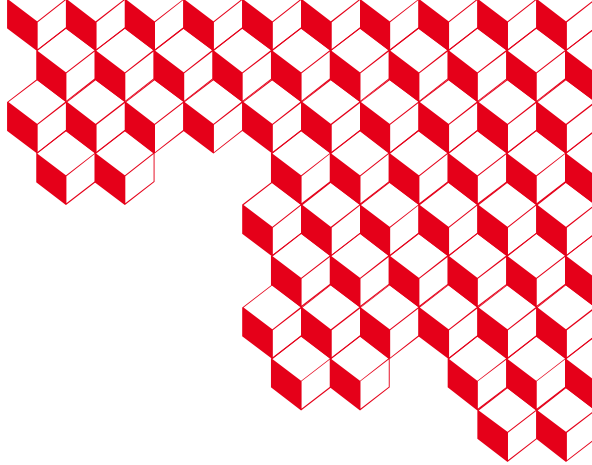


Vers des processeurs d'applications RISC-V intrinsèquement sécurisés

Olivier SAVRY

PROGRAMS VULNERABILITIES

Spatial & temporal memory safety

FAULT INJECTION & SIDE-CHANNEL

Spectre & Meltdown

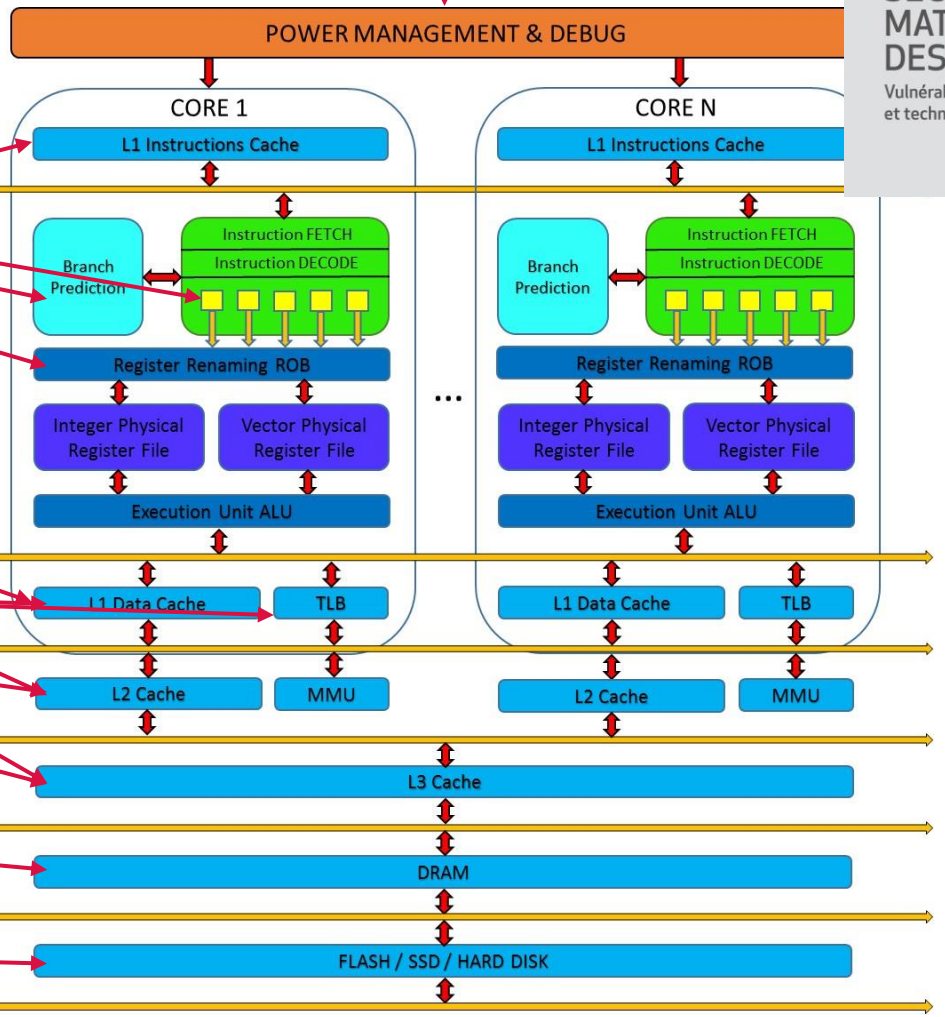
cache timing side-channel
- Flush + Reload
- Prime + Probe
- Flush + flush
- Cachebleed
- Evict + Time
- TLB

SRAM Memory:
- Cell to cell coupling -> sofware fault injection in cache memories with transistors < 10 nm

DRAM Memory:
- RowHammer
- DRAMA

FLASH Memory
- bit to bit programming interference with MLC

Power Management:
- DVFS Exploitation : CLKScrew



Problems : lacks of confidentiality, authenticity and integrity of data and instructions in SoC

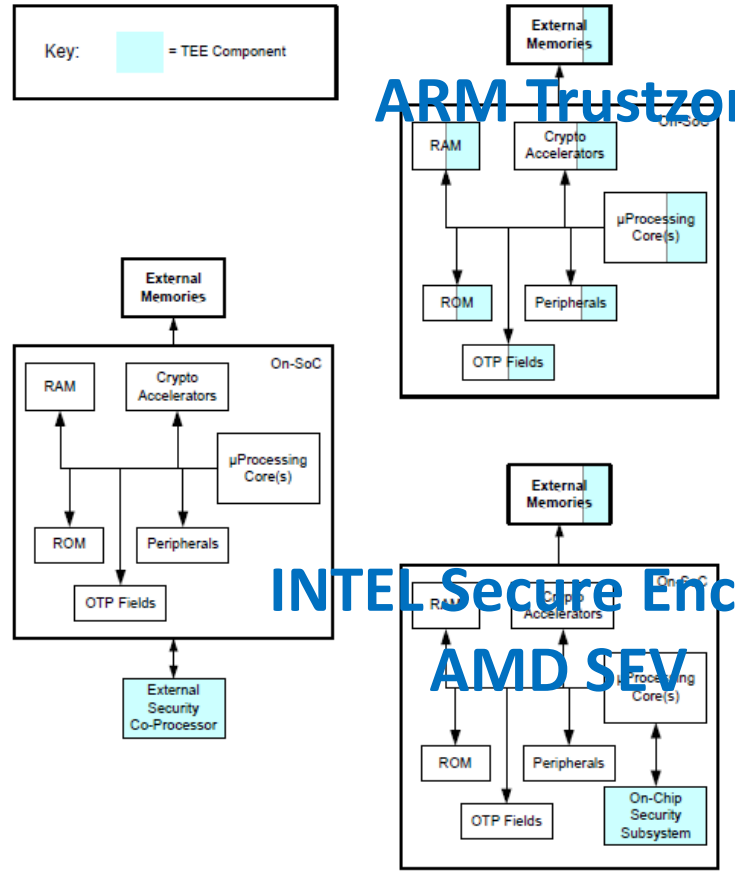
Secure Processors benchmarking (1/2)

- **Secure processors turn around the TEE (Trusted Execution Environment)**
  - Certified by globalplatform
  - inherited from Smart Cards processor (Secure Element)

- **3 types of architecture**
- **Security Objectives :**
  - Isolation from the Rich OS
  - Isolation from other Trusted Applications (TAs)
  - Authenticated modification of the TA
  - Identification and binding
  - Trusted Storage
  - Trusted access to peripherals
  - State of the art cryptography

- **A certification of TAs and TEE**
  - **What about security of applications in Rich OS ?**

Figure 2-3: Examples of TEE Realizations



ARM Trustzone

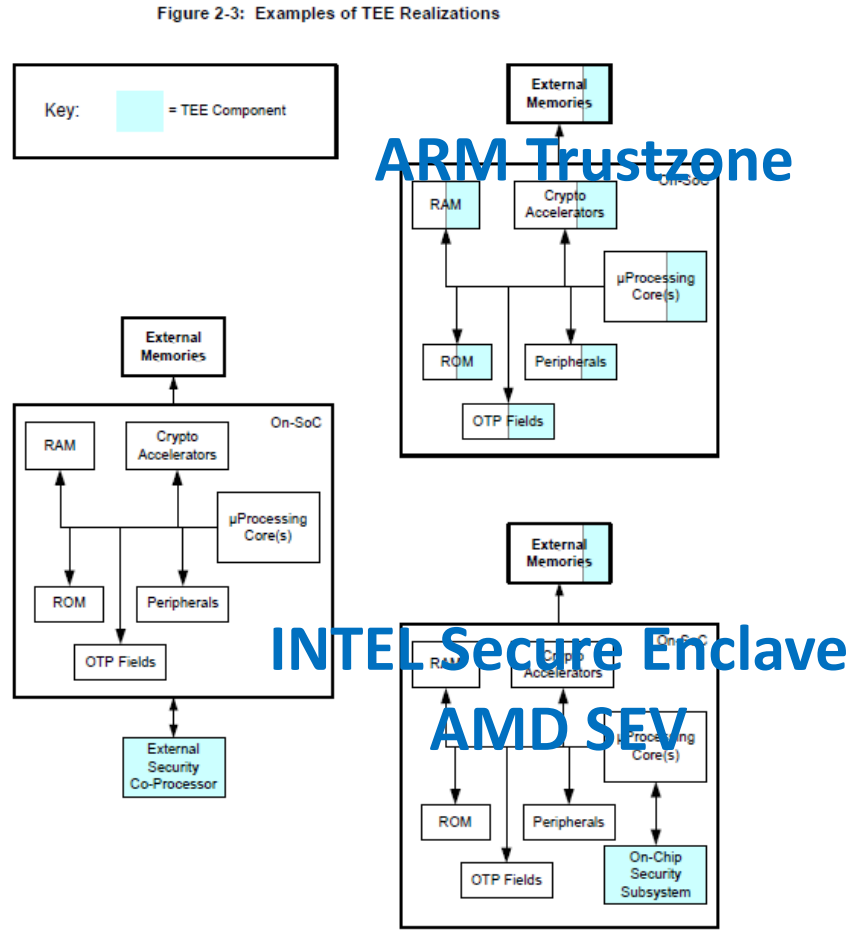
INTEL Secure Enclave

AMD SEV

Secure Processors benchmarking (2/2)

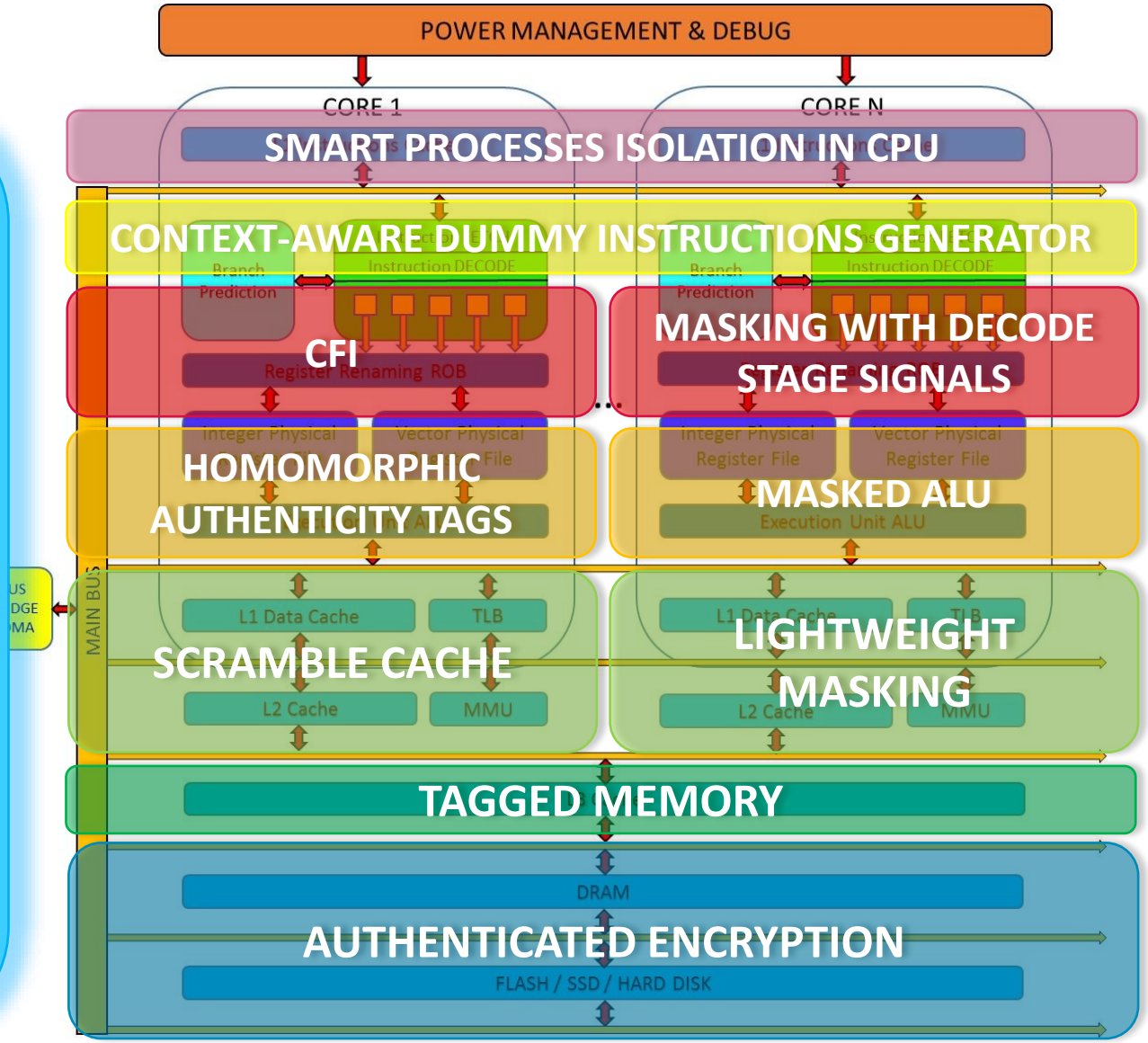
- **The real requirement is security of all applications and mainly those in the Rich OS (in the application processor)**
- **With TEE, the interface with Rich OS is leaking :**
  - Buffer overflow, ROP, Cache timing attacks, rowhammer, spectre, meltdown, side channel attacks, fault injections, energy based attacks
  - Are still possible

- Conceptual problems :
  - **TEE is an externalisation of security** : a guard monitoring an entrance door but leaving the windows open
  - **Security of TA not Rich OS Applications** : the real ones that need to be protected !
  - Running secure applications (TA) alongside applications for which security is not considered to be necessary **leads to increases in the attack surface** that are difficult to manage
  - **Development requires 2 teams** : one for Rich OS applications, the other for the TEE
  - **TEE does not enable to certify security of application processor** and of their Rich OS
  - What about **confidentiality, integrity and authenticity during runtime ?** They are only ensured at the reset in TEE
  - What about **availability and resilience ?**

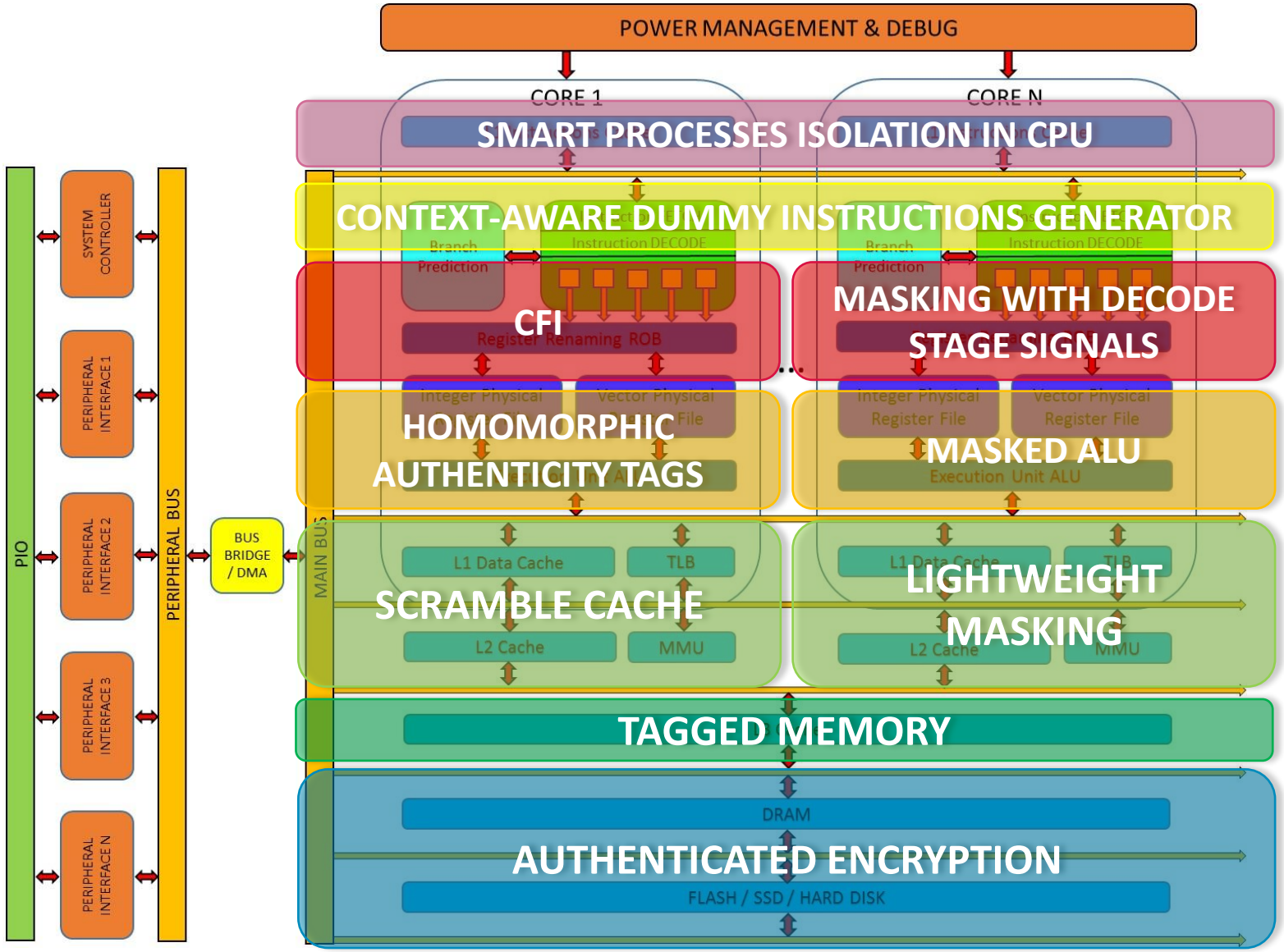


AMBITION : ADRESSER ALL VULNERABILITIES

HOLISTIC APPROACH
FOR END TO END
CONFIDENTIALITY,
INTEGRITY AND
AUTHENTICITY
WITH SECURITY + SAFETY
TOWARDS RESILIENCE
FOR A MINIMIZED
IMPACT ON
PERFORMANCES
FOR A NEW KIND OF
INTRINSICALLY SECURE
PROCESSOR



AMBITION : ADRESSER ALL VULNERABILITIES

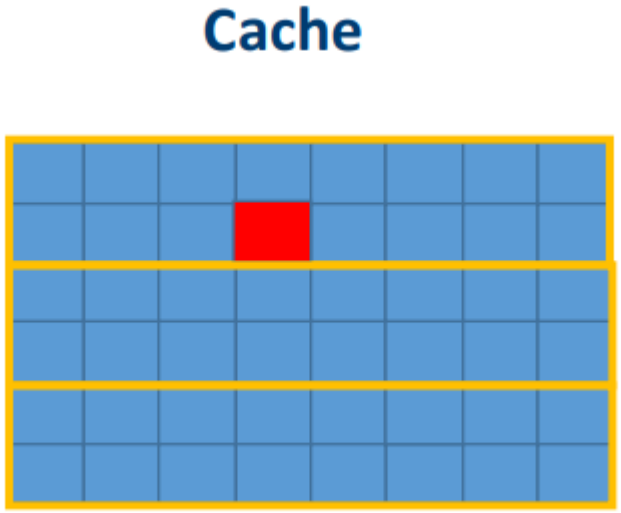


Isolation des processus dans le CPU

- Les processeurs actuels sont agnostiques des différents processus qui sont exécutés en leur sein
  - Les unités fonctionnelles exécutent des instructions qui peuvent venir de processus différents sans le savoir
- Seule la MMU et l'espace d'adressage virtuel qu'elle fournit permet une isolation des processus dans seulement quelques microarchitectures du CPU
- Les attaques Spectre ont montré :
  - La spéculation permet de modifier les états micro-architecturaux du CPU sans montrer de conséquence sur l'état architectural
  - Ces modifications micro-architecturales peuvent être exploitées pour extraire des données secrètes d'un processus vers un autre via un « covert channel »
  - Il existe une quantité importante de ces « covert channels » et pas seulement les caches

FLUSH + RELOAD

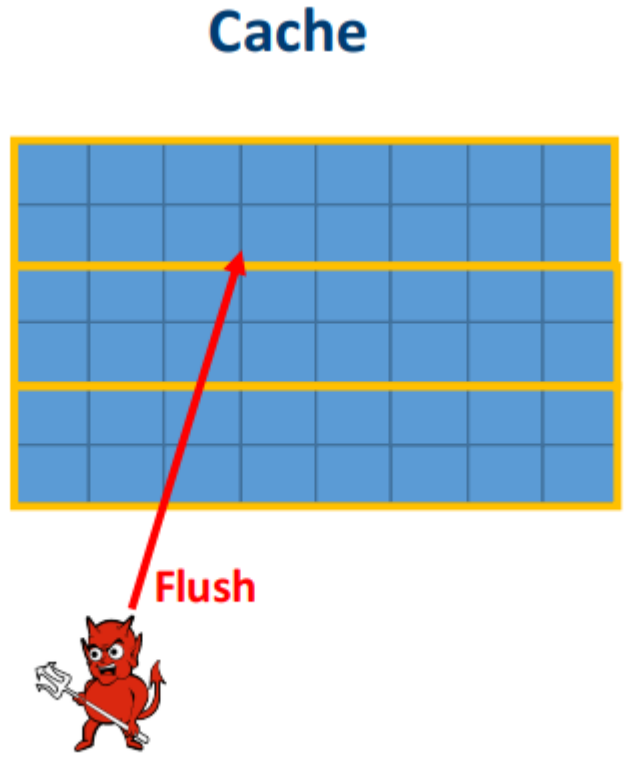
- First, the attacker and the victim have their own process
- To use the Clflush instruction, it is necessary for the attacker to share the same address space than the victim
- The attacker mmap the same executable file or the same shared library
- Normally, the OS will see that the 2 processes uses the same pages and then it will create a deduplication of the pages : the pages used by the 2 processes are physically the same !
- The attacker tries to know if the red data is accessed by the victim process



Yarom, Yuval, and Katrina Falkner. "{FLUSH+ RELOAD}: A high resolution, low noise, l3 cache {Side-Channel} attack."*23rd USENIX security symposium (USENIX security 14)*. 2014.

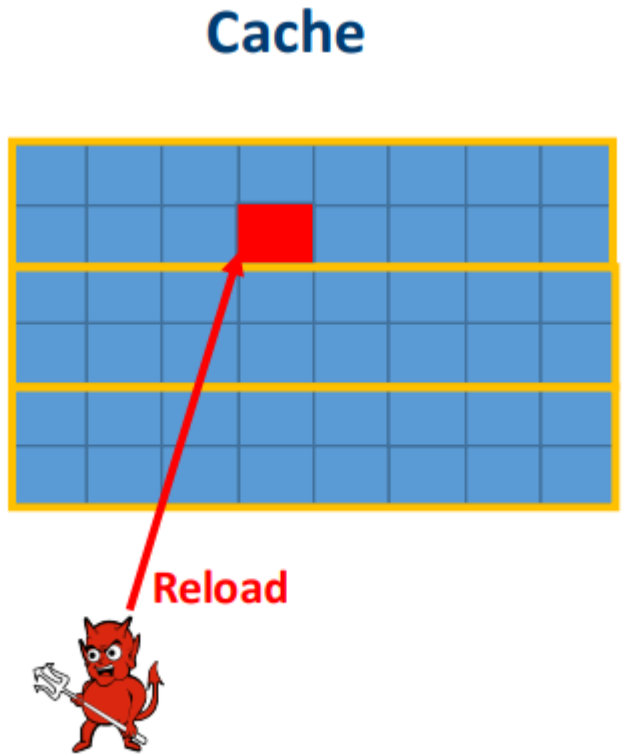
FLUSH + RELOAD

- The attacker tries to know if the red data is accessed by the victim process
- The attacker flushes the red data



FLUSH + RELOAD

- The attacker tries to know if the red data is accessed by the victim process
- The attacker flushes the red data
- The victim accesses/does not access to the red data
- The attacker re-accesses red data and
- measure the access time
  - Fast access time -> victim accessed
  - Slow access time -> victim did not access

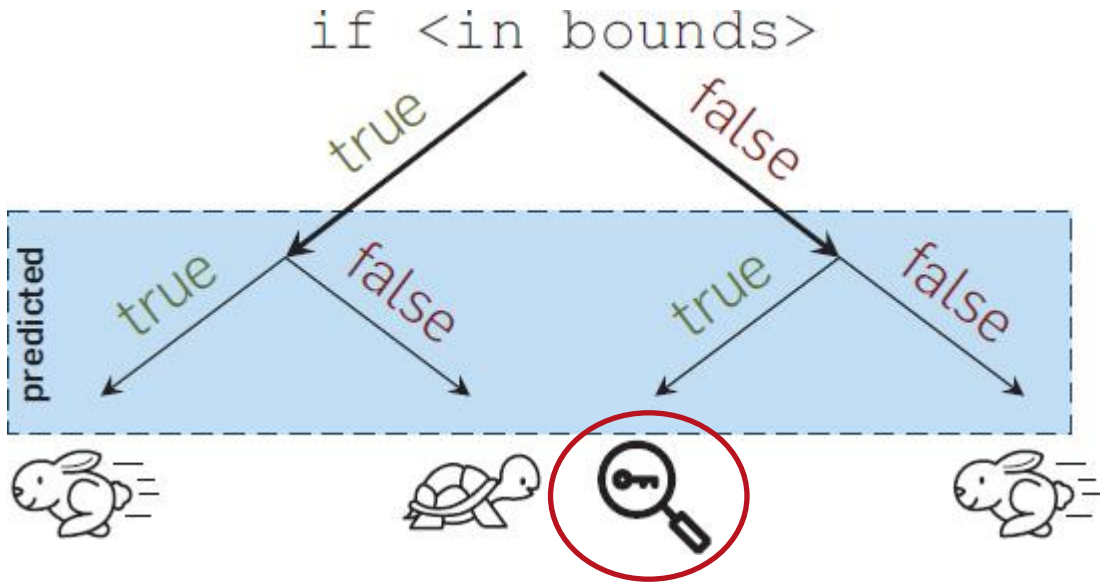


SPECTRE PRINCIPLE : VARIANT 1

- Let's imagine that a victim process accesses an array:

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

- Well-written code that checks the limits of the table!
- The different cases of speculative execution



Before the correct result of the bounds check is known,

- the branch predictor continues with the most likely target branch,
  - resulting in an overall acceleration of execution if the result has been correctly predicted.
- However, if the bounds check is incorrectly predicted as true,
  - an attacker may disclose secret information in certain scenarios.

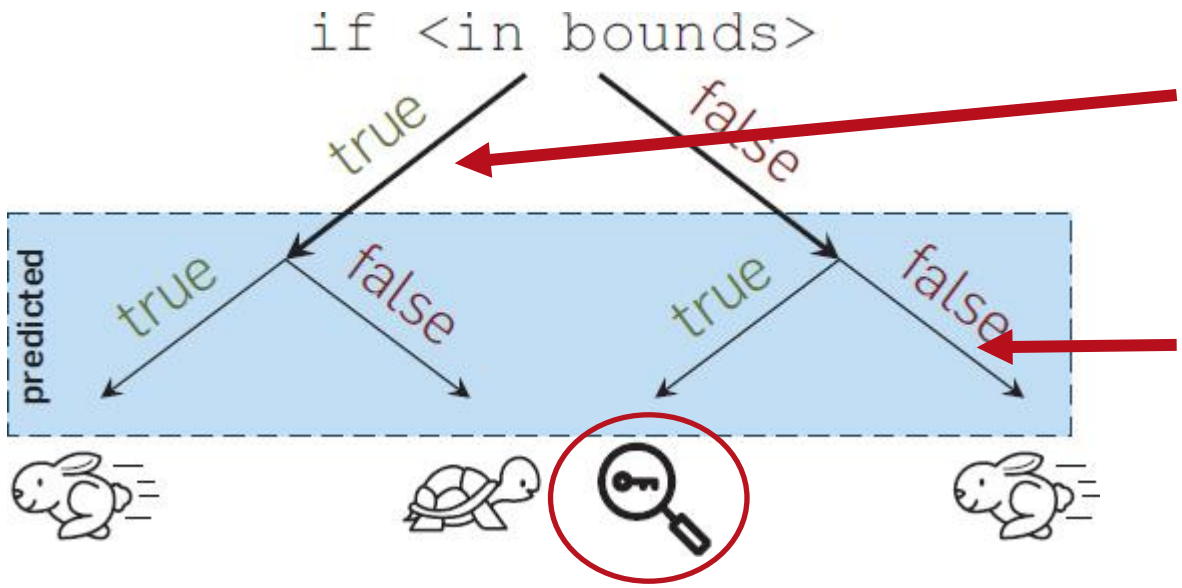
Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., ... & Yarom, Y. (2020). Spectre attacks: Exploiting speculative execution. *Communications of the ACM*, *63*(7), 93-101.

SPECTRE : PRINCIPLE

- Let's imagine that a victim process accesses an array:

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

- Well-written code that checks the limits of the table!
- The different cases of speculative execution



We train the branch predictor on this path

And we prevent it from going here!

Configuration:

- The value of x is maliciously chosen (out of bounds), so that array1[x] resolves to a secret byte k somewhere in the victim's memory;
- array1_size and array2 are flushed, but k (array[1]) is cached; and
- the preceding operations received valid x values, leading the branch predictor to assume that the if value is likely to be true.

SPECTRE : PRINCIPE

- **Let's imagine that a victim process accesses an array:**

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

Configuration:

- The value of x is maliciously chosen (out of bounds), so that array1[x] resolves to a secret byte k somewhere in the victim's memory;
- array1_size and array2 are flushed, but k is cached; and
- the preceding operations received valid x values, leading the branch predictor to assume that the if value is likely to be true.

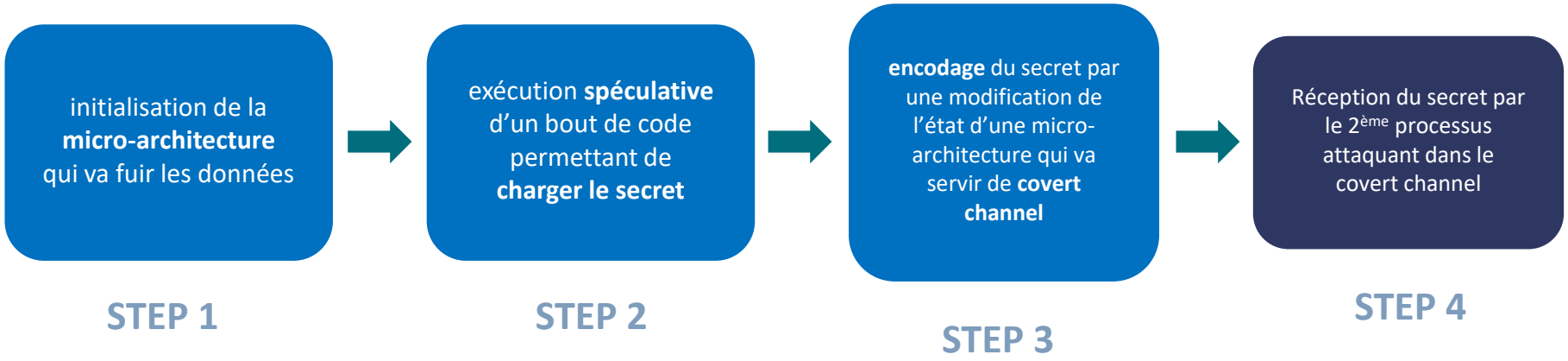
Realization:

- the processor starts by comparing the malicious value of x with array1_size. Reading array1_size causes a miss cache, and the processor faces a substantial delay until its value is available from DRAM.
- In the meantime, the branch predictor assumes that the if value will be true, then speculative execution adds x to the base address of array1 and requests data at the resulting address.
- This read is a cache hit and quickly returns the value of secret byte k (or array[1]).
- Speculative execution continues, using k to calculate the address of array2, and sending a request to read this address from memory (resulting in a cache miss).
- At some point after the read from array2[k*4096] is initiated, the processor realizes that its speculative execution was erroneous and rewinds its register state.
- However, the speculative read from array2 affects the cache state in an address-specific way, where address depends on k.

→ **Flush + Reload**

Spectre

- Modèle d'attaquant :
  - l'attaquant peut exploiter un bout de code (troyen) dans le processus de la victime
    - gadgets dans une bibliothèque partagée
    - Script malicieux dans navigateur
  - Il a la main sur un autre processus appelé « processus attaquant »
- Une attaque Spectre se déroule en 4 étapes



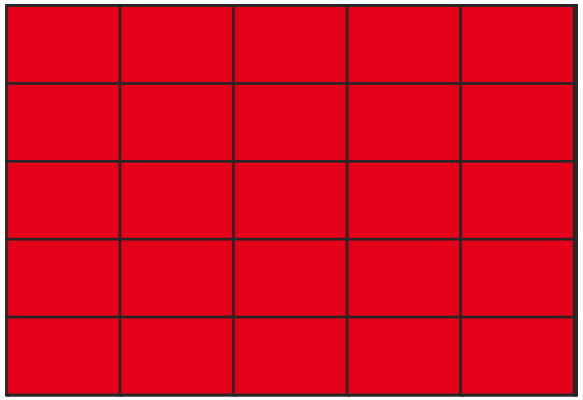
- L'absence de l'une de ces étapes suffit à éviter les attaques Spectre
- … mais pas les fuites de données

Spectre

- Problème :
  - Même en l'absence de spéculation les « covert channels » restent exploitables pour faire fuir les données
- Les covert channels sont multiples :
  - Caches L1, L2, LCC
  - MMU
  - TLB
  - Prédiction de branchement : BHT, BTB, RSB
  - Contention de port
  - Contention de bus
  - Load-store unit
  - FPU, VPU
  - …
- En fait, tout registre, mémoire, buffer partagé entre processus dans le CPU sont susceptibles de fuir

Spectre

- Principe général de fonctionnement d'un covert channel (canal entre 2 processus attaquant : un émetteur et un récepteur)
  1. Initialisation : remplissage total du buffer (aussi petit soit-il !) par le récepteur
     - Un effacement total (flush) est également envisageable (état initial connu)
  2. Modification du buffer par le processus attaquant émetteur : 2 états sont nécessaires
     - Modification ou non de l'état
  3. Sondage du buffer avec une simple utilisation
     - Souvent selon la microarchitecture utilisée, une mesure du temps d'accès est effectuée pour révéler la donnée

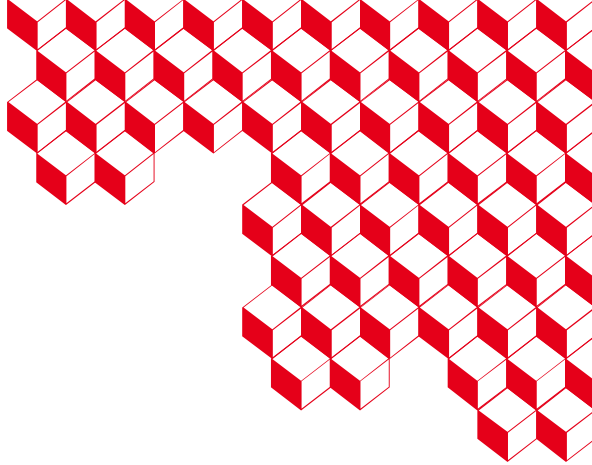


Accès par le récepteur, et mesure du temps d'accès :
Si temps long : l'émetteur a modifié le buffer (bit 1)
Si temps court : pas de modification (bit 0)

Spectre

- 2 solutions simples et déterministes :
  - Partitionnement du buffer par processus
  - Flush du buffer à chaque changement de contexte
- Mais impact sur les performances violent !
- N’existent-il pas des moyens (probabilistes) entre les deux pour éviter la fuite tout en conservant des performances acceptables ?
- Faut-il repenser complètement l’architecture des processeurs ?

- Doctorat dans le cadre du PEPR Arsène (début : juin 2024)
  - Plateforme d’étude : NaxRiscv (OoO) 64 bits sur FPGA
  - Trouver de façon exhaustive les covert channels (méthodes formelles ?)
  - Au cas par cas, trouver des contremesures avec un bon compromis performances/sécurité
  - Evaluer la sécurité



Merci

Prénom NOM (premier niveau)

Prenom.nom@cea.fr (deuxième niveau)

06 00 00 00 00 (deuxième niveau)