $n$  et l'hypothèse de clé  $k$  ( $0 \leq k < K$ )
  - Remarque : cette valeur dépend du *plaintext*, mais pas du point dans le temps
- On peut ensuite voir à quel point le modèle et les mesures correspondent pour chaque hypothèse de clé  $k$  et à chaque instant  $i$  en calculant :

$$r_{k,i} = \frac{\sum_{n=1}^N (h_{n,k} - \bar{h}_k)(t_{n,i} - \bar{t}_i)}{\sqrt{\sum_{n=1}^N (h_{n,k} - \bar{h}_k)^2 \sum_{n=1}^N (t_{n,i} - \bar{t}_i)^2}} \quad (5)$$

où  $\bar{h}_k$  et  $\bar{t}_i$  sont respectivement les valeurs moyennes de la consommation modélisée et de la consommation mesurée à l'instant  $i$ .

- Puis, en prenant le maximum des  $r_{k,i}$  pour toutes les valeurs de  $i$  et  $k$ , on peut en déduire quelle hypothèse de clé est la plus probable

- **Remarque** : on peut montrer que  $r_{i,k}$  peut être calculé avec la formule suivante, qui présente l'avantage de permettre un calcul "*online*" (en un seul parcours de l'ensemble des données) :

$$r_{k,i} = \frac{N \sum_{n=1}^N h_{n,k} t_{n,i} - \left( \sum_{n=1}^N h_{n,k} \right) \left( \sum_{n=1}^N t_{n,i} \right)}{\sqrt{\left( \left( \sum_{n=1}^N h_{n,k} \right)^2 - N \sum_{n=1}^N h_{n,k}^2 \right) \left( \left( \sum_{n=1}^N t_{n,i} \right)^2 - N \sum_{n=1}^N t_{n,i}^2 \right)}} \quad (6)$$

## Avantages de n'avoir qu'une passe sur les données

- Quand le nombre de traces est très élevé, permet de calculer les valeurs de CPA à la volée, sans avoir à stocker les traces
- Permet de regarder régulièrement les valeurs de corrélations obtenues pour les meilleures hypothèses (par exemple toutes les 1000 traces) et de s'arrêter dès que l'écart est jugé significatif : évite de regarder toutes les traces

## Attaques

Rappel : Consommation des instructions et des données

Principe de la DPA : Differential Power Analysis

Principe de la CPA : Correlation Power Analysis

**Attaque par template**

Masquage

- **Objectif** : retrouver la clé secrète d'un algorithme cryptographique à partir de la consommation d'encryptions matérielles ou logicielles
- **Hypothèse de base** : la valeur de consommation d'un calcul un instant donné dépend de la valeur du résultat du calcul
- Attaque basée sur un **modèle de consommation d'énergie**
- De même que pour la DPA et la CPA, l'attaque cible un point particulier dans le temps au cours de l'exécution
- Contrairement à la DPA et à la CPA, cette attaque nécessite de pouvoir **faire changer la clé entre deux exécutions**

- Le principe de cette attaque est de construire un “profil” de consommation du dispositif en fonction de la valeur de la clé
- Une fois le profil construit, il est possible de retrouver la valeur de la clé à partir de mesures, en regardant à quel profil correspondent le mieux les valeurs observées
- ⇒ Attaque en deux phases :
  - **Phase d'acquisition** : construction des profils, nécessite beaucoup de traces avec la clé qui varie et qui est connue
  - **Phase d'exploitation** : la clé est fixée et inconnue, le but de l'attaque est de la retrouver ; cette phase requiert peu de traces



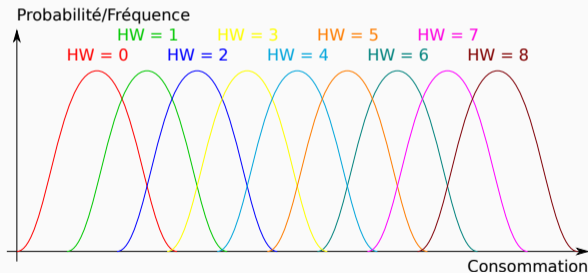
## Principe d'une attaque par template : première phase

- Enregistrer **beaucoup de traces** (de l'ordre de 100 000) avec clé et plain text connus et variables
- Choisir quelques **instants d'intérêts** (typiquement 5), qui correspondent aux instants qui maximisent la variance inter-traces
- **Classifier les traces** selon un critère, qui peut être par exemple (pour l'attaque d'un octet de clé) :
  - La valeur en sortie de la première SBox
  - Le poids de Hamming de la valeur en sortie de la première SBox
- Pour chaque valeur du critère, reconstruire la loi normale multidimensionnelle dont les paramètres sont ceux obtenus de manière empirique
  - Le nombre  $N$  de dimensions est le nombre d'instants d'intérêt
  - Les paramètres de la loi sont le vecteur des moyennes  $\mu$  de taille  $N$  et la matrice de covariance  $\Sigma$  entre les dimensions (de taille  $N \times N$ )
- **Remarque** : la densité de la loi normale multidimensionnelle pour un vecteur  $x$  de taille  $N$  est

$$f_{\mu, \Sigma}(x) = \frac{1}{(2\pi)^{N/2} |\Sigma|^{1/2}} e^{-\frac{1}{2}(x-\mu)^T \Sigma^{-1} (x-\mu)}$$

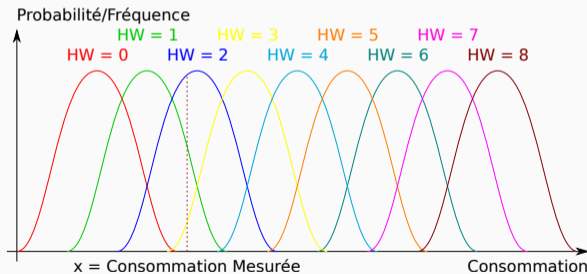
## Principe d'une attaque par template : exemple première phase

- Exemple pour une attaque considérant un seul instant d'intérêt  $t_0$  (considère la consommation à un seul sample) : loi normale de dimension 1
- On trie les traces en fonction de la valeur du poids de Hamming en sortie de la 1ère Sbox
- $\Rightarrow$  On obtient 9 distributions empiriques
- Pour chaque distribution empirique, on reconstruit une distribution de probabilité en estimant les paramètres  $\mu$  et  $\sigma$  de la loi normale



## Principe d'une attaque par template : exemple première phase

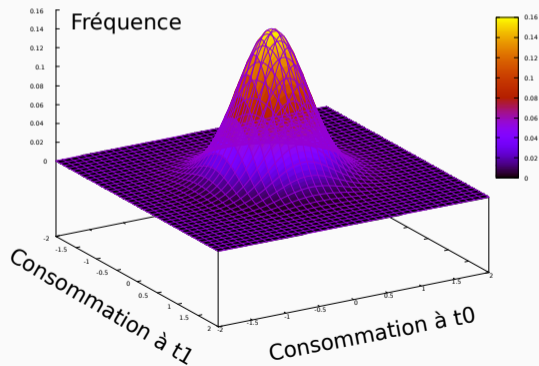
- Exemple pour une attaque considérant un seul instant d'intérêt  $t_0$  (considère la consommation à un seul sample) : loi normale de dimension 1
- On trie les traces en fonction de la valeur du poids de Hamming en sortie de la 1ère Sbox
- $\Rightarrow$  On obtient 9 distributions empiriques
- Pour chaque distribution empirique, on reconstruit une distribution de probabilité en estimant les paramètres  $\mu$  et  $\sigma$  de la loi normale



- Pour une mesure de consommation  $x$ , chaque distribution donne la "probabilité" d'obtenir la valeur  $x$  pour un poids de Hamming donné

## Principe d'une attaque par template : exemple première phase

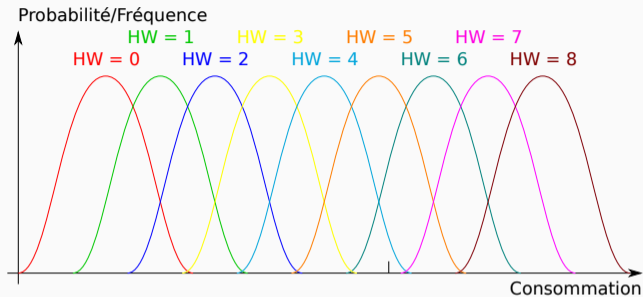
- Exemple d'une distribution reconstruite pour 2 instants d'intérêt  $t_0$  et  $t_1$  pour un HW donné
- Pour le poids de Hamming correspondant, associe une probabilité au fait d'observer la consommation  $c_0$  à  $t_0$  et  $c_1$  à  $t_1$
- Même principe quand on a un nombre plus élevé de points d'intérêt (et de dimensions dans la loi normale)



## Principe d'une attaque par template : deuxième phase

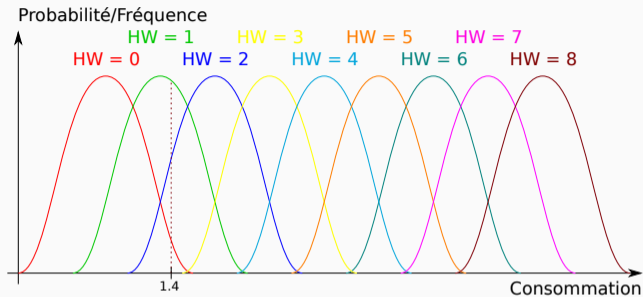
- La **deuxième phase** correspond à l'attaque d'un système similaire à celui profilé mais dont on ne connaît pas la clé (constante)
- Enregistrement des traces avec **plaintext variable et connu** (de l'ordre d'une dizaine à une cinquantaine)
- Pour chaque trace :
  - Création du vecteur de consommation  $c$  correspondant aux points d'intérêt
  - Puis, pour chaque hypothèse de clé :
    - Calcul de la valeur  $v$  du critère (ex : poids de Hamming de la sortie de la SBox), elle-même fonction du plaintext et de l'hypothèse de clé
    - Calcul du score  $s = p_v(c)$  de la distribution associée à ce critère pour ce vecteur de consommation : plus la valeur est grande, plus le modèle est satisfaisant
- Pour chaque hypothèse de clé, on multiplie tous les scores obtenus : le score total le plus grand est normalement obtenu pour la bonne clé
  - Permet d'éliminer les mauvaises hypothèses de clé, qui auront un score très faible pour au moins une sous-partie des *plaintexts* car le HW ne sera toujours correct que pour la bonne hypothèse de clé
  - **Remarque** : En réalité, pour éviter les problèmes de stabilité numérique, on somme des log au lieu de faire des multiplications (ne change pas le max)

## Principe d'une attaque par template : exemple deuxième phase



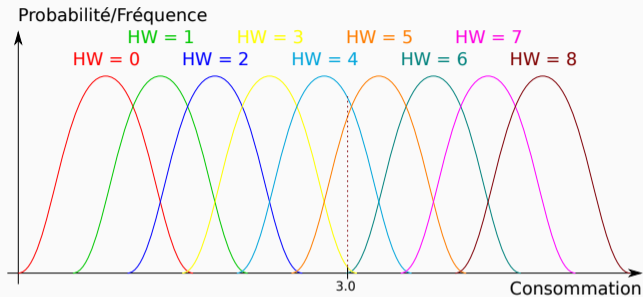
|              |  | Hypothèse de clé   HW / Proba |      |      |      |      |     |
|--------------|--|-------------------------------|------|------|------|------|-----|
| Consommation |  | 0x00                          | 0x01 | 0x02 | 0x03 | 0x04 | ... |
| Trace 0      |  |                               |      |      |      |      |     |
| Trace 1      |  |                               |      |      |      |      |     |
| Trace 2      |  |                               |      |      |      |      |     |
| Trace 3      |  |                               |      |      |      |      |     |

## Principe d'une attaque par template : exemple deuxième phase



|              |     | Hypothèse de clé   HW / Proba |         |         |         |                |     |
|--------------|-----|-------------------------------|---------|---------|---------|----------------|-----|
| Consommation |     | 0x00                          | 0x01    | 0x02    | 0x03    | 0x04           | ... |
| Trace 0      | 1.4 | 6 / $\epsilon$                | 0 / 0.2 | 2 / 0.6 | 1 / 0.9 | 4 / $\epsilon$ |     |
| Trace 1      |     |                               |         |         |         |                |     |
| Trace 2      |     |                               |         |         |         |                |     |
| Trace 3      |     |                               |         |         |         |                |     |

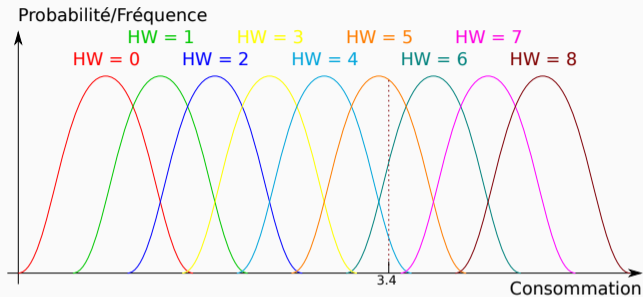
## Principe d'une attaque par template : exemple deuxième phase



|              |     | Hypothèse de clé   HW / Proba |         |                |         |                |     |
|--------------|-----|-------------------------------|---------|----------------|---------|----------------|-----|
| Consommation |     | 0x00                          | 0x01    | 0x02           | 0x03    | 0x04           | ... |
| Trace 0      | 1.4 | 6 / $\epsilon$                | 0 / 0.2 | 2 / 0.6        | 1 / 0.9 | 4 / $\epsilon$ |     |
| Trace 1      | 3.0 | 4 / 0.8                       | 5 / 0.7 | 7 / $\epsilon$ | 4 / 0.8 | 3 / 0.1        |     |
| Trace 2      |     |                               |         |                |         |                |     |
| Trace 3      |     |                               |         |                |         |                |     |

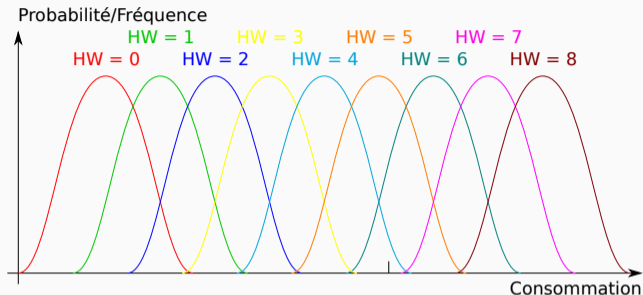


## Principe d'une attaque par template : exemple deuxième phase



|              |     | Hypothèse de clé   HW / Proba |                |                |         |                |     |
|--------------|-----|-------------------------------|----------------|----------------|---------|----------------|-----|
| Consommation |     | 0x00                          | 0x01           | 0x02           | 0x03    | 0x04           | ... |
| Trace 0      | 1.4 | 6 / $\epsilon$                | 0 / 0.2        | 2 / 0.6        | 1 / 0.9 | 4 / $\epsilon$ |     |
| Trace 1      | 3.0 | 4 / 0.8                       | 5 / 0.7        | 7 / $\epsilon$ | 4 / 0.8 | 3 / 0.1        |     |
| Trace 2      | 3.4 | 2 / $\epsilon$                | 7 / $\epsilon$ | 4 / 0.15       | 5 / 0.9 | 6 / 0.5        |     |
| Trace 3      |     |                               |                |                |         |                |     |

# Principe d'une attaque par template : exemple deuxième phase



|              |     | Hypothèse de clé   HW / Proba |                |                |         |                |     |
|--------------|-----|-------------------------------|----------------|----------------|---------|----------------|-----|
| Consommation |     | 0x00                          | 0x01           | 0x02           | 0x03    | 0x04           | ... |
| Trace 0      | 1.4 | 6 / $\epsilon$                | 0 / 0.2        | 2 / 0.6        | 1 / 0.9 | 4 / $\epsilon$ |     |
| Trace 1      | 3.0 | 4 / 0.8                       | 5 / 0.7        | 7 / $\epsilon$ | 4 / 0.8 | 3 / 0.1        |     |
| Trace 2      | 3.4 | 2 / $\epsilon$                | 7 / $\epsilon$ | 4 / 0.15       | 5 / 0.9 | 6 / 0.5        |     |
| Trace 3      | ... | ...                           | ...            | ...            | ...     | ...            |     |

Pour chaque hypothèse, on multiplie tous les scores ; clé = max

Attaques

Masquage

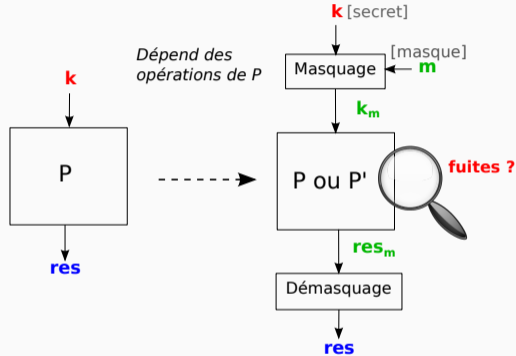
Masquage Matériel

Masquage logiciel

Exemple de l'AES, schéma de Herbst

Vérification du masquage

# Masquage : introduction



- Applicable au niveau **logiciel** ou au niveau **matériel** sur un circuit

# Principe du Masquage

- **Objectif** : découpler les valeurs manipulées par l'algorithme ou le programme des valeurs secrètes
- **Plus précisément** : faire en sorte que la distribution des valeurs des expressions intermédiaires soit indépendante des valeurs secrètes : **Threshold Probing Security** (TPS) à l'ordre 1
- Introduction de **variables de type masque** : variables dont la valeur change à chaque exécution (distribution aléatoire uniforme sur  $n$  bits)
- **Intérêt** : si  $m$  est un masque, alors l'expression  $e = m \oplus e'$  a une distribution uniforme si  $m$  n'apparaît pas dans  $e'$

| e | m | $e \oplus m$ |
|---|---|--------------|
| 0 | 0 | 0            |
|   | 1 | 1            |
| 1 | 0 | 1            |
|   | 1 | 0            |

- Representation avec des **secrets et masques** ou avec des **shares** selon la propriété de sécurité vérifiée
  - **Secrets et masques** : manipule des expressions de type  $m$  et  $k \oplus m$
  - **Shares** : manipule des shares, par exemple  $s_0$  et  $s_1$  tels que  $s_0 \oplus s_1 = s$

Attaques

Masquage

**Masquage Matériel**

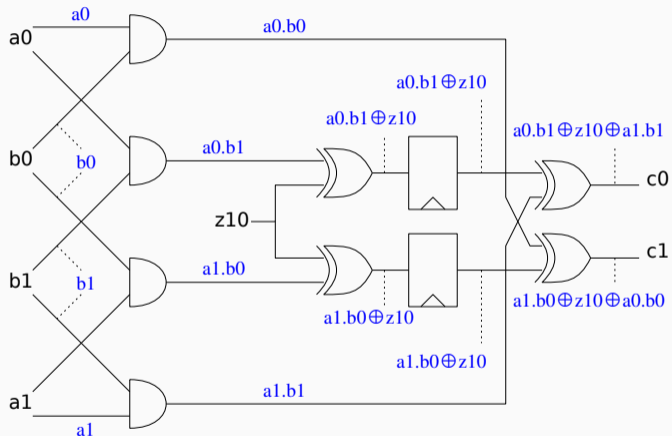
Masquage logiciel

Exemple de l'AES, schéma de Herbst

Vérification du masquage

- **Problème** : Soit 2 entrées secrètes sur  $n$  shares  $a$  et  $b$ ; comment calculer une sortie  $c$  sur  $n$  shares telle que  $c = a.b$  de manière sécurisée?
- Par exemple, pour  $n = 2$  :
  - On a  $a = a_0 \oplus a_1$  et  $b = b_0 \oplus b_1$
  - On veut calculer  $c_0$  et  $c_1$  tels que  $c_0 \oplus c_1 = (a_0 \oplus a_1).(b_0 \oplus b_1)$  sans jamais recalculer  $a$  et  $b$
- De nombreux schémas existants

# Exemple : Domain-Oriented Masking AND [Groß et al., 2017] à l'ordre 1

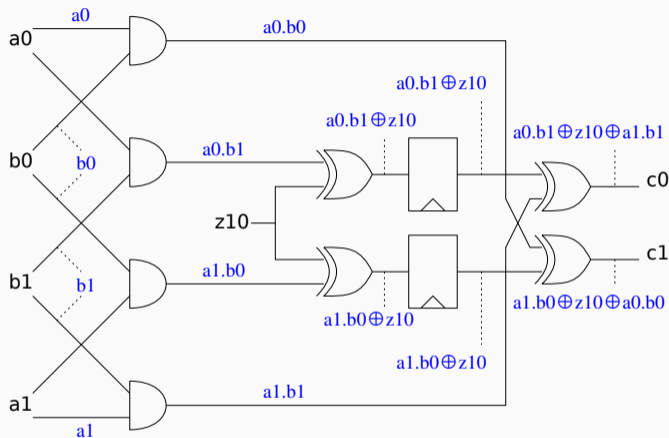




- **Attaque à l'ordre  $n$**  : une attaque à l'ordre  $n$  peut utiliser  $n$  valeurs intermédiaires (fils, ou **probes**)
  - **Idée** : les mesures des différentes probes peuvent être combinées
  - Intuitivement, si manipule  $m$  et  $k \oplus m$ , on peut retrouver  $k$  en 2 observations, tandis que si l'on manipule  $m_1$ ,  $m_2$  et  $k \oplus m_1 \oplus m_2$ , il faut 3 observations
  - En pratique, peut résulter d'une mesure de consommation globale
- **Sécurité à l'ordre  $n$**  : la sécurité à l'ordre  $n$  fournit une résistance face aux attaques à l'ordre  $n$ 
  - Implique que le nombre de shares  $t > n$
- La sécurité à l'ordre  $n$  requiert d'énumérer tous les tuples possibles de taille  $n$ , et de montrer que la propriété est vérifiée pour chacun
- Le circuit DOM-AND à l'ordre 1 n'offre pas de sécurité à l'ordre 2 : la distribution jointe des expressions  $(a_0, a_1)$  n'est pas indépendante de  $a$

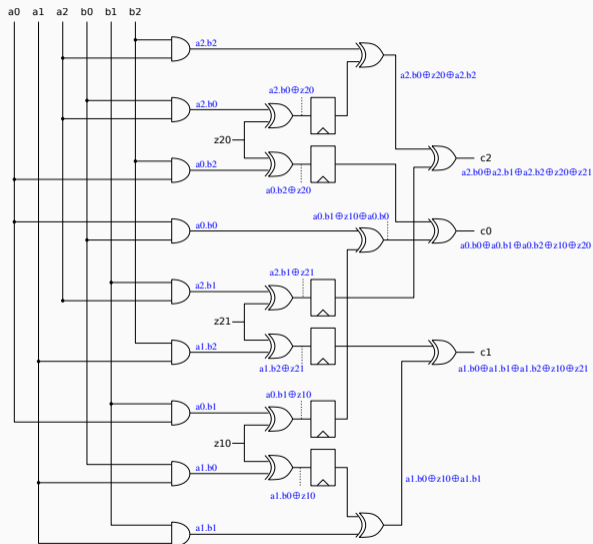
- **Non-Interférence** (NI) : la distribution de chaque tuple de  $t$  expressions dépend au plus de  $t$  shares par entrée
  - Implique TPS si les sharings sont uniformément distribués
- **Strong Non-Interférence** (SNI) : la distribution de chaque tuple de  $t$  expressions dépend au plus de  $t$  shares par entrée, moins le nombre de sorties du circuit parmi les expressions du tuple
  - Implique Non-Interférence
- **Probe Isolating Non-Interference** (PINI)
  - Implique NI et est impliqué par SNI

## Exemple : Domain-Oriented Masking AND [Groß et al., 2017] à l'ordre 1



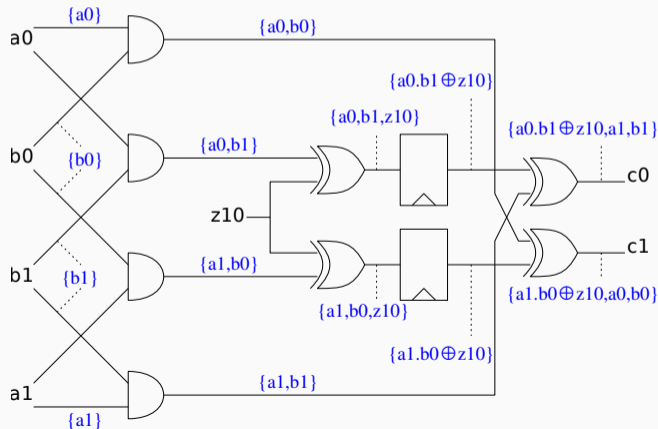
- 1-NI : chaque distribution d'expression dépend au plus d'un share par entrée
- 1-SNI : chaque distribution d'expression dépend au plus d'un share par entrée, excepté pour les sorties qui ne doivent dépendre d'aucun share

## Ordre Supérieur : DOM-AND à l'ordre 2



## Prise en Compte des Glitches

- Considère que les valeurs temporaires sur les fils sont suffisantes pour fuir de l'information, et sont incluses dans le modèle de fuite
- Modifie la fuite associée à chaque probe
- Orthogonal à la propriété de sécurité

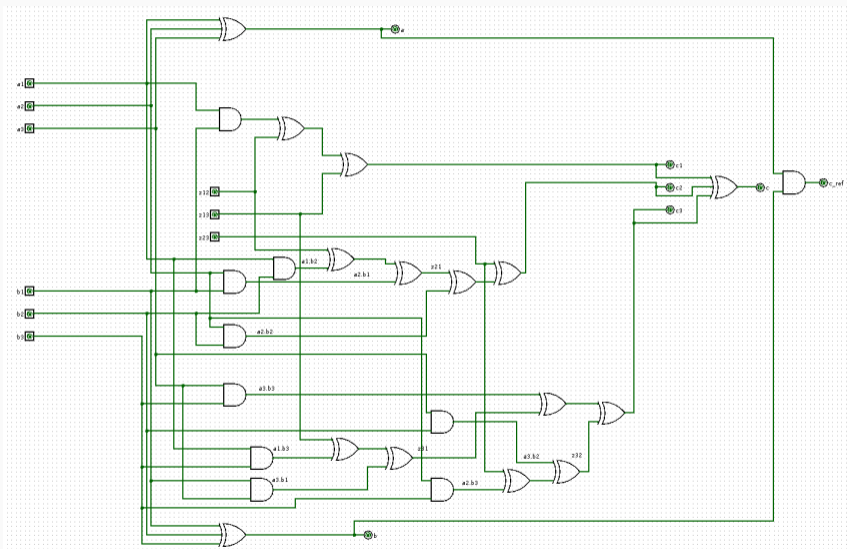


- **DOM AND** : Schéma dit **Domain Oriented Masking** [Groß et al., 2017], résistant aux glitches
- **ISW AND** : Premier schéma proposé en 2006 [Ishai et al., 2003]
- **ISW AND refresh** : Variante avec un **refresh** sur une des entrées [De Cnudde et al., 2016]
- **TI AND** : Schéma dit **Threshold Implementation** n'utilisant pas de random intermédiaires [Nikova et al., 2006]
- **NI Mult and SNI Mult** : Schémas vérifiant les propriétés NI et SNI [Bordes and Karpman, 2021]
- **PINI Mult** : Schéma implémentant la propriété PINI, introduit dans [Wang et al., 2023]
- **GMS AND** : Deux implémentations du AND utilisant le **Generalized Masking Scheme**, décrit dans l'article, utilisant respectivement 3 et 5 shares [Reparaz et al., 2015]
- Et bien d'autres...
- Les schémas diffèrent sur les propriétés de sécurité vérifiées, le nombre de portes, le nombre de registres, etc.

## Description du schéma ISW à l'ordre $t$ pour une porte AND

- Soient  $a = a_1 \oplus \dots \oplus a_{t+1}$  et  $b = b_1 \oplus \dots \oplus b_{t+1}$  deux entrées secrètes découpées en  $t + 1$  shares
- Pour  $1 \leq i < j \leq t + 1$  ( $i \neq j$ ), on définit  $z_{i,j}$  et  $z_{j,i}$  de la manière suivante :
  - $z_{i,j}$  est un bit random
  - $z_{j,i} = (z_{i,j} \oplus a_i b_j) \oplus a_j b_i$
- On calcule les bits  $c_i$  de la sortie de la manière suivante :
$$c_i = a_i b_i \oplus \bigoplus_{j \neq i} z_{i,j}$$
- Le circuit résultant calcule  $a.b$  et est TPS à l'ordre  $t$
- Une porte AND est ainsi convertie en  $O(t^2)$  portes
- **Remarque** : ce schéma ne prend pas en compte les *glitches*

## Exemple ISW : porte AND pour $t = 2$





- **Cadre** : fonctions combinatoires, plusieurs entrées et plusieurs sorties constituées de  $n$  shares
  - Comme précédemment, le codage pour une valeur donnée d'une entrée doit être uniforme parmi les codages possibles
- **Contrainte** : n'utilise pas de valeur aléatoire au milieu du calcul
- **Idée** : si chaque sortie ne dépend pas d'un indice (identique) pour toutes les entrées, alors chaque sortie est indépendante de chacune des valeurs secrète
  - Exemple :  $z_1$  ne dépend pas de  $x_1, y_1$  ;  $z_2$  ne dépend pas de  $x_2, y_2$ , etc.
- Chaque caractéristique de chaque fonction  $z_i$  (consommation, rayonnement EM, etc.) est indépendante de  $x, y$  et  $z$
- Marche aussi pour les valeurs intermédiaires

- $x = x_1 \oplus x_2 \oplus x_3, y = y_1 \oplus y_2 \oplus y_3$
- Les 3 fonctions de sorties :
  - $z_1 = x_2y_2 \oplus x_2y_3 \oplus x_3y_2$
  - $z_2 = x_3y_3 \oplus x_1y_3 \oplus x_3y_1$
  - $z_3 = x_1y_1 \oplus x_1y_2 \oplus x_2y_1$
- Sont une implémentation TI de  $x.y$
- **Problème** de cette réalisation : si la sortie  $z$  est utilisée en entrée d'un autre circuit TI, il faut que le codage de chaque valeur soit uniforme parmi les codages possibles (hypothèse requise pour garantir la sécurité) : ce n'est pas le cas ici (propriété dite d'équilibre ou de *balance*)

## Masquage T1 : exemple du AND

- On s'intéresse au cas  $x = 1$  et  $y = 1$  (seule possibilité pour que la sortie vaille 1)
- Sous l'hypothèse d'uniformité de codage et d'indépendance des entrées, chacune des lignes a la même probabilité
- On observe que le codage en sortie n'est **pas uniforme**

| x3 | x2 | x1 | y3 | y2 | y1 | z3 | z2 | z1 |
|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 0  |
| 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  |
| 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 0  |
| 0  | 0  | 1  | 1  | 1  | 1  | 0  | 1  | 0  |
| 0  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  |
| 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  |
| 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  |
| 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 0  |
| 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  |
| 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  |
| 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  |
| 1  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 1  |
| 1  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  |
| 1  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  |
| 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  |
| 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  |

- Une implémentation équilibrée n'est pas possible avec seulement 3 shares

- Implémentation équilibrée à 4 shares :
  - $z_1 = (x_3 \oplus x_4)(y_2 \oplus y_3) \oplus y_2 \oplus y_3 \oplus y_4 \oplus x_2 \oplus x_3 \oplus x_4$
  - $z_2 = (x_1 \oplus x_3)(y_1 \oplus y_4) \oplus y_1 \oplus y_3 \oplus y_4 \oplus x_1 \oplus x_3 \oplus x_4$
  - $z_3 = (x_2 \oplus x_4)(y_1 \oplus y_4) \oplus y_2 \oplus x_2$
  - $z_4 = (x_1 \oplus x_2)(y_2 \oplus y_3) \oplus y_1 \oplus x_1$
- Beaucoup de portes pour une protection à l'ordre 1

Attaques

Masquage

Masquage Matériel

Masquage logiciel

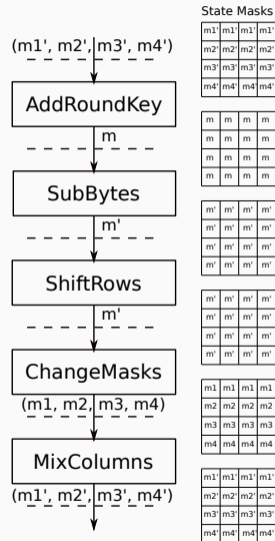
Exemple de l'AES, schéma de Herbst

Vérification du masquage

- **But** : rendre aléatoires les valeurs intermédiaires du calcul, les décorrélérer des valeurs des secrets
- Très dépendant de l'algorithme
- Peut être ajouté dans le source code
- Même si en pratique, les programmes masqués sont souvent écrits directement en assembleur par des experts
- **Problème résiduel** : il y a toujours une sensibilité avant/pendant le masquage
- 2 grands **types de masquage** : booléen ( $\text{xor } \oplus$ ) et arithmétique ( $+ \text{ mod } 2^n$ )

# Exemple de masquage de l'AES (Schéma de Herbst [Herbst et al., 2006])

- Pour chaque exécution, utilisation de **6 masques** sur 8 bits :  $m$ ,  $m'$ , et  $(m_1, m_2, m_3, m_4)$  (masques par ligne)
- Calcul de  $(m_1', m_2', m_3', m_4')$ , image de  $(m_1, m_2, m_3, m_4)$  par le MixColumns (linéaire par rapport à l'addition dans  $GF(2^8)$  et donc au  $\oplus$ )
- Ajout d'une fonction ChangeMasks
- Adaptation de la génération des clés de ronde en conséquence



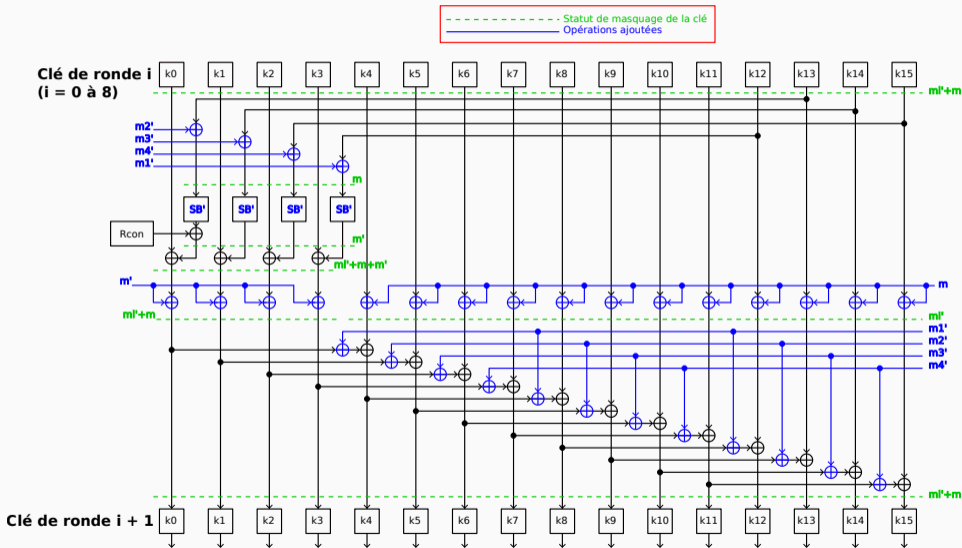
- **Conséquence** : il faut adapter certaines opérations
- **AddRoundKey** : au lieu de xorer un octet d'une ligne  $i$  avec  $K$ , il faut le xorer avec  $K \oplus m_i' \oplus m$ 
  - Adaptation de la génération des clés de ronde en conséquence
- **SBox** : il faut calculer une autre permutation, l'unique permutation  $SBox'$  telle que  $SBox'[x \oplus m] = SBox[x] \oplus m'$
- **ShiftRows** : ne modifie pas les masques, pas de modification
- **MixColumns** : pas de modifications, mais il faut précalculer les  $m_i'$  au début

### Problème

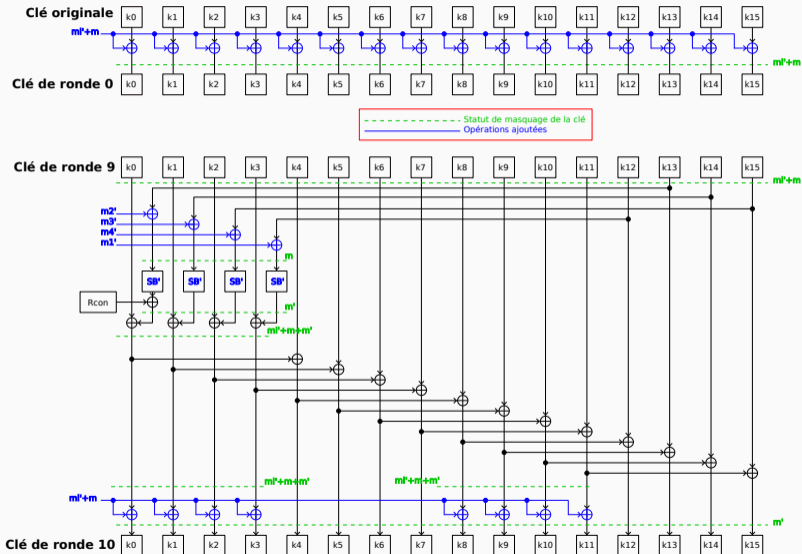
- La génération des clés de ronde utilise les clés en clair
- $\Rightarrow$  On peut aussi la masquer



# Exemple de masquage de l'AES : Key Schedule



# Exemple de masquage de l'AES : Key Schedule



Attaques

Masquage

Masquage Matériel

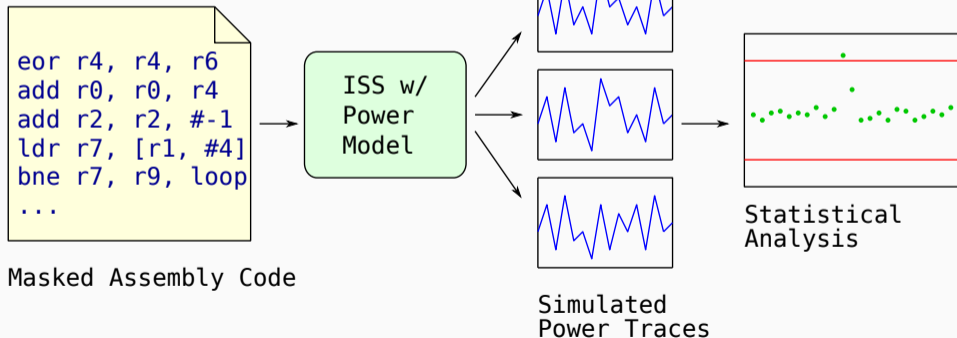
Masquage logiciel

Exemple de l'AES, schéma de Herbst

Vérification du masquage

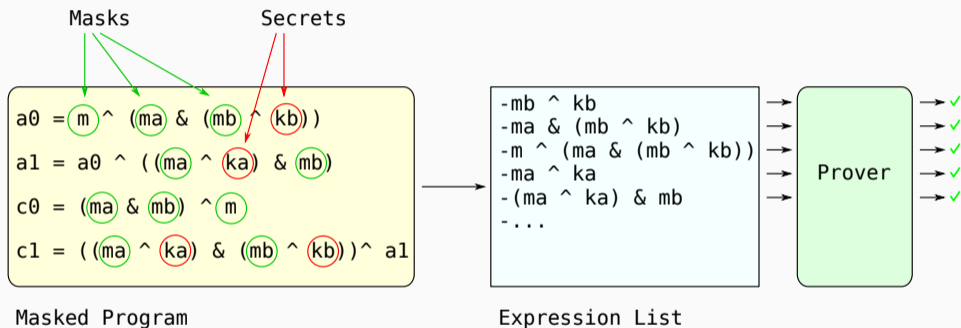
## Comment Vérifier un Schéma de Masquage ?

- **Empiriquement** : Effectuer des simulations ou des captures, et utiliser des tests statistiques comme le t-test
- **Inconvénient** : Ne donne pas de garantie (dépendant de facteurs d'implémentation), localisation des fuites



## Comment Vérifier un Schéma de Masquage ?

- **Formellement** : vérifier que tous les calculs intermédiaires (valeurs) ont une distribution statistiquement indépendante des secrets (TPS)



- **1ère idée** : énumérer toutes les valeurs des entrées, des secrets et des masques, et regarder si les distributions obtenues sont identiques pour toutes les valeurs des secrets
  - Ne passe pas à l'échelle
- $\Rightarrow$  Les outils de vérification formelle sont symboliques

- Plusieurs familles de techniques, parmi lesquelles :
- La vérification par **Inférence**
  - Établir des propriétés pour des expressions simples
  - Inférer des propriétés lors d'opérations entre expressions, à partir des propriétés de ces expressions et de l'opérateur
- La vérification par **Substitution**
  - Transformer successivement une expression grâce aux masques jusqu'à ce que la propriété de sécurité soit vérifiée
- Les 2 techniques peuvent être utilisées à la fois pour la vérification logicielle et matérielle

- Bibliothèque Python pour la vérification d'expressions symboliques (HW et SW)
- Chaque variable a un type parmi **secret**, **mask** et **public**
- Les variables et constantes ont une taille fixe arbitraire (ex : 9 bits)
- Support pour les opérations booléennes et arithmétiques
- Possibilité d'utiliser des valeurs concrètes pour la simulation
- Beneficie du flot de contrôle Python
- Représentations possibles avec shares ou secrets
- Constructions matérielles (portes, registres), glitches
- Implémente différentes propriétés de sécurité
- Vérifications aux ordres supérieurs

## Circuit DOM-AND en VerifMSI

```
# Secret and Masks declaration
a  = symbol('a', 'S', 1) # 1-bit secret
b  = symbol('b', 'S', 1) # 1-bit secret
z10 = symbol('z10', 'M', 1) # 1-bit mask

# Do the sharing for 'a' and 'b'
a0, a1 = getPseudoShares(a, 2)
b0, b1 = getPseudoShares(b, 2)

# Create input gates
a0 = inputGate(a0)
a1 = inputGate(a1)
b0 = inputGate(b0)
b1 = inputGate(b1)
z10 = inputGate(z10)

# Cross products
a0b0 = andGate(a0, b0)
a0b1 = andGate(a0, b1)
a1b0 = andGate(a1, b0)
a1b1 = andGate(a1, b1)
```

```
# Remaining gates and registers
a1b0 = xorGate(a1b0, z10)
a1b0 = Register(a1b0)
a0b1 = xorGate(a0b1, z10)
a0b1 = Register(a0b1)
c0    = a0b0
c0    = xorGate(c0, a0b1)
c1    = a1b1
c1    = xorGate(c1, a1b0)

# Check the TPS security property
checkSecurity(order, wGlitches, 'tps', c0, c1)
```



---

## Algorithm 1 Algorithm for verifying threshold probing security

---

**Require:** `nodeIn` is the root of the expression to analyse

**Ensure:** `False` is returned if the distribution of the expression `nodeIn` is dependent from a secret it contains. Otherwise, `True` is probably returned.

**procedure** `ThresholdProbingSecurity(nodeIn)`

`n`  $\leftarrow$  `simplify(nodeIn)`

`masksTaken`  $\leftarrow$  `set()`

**while** `True` **do**

**if** `secretVarOcc(n, .) = 0` **then**

**return** `True`

▷ No more secret

**if** `maskedBy(n, ., .)` **then**

**return** `True`

▷ A mask is masking the current node

`mask, nodeToReplace`  $\leftarrow$  `SelectMask(n, masksTaken)`

▷ `mask` is the mask node masking the node to replace

**if** `mask = None` **then**

**return** `False`

`masksTaken.add(mask)`

`n`  $\leftarrow$  `GetReplacedGraph(n, mask, nodeToReplace)`

`n`  $\leftarrow$  `simplify(n)`

---

Types de distributions définis :

- *Random Uniform* (RUD)
- *Unknown* (UKD)
- *Constant* (CST)
- *(Statistically) Independent from Secrets* (ISD) : même distribution pour toutes les valeurs du secret.

$$e = (k \oplus m_1) \& m_2$$

| k | m <sub>1</sub> | m <sub>2</sub> | e |
|---|----------------|----------------|---|
| 0 | 0              | 0              | 0 |
|   | 0              | 1              | 0 |
|   | 1              | 0              | 0 |
|   | 1              | 1              | 1 |
| 1 | 0              | 0              | 0 |
|   | 0              | 1              | 1 |
|   | 1              | 0              | 0 |
|   | 1              | 1              | 0 |

$$\left. \begin{array}{l} P(e=0|k=0) = \frac{3}{4} \\ P(e=1|k=0) = \frac{1}{4} \end{array} \right\}$$

$$\left. \begin{array}{l} P(e=0|k=1) = \frac{3}{4} \\ P(e=1|k=1) = \frac{1}{4} \end{array} \right\}$$

$$e' = (k \oplus m_1) \& m_1$$

| e' |
|----|
| 0  |
| 0  |
| 1  |
| 1  |
| 0  |
| 0  |
| 0  |
| 0  |

$$\left. \begin{array}{l} P(e'=0|k=0) = \frac{1}{2} \\ P(e'=1|k=0) = \frac{1}{2} \end{array} \right\}$$

$$\left. \begin{array}{l} P(e'=0|k=1) = 1 \\ P(e'=1|k=1) = 0 \end{array} \right\}$$

$(k \oplus m) \& m = (RUD) \& RUD = ISD \implies$  faux

- ▶ Nécessaire de mémoriser les dépendances *structurelles*

$(k \oplus m) \& m = (RUD\{m,k\}) \& RUD\{m\} = UKD$

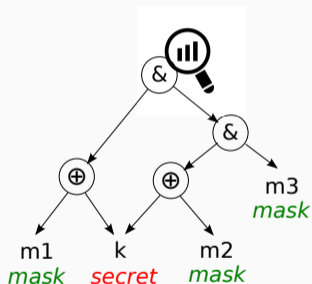
## Types garantis sans fuite

- ▶  $eRUD$
- ▶  $eISD$
- ▶  $eUKD$  sans dépendance structurelle sur un secret

## Type potentiellement vulnérable

- ▶  $eUKD$  avec dépendance structurelle sur un secret

```
# r0 ← k; r1 ← m1; r2 ← m2; r3 ← m3
1 eor r4, r0, r1 # k ⊕ m1
2 eor r5, r0, r2 # k ⊕ m2
3 and r5, r5, r3 # (k ⊕ m2) & m3
4 and r5, r5, r4 # (k ⊕ m1) & ((k ⊕ m2) & m3)
```



Arbre d'expression de la dernière instruction

La distribution de la racine est elle statistiquement indépendante de  $k$  ?

- ▶ Étiqueter les variables d'entrée selon leur type
- ▶ Combinaison ascendante de types de distribution utilisant des règles d'inférence

- expression  $e = e' \oplus m / e = e' + m \text{ mod } 2^n$
- **mRUD**
- **m** n'apparaît pas dans **e'**

⇒ **eRUD** et **m** est un **masque dominant** de **e**.

Pour chaque expression, 2 ensembles :  $\text{dom}_{\oplus}(e)$ ,  $\text{dom}_+(e)$

Exemples :

- $\text{dom}_{\oplus}((k + m1) \oplus (k \oplus m1 \oplus m2)) = m2$
- $\text{dom}_+((k + m1) \oplus 0) = \text{dom}_+(k + m1) = m1$

- Pour  $\oplus, + \text{ mod } 2^n$
- Pour AND, OR
- Distribution des accès SBox = distribution de l'expression d'accès (propriété de permutation)

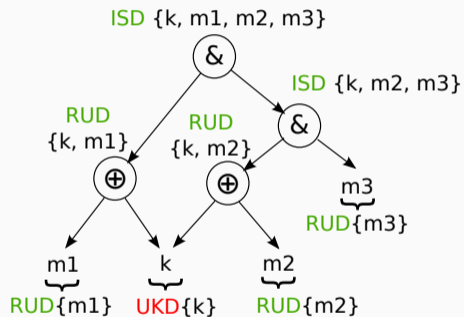
### Règle Disjoint

- Deux expressions  $uISD$  et  $vISD$
- Aucun masque en commun entre  $u$  et  $v$

$\implies (u \text{ op } v)ISD$  pour tout opérateur binaire  $op$

## Exemple d'analyse de distribution

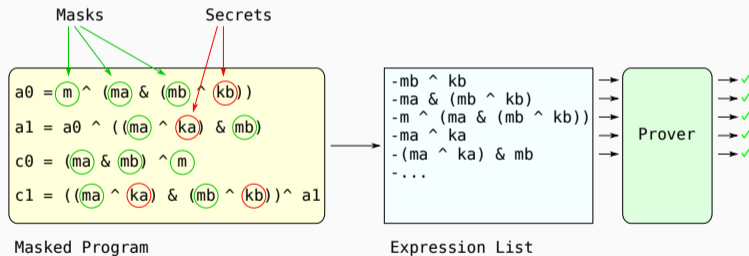
Dernière instruction  $i4$  :  $(k \oplus m_1) \& ((k \oplus m_2) \& m_3)$



▷  $i4$  est statistiquement indépendante de  $k$

## Comment Vérifier un Schéma de Masquage ?

- **Formellement** : vérifier que tous les calculs intermédiaires (valeurs) ont une distribution statistiquement indépendante des secrets (TPS)

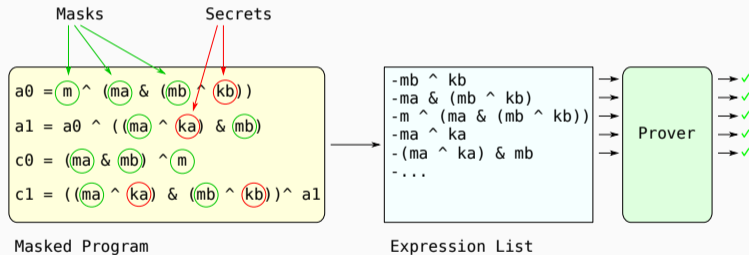


- **1ère idée** : énumérer toutes les valeurs des entrées, des secrets et des masques, et regarder si les distributions obtenues sont identiques pour toutes les valeurs des secrets
  - Ne passe pas à l'échelle
- $\Rightarrow$  Les outils de vérification formelle sont symboliques



## Comment Vérifier un Schéma de Masquage ?

- **Formellement** : vérifier que tous les calculs intermédiaires (valeurs) ont une distribution statistiquement indépendante des secrets (TPS)
- **Inconvénients** : Les expressions considérées pour la preuve ne modélisent pas forcément bien les fuites réelles
- $\Rightarrow$  Un schéma de masquage prouvé peut fuir en pratique



- **1ère idée** : énumérer toutes les valeurs des entrées, des secrets et des masques, et regarder si les distributions obtenues sont identiques pour toutes les valeurs des secrets
  - Ne passe pas à l'échelle
- $\Rightarrow$  Les outils de vérification formelle sont symboliques

# Reduire l'Écart Entre les Expressions Analysées et les Fuites Réelles

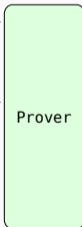
- 2 techniques de vérification ont été proposées pour remplacer la vérification des valeurs intermédiaires
  - Vérifier les transitions entre **variables**
  - Vérifier les transitions entre **Registres Généraux (GPR)**

## Use Variables

```
t0 = mb ^ kb
t1 = t0 & ma
a0 = t1 ^ m
t0 = ma ^ ka
t1 = t0 & mb
a1 = t1 ^ a0
...
```

Transition for **t0**:  
 $(ma \wedge ka) \wedge (mb \wedge kb)$

Transition for **t1**:  
 $((mb \wedge kb) \wedge ma) \wedge ((ma \wedge ka) \wedge mb)$



## Analyse Assembly Code

```
eor r0, r8, r9
and r1, r0, r6
eor r1, r1, r5
eor r2, r6, r7
and r3, r2, r8
eor r3, r3, r1
...
```

r5: m    r8: mb  
r6: ma    r9: kb  
r7: ka

Transition for **r1**:  
 $m$

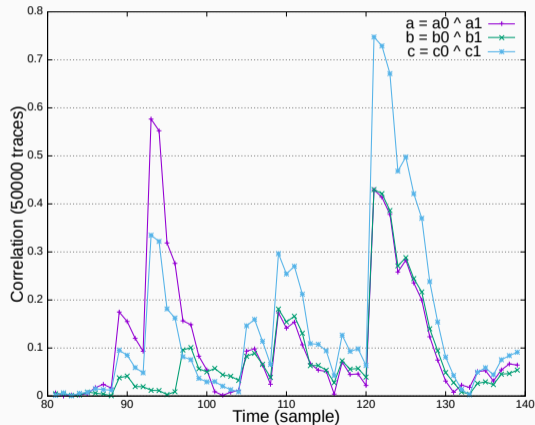
Transition for **r3**:  
 $((mb \wedge kb) \wedge ma) \wedge m$



# Prouvé sans Fuite Secrète ?

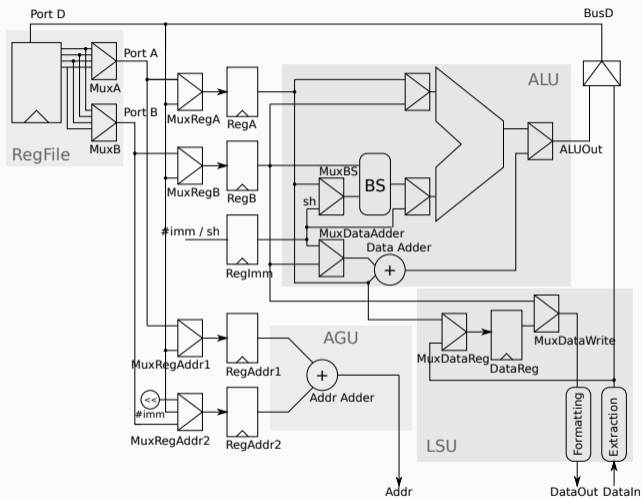
- Cas du “ISW And”
  - Prouvé sans fuite dans le modèle en valeur
  - Prouvé sans fuite dans le modèle en transition entre GPR

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



- Comment se rapprocher des fuites réelles ?

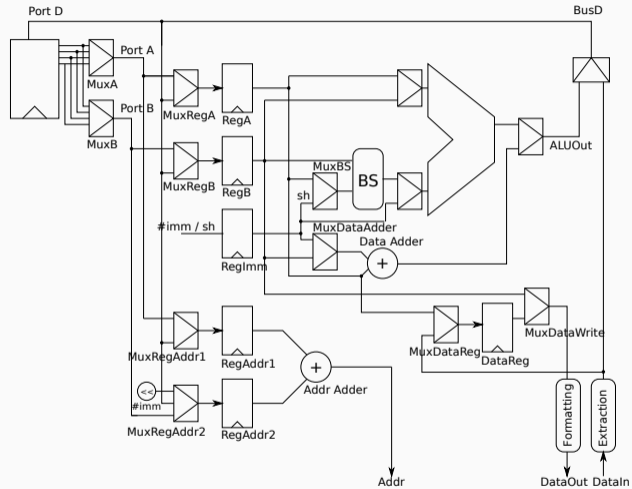
## Cas Étudié : Board STM32F1 [De Grandmaison et al., 2022]



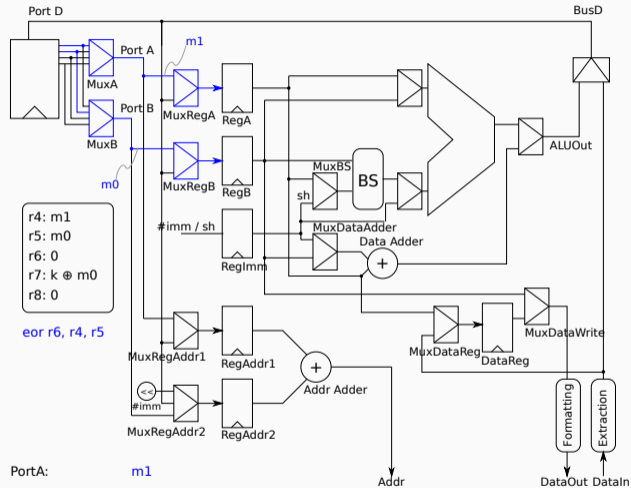
- Arm Cortex-M3 : Modélisé à partir d'une description RTL
- Mémoire : Modèle en boîte noire (pas de RTL)
- Conception et Implémentation de nombreux "leakage test vectors" :
  - Détection des sources de fuite (black-box)
  - Validation (white-box)
  - Classement

- Accessible en ligne : <https://www-soc.lip6.fr/armistice>

# Arm Cortex-M3 : Exemple



# Arm Cortex-M3 : Exemple



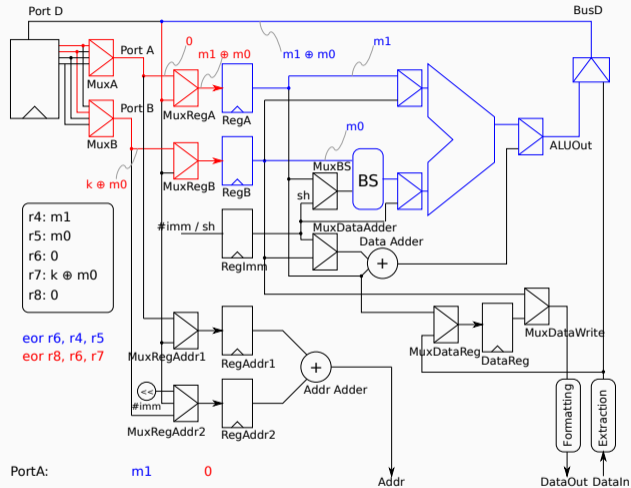
```

r4: m1
r5: m0
r6: 0
r7: k ⊕ m0
r8: 0
    
```

eor r6, r4, r5

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

# Arm Cortex-M3 : Exemple

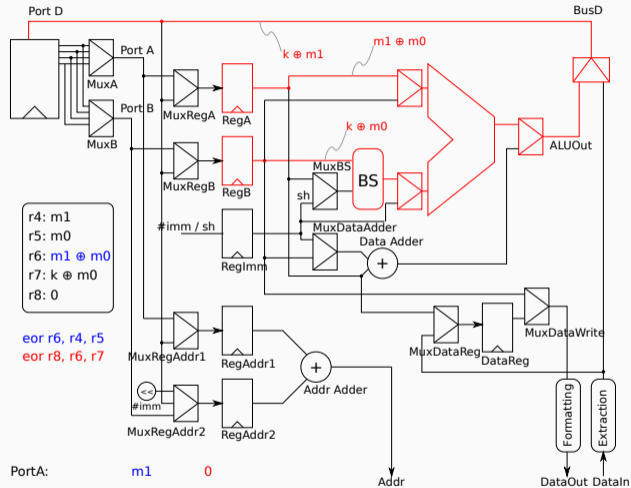


r4: m1  
r5: m0  
r6: 0  
r7: k ⊕ m0  
r8: 0

eor r6, r4, r5  
eor r8, r6, r7

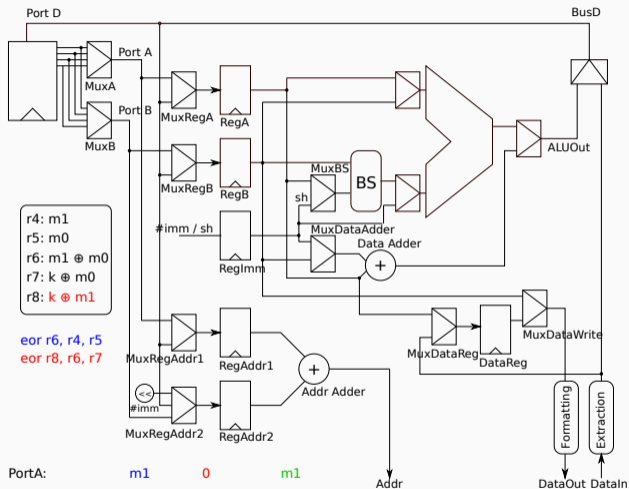
|                 |         |         |
|-----------------|---------|---------|
| PortA:          | m1      | 0       |
| PortB:          | m0      | k ⊕ m0  |
| MuxRegA / RegA: | m1      | m1 ⊕ m0 |
| MuxRegB / RegB: | m0      | k ⊕ m0  |
| ALUOut:         | m1 ⊕ m0 |         |

# Arm Cortex-M3 : Exemple





# Arm Cortex-M3 : Exemple



r4: m1  
 r5: m0  
 r6: m1 ⊕ m0  
 r7: k ⊕ m0  
 r8: k ⊕ m1

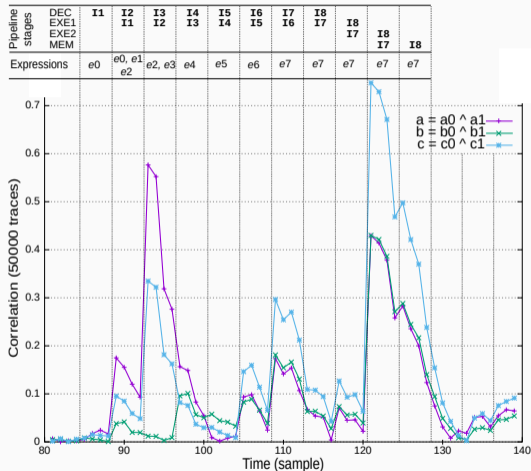
eor r6, r4, r5  
 eor r8, r6, r7

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

# Retour sur le cas du "ISW And"

|    | Instructions     | Leaks : <i>expr. name</i>                        |
|----|------------------|--|
| I1 | and.wr5, r2, r1  | MuxRegA, RegA : e0<br>RegB : e1                  |
| I2 | ands r0, r1      | PortA, RegA : e2<br>AluOut : e3                  |
| I3 | ands r3, r2      | AluOut : e4                                      |
| I4 | eors r4, r5      | RegB : e5  |
| I5 | eors r0, r7      | AluOut : e6                                      |
| I6 | eors r4, r3      | AluOut : e7                                      |
| I7 | str r0, [r6, #0] | -  |
| I8 | str r4, [r6, #4] | PortB, RegB, DataReg,<br>DataOut, BufferMem : e7 |




| Nom | Expression   | Fuites  |
|-----|--|---------|
| e0  | $a0 \cdot b1 \oplus a1$  | a, c    |
| e1  | $a0 \cdot b1 \oplus b0$  | b, c    |
| e2  | $a0 \oplus a1$   | a, c    |
| e3  | $a0 \cdot b0 \oplus a1 \cdot b0$                                       | a, c    |
| e4  | $a0 \cdot b0 \oplus a1 \cdot b1$                                       | a, b, c |
| e5  | $a1 \cdot b0 \oplus b1$  | b, c    |
| e6  | $a0 \cdot b0 \oplus a0 \cdot b1 \oplus a1 \cdot b0$                    | a, b, c |
| e7  | $a0 \cdot b0 \oplus a0 \cdot b1 \oplus a1 \cdot b0 \oplus a1 \cdot b1$ | a, b, c |









Merci !

Contact :

Email : `quentin.meunier@lip6.fr`

-  Ben El Ouahma, I., Meunier, Q. L., Heydemann, K., and Encrenaz, E. (2019).  
**Side-channel robustness analysis of masked assembly codes using a symbolic approach.**  
Journal of Cryptographic Engineering, 9 :231–242.
-  Bordes, N. and Karpman, P. (2021).  
**Fast verification of masking schemes in characteristic two.**  
In Advances in Cryptology–EUROCRYPT 2021 : 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17–21, 2021, Proceedings, Part II, pages 283–312. Springer.
-  De Cnudde, T., Reparaz, O., Bilgin, B., Nikova, S., Nikov, V., and Rijmen, V. (2016).  
**Masking aes with shares in hardware.**  
In Cryptographic Hardware and Embedded Systems–CHES 2016 : 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings, pages 194–212. Springer.

-  De Grandmaison, A., Heydemann, K., and Meunier, Q. L. (2022).  
**Armistice : Microarchitectural leakage modeling for masked software formal verification.**  
IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,  
41(11) :3733–3744.
-  Groß, H., Mangard, S., and Korak, T. (2017).  
**An efficient side-channel protected aes implementation with arbitrary protection order.**  
In Topics in Cryptology–CT-RSA 2017 : The Cryptographers Track at the RSA Conference 2017,  
San Francisco, CA, USA, February 14–17, 2017, Proceedings, pages 95–112. Springer.
-  Herbst, C., Oswald, E., and Mangard, S. (2006).  
**An aes smart card implementation resistant to power analysis attacks.**  
In ACNS, volume 3989, pages 239–252. Springer.
-  Ishai, Y., Sahai, A., and Wagner, D. (2003).  
**Private circuits : Securing hardware against probing attacks.**  
In Annual International Cryptology Conference, pages 463–481. Springer.

-  Meunier, Q. and Taleb, A. (2023).  
**Verifmsi : Practical verification of hardware and software masking schemes implementations.**  
In 20th International Conference on Security and Cryptography, volume 1, pages 520–527.  
SciTePress.
-  Nikova, S., Rechberger, C., and Rijmen, V. (2006).  
**Threshold implementations against side-channel attacks and glitches.**  
In International conference on information and communications security, pages 529–545. Springer.
-  Reparaz, O., Bilgin, B., Nikova, S., Gierlichs, B., and Verbauwhede, I. (2015).  
**Consolidating masking schemes.**  
In Annual Cryptology Conference, pages 764–783. Springer.
-  Wang, W., Ji, F., Zhang, J., and Yu, Y. (2023).  
**Efficient private circuits with precomputation.**  
IACR Transactions on Cryptographic Hardware and Embedded Systems, pages 286–309.