Side-Channel Attacks : Sécurité et Attaques par Canaux Auxiliaires

Tests statistiques, Attaque en Faute, Attaques de cache et micro-architecturales

Quentin Meunier

2025

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

Outline

Tests statistiques

- Motivation
- Généralités
- Application des tests statistiques à l'évaluation des fuites
- t-test spécifique et non spécifique
- Attaques et tests multi-variés

Differential Fault Analysis

Attaques de cache

Outline

Tests statistiques

- Motivation
- Généralités
- Application des tests statistiques à l'évaluation des fuites
- t-test spécifique et non spécifique
- Attaques et tests multi-variés

Differential Fault Analysis

Attaques de cache

Comment savoir si un code fuit des informations secrètes ?

- Première idée : Faire plein d'attaques et s'assurer qu'elles échouent
- Suffisant ?
  - Comment s'assurer qu'avec plus de traces l'attaque ne marchera pas non plus ?
  - Comment s'assurer qu'il n'existe pas une attaque qu'on ne connait pas et qui marche ?
- $\Rightarrow$ Faire des attaques n'est en général pas suffisant

Des questions simples aux réponses moins simples

- Plus généralement, on souhaite pouvoir dire des choses sur la base d'observations
- Exemple : un nouveau médicament réduit-il vraiment la fièvre plus vite ou plus efficacement que l'ancien ?
  - Les données que l'on observe varient toujours d'un patient à l'autre
  - Comment savoir si la différence observée n'est pas simplement due au hasard ? (comment prendre en compte la variablilité)
- $\Rightarrow$ Les observations sont souvent à la base de la prise de décision

Approche statistique

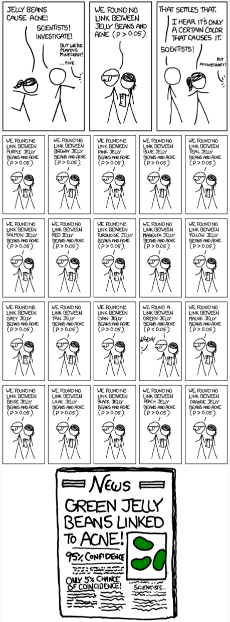
- Statistiques : science de pouvoir dire des choses à partir d'observations
- Certains cas faciles : si on lance une pièce 20 fois et qu'on obtient 19 faces $\Rightarrow$ Grande confiance dans le fait que la pièce est truquée
- Mais si on obtient 15 fois face ?
- Solution : Faire plus de lancer pour savoir
- Problèmes :
  - Il n'est pas toujours possible d'augmenter le nombre d'observations
  - Certains effets peuvent être faibles (par exemple, si la pièce fait face 51% du temps), mais on voudrait quand même s'assurer de leur absence
- Mais les statistiques permettent aussi de conclure parfois avec peu de données, même quand on pourrait croire que c'est impossible
- $\Rightarrow$ Besoin de quantification précise

Principe de l'approche en statistique

- Évaluer la vraisemblance (= probabilité) des observations sous certaines hypothèses
- En déduire quelle hypothèse est la plus crédible
- À la fin, on veut une réponse oui/non, il faut donc définir a priori les conditions d'acceptation / de rejet, en termes de probabilité
- Approche intuitive pour l'exemple de la pièce :
  - Si elle n'est pas truquée, quelle est la probabilité d'observer 15 fois face sur 20 lancers ?
  - On fixe un seuil a priori, par exemple 5% : on conclura donc que la pièce est truquée si cette probabilité est inférieure à 5%
  - P(15 faces sur 20 lancers) pour une pièce équilibrée $= \sum_{i=15}^{20} C_{20}^{i} (\frac{1}{2})^{i} \times (\frac{1}{2})^{20-i} = 2.07\%$
  - On en conclut donc que les observations sont très peu compatibles avec une pièce équilibrée
  - On peut toujours se tromper, mais on sait quantifier notre risque d'erreur

Remarques sur l'exemple

- On ne peut pas vraiment choisir d'autre hypothèse pour les calculs : trop de cas de pièces truquées
  - $\Rightarrow$ On doit donc se contenter de la probabilité d'observation pour une pièce non truquée et conclure avec cela
- Le seuil dépend beaucoup du contexte et des conséquences associées (particule qui va plus vite que la lumière, risque d'effet secondaire grave, etc.)
  - $\Rightarrow$ dépend du risque d'erreur que l'on tolère
  - 5% fréquent, mais : si on tolère un risque d'erreur de 5% (conlusion à tort que la pièce est truquée), cela signifie que si on fait l'expérience 20 fois pour une pièce non truquée, on se trompera en moyenne 1 fois sur les 20



JELLY BEANS CAUSE ACNE!
SCIENTISTS! INVESTIGATE!
BUT WE'RE PLAYING MINECRAFT!
...FINE.
WE FOUND NO LINK BETWEEN JELLY BEANS AND ACNE (P > 0.05).
THAT SETTLES THAT.
I HEAR IT'S ONLY A CERTAIN COLOR THAT CAUSES IT.
SCIENTISTS!
BUT MIIIINECRAFT!
WE FOUND NO LINK BETWEEN PURPLE JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN BROWN JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN PINK JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN BLUE JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN TEAL JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN SALMON JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN RED JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN TURQUOISE JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN MAGENTA JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN YELLOW JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN GREY JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN TAN JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN CYAN JELLY BEANS AND ACNE (P > 0.05).
WE FOUND A LINK BETWEEN GREEN JELLY BEANS AND ACNE (P < 0.05).
WHOA!
WE FOUND NO LINK BETWEEN MAUVE JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN BEIGE JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN LILAC JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN BLACK JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN PEACH JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN ORANGE JELLY BEANS AND ACNE (P > 0.05).
NEWS
GREEN JELLY BEANS LINKED TO ACNE!
95% CONFIDENCE
ONLY 5% CHANCE OF COINCIDENCE!
SCIENTISTS...

Pourquoi a-t-on besoin de statistiques ?

- On observe des données toujours bruitées
- Les intuitions sont parfois trompeuses
- Beaucoup de questions nécessitent de conclure même quand :
  - Le nombre de données est petit
  - L'effet est faible
  - Les mécanismes sont complexes
- Le test statistique permet de :
  - Quantifier l'incertitude
  - Mesurer la plausibilité
  - Prendre une décision avec un risque d'erreur maîtrisé
- Les statistiques permettent de mettre en évidence des effets mais ne les expliquent pas

- But d'un test statistique : étant donné un échantillon, on souhaite tester la vraisemblance d'une hypothèse H_0 contre une autre H_1
- H_0 est dite l'hypothèse nulle, H_1 l'hypothèse alternative
- En général, on choisit H_1 comme étant le complément de H_0 mais ce n'est pas nécessairement le cas
- Dans le langage courant, l'hypothèse nulle est l'hypothèse qui dit qu'il n'y a pas d'effet, pas de corrélation, etc. (pas nécessairement le cas dans le test)
- Un test statistique permet de rejeter H_0 avec un certain seuil de confiance
- Si H_1 est le complément de H_0, rejeter H_0, c'est accepter H_1 ; en revanche, ne pas rejeter H_0 n'est pas accepter H_0

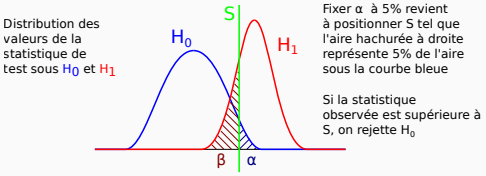
- La statistique de test S est une variable aléatoire définie indépendamment des données observées, qui résume l'information sur un échantillon
- On choisit S de façon à connaître sa loi sous H_0
- La valeur que prend cette variable aléatoire pour les données observées est appelée statistique observée
- La construction d'un test donne la forme de la région de rejet de H_0
- On parle de test unilatéral (à droite) lorsque l'on rejette H_0 si la statistique observée est trop grande
- On parle de test bilatéral lorsque l'on rejette H_0 si la statistique observée est trop grande ou trop petite

- Le risque dit de première espèce (α) est le risque de rejeter H_0 à tort
- Le risque dit de deuxième espèce (β) est le le risque de ne pas rejeter H_0 à tort

	H_0 vraie	H_0 fausse
H_0 non rejetée	Bonne décision (1 - α)	Risque β
H_0 rejetée	Risque α	Bonne décision (1 - β)

- α est choisi arbitrairement, on prend typiquement $\alpha = 0.05$ ou $\alpha = 0.01$

- α et β sont liés : plus on diminue l'un, et plus on augmente l'autre



- La valeur limite du seuil α qui conduit au rejet de H_0 est appelée p-valeur : c'est la probabilité d'observer une réalisation de la statistique de test aussi éloignée de son espérance lorsque H_0 est vraie
- En d'autres termes, c'est la probabilité d'observer quelque chose d'au moins aussi surprenant que ce que l'on observe sous l'hypothèse que H_0 est vraie (on dit dès fois que c'est la probabilité que ce qu'on observe soit "dû au hasard")
- Si la p-valeur est supérieure au α fixé, on considère qu'il n'est pas exceptionnel sous H_0 d'observer la valeur effectivement observée ; par conséquent, H_0 n'est pas rejetée
- Si la p-valeur est inférieure au α fixé, la valeur observée est jugée exceptionnelle sous H_0 ; on décide alors de rejeter H_0

- Il existe plusieurs choses que l'on peut souhaiter tester sur un échantillon :
  - L'égalité de la moyenne de l'échantillon à une valeur fixée
  - La comparaison de la moyenne de l'échantillon à une valeur fixée (inférieure/supérieure)
  - L'adéquation d'un échantillon à une distribution donnée
  - etc.
- Chaque test statistique répond à un problème donné
- Construire un test statistique *from scratch* est difficile car à la fin il faut trouver une fonction (variable aléatoire) pour la statistique de test dont la distribution ne dépend pas des paramètres du modèle

Problème : comment savoir si une implémentation est résistante aux SCA ?

- Point de vue du concepteur
- Différent du point de vue d'un attaquant : chercher à protéger le système de toutes les attaques
- Hypothèses différentes aussi : on connait la clé
- 1ère approche : tester toutes les attaques connues et vérifier que toutes échouent
- 2e approche : caractériser les fuites secrètes de manière statistique
- Dans ce cas, H_0 correspond à l'hypothèse : "il n'y a pas de lien entre les valeurs manipulées à un instant donné et la consommation à cet instant"
- On rejette H_0 si la probabilité sous H_0 d'observer ce que l'on observe est trop faible
- Ce que l'on obtient est une probabilité de fuite secrète, que l'on veut la plus petite possible

Test de Student (ou t-test) de Welch

- Test pour comparer la moyenne de deux échantillons de tailles respectives n_0 et n_1 et de variances différentes
- La statistique de test est donnée par : $t = \frac{|\overline{x_{n_0}} - \overline{x_{n_1}}|}{\sqrt{\frac{s_0^2}{n_0} + \frac{s_1^2}{n_1}}}$
- Le nombre de degrés de liberté à utiliser pour la loi de Student associée peut être approximé par : $v = \frac{\left(\frac{s_0^2}{n_0} + \frac{s_1^2}{n_1}\right)^2}{\frac{\left(\frac{s_0^2}{n_0}\right)^2}{n_0 - 1} + \frac{\left(\frac{s_1^2}{n_1}\right)^2}{n_1 - 1}}$
- Avec :
  - $\overline{x_n} = \frac{1}{n} \sum_{i=1}^{n} x_i$ la moyenne empirique d'un échantillon
  - $s_n^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \overline{x_n})^2$ la variance empirique non biaisée d'un échantillon
- Remarque : lorsque n_0 et n_1 grandissent, le nombre de degrés de liberté grandit également ; dans ce cas, la loi de Student tend vers la loi normale (plus besoin de calculer v)

Test de Student (ou t-test) de Welch : exemple

- Un concepteur veut savoir si son implémentation de l'AES est résistante aux SCA
- Pour cela, il fait un t-test sur chacun des bits i de l'octet de clé k en sortie de la première SBox
- Pour un bit i donné, il sépare les traces en 2 échantillons :
  - Un échantillon avec les traces où le bit i vaut 0
  - Un échantillon avec les traces où le bit i vaut 1
- Ici, l'hypothèse H_0 est : "Les 2 échantillons ont la même moyenne", ce qui correspond à une absence de fuite
- Pour les traces où le bit i vaut 0, il obtient les mesures : 0.66, 0.54, 0.59, 0.68, 0.52, 0.61, 0.63, 0.55
- Pour les traces où le bit i vaut 1, il obtient les mesures : 0.56, 0.55, 0.58, 0.54, 0.59, 0.54, 0.57, 0.53
- On obtient $\overline{x_0} = 0.5975$, $\overline{x_1} = 0.5575$, $s_0^2 = 0.0034$, $s_1^2 = 0.00045$, $t = 1.832$, $v = 8.84$

Test de Student (ou t-test) de Welch : exemple

- Pour convertir ces valeurs de t et v en probabilité, on utilise un logiciel ou on regarde dans une table

$1-\alpha$	75 %	80 %	85 %	90 %	95 %	97,5 %	99 %	99,5 %	99,75 %	99,9 %	99,95 %
k											
1	1,000	1,376	1,963	3,078	6,314	12,71	31,82	63,66	127,3	318,3	636,6
2	0,816	1,061	1,386	1,886	2,920	4,303	6,965	9,925	14,09	22,33	31,60
3	0,765	0,978	1,250	1,638	2,353	3,182	4,541	5,841	7,453	10,21	12,92
4	0,741	0,941	1,190	1,533	2,132	2,776	3,747	4,604	5,598	7,173	8,610
5	0,727	0,920	1,156	1,476	2,015	2,571	3,365	4,032	4,773	5,893	6,869
6	0,718	0,906	1,134	1,440	1,943	2,447	3,143	3,707	4,317	5,208	5,959
7	0,711	0,896	1,119	1,415	1,895	2,365	2,998	3,499	4,029	4,785	5,408
8	0,706	0,889	1,108	1,397	1,860	2,306	2,896	3,355	3,833	4,501	5,041
9	0,703	0,883	1,100	1,383	1,833	2,262	2,821	3,250	3,690	4,297	4,781
10	0,700	0,879	1,093	1,372	1,812	2,228	2,764	3,169	3,581	4,144	4,587
11	0,697	0,876	1,088	1,363	1,796	2,201	2,718	3,106	3,497	4,025	4,437
12	0,695	0,873	1,083	1,356	1,782	2,179	2,681	3,055	3,428	3,930	4,318

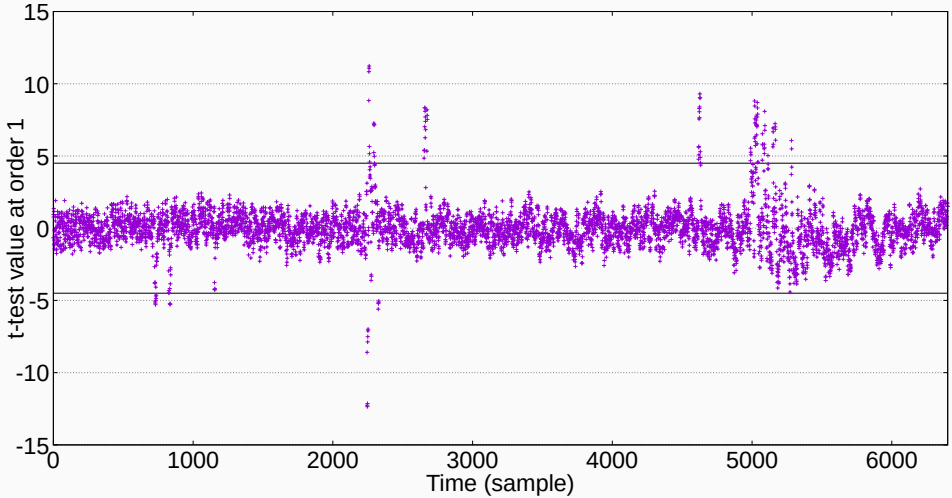
- α est très proche de 5% : on a environ 5% de chances de se tromper en rejetant H_0, c'est-à-dire 5% de chances de se tromper en concluant qu'il y a une fuite secrète sur ce bit
- Remarque : Dans la pratique, le nombre de traces est souvent très grand, donc pas besoin de calculer $v \to$ on prend la loi normale (ligne $k = \infty$)

t-test spécifique

- Un test est dit spécifique lorsque l'on cherche à caractériser la fuite sur un endroit précis d'un programme (ou circuit)
  - Par exemple sur l'AES, sur le bit 0 de l'octet 0 après la première SBox
- La personne qui effectue cette caractérisation connait le programme ou circuit, et donc la valeur de clé
  - Découpage des traces en deux échantillons en fonction de la valeur du bit, calcul des moyennes et variances, puis calcul de la statistique de test
- Problème de cette approche : il faut tester tous les bits internes pour s'assurer qu'il n'y a pas de fuite... ($\tilde{}128 \times 4 \times 10$ au minimum pour l'AES 128)
- ... mais aussi bien d'autres cas : valeur sur plusieurs bits (par exemple, valeur d'un octet à 0 *vs.* toutes les autres valeurs), combinaisons de valeurs intermédiaires, etc.
- Le plus commun en multi-bit : discriminer selon le poids de Hamming (HW), selon qu'il soit inférieur ou supérieur à 4

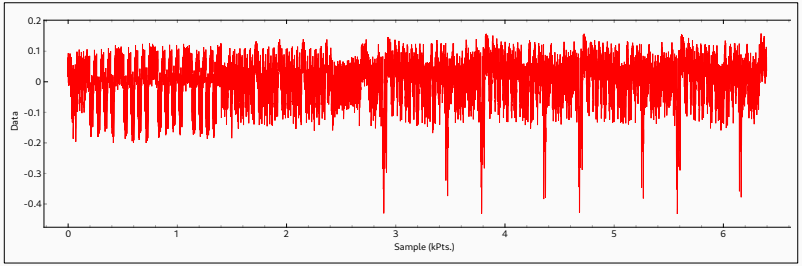
t-test spécifique : exemple

- t-test spécifique sur le bit 3 de l'octet [3][1] de l'état en sortie de SBox, pour la première ronde de l'AES (10 000 traces)



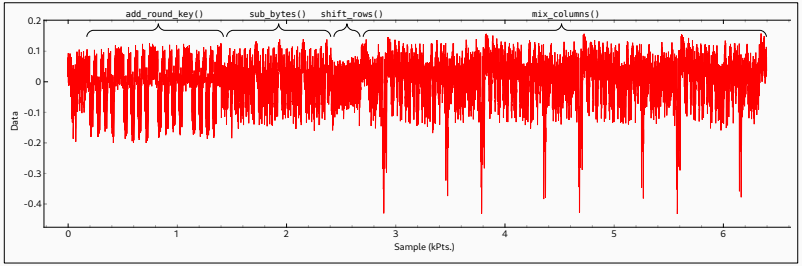
t-test spécifique : exemple

- Trace de consommation correspondante (première ronde de l'AES)



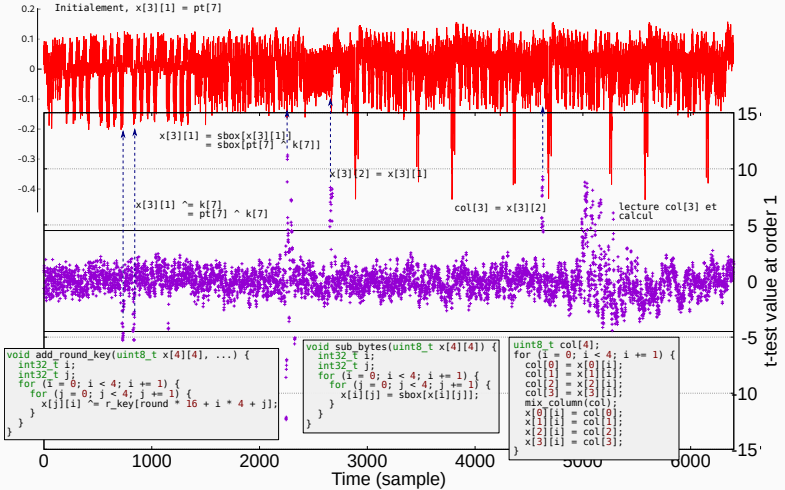
t-test spécifique : exemple

- Trace de consommation correspondante (première ronde de l'AES)



t-test spécifique : exemple

- t-test spécifique sur le bit 3 de l'octet [3][1] de l'état en sortie de SBox, pour la première ronde de l'AES (10 000 traces)

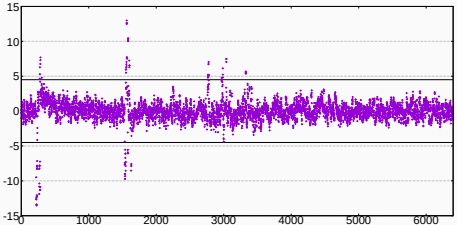


t-test spécifique : remarques et difficultés

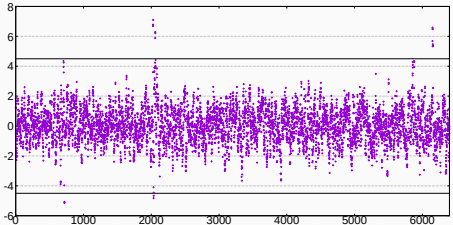
- Sur 8 bits, corrélation entre le HW d'une valeur et le HW de la valeur $\oplus$ une constante `cst` : 1 - 0.25 $\times$ HW(`cst`)
  - En théorie, les deux sont corrélés sauf quand HW(`cst`) = 4
  - En pratique, si la fuite sur la valeur est elévée, on observe une fuite quelque soit `cst`
  - Conséquence : ce n'est pas parce que l'on observe une fuite sur `SBox[k` $\oplus$ `pt]` que `SBox[k` $\oplus$ `pt]` fuit (et donc que `k` fuit) : il peut s'agir uniquement d'une fuite sur `pt` qui "traverse" la SBox
- Corrélation entre les HW en entrée et en sortie de la SBox : 2.5%
- Corrélation entre les valeurs des bits en entrée et en sortie de la SBox :
  - 9.4% pour les bits 0 et 6
  - 6.3% pour les bits 1 et 4
  - 4.7% pour le bit 2
  - 3.1% pour le bit 7
  - 0 pour les bits 3 et 5
- $\Rightarrow$ Pose la question du modèle de fuite : valeurs lues/écrites en mémoire ? Lues/écrites en registres ? $\oplus$ de l'ancienne et de la nouvelle valeur lors de l'écriture en registre ?

t-test spécifique : autres exemples

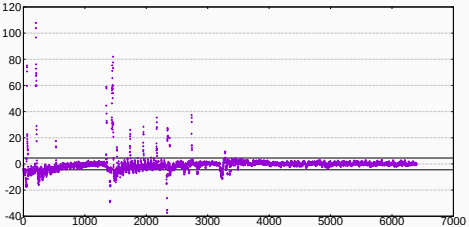
IV : HW(SBox[pt0 ^ k0])



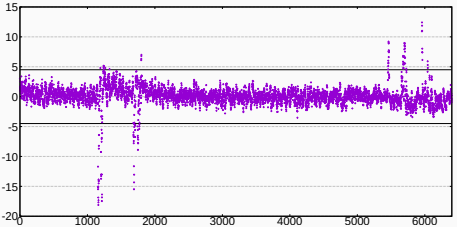
IV : SBox[pt6 ^ k6] & 0x4



IV : HW(pt15)



IV : HW(SBox[pt12 ^ k12])

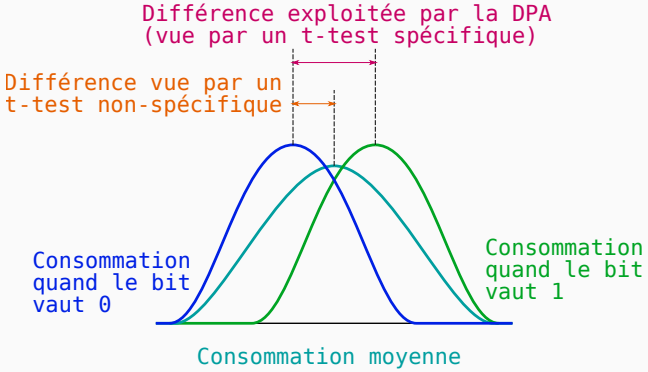


t-test spécifique : remarques et difficultés

- Grande variabilité en fonction du bit, de l'octet et de la valeur ciblée
- Effets bizarres
  - Fuites sur `pt[15]` à chaque itération externe de `sub_bytes`
  - Valeurs à la fin de `mix_columns`
  - Fuite plus grande avant la SBox alors que la valeur ciblée est après
- Certaines observations ne peuvent pas être expliquées sans connaitre le détail du processeur
- Remarque : pour N = 10000, p-valeurs pour des valeurs de la statistique :
  - 4.5 $\rightarrow$ 0.000008
  - 6 $\rightarrow$ 0.000000005
  - 8 $\rightarrow$ 0.000000000000001
  - 9 $\rightarrow$ 0 avec la précision maximale dans R (22 chiffres après la virgule)

t-test non-spécifique

- Cherche à répondre au problème de l'infaisabilité en pratique de tous les t-test spécifiques
- Idée : si un bit interne fuit de l'information (consommation) selon sa valeur 0 ou 1, alors la moyenne entre 2 ensembles de traces, dont l'un correspond à une valeur du bit fixe et l'autre à une valeur aléatoire (uniforme), doit avoir un écart observable



t-test non-spécifique

- Soit un bit intermédiaire du calcul w, et à l'instant de son calcul, les moyennes de consommation $\mu_{w=0}$ et $\mu_{w=1}$ des traces pour lesquelles il vaut respectivement 0 et 1
- Si ces moyennes sont suffisamment différentes (i.e. que w fuit de l'information), alors elles sont chacunes éloignées de la moyenne de toutes les traces $\mu \simeq \frac{\mu_{w=0}+\mu_{w=1}}{2}$ (si $n_0 \simeq n_1$)
- Il suffit donc de tester le programme avec des entrées fixées d'un côté *vs.* des entrées aléatoires de l'autre et de regarder pour chaque instant si on observe un écart de moyenne entre les deux ensembles de traces associés
- La statistique de test au moment du calcul de w sur un test non-spécifique est forcément plus petite que celle sur un test spécifique sur la valeur de w si w fuit de l'information, puisque $\mu_{w=0} < \mu < \mu_{w=1}$ ou $\mu_{w=1} < \mu < \mu_{w=0}$
  - Pour les variances, on doit avoir $s^2 > s^2_{w=1}$ et $s^2 > s^2_{w=0}$ donc le dénominateur de la statistique de test est plus grand pour un test non-spécifique
- $\Rightarrow$ Ces 2 effets font que le test non spécifique est moins sensible
- Une fuite détectée sur un t-test non spécifique $\Rightarrow$ une fuite détectée un t-test spécifique
- Mais le t-test non spécifique ne dit pas de quel bit il s'agit...

t-test non spécifique : cas pratique

- En pratique, on peut avoir un algorithme avec des variables secrètes, publiques, et des masques
- Qu'est-ce qui est fixe en *fixed* et qu'est-ce qui varie en *random* ?
  - Les masques devraient toujours varier (en *fixed* et *random*)
  - Cas 1 : (PT random, K fixe) *vs.* (PT random, K random)
  - Cas 2 : (PT fixe, K fixe) *vs.* (PT random, K random)
  - Cas 3 : (PT fixe, K fixe) *vs.* (PT random, K fixe)
  - Cas 4 : (PT fixe, K fixe) *vs.* (PT fixe, K random)
- Problème cas 1 : le PT peut ici agir comme un masque, or dans le pire cas il pourrait être constant
- Problème cas 2 : détecte les fuites sur le plaintext et la clé, pas uniquement la clé
- Problème cas 3 : détecte les fuites sur le plaintext
- $\Rightarrow$ Le cas 4 devrait être utilisé de préférence quand c'est possible (clé variable), sinon cas 3 :
  - Équivalent dans certains cas, dans l'AES par exemple car on commence par xorer clé et plaintext
  - Mais ne permet pas de capturer les fuites sur le Key Schedule : pour capturer les fuites sur les bits de la clé, il faut que la clé varie en *random*

Comment calculer une moyenne ?

- Ne pas faire : $x_1 + x_2 + ... + x_n$ puis diviser par n
  - Raison : quand le nombre de termes à sommer est grand, différence progressive entre les ordres de grandeurs des termes sommés
- Faire : $\overline{x_0} = 0$, puis itérativement : $\overline{x_i} = \overline{x_{i-1}} + \frac{x_i - \overline{x_{i-1}}}{i}$
- $\Rightarrow$ La même idée peut être appliquée au calcul de la variance pour n'avoir qu'une passe sur les données

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

- Permet de calculer régulièrement la statistique de test (par exemple toutes les 1000 traces) et de s'arrêter dès que le seuil est dépassé : évite de regarder toutes les traces si une fuite apparait dès 1000 traces
- Quand le nombre de traces est très élevé, permet de calculer la statistique sans avoir à stocker les traces

Attaques et tests multi-variés

- Jusqu'à présent, nous n'avons considéré que la fuite en un point (univarié)
- Une attaque multi-variée cherche à mettre en relation des mesures de différents instants
- Il faut savoir quels instants on veut mettre en relation : sinon, explosion combinatoire
  - Exemple en bi-varié pour 3000 samples : 9 millions de combinaisons possibles
- Le but est de choisir des instants où les mesures devraient être corrélées, et où leur combinaison permet de réveler de l'information
- Utile pour attaquer les codes masqués : si l'on ne considère qu'un seul instant, il n'y a pas de fuite d'information
- Par exemple, une CPA bi-variée (dite à l'ordre 2) permet de retrouver la clé d'un code masqué à l'ordre 1
- De la même manière, il y a des tests statistiques multi-variés

Exemple d'attaque multi-variée : CPA à l'ordre 2 sur AES masqué

- Dans le schéma présenté, en sortie de SBox à la première ronde, expressions du type `SBox[k` $\oplus$ `pt]` $\oplus$ `m'`
- $\Rightarrow$ Plus de fuite selon le modèle de fuite en valeur
- Supposons que l'on ait accès à la consommation de l'expression `m'`
- Comment combiner les consommations aux deux instants pour annuler `m'` ?
  - On ne peut pas faire un "$\oplus$" sur les consommations

CPA à l'ordre 2 sur AES masqué

- Idée : il faut trouver un moyen de corréler une fonction des consommations avec le $\oplus$ des deux valeurs
- Sous l'hypothèse que la consommation est le HW, il faut donc trouver une fonction `f(HW(SBox[k ⊕ pt] ⊕ m'), HW(m'))` qui ait une corrélation non nulle avec `HW((SBox[k ⊕ pt] ⊕ m') ⊕ m') = HW(SBox[k ⊕ pt])`
- Fonction possible : valeur absolue de la différence
- En effet sur 8 bits, si `u` et `v` sont uniformes et indépendants :
  ρ(`|HW(u) - HW(v)|, HW(u ⊕ v)`) = 0.24
  - Sous l'hypothèse que la consommation est le HW, `|HW(u) - HW(v)| = |P(u) - P(v)|`
- $\Rightarrow$ On cherche donc à maximiser la corrélation entre `|P(SBox[k ⊕ pt] ⊕ m') - P(m')|` et `HW(SBox[k ⊕ pt])`
- Si impossible d'obtenir la consommation correspondant à l'expression `m'`, possible de faire l'attaque sur 2 expressions masquées : `HW(SBox[k0 ⊕ pt0] ⊕ m')` et `HW(SBox[k1 ⊕ pt1] ⊕ m')`
- Inconvénient : requiert de tester 65 536 hypothèses de clé au lieu de 256

CPA à l'ordre 2 sur AES masqué

- Mais en réalité, on en revient à l'architecture et au code assembleur
- Même registre pour `m'` et `SBox[k` $\oplus$ `pt]` $\oplus$ `m'` :
  fuite en transition de `SBox[k` $\oplus$ `pt]`
  - $\Rightarrow$ CPA à l'ordre 1
- Sinon, si la SBox est calculée dans une boucle, même registre utilisé pour toutes les expressions SBox
- Au moment de la première transition :
  fuite de `HW(SBox[k0` $\oplus$ `pt0]` $\oplus$ `SBox[k4` $\oplus$ `pt4])`
  - $\Rightarrow$ CPA à l'ordre 1 possible avec 65 536 hypothèses

Outline

Tests statistiques

Differential Fault Analysis

Attaques de cache

Outline

Tests statistiques

Differential Fault Analysis

Attaques de cache

Objectifs

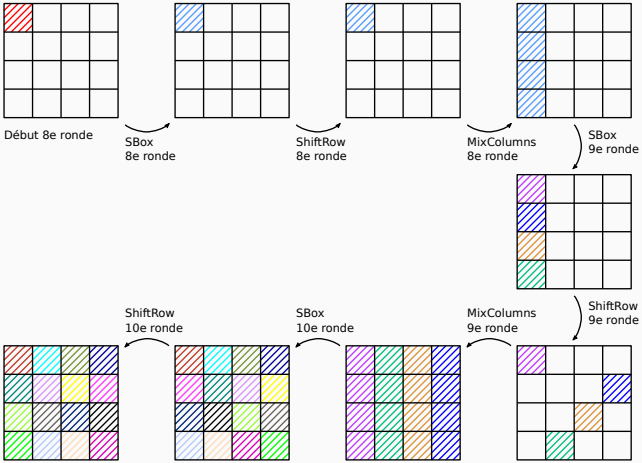
- Retrouver un secret à partir d'une faute lors de l'exécution d'un algorithme de chiffrement
- L'effet de la faute doit en général être assez précis par rapport :
  - Au moment où elle intervient durant l'exécution
  - À la donnée modifiée et sa taille (bit, mot)
- En général, on n'obtient pas directement la clé recherchée, mais une réduction de l'espace d'états (valeurs possibles pour la clé)
- L'attaquant se base sur la valeur du ciphertext fauté, en la comparant avec le ciphertext non fauté par une analyse différentielle
- Hypothèse : pouvoir générer plusieurs chiffrements avec le même plaintext

Moyens possibles de réalisation de fautes

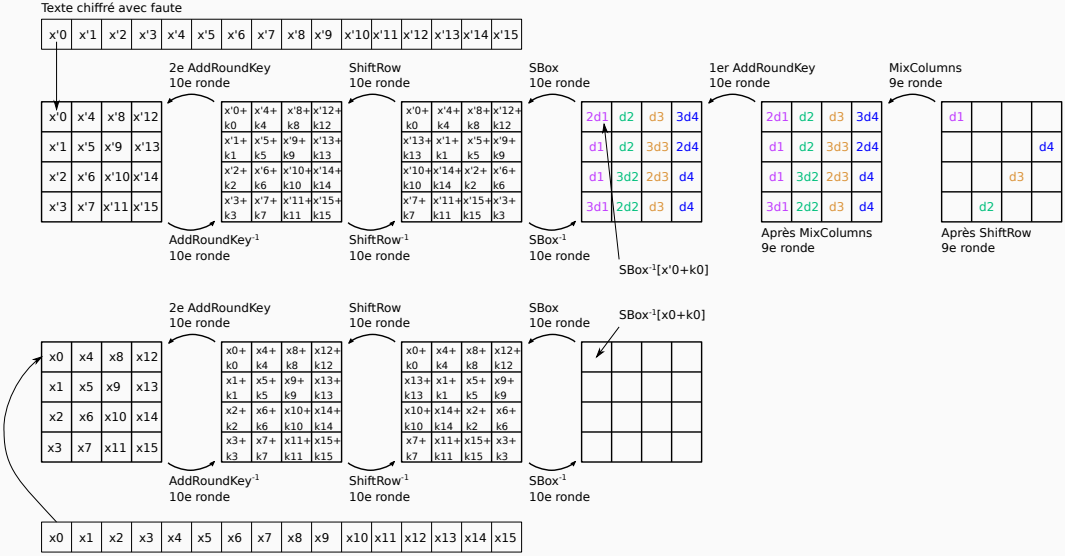
- Glitch d'horloge : front montant qui arrive plus tôt que prévu
- Glitch de tension : surtension pendant un très court instant
- Impulsion électromagnétique proche du circuit
- Laser : impulsion laser sur le circuit décapsulé

Exemple de DFA : “single-fault” sur l’AES

- Hypothèse : la faute se produit au début de la 8e ronde et affecte un octet du *state* (n’importe comment mais un seul octet)
- Cas du premier octet avec une faute δ : `x' = x` $\oplus$ δ



Exemple de DFA : “single-fault” AES



Exemple de DFA : “single-fault” sur l’AES

On en tire donc les 4 équations suivantes

- $2\delta = SBox^{-1}[x_0 \oplus k_0] \oplus Sbox^{-1}[x_0^{'} \oplus k_0]$
- $\delta = SBox^{-1}[x_{13} \oplus k_{13}] \oplus Sbox^{-1}[x_{13}^{'} \oplus k_{13}]$
- $\delta = SBox^{-1}[x_{10} \oplus k_{10}] \oplus Sbox^{-1}[x_{10}^{'} \oplus k_{10}]$
- $3\delta = SBox^{-1}[x_7 \oplus k_7] \oplus Sbox^{-1}[x_7^{'} \oplus k_7]$

- δ, k_0, k_{13}, k_{10} et k_7 sont des inconnues, les x_i et $x_i^{'}$ sont connus
- Résolution : pour chaque valeur de δ, on énumère pour chaque équation les valeurs possibles de k_i.
- Chaque équation retourne 0, 2 ou 4 valeurs possibles pour k_i
- Si aucune des 4 équations ne retourne 0 valeur possible pour k_i, alors cette valeur de δ est possible, et la combinaison des k_i doit être explorée
- Il y a (toujours ?) 15 valeurs de δ satisfaisant les 4 équations, presque toujours 2 valeurs possibles pour les k_i dans ce cas, soit 240 ou 256 combinaisons à explorer pour 4 octets de clé

Exemple de DFA : “single-fault” sur l’AES

- En répétant le processus avec les équations pour les autres colonnes, il y a moins de 2^{32} valeurs possibles pour la clé entière
  - Nécessite un *key schedule* inverse puis un chiffrement complet pour chaque hypothèse de clé complète
- $\Rightarrow$ On peut faire une énumération exhaustive ($\sim$30 minutes sur ma machine pour un code séquentiel compilé en -O3)
- Pour considérer une faute sur les autres octets de `x` au début de la 8^e ronde, on peut calculer les équations correspondantes
  - Il n’y a en réalité que 4 ensembles de 16 équations car chaque ensemble de 16 équations est valide pour 4 octets différents
- On peut retrouver la clé en environ 2h (code séquentiel sur une machine moyenne) pour une faute sur n’importe quel octet de `x`

Outline

Tests statistiques

Differential Fault Analysis

Attaques de cache

Outline

Tests statistiques

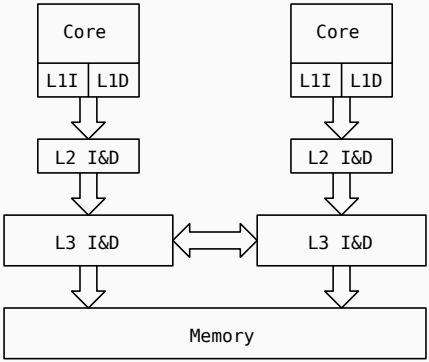
Differential Fault Analysis

Attaques de cache

Attaques de cache

- Attaques plutôt récentes
- Concernent un grand nombre de processeurs
- Atténuation des effets de ces attaques pas facile
- Peuvent être utilisées en combinaison d'autres attaques (ex : Meltdown)

Hiérarchie mémoire des processeurs Intel



- Le cache L3 est partagé entre les coeurs
- Les données incluses dans les caches L1/L2 sont forcément dans le cache L3 : caches inclusifs
- Comment utiliser cela pour faire une attaque ?

Remarques sur la hiérarchie mémoire

- Question : puisque les pages physiques des processus sont disjointes (mécanisme de mémoire virtuelle), quel intérêt ?
  - Librairies dynamiques
  - *Memory deduplication* (ou *Kernel Same-page Merging*) : technique utilisée dans les hyperviseurs de machines virtuelles (mais aussi dans les OS) pour réduire l'empreinte mémoire
- $\Rightarrow$ Il y a donc des pages partagées (en lecture) entre les processus
- 1^{ere} Remarque : si on invalide (*flush*) une ligne de cache du L3, elle est invalidée dans tous les caches L1/L2 (conséquence de la propriété d'inclusivité)
- $\Rightarrow$ On peut donc s'amuser à ralentir les autres processus
  - Mais ça ne fait pas une attaque...
- 2^{e} Remarque : un hit en cache est plus rapide qu'un miss
- Et donc...

Remarques sur la hiérarchie mémoire

- ...Si un autre processus exécute du code d'une librairie partagée, on peut savoir si des instructions contenue dans une ligne de cache ont été exécutées
- Principe :

```
1       flush(line with some insts);
2       wait();
3       a = timestamp();
4       load(line with some insts)
5       b = timestamp()
6       line_acessed = (b - a) < threshold
```

- OK, mais je n'ai rien à cacher...

Remarques sur la hiérarchie mémoire

- ...Si un autre processus exécute du code d'une librairie partagée, on peut savoir si des instructions contenue dans une ligne de cache ont été exécutées
- Principe :

```
1       flush(line with some insts);
2       wait();
3       a = timestamp();
4       load(line with some insts)
5       b = timestamp()
6       line_acessed = (b - a) < threshold
```

- OK, mais je n'ai rien à cacher...
- ...Sauf quand j'utilise RSA !

RSA

- Algorithme de chiffrement asymétrique avec paire (clé publique, clé privée) pour l'envoi de messages chiffrés à un destinataire identifié et la signature de messages
- Utilisé pour l'échange de clé des algorithmes symétriques en général
- Étapes pour la génération des clés :
  - Choisir p, q deux nombres entiers premiers et calculer $n = pq$
  - Choisir un exposant public e (par exemple $e = 65537$)
  - Calculer l'exposant privé $d = e^{-1} \mod (p-1)(q-1)$
- La clé privée est (p, q, d)
- La clé publique est (n, e)
- Chiffrement (ou vérification de signature) : $E(m) = m^e \mod n$
- Déchiffrement (ou signature) : $D(c) = c^d \mod n$

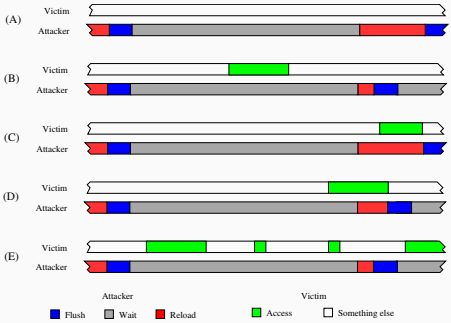
Pseudo-code de l'exponentiation modulaire

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

- Connaitre la séquence des fonctions appelées parmi `square`, `modulo` et `mult` permet de retrouver l'exposant
- Bien sûr, ce code se trouve dans une librairie dynamique
- Et donc un autre processus que celui qui fait le déchiffrement peut retrouver l'exposant secret d
  - En réalité, ce n'est pas aussi immédiat pour CRT-RSA, mais retrouver l'exposant lors du déchiffrement permet quand même de retrouver la clé secrète

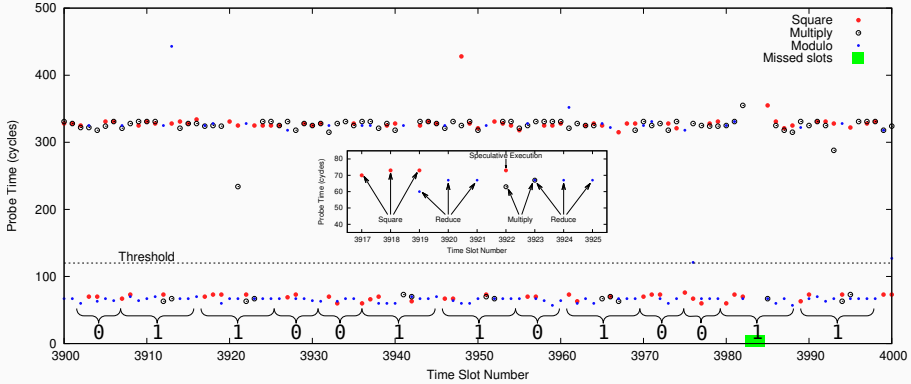
Principe de l'attaque Flush+Reload [Yarom and Falkner, 2014]

- Scénarios (source image : [Yarom and Falkner, 2014]) :



- A : Pas d'accès du processus victime
- B : Accès du processus victime
- C : Accès du processus victime durant le reload (manqué)
- D : Reload durant l'accès du processus victime (potentiellement manqué)
- E : Multiples accès du processus victime

Flush+Reload sur l'exponentiation de RSA



(source image : [Yarom and Falkner, 2014])

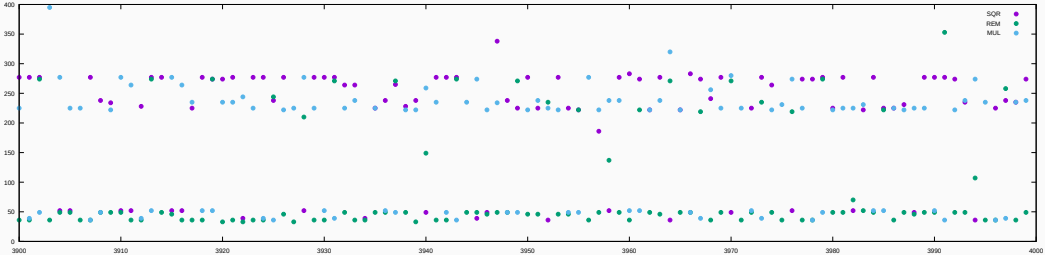
- Difficulté : compromis pour trouver la bonne durée d'attente
  - Trop courte : plus grande chance de manquer des accès à cause de recouvrements
  - Trop longue : détection d'un seul accès au lieu de plusieurs
- Configuration préalable : détermination du seuil qui discrimine un hit d'un miss sur la machine

Flush+Reload en pratique : Sources d'erreur

- Code d'autres applications tournant en même temps sur la machine
- Interruptions
- Préfetch des lignes de cache
  - $\Rightarrow$ Éviter les débuts de fonction
- Compliqué à faire soi-même ?

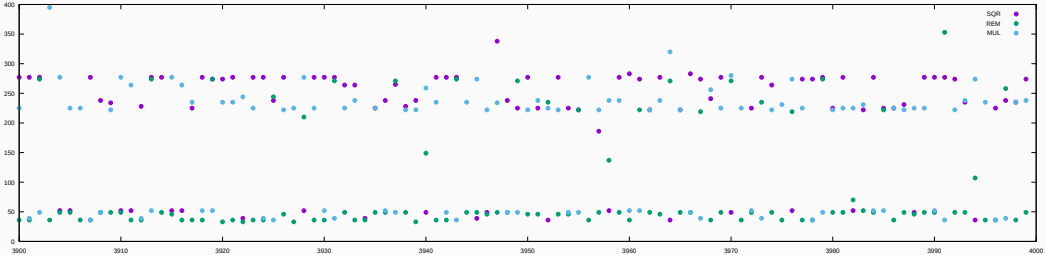
Flush+Reload en pratique : Sources d'erreur

- Code d'autres applications tournant en même temps sur la machine
- Interruptions
- Préfetch des lignes de cache
  - $\Rightarrow$ Éviter les débuts de fonction
- Compliqué à faire soi-même ?



Flush+Reload en pratique : Sources d'erreur

- Code d'autres applications tournant en même temps sur la machine
- Interruptions
- Préfetch des lignes de cache
  - $\Rightarrow$ Éviter les débuts de fonction
- Compliqué à faire soi-même ?



- Que des 1 ?

Flush+Reload en pratique

- Extrait du code dans la fonction d'exponentiation de la lib MPI (Multi-Precision Integer)

```
        _gcry_mpih_sqr_n (xp, rp, rsize, tspace);
      }

    xsize = 2 * rsize;
    if ( xsize > msize )
      {
        _gcry_mpih_divrem(xp + msize, 0, xp, xsize, mp, msize);
        xsize = msize;
      }

    tp = rp; rp = xp; xp = tp;
    rsize = xsize;

    /* To mitigate the Yarom/Falkner flush+reload cache
     * side-channel attack on the RSA secret exponent, we do
     * the multiplication regardless of the value of the
     * high-bit of E.  But to avoid this performance penalty
     * we do it only if the exponent has been stored in secure
     * memory and we can thus assume it is a secret exponent.  */
    if (esec || (mpi_limb_signed_t)e < 0)
      {
        /*mpih_mul( xp, rp, rsize, bp, bsize );*/
        if( bsize < KARATSUBA_THRESHOLD )
          _gcry_mpih_mul ( xp, rp, rsize, bp, bsize );
        else
          _gcry_mpih_mul_karatsuba_case (xp, rp, rsize, bp, bsize,
                                         &karactx);

        xsize = rsize + bsize;
        if ( xsize > msize )
          {
            _gcry_mpih_divrem(xp + msize, 0, xp, xsize, mp, msize);
            xsize = msize;
          }
      }
```

Obstacles à l'attaque Flush+Reload et variantes

- Si les caches L3 ne sont pas partagés par les différents coeurs : Flush+Flush [Gruss et al., 2016]
  - La durée de l'instruction `flush` diffère selon que la ligne se trouve ou non dans la hiérarchie mémoire
  - Comme Flush+Reload, mais `flush` au lieu du reload
  - Pas d'accès mémoire requis, marche aussi quand les caches sont partagés
- Pas d'accès à une instruction comme `rdtsc` pour mesurer le temps
  - Utiliser des API comme `clock()`, `clock_gettime()` (la précision peut ne pas être assez importante)
  - En dernier recours, utiliser un thread qui incrémente un compteur volatile

Obstacles à l'attaque Flush+Reload et variantes

- Si l'architecture ne contient pas d'instruction `flush` (ou si celle-ci est privilégiée) : Evict+Reload [Gruss et al., 2015]
  - $\Rightarrow$ Remplir les sets du cache correspondant à la ligne à espionner en faisant plein d'accès à des lignes mappées dans ce set
  - Peut nécessiter de faire du rétro-engineering sur la politique de remplacement du cache
  - Avantage : les pages n'ont plus besoin d'être partagées entre les processus
- Variante : Prime+Probe [Osvik et al., 2006]
  - Remplir tous les ways des sets à observer
  - Attendre
  - Déterminer quels sets sont toujours occupés
- Ces attaques ont une moins bonne précision et une moins bonne granularité que Flush+Reload

Utilisation des attaques de cache pour faire une Covert Channel

- Covert channel : canal de communication (caché) entre deux processus dont le système n'a pas connaissance : permet de casser l'isolation des processus (un processus peut fuiter des informations sans en avoir les droits)
- Les deux processus peuvent s'envoyer des messages en faisant des lectures en cache à des adresses prédéfinies
- Un processus fait une lecture à une adresse correspondant à 0 et 1 (ou de 0 à 255), puis l'autre processus regarde quelle ligne fait hit et en déduit le caractère transmis

Covert Channel en pratique

```
\subffigure[Sémantique dee la primiitive CASS-2 een pseuddo-code]{
\begin{(minipage){0.57\\linewidth}}
   \code{\\textbf{bool} CAS2(T  *addr1,, TTT *adddr2, T old1, T old2, T nneew1, TT new2)\{}\\
   \hssppace*{0.5cm}\code{<atomic>}\\
   \hspace*{{0..5cm}\code{\\texxttbbf{if} (*addr1 == oldd1 \&\& *addr2 == old2 )\{}\\\
     \hspace*{1cm}\code{*addr1 = new11;}\\
   \hssppacee*{1cm}}\\code{*addr2  = neew2;}\\\
   \hspace*{1cm}\code{\textbf{returnn true};}}\\
   \hspaccee*{0.5cmm}\codee{\}}\\
   \hsspace*{0.5cm}\ccode{\textbf{elsee} \{}\\
   \hspace**{1cm}\coode{\textbf{return false};}\\
    \hspace*{00.5cm}\code{\}}}\\
   \hspacce*{0..5cm}\ccodde{<end atomic>}\\
    \code{\}}}\\\

   \label{{figg:semantiquee_cas2}
\end{minipage}} \hfill
\caaptioon{Sémantique  ddes primmitives  CAS et CAS-2 en  pseudo-code}
\labeel{ffig:semantique_caas_cas2}
\end{figure}
```

Principalement deux types d'ammélioratıons existent pour ceette instruuction : la meilleure consiste à pouvoir oopérrer un CASS sur un nommbre de
que du CCAS--n pouur $n = 2$ est illustrÁéÒe figure-\reeef{fig::semantique_cas2}. Cette instruction est couramment nommée CAS-2 ou DDCCAS (p
ble,, coonsiste à pouvoir eeffectuer des CAS sur dees mots de pluss grande taille ;; ces instructiions peuvent gtre vuess commmme ddes CAS-n avec
nt être contiggus. Sii l'instruction CAAS ooperanntt sur 2 mots est relativemeent répandue dans les jeux d'instructions des processeurs, l'inst
ctures. L'insttruction CCAS-n pour $n > 2$ n'existee dans aucuune architecture commme eÒtannt directeement immmplantée en matéÒriel.

Malheureusement poour ce modèle dde programmattiooonn, la primmittive CASS seulee,, mgme opérant surr deeux motts, reste légère pour beauucoupp dd'al
t nécessairre pour que ce modèle de programmmation puisse aavoir un nommbre plus élevé d'implémentations de strruuctuuures de données effficaceess
ise, il aurait fallu qquuee cce modÁèle de programmmaation ne soit pas si ddiifficile à uutiliisser.

\subsection{Le modèlle de programmmattion trranssactionnel}

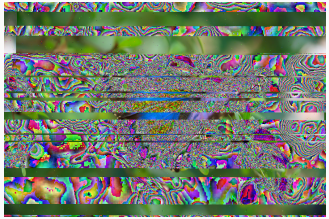
Une auttre alternnattivee auxx verrouss est le mmodèle dde programmaation transactionnel, introduit een 11977 par Lomet~\cite{conf/ldrs/Lommet77}.
 d'innstructions réalisées de mmanière atomique duu point de vue ddes autres tâches s''exécuttant de maniÂère concurrrente. Cela offree une abstrac
s, puisqu'un programmeur peut ainsi rraisonnerr suur la correctionn de sson code Àà l'intérieur d'une transaction sans se soucier dees iinteractions

Isssue du domainne dees bases de doonnées, la sémantique des transactiions mène à quuatre propriétés, connues souus le nom dde ``propriéÒtés ACID''

Lee modèle dde pproogrammatioon transactionnel rresstreint la notion de transaaction à deux seulement de cces ppproprriéttéÒss : attomiccité et iso
i du programme résultant dde l'exécution du code contenu dans unne trransaction sont indivisibblee du point de vue des autres threads.. En dd''
abllees à travers une série d'afffectations, un auuttre thread ne ppeut obssseerver qque l'état immédiatement avantt ou immmédiattement après la
n garantitt que leees ttâches s'exécuutant de manière conncurrente ne peuvennt pass affecterr llee résulltat d'une transsaction allantt à son ter
  Ainsi, une transaction ddoiit produirre le mgmee résultaat que ssi aucune autrre tâcchhhe ne s'exécutaittt een même temps.

Covert Channel en pratique

- Il faut ajouter de la redondance et des mécanismes de correction d'erreur + gérer le descheduling/rescheduling des threads
- Peut alors être très robuste : implémentation du protocole SSH sur des caches entre 2 machines virtuelles allouées sur des serveurs amazon [Maurice et al., 2017]



(a) Image distortion caused by insertion and deletion errors due to scheduling.

(b) Noisy image after handling insertion and deletion errors.

(c) The image after applying error correction. It is equivalent to the original image.

(Source [Maurice et al., 2017])

Meltdown

- Bug rendu public le 3 Janvier 2018
- Affecte tous les processeurs Intel du marché
- Permet de lire la mémoire de tous les processus tournant sur une machine
- Attaque dite micro-architecturale : combine une attaque de cache avec une attaque sur l'exécution out-of-order
- Source de toutes les images : [Lipp et al., 2018]

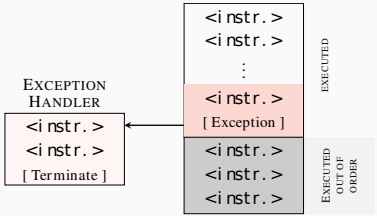
Meltdown : Exécution out-of-order

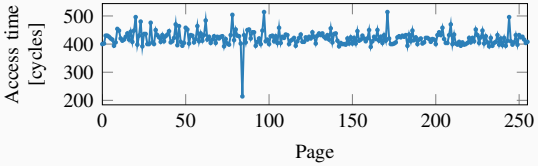
- Mécanisme matériel pour augmenter la performance d'un processeur
- Une instruction assembleur donnée peut s'exécuter avant une autre instruction assembleur qui la précède dans le code source
- Souvent combiné à de l'exécution spéculative, i.e. sans savoir si l'instruction exécutée sera vraiment exécutée, suite à un branchement
- En cas d'exécution spéculative à tort ou d'exception : tous les registres sont réinitialisés à une valeur correcte ou par défaut

Meltdown : Combiner l'exécution out-of-order avec Flush+Reload

- Possible de voir l'effet d'une instruction jamais exécutée

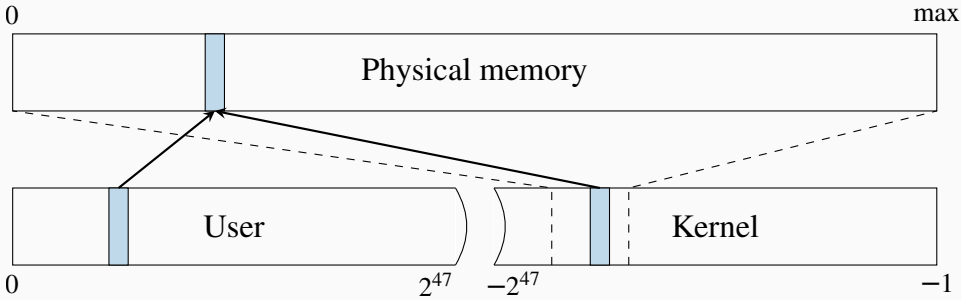
```
1 raise_exception();
2 // the line below is never reached
3 access(probe_array[data * 4096]);
```





Meltdown : Isolation des processus

- Mémoire virtuelle : tous les accès mémoire faits par un processeur sont traduits en adresse physique
- Un processus ne peut donc pas lire/écrire une donné d'un autre processus
- Mais tout le kernel est en général mappé dans l'espace virtuel du processus...
- ...et toute la mémoire physique est mappée dans le kernel
- En cas d'accès à une adresse virtuelle du kernel : exception
  - Les vérifications des droits et l'accès lui-même sont faits en parallèle...



Meltdown : Principe

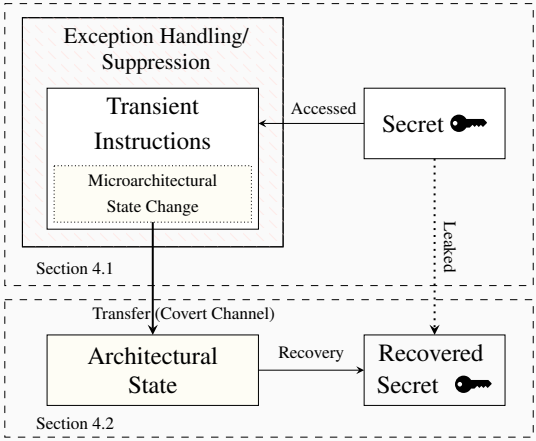
- Faire un accès en lecture à une adresse kernel qui provoque une exception
- Modifier l'état du cache en fonction du résultat de la valeur lue avec une instruction exécutée out-of-order
- Consulter l'état du cache pour savoir quelle valeur a été lue dans le kernel

```
; rcx = kernel address, rbx = probe_array
xor rax, rax
mov al, byte [rcx]
shl rax, 0xc
mov rbx, qword [rbx + rax]
```

Meltdown : Problèmes

- Faire en sorte que l'exception ne termine pas le programme
  - Forker un processus fils avant de faire l'accès et lire le résultat dans le processus père
  - Capturer le signal SIGINT
  - Utiliser la mémoire transactionnelle
- Prefetch $\Rightarrow$ Utiliser des pages différentes
- Lecture fréquente de la valeur 0
  - Inclure une boucle de retry (exécutée spéculativement)
  - Refaire l'attaque si la valeur 0x0 est lue
  - Compromis taux d'erreur $\Leftrightarrow$ Vitesse de lecture de la mémoire

Meltdown : Résumé et résultats



- Permet de lire la mémoire kernel à une vitesse de 500 KB/s
- Avec un taux d'erreur de 0.003% (sur un Core i7)
- Attaque réussie sur tous les processeurs Intel testés, sur tous les OS

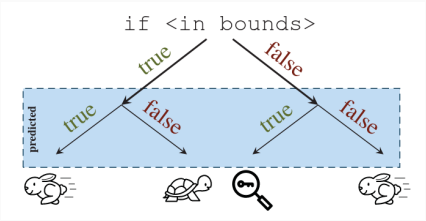
Meltdown : Limites et contre-mesures

- Attaque réussie sur 1 seul processeur ARM, aucun AMD
- Des variantes de l'attaque possible sur ARM en utilisant une autre covert channel
- Contre-mesures logicielles :
  - Ne pas mapper toute la mémoire physique dans le kernel (patch en cours de réalisation à l'époque pour une autre raison, dont Meltdown a accéléré l'intégration)
- Contre-mesures matérielles :
  - Séquentialiser la vérification de permission dans la TLB et le load de la valeur
  - Segmenter l'espace virtuel en deux : Kernel vs. user $\Rightarrow$ Vérification triviale (cf. Mips)
- Meltdown n'est que la première attaque d'une classe d'attaques appelées attaques micro-architecturales ; la 2e plus connue est Spectre

Spectre [Kocher et al., 2020]

- Principe de l'attaque :

```
if (x < array1_size) {
    y = array2[array1[x] * 4096];
}
```



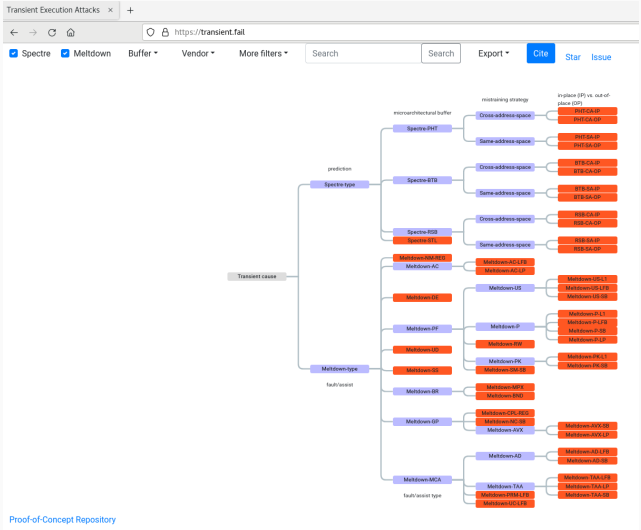
- Exploite le prédicteur de branchement (source : article Spectre)
- Supposons que :
  - x est choisi de manière malicieuse (hors borne), et tel que `array1[x]` soit un octet secret `k` dans la mémoire du processus victime
  - `array1_size` et `array2` sont non cachés, mais `k` est caché
  - Les opérations précédentes ont reçues des valeurs de x valides, menant le prédicteur à supposer que le branchement va échouer.
- $\Rightarrow$ Fuite possible de l'octet secret `k`

Autre attaques micro-architecturales

- Récemment, de nombreuses autres attaques exploitant des failles micro-architecturales
- Foreshadow [Van Bulck et al., 2018, Weisse et al., 2018] : variante de Meltdown, permet de lire des données à l'intérieur d'une enclave SGX
- ZombieLoad [Schwarz et al., 2019] : variante de Meltdown qui exploite un *Fill-Buffer*
- RIDL [van Schaik et al., 2019] : variante de Meltdown qui exploite des *Line-Fill buffers*
- Fallout [Canella et al., 2019] : variante de Meltdown qui exploite le *store buffer*
- Pour plusieurs de ces attaques, les vols de données peuvent se faire avec des hypothèses restreintes (ex : code javascript)

Classification des attaques de type Meltdown et Spectre

- Classification sur le site `https://transient.fail` (TU Graaz)



Autres attaques micro-architecturales

- Hertzbleed [Wang et al., 2022] : exploite la variation de fréquence induite par le changement de consommation, permet de déduire des clés secrètes
- IChannels [Haj-Yahya et al., 2021] : exploite le temps pris par des instructions “couteuses” (SIMD) pour que la fréquence et le voltage s'adaptent, permet de faire des covert channels
- Attaques exploitant la température, etc.

Merci !

Contact :
Email : quentin.meunier@lip6.fr


Canella, C., Genkin, D., Giner, L., Gruss, D., Lipp, M., Minkin, M., Moghimi, D., Piessens, F., Schwarz, M., Sunar, B., Van Bulck, J., and Yarom, Y. (2019).
Fallout : Leaking data on meltdown-resistant cpus.
In Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS). ACM.


Gruss, D., Maurice, C., Wagner, K., and Mangard, S. (2016).
Flush+ flush : a fast and stealthy cache attack.
In International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, pages 279–299. Springer.


Gruss, D., Spreitzer, R., and Mangard, S. (2015).
Cache template attacks : Automating attacks on inclusive last-level caches.
In 24th {USENIX} Security Symposium ({USENIX} Security 15), pages 897–912.

References ii


Haj-Yahya, J., Orosa, L., Kim, J. S., Luna, J. G., Yağlıkçı, A. G., Alser, M., Puddu, I., and Mutlu, O. (2021).
Ichannels : exploiting current management mechanisms to create covert channels in modern processors.
In 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA), pages 985–998. IEEE.


Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., et al. (2020).
Spectre attacks : Exploiting speculative execution.
Communications of the ACM, 63(7) :93–101.


Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. (2018).
Meltdown : Reading kernel memory from user space.
In 27th USENIX Security Symposium (USENIX Security 18).


Maurice, C., Weber, M., Schwarz, M., Giner, L., Gruss, D., Boano, C. A., Mangard, S., and Römer, K. (2017).
Hello from the other side : Ssh over robust cache covert channels in the cloud.
In NDSS, volume 17, pages 8–11.


Osvik, D. A., Shamir, A., and Tromer, E. (2006).
Cache attacks and countermeasures : the case of aes.
In Cryptographers' track at the RSA conference, pages 1–20. Springer.


Schwarz, M., Lipp, M., Moghimi, D., Van Bulck, J., Stecklina, J., Prescher, T., and Gruss, D. (2019).
ZombieLoad : Cross-privilege-boundary data sampling.
In CCS.


Van Bulck, J., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T. F., Yarom, Y., and Strackx, R. (2018).
Foreshadow : Extracting the keys to the Intel SGX kingdom with transient out-of-order execution.
In Proceedings of the 27th USENIX Security Symposium. USENIX Association.
See also technical report Foreshadow-NG [Weisse et al., 2018].


van Schaik, S., Milburn, A., Österlund, S., Frigo, P., Maisuradze, G., Razavi, K., Bos, H., and Giuffrida, C. (2019).
RIDL : Rogue in-flight data load.
In S&P.


Wang, Y., Paccagnella, R., He, E. T., Shacham, H., Fletcher, C. W., and Kohlbrenner, D. (2022).
Hertzbleed : Turning power {Side-Channel} attacks into remote timing attacks on x86.
In 31st USENIX Security Symposium (USENIX Security 22), pages 679–697.


Weisse, O., Van Bulck, J., Minkin, M., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Strackx, R., Wenisch, T. F., and Yarom, Y. (2018).
Foreshadow-NG : Breaking the virtual memory abstraction with transient out-of-order execution.
Technical report.
See also USENIX Security paper Foreshadow [Van Bulck et al., 2018].


Yarom, Y. and Falkner, K. (2014).
Flush+ reload : a high resolution, low noise, l3 cache side-channel attack.
In 23rd USENIX Security Symposium (USENIX) Security 14, pages 719–732.