Architecture Matérielle 3^e année
Module Sécurité

Vincent Migliore

vincent.migliore@insa-toulouse.fr

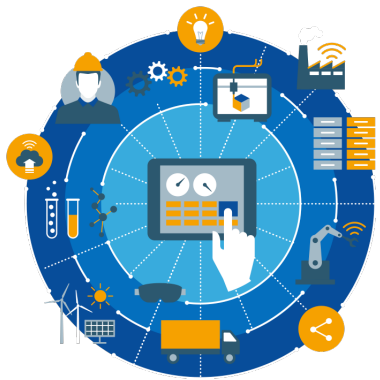
INSA-TOULOUSE / LAAS-CNRS

Contexte

L'émergence de l'outil numérique a donné lieu à la mise en place de services nouveaux tirant partie des capacités de calcul des ordinateurs :

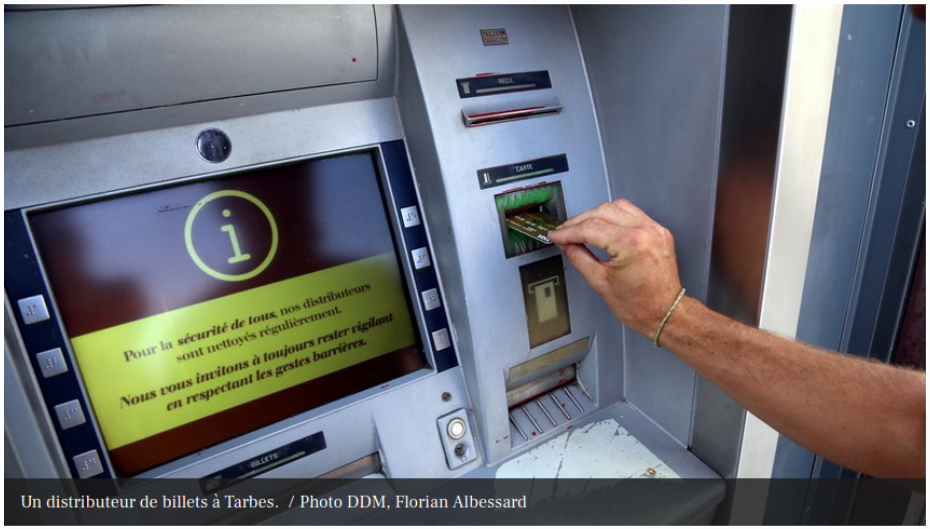
- Automatisation des transactions bancaires (distributeurs automatiques de billets, terminaux de paiement, ...) ;
- Systèmes de transports autonomes (transports publics sans conducteurs, système de détection de panneaux, de collision, ...) ;
- Industrie 4.0 ;
- Réseaux énergétiques intelligents ;
- ...

Contrairement aux systèmes d'information historiques centralisés dans des environnements protégés, les nouveaux systèmes d'information se trouvent de plus en plus déployés dans l'environnement et plus facilement accessibles par des personnes malveillantes.



De nouvelles menaces émergent et ne tirent plus uniquement partie de failles logicielles (fortuitement introduites lors des phases de conception logicielles) mais également de caractéristiques matérielles pour réduire le niveau de sécurité attendu voire contourner totalement les mécanismes de sécurité.

Jackpotting



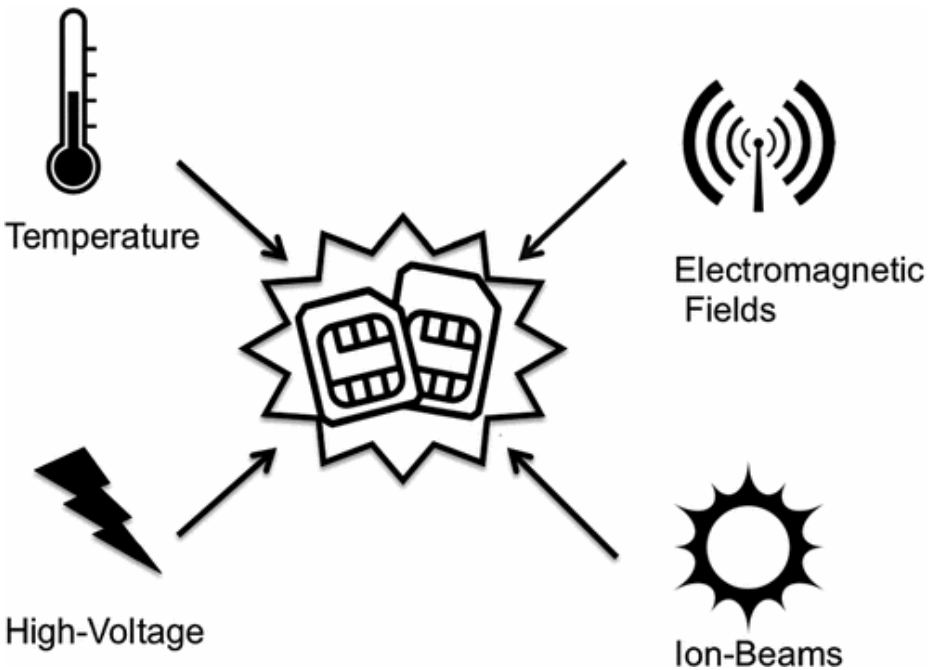
Un distributeur de billets à Tarbes. / Photo DDM, Florian Albessard

Methode :

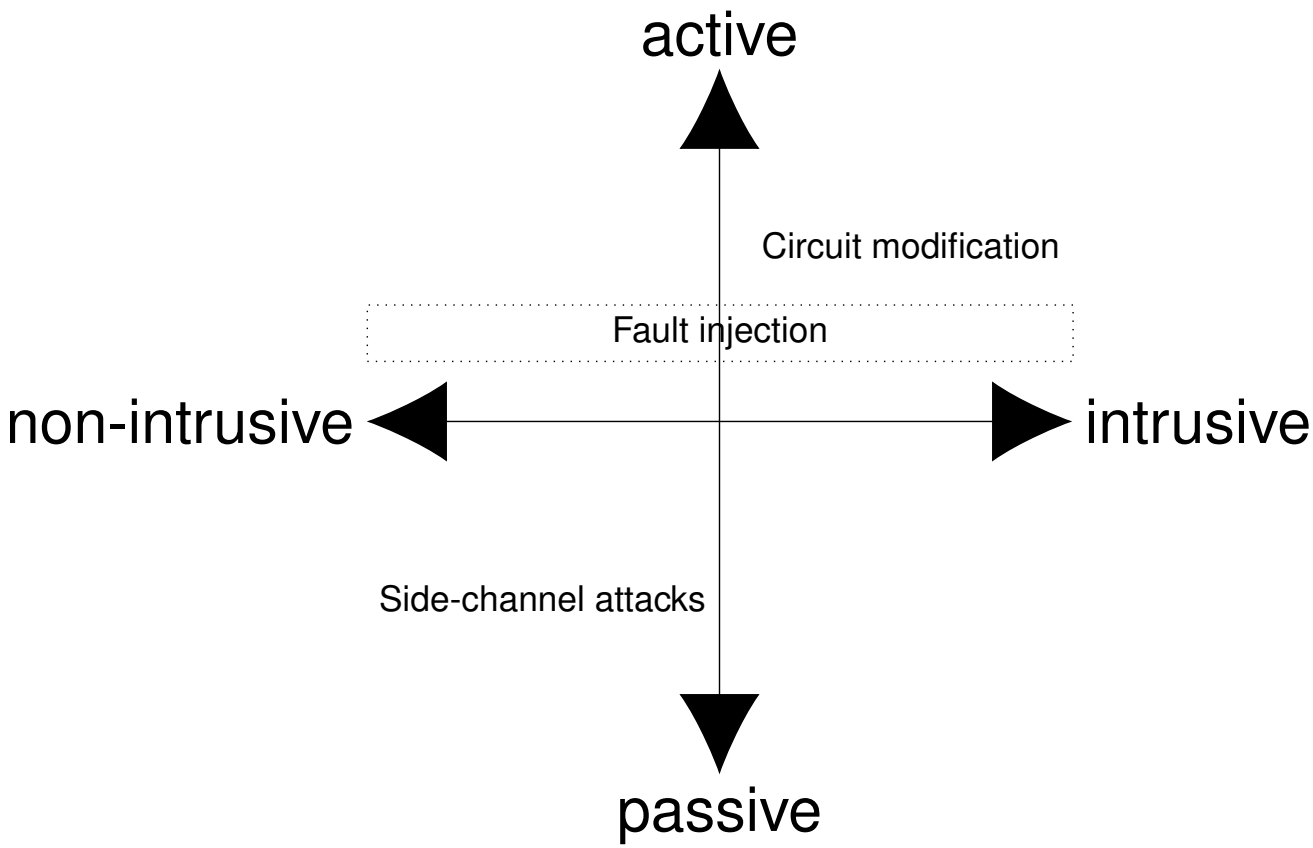
- Prendre un chalumeau ;
- Ouvrir avec le DAB ;
- Injecter sur le bus des requêtes de distribution de billet ;
- ...
- Profit.

Exemples d'attaques

- Injection de messages malveillants sur les bus de communication ;
- Injection de fautes dans un composant au niveau de son alimentation, de son horloge matérielle ou encore par arc électrique ;
- Attaques temporelles sur les caches de processeur ;
- Exploitation des prédicteurs de branche et d'exécution spéculative des processeurs (Spectre, Meltdown) ;
- Exploitation du phénomène de perturbation en lecture des mémoires RAM (Rowhammer) ;
- Attaques par écoute de la consommation énergétique ou des émanations électromagnétiques d'un composant.

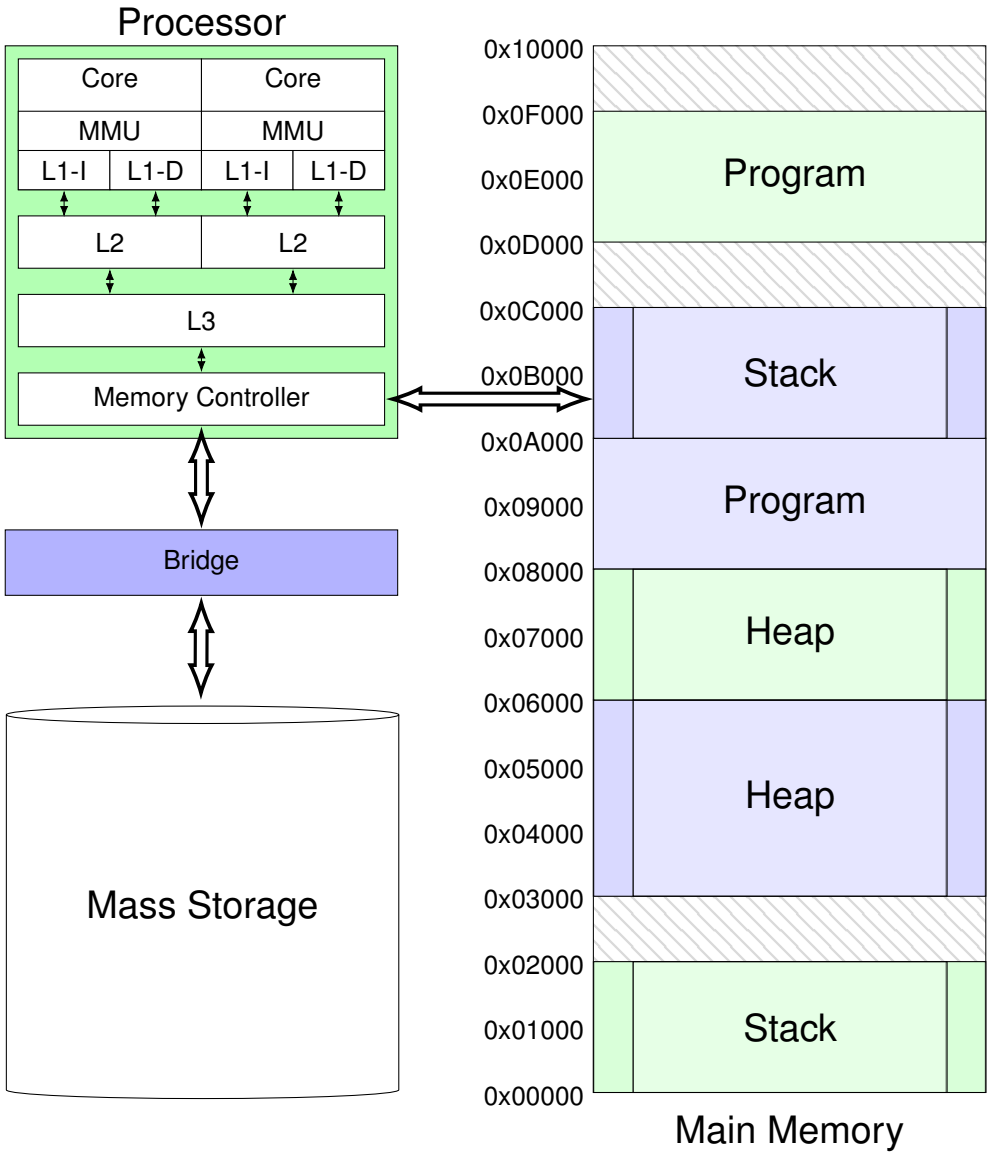


Classes d'attaque



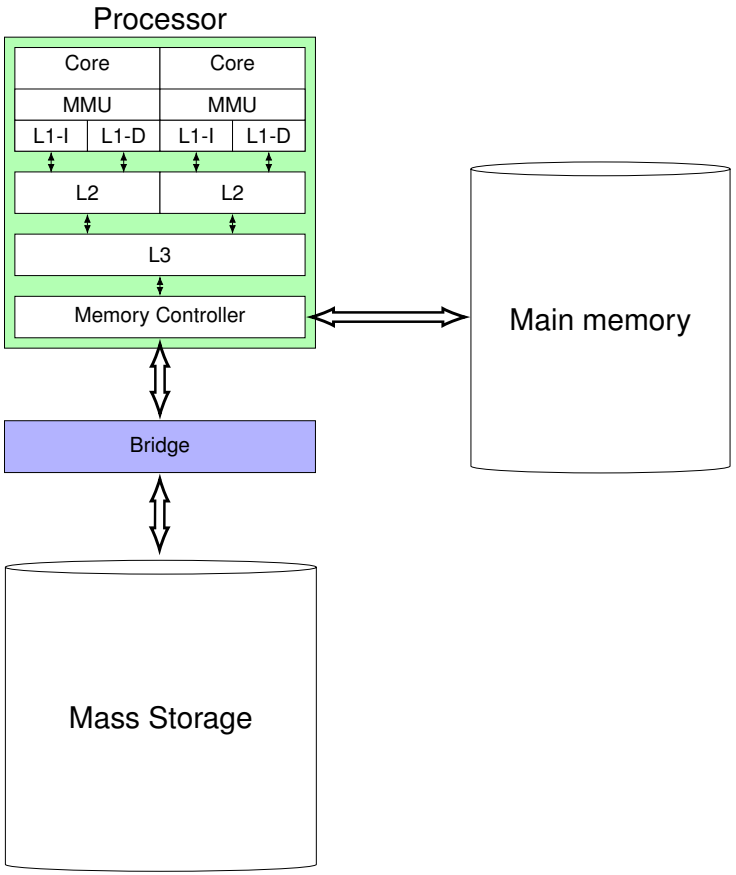
- Passive: Observer et déduire.
- Active: Perturber et conclure.
- Intrusive: Ouvrir le boîtier, ce brancher au composant.
- Non-Intrusive: Pas de contact.

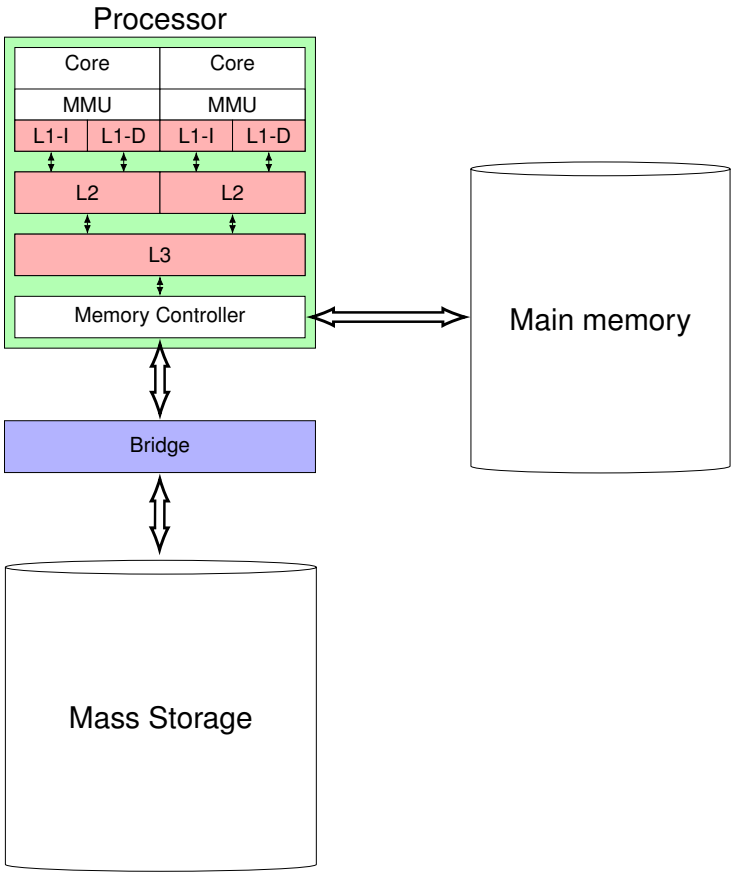
Cas partculier des attaques sur processeur



- Mémoire virtualisée → Isolation logique entre deux processus ;
- Mais pas d'isolation physique, les ressources matérielles sont partagés.

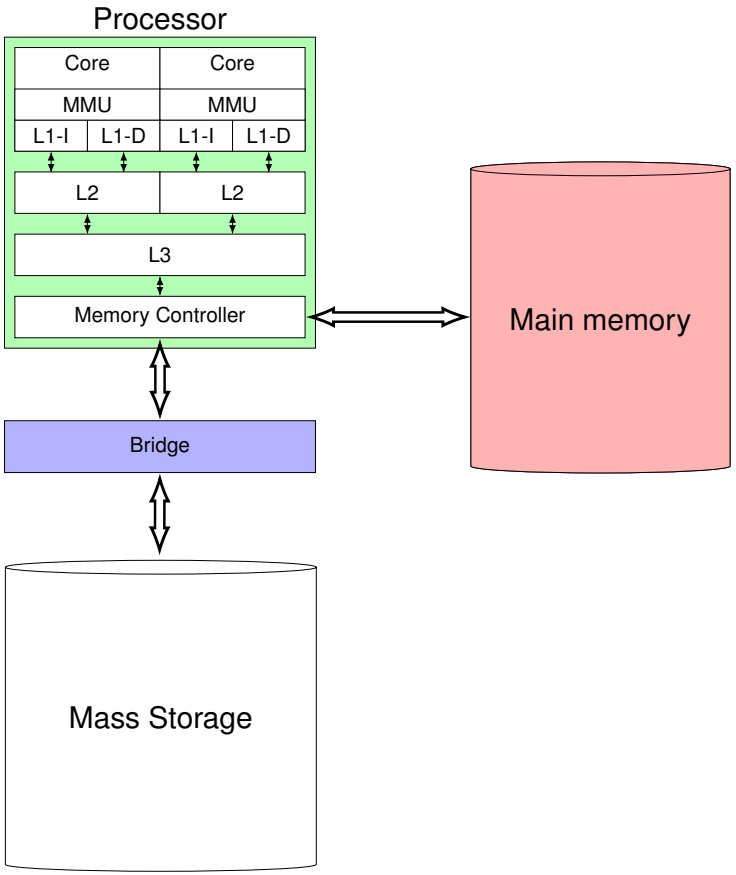
Attack space





Caches

Small SRAM-based memories storing data from main memory locally to speed-up access times.



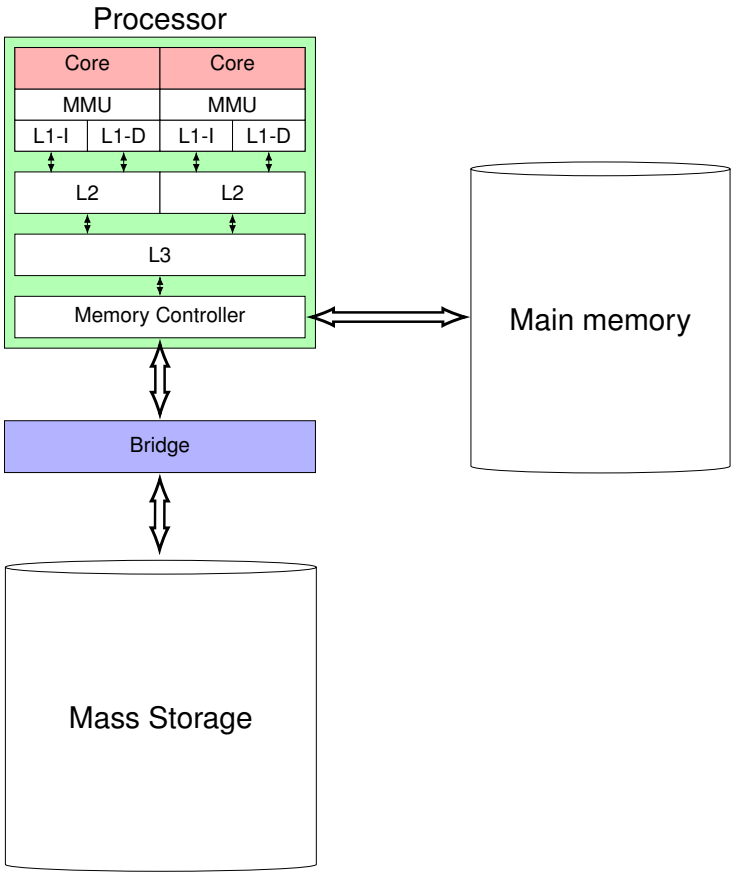
Caches

Small SRAM-based memories storing data from main memory locally to speed-up access times.

Main memory

Large DRAM-based memory storing data under used by running programs.

Attack space



Caches

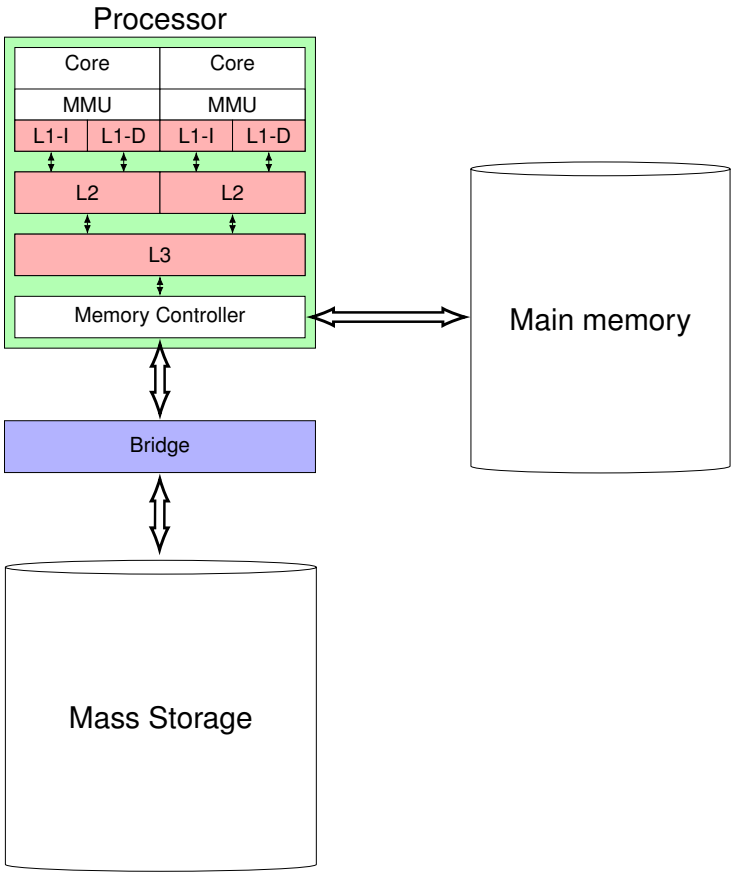
Small SRAM-based memories storing data from main memory locally to speed-up access times.

Main memory

Large DRAM-based memory storing data under used by running programs.

Internal core

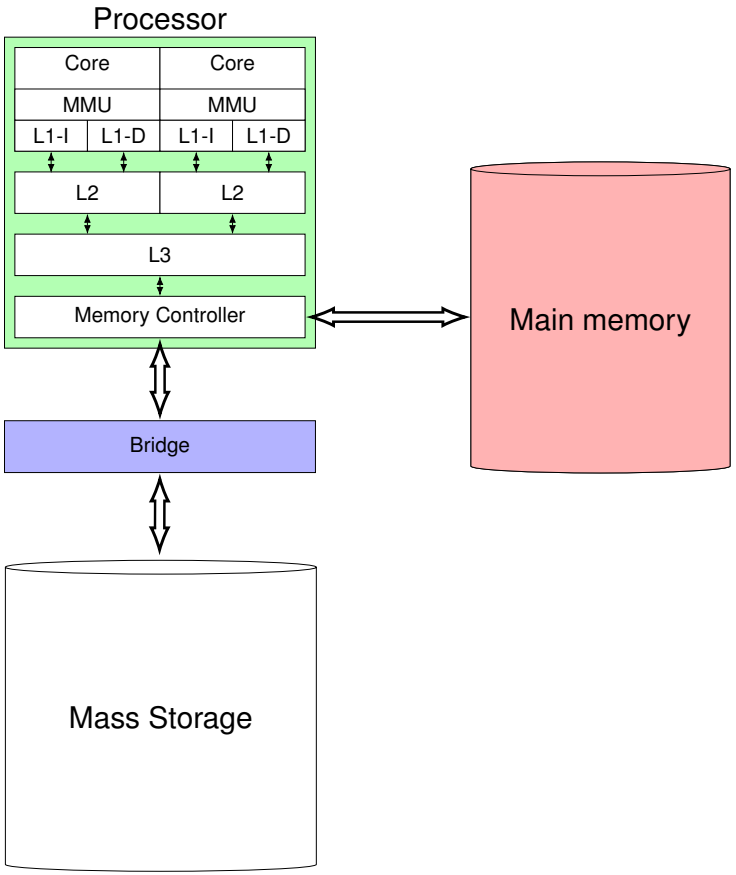
Large DRAM-based memory storing data under used by running programs.



Example of cache exploitation

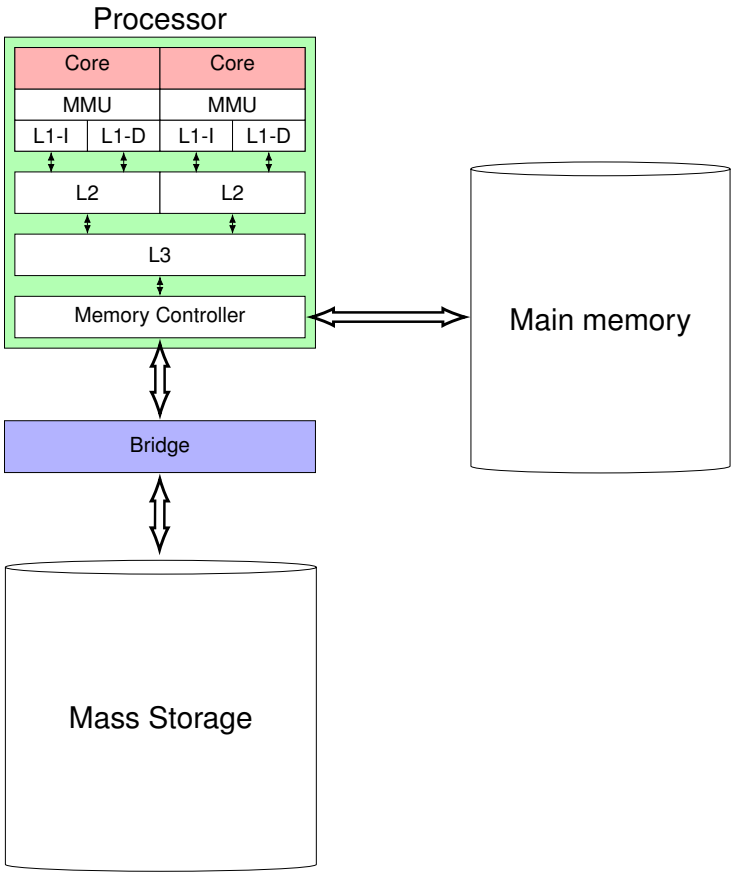
Abusing the time of accessing data through a cache when data is present or not:

- In 2015, this method has been used to spy the use of web browser using javascript [OREN15].
- The same year, AES secret keys of ARM TrustZone has also been extracted [BRM15].
- In 2016, RSA secret keys stored into amazon EC2 has also been extracted [INCI16].



Example of Main Memory exploitation

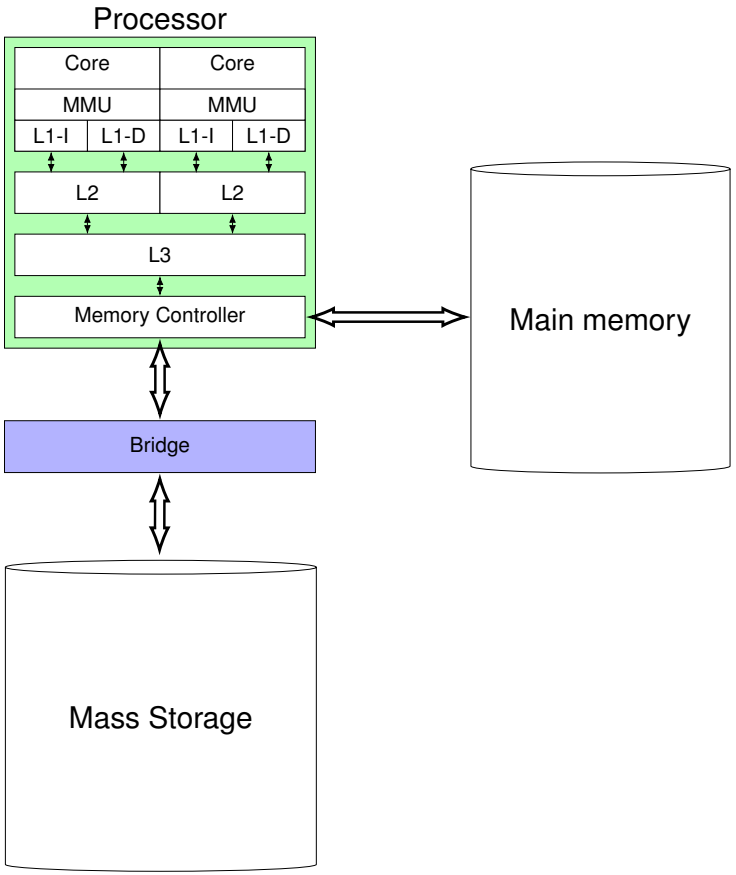
Exploitation of the “disturbance phenomenon” in DRAM when accessing adjacent lines, altering the content of memory cells (Rowhammer) [KDKFL+15].



Processor's pipeline exploitation

Exploitation of out-of-order execution and branch prediction to break memory isolation (Spectre/Meltdown) [LSGPH18].

Attack space



Challenge

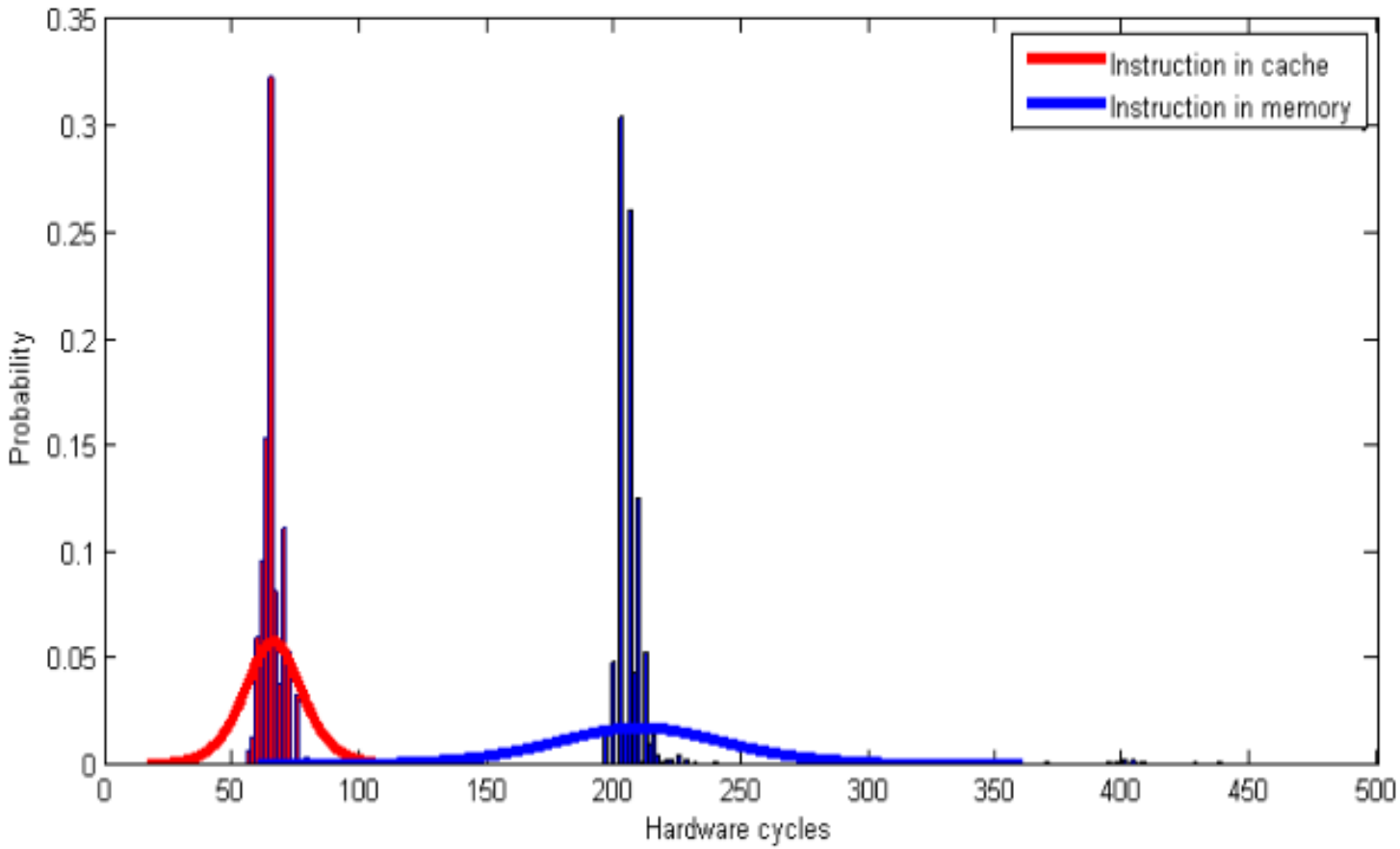
All these attacks are hard to mitigate because they target micro-architecture, i.e. invisible from the programmer's perspective (ISA). In addition, mitigations should also have :

- Small overhead.
- Reliable.
- Updatable (in case of new or variant attacks are discovered).

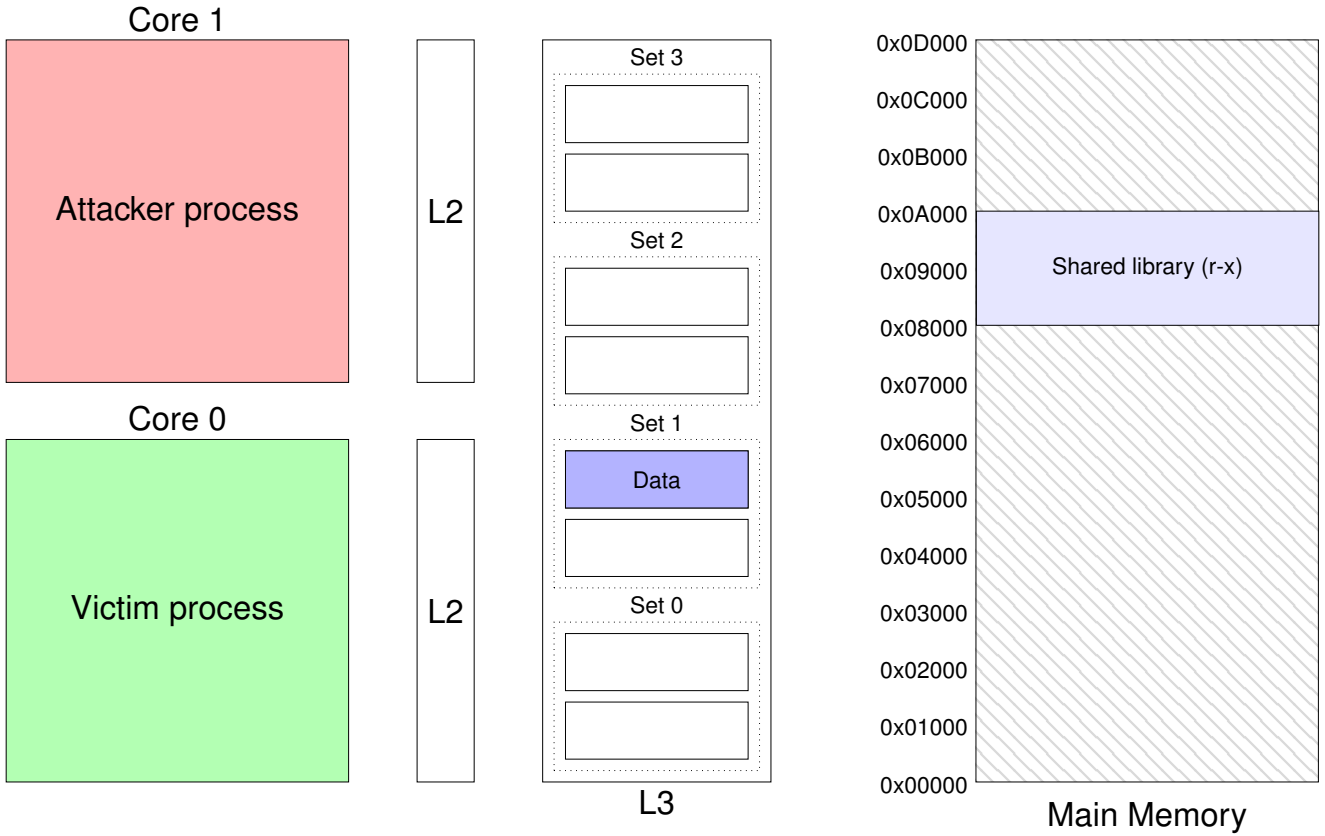
Exploitation des caches de processeur

Principe

Exploitation de différences de temps d'accès au cache lors d'un cache hit et un cache miss. Cette information est modifiée par l'activité des autres processus, qui peuvent venir remplir le cache avec leurs propres données.

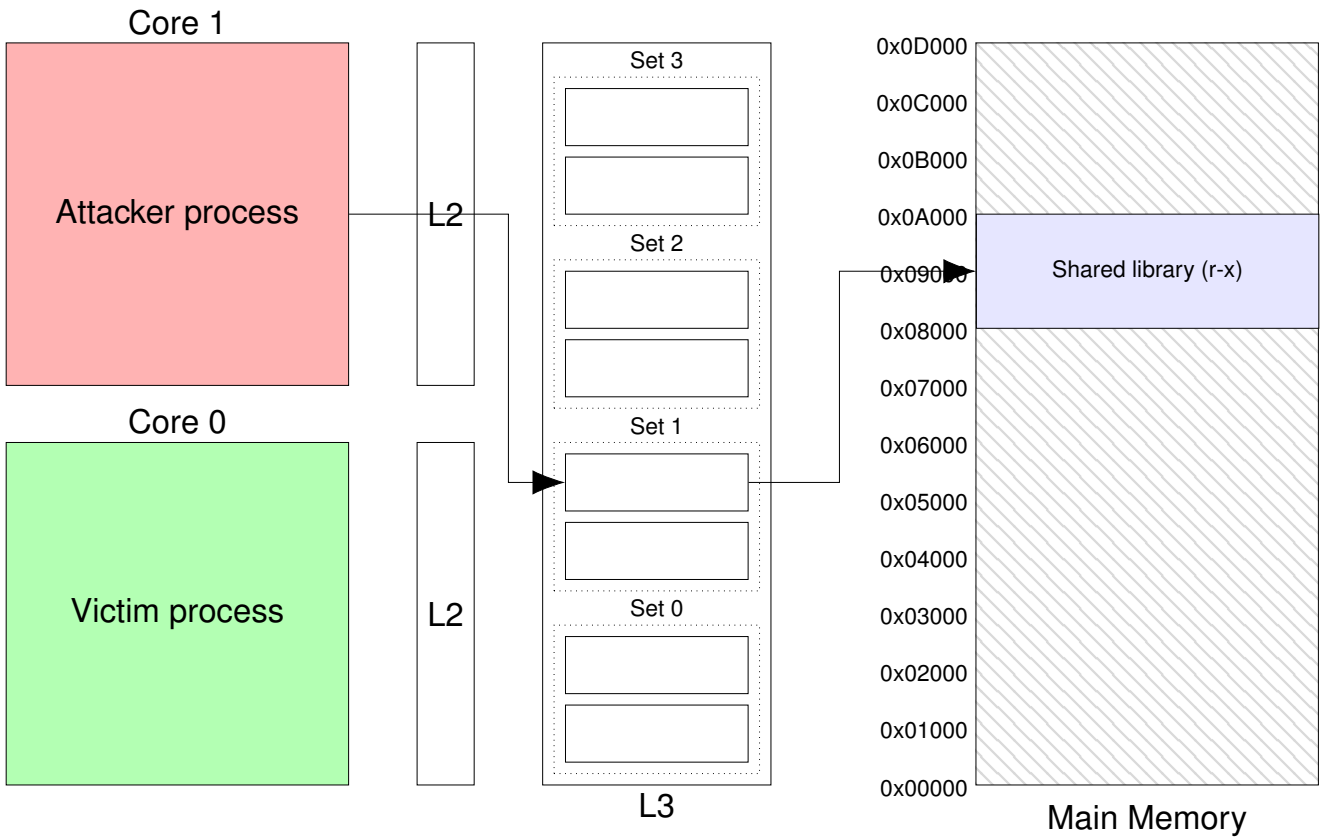


Cache Timing Attacks assumptions

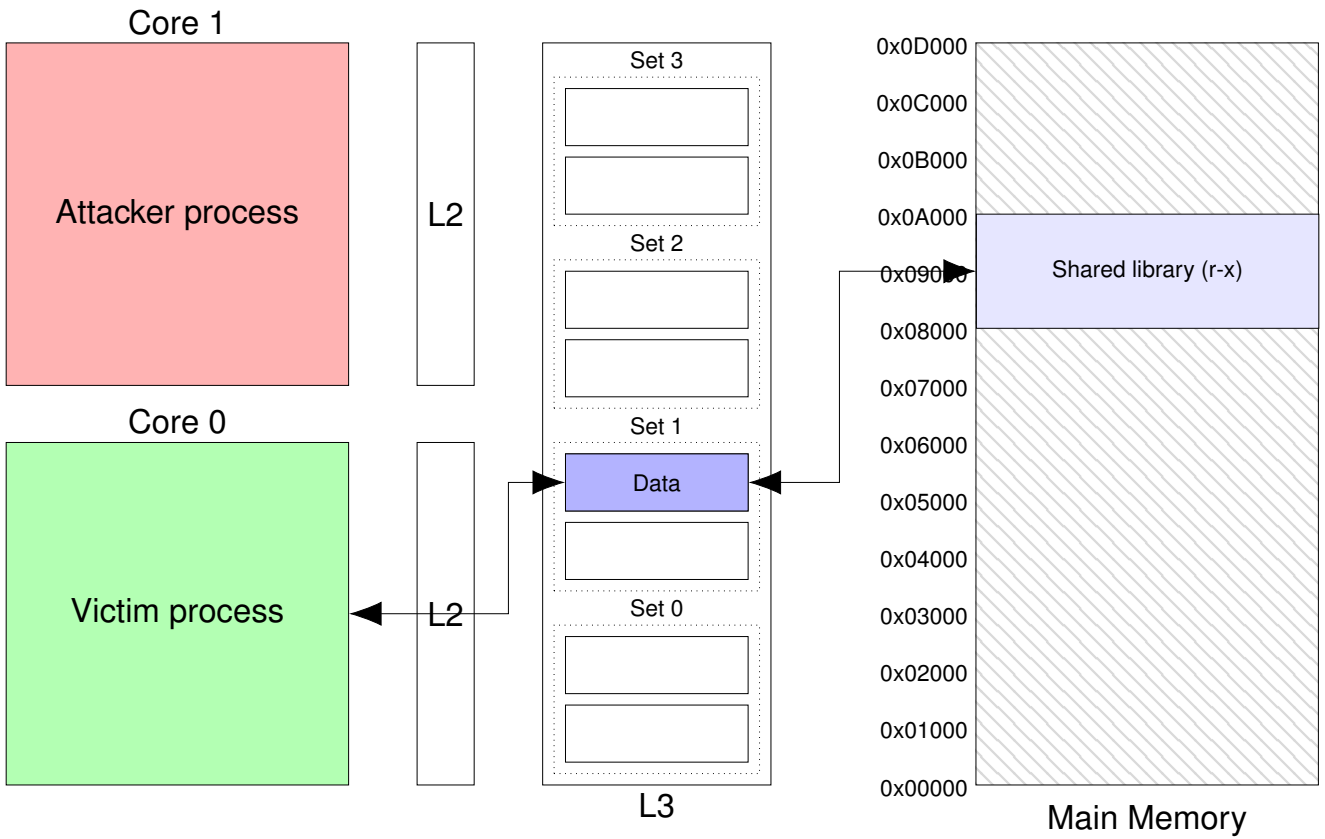


Cache timing attacks aim at infering information on a process under execution by exploiting access times to the caches. It requires an access to a sufficiently precise time measure. Two main class of attacks exist: 1) Flush + Reload; 2) Prime and probe.

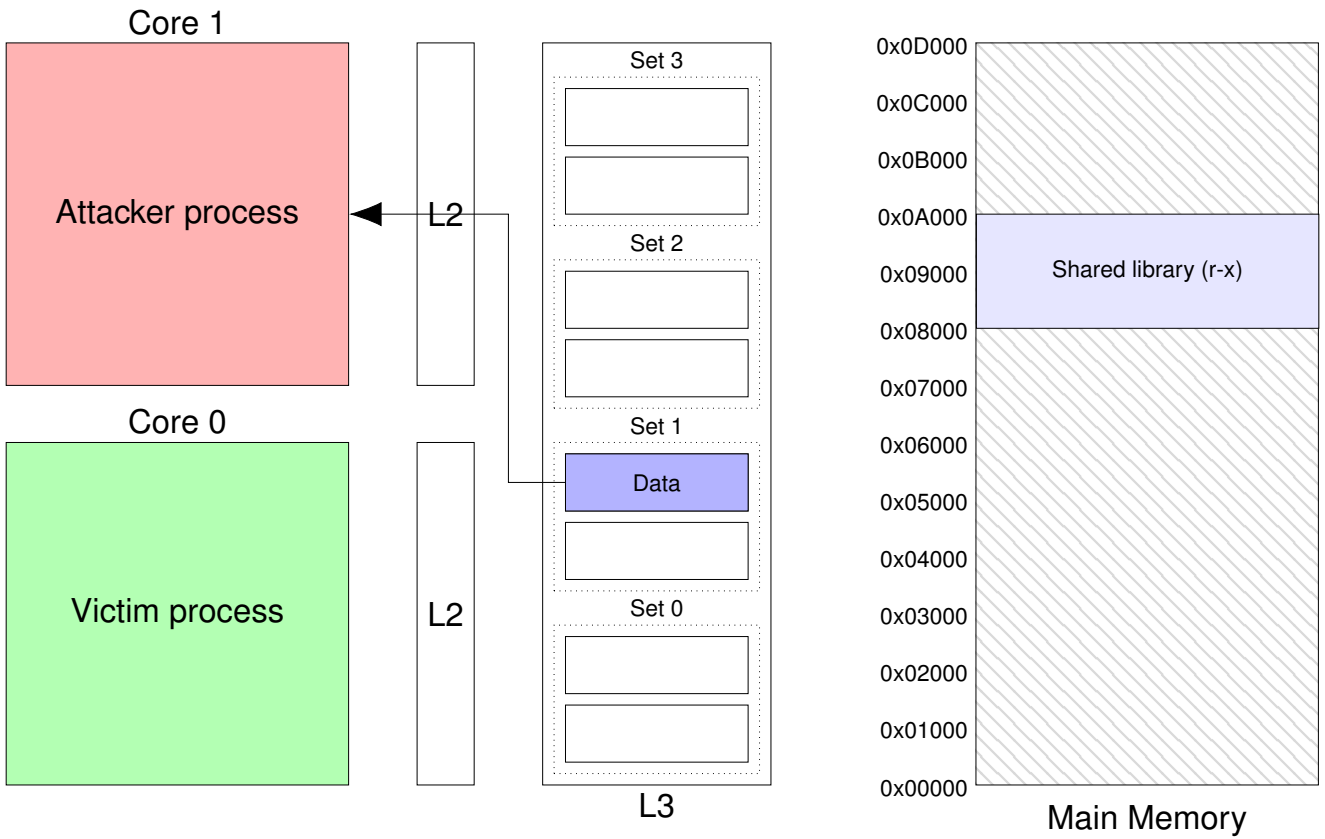
Flush + Reload



Flush + Reload requires a shared library between two processes. The attacker flushes the cache line (i.e. removing it from the cache) where a data from the shared library is located.

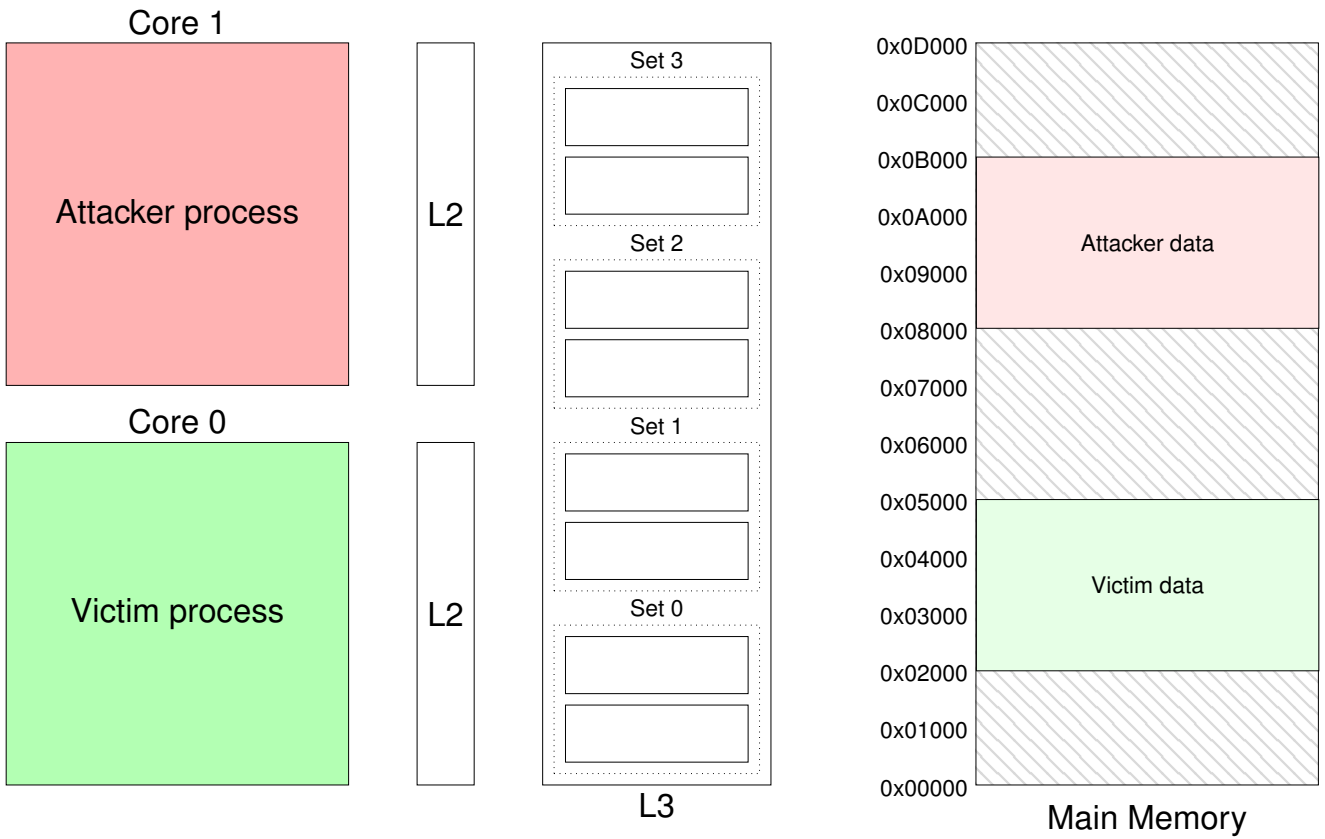


In parallel, the victim process may access during a short time frame to this data, bringing it back to the cache.

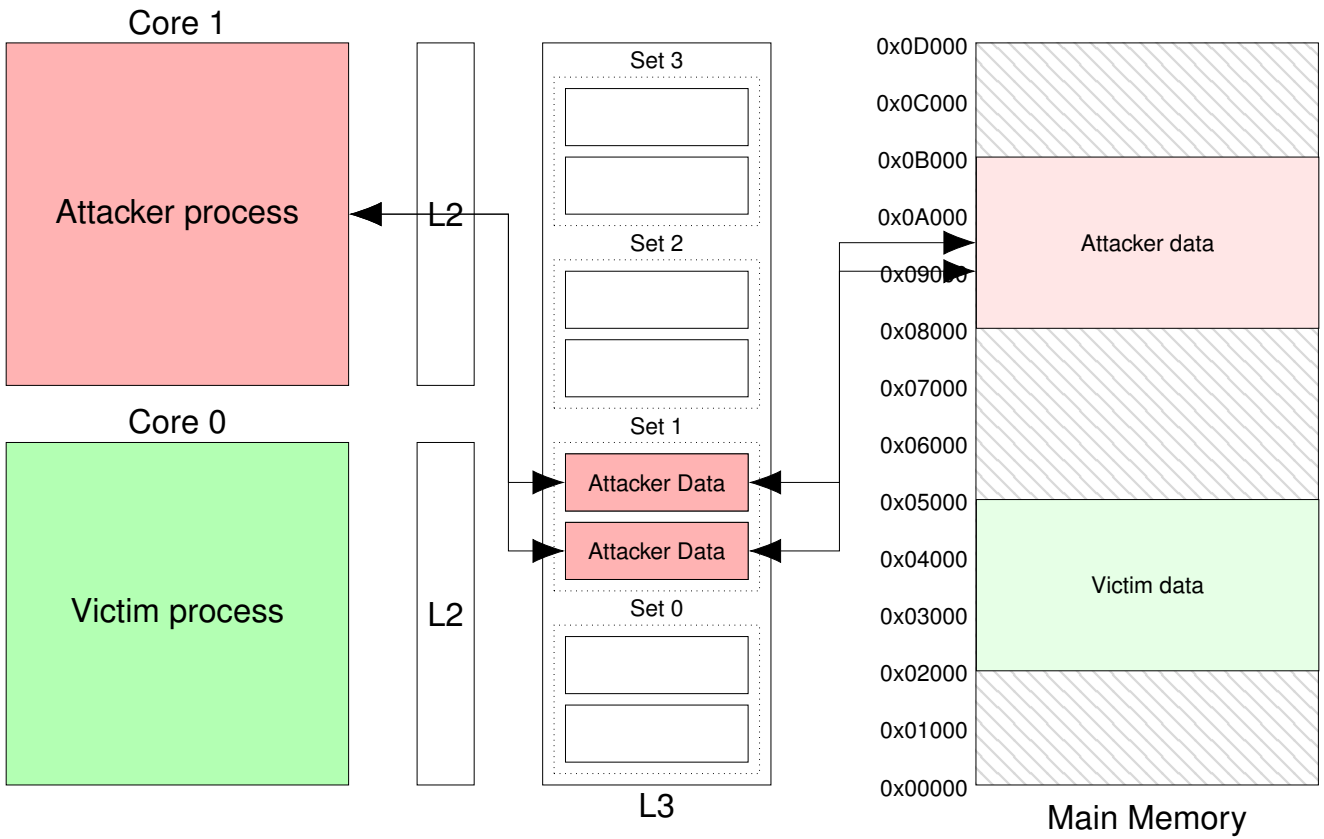


Then the attacker capture the time to access this data. If the time is a few processor's cycles, it is a cache hit and it can deduce that the victim process has accessed to the data.

Prime and Probe

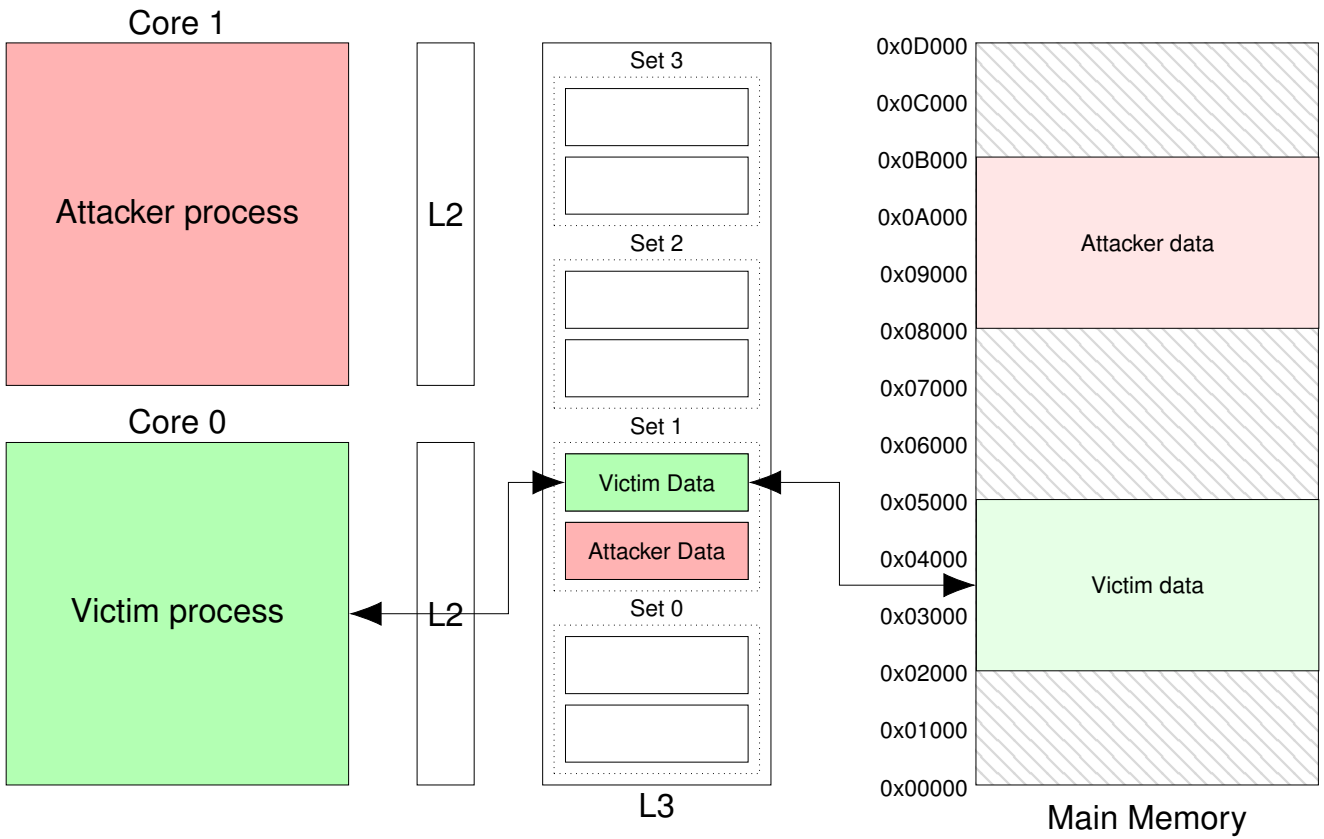


Prime and Probe does not require any asumption.

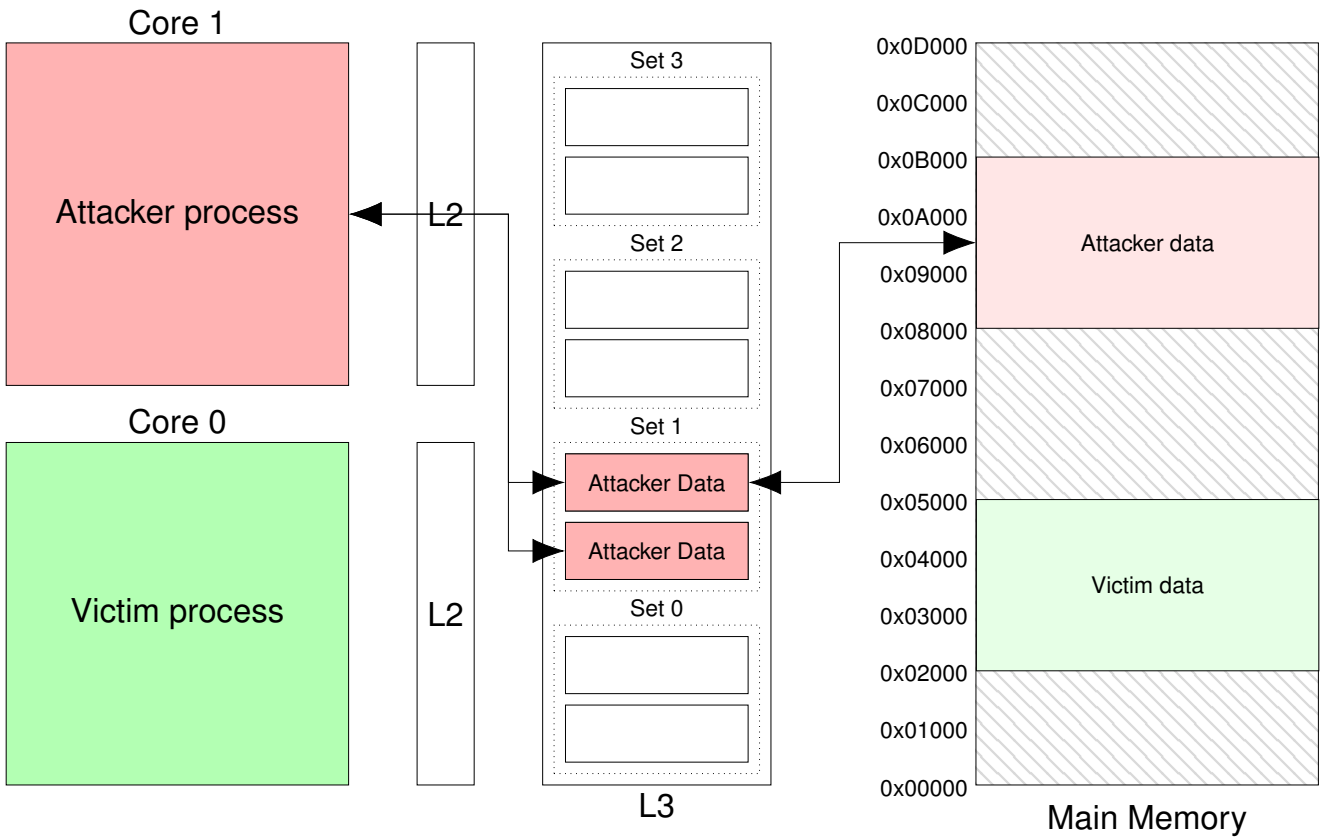


The attacker primes the cache (i.e. fill an entire set with its own data).

Prime and Probe



In parallel, the victim process may access during a short time frame to its data, removing attacker data from the cache.



Then the attacker capture the time to access all the lines in the set. If the time is a few processor's cycles, it is a cache hit and it can deduce that the victim process has accessed its data.

Sample code for timing cache attack

Minimal code to perform a timing cache attack:

```
code:
clflush (X)      // Flush cache for address X
mfence           // Wait execution of pending memory accesses
lfence           // Wait execution of pending instructions
rdtscp           // Read performance counter to EAX
mov %eax, (T1)   // Store performance counter to T1
mov (X), %eax    // Read from address X
rdtscp           // Read performance counter to EAX
mov %eax, (T2)   // Store performance counter to T2
jmp code
```

Example: Cache Attack to break RSA

RSA is a famous encryption algorithm that can be used to send securely a message m. The decryption is critical in the algorithm with a secret exponent d:

$$\text{RSA-ENCRYPT}(m, e, N) = m^e \mod N = c$$

$$\text{RSA-DECRYPT}(c, d, N) = c^d \mod N = m$$

Example: Cache Attack to break RSA

```
void RSA_DECRYPT(int* m, int c, int d, int N)
{
        int i = 0;
        *m = c;
        for(i = 0; i < d; ++i) {
                *m = (*m) * c % N;
        }
        return c;
}
```

By using Prime and Probe on the address of the instruction inside the loop, the attacker is able to count the number if time the loop has been called, and extract the exponent d.

Mitigation 1: Leakage Free Code Design

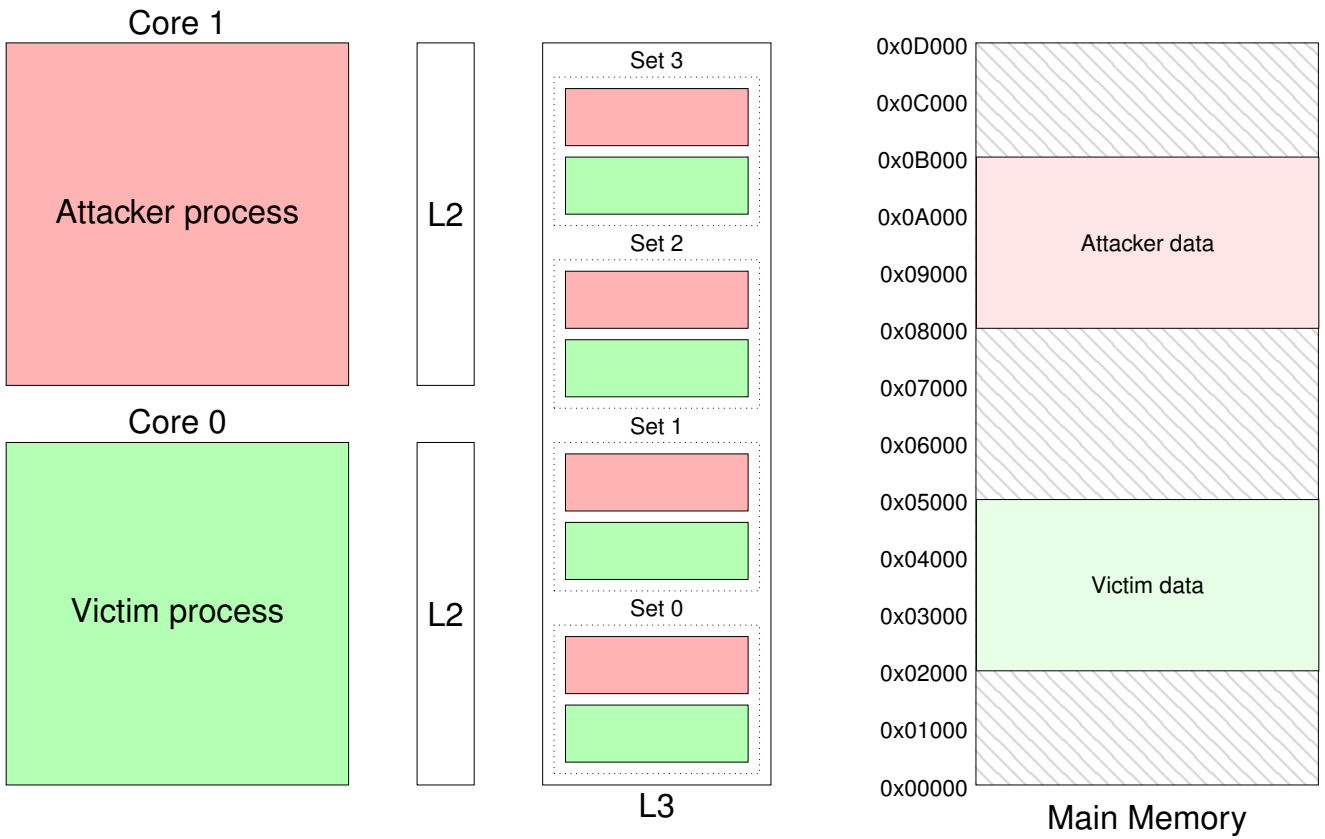
Making secret indepndent from, the execution flow and the memory accesses.

```
// D = [d0,d1,d2,d3, ...,d_k-1] i.e. Binary digits of d
void RSA_DECRYPT(int* m, int c, int* D, int N)
{
        int i = 0; j = 0;
        int C[2]; C[0] = c; C[1] = 1;

        for(i = 0; i < k; ++i) {
                for(j = 0; j < i+1; ++i) {
        *m = (*m)*((C[0]*D[i])+(C[1]*not(D[i])))% N;
        }
        return c;
}
```

Warning: This is just a concept and not a leakage free design.

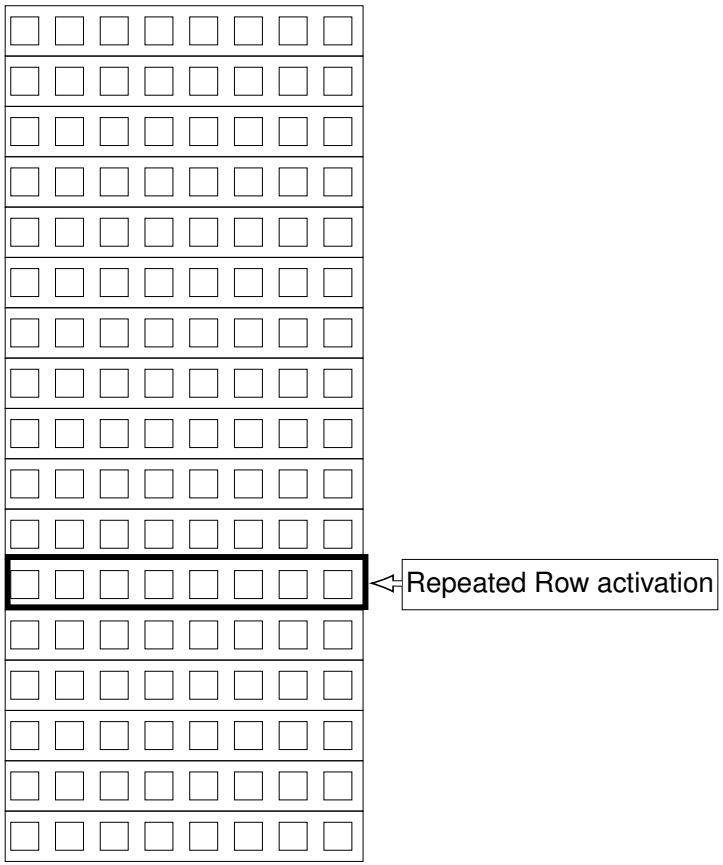
Mitigation 2: Page coloring



Avoiding Collisions in the LLC by creating a sub-set of lines dedicated to specific pages. An attacker cannot access lines reserved for the victim process. A color tag is attached to lines and pages, avoiding pages to be filled into lines with the wrong color. Example: Intel CAT technology.

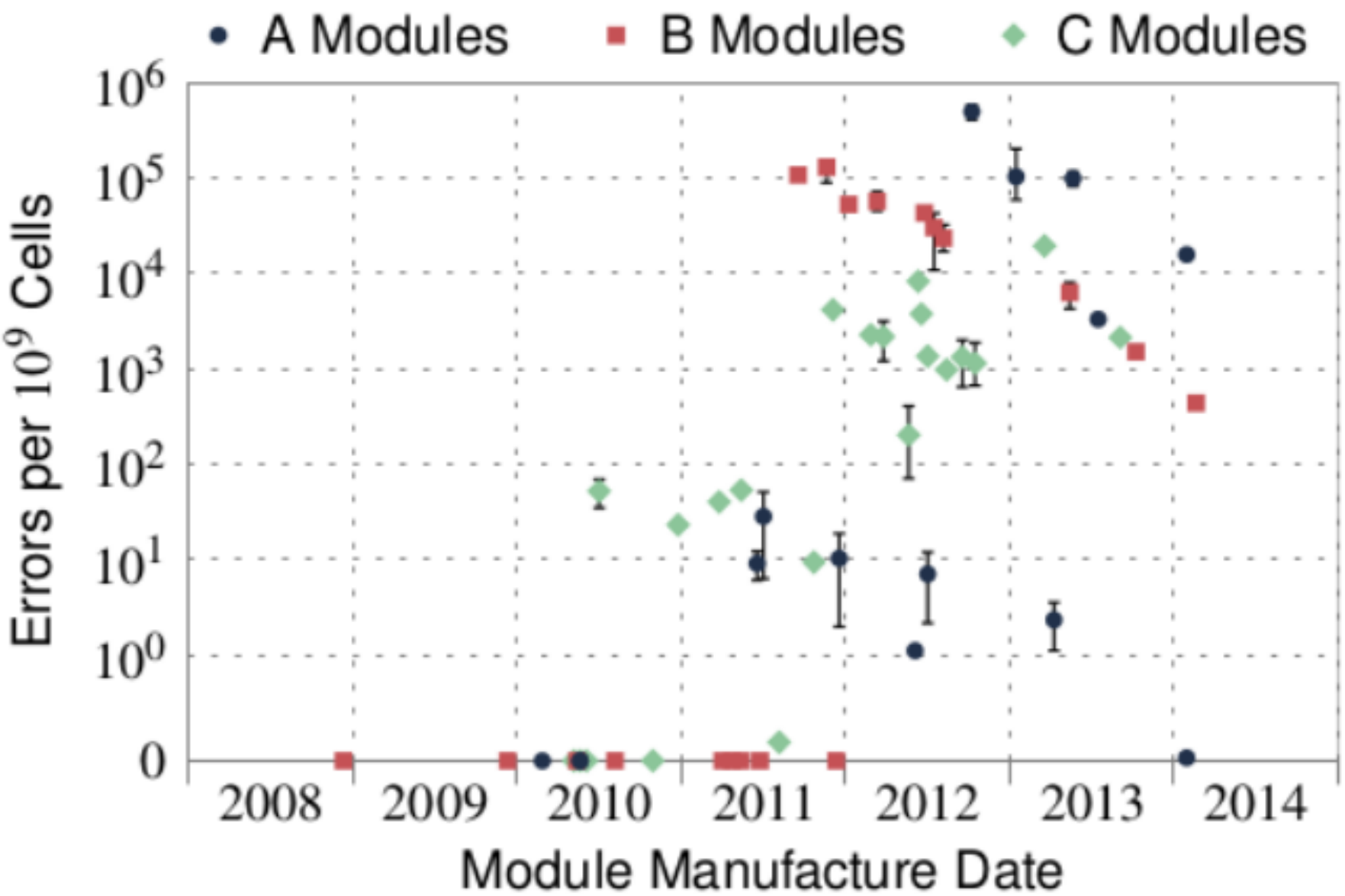
Attaque sur la DRAM : Rowhammer

Other attack: Rowhammer



- → RowHammer is an attack targetting DRAMs (i.e. main memory);
- → When a memory line is activeted repeatedly, the voltage wordline fluctuate generating a “disturbance error”;
- → This error is responsible of possible bit-flips on adjacent rows;
- → It targets recent memories since memory cells are more and more close each other (2010 onwards);

Other attack: Rowhammer



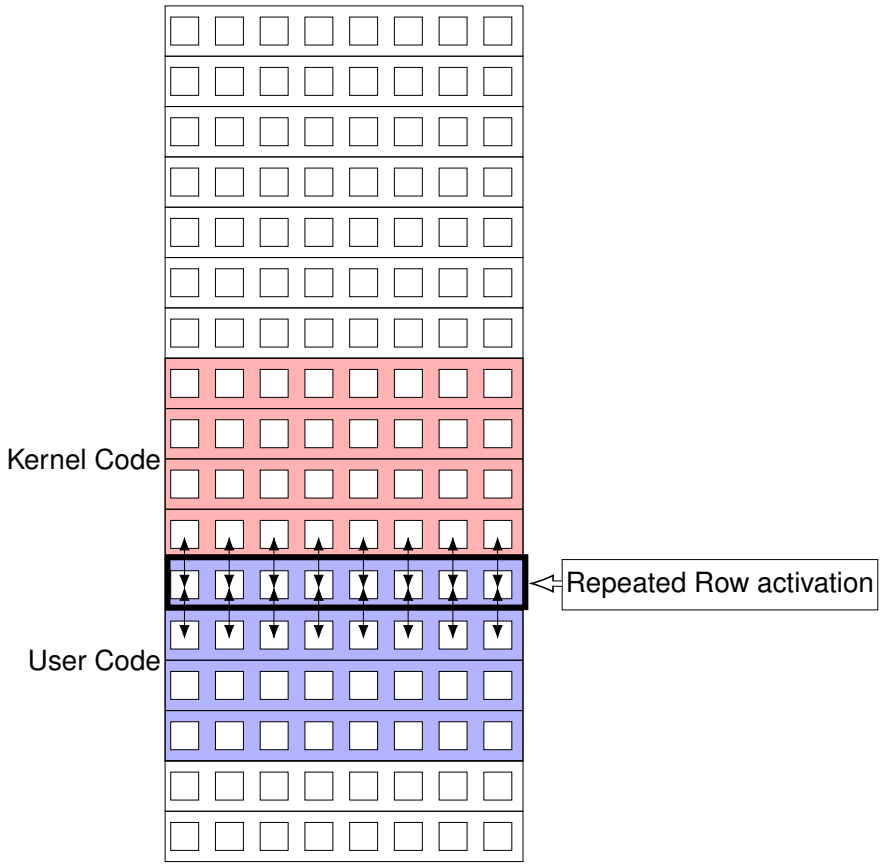
Sample code for RowHammer

Minimal code to trigger RowHammer bug:

```
code:
mov (X), %eax   // Read from address X
mov (Y), %ebx   // Read from address Y
clflush (X)     // Flush cache for address X
clflush (Y)     // Flush cache for address Y
jmp code
```

- → The clflush flushes the cache line to bypass the cache.
- → Some rows are more vulnerable than others (exhaustive search);
- → Two different addresses are required to maximize the disturbance error (known as Row-conflict address pair). Since DRAM are split into banks, the addresses X and Y must be in the same bank.

Rowhammer to gain Kernel privilege



→ In 2015, RowHammer bug has been used to gain Kernel Privileges by the modification of the process page table [SB2015][a].

[a] M. Seaborn and T. Dullien, “Exploiting the DRAM RowHammer Bug to GainKernel Privileges”, BlackHat, 2016.

Vulnérabilité des prédicteurs de branche et de l'exécution spéculative

Branch Prediction and Speculative Execution

Modern processors use branch prediction and speculative execution to maximise performance. Speculative execution occurs when execution reaches a conditional branch instruction whose direction depends on preceding instructions not completed yet. The processor will *speculatively* execute the instruction inside one branch, and commit (i.e., save) if its turns out to be correct. Otherwise, it abandons the work it performed speculatively by reverting its register state and resuming along the correct path.

```
if( x < array_size) {
        out = array[x];
}
```

In this example, if array_size is not available at the time of condition is evaluated (for exemple because it is not present in the cache), the speculative execution will execute (out = array[x]) *speculatively* before checking if *x* is in the right bounds.

Branch prediction is one step further of speculative execution. The processor make a guess on the branch that is more likely to be executed, and then perform speculative execution on it.

```
if( x < array_size) {
        out = array[x];
}
```

- → If *x* is usually in the right bounds, processor will execute *speculatively* the memory access (out = array[x];)
- → If *x* is usually not in the right bounds, the processor will skip the execution of (out = array[x];).

Spectre is an attack exploiting Branch Prediction, speculative execution and cache timing attack to read arbitrary memory address on a victim process.

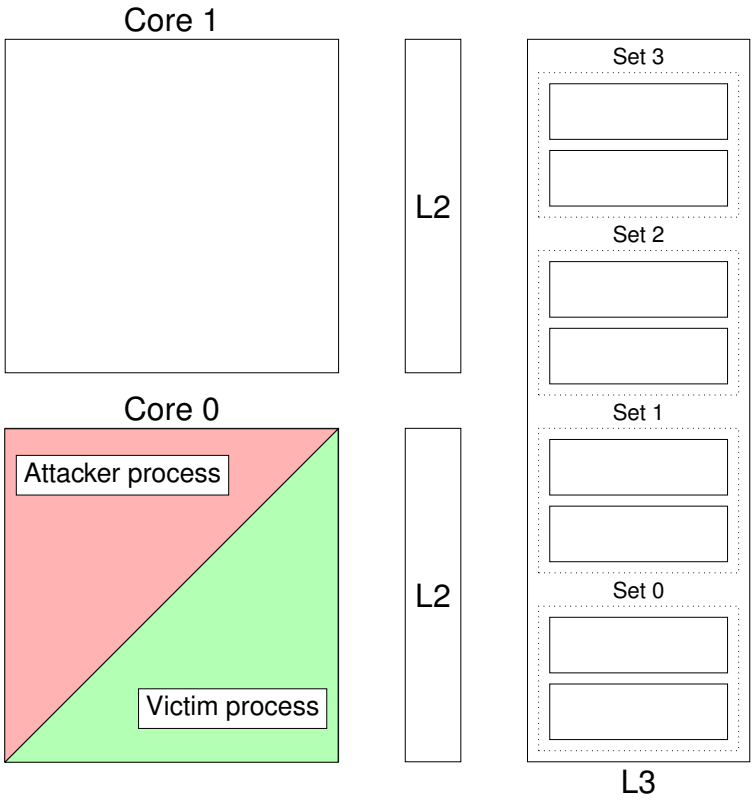
It targets a slightly different code than discussed in the previous slides:

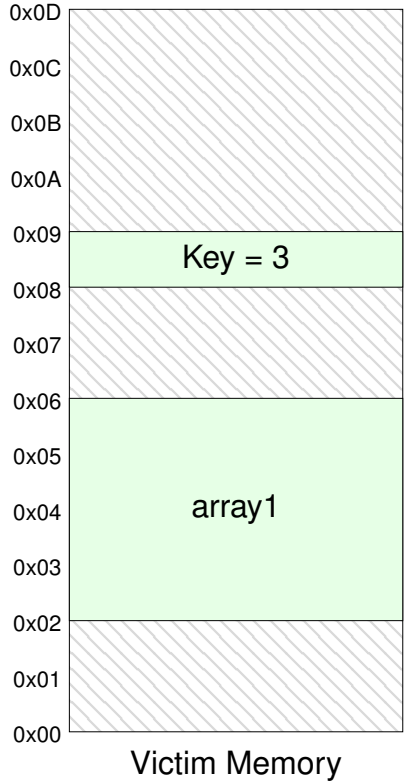
```
if( x < array1_size) {
        out = array2[array1[x] * 32];
}
```

Attacker Model

- → The Attacker can control the value of x (typically by a communication process $\Leftrightarrow$ Kernel or process $\Leftrightarrow$ process);
- → The Attacker knows the addresses of array1_size and array1 (but not the content);
- → Attacker and victim are in the same processor's core.

Remark: The factor 32 refers to a cache line size in bytes. Since Attacker knows array1 address and controls x, it is just a variable substitution.

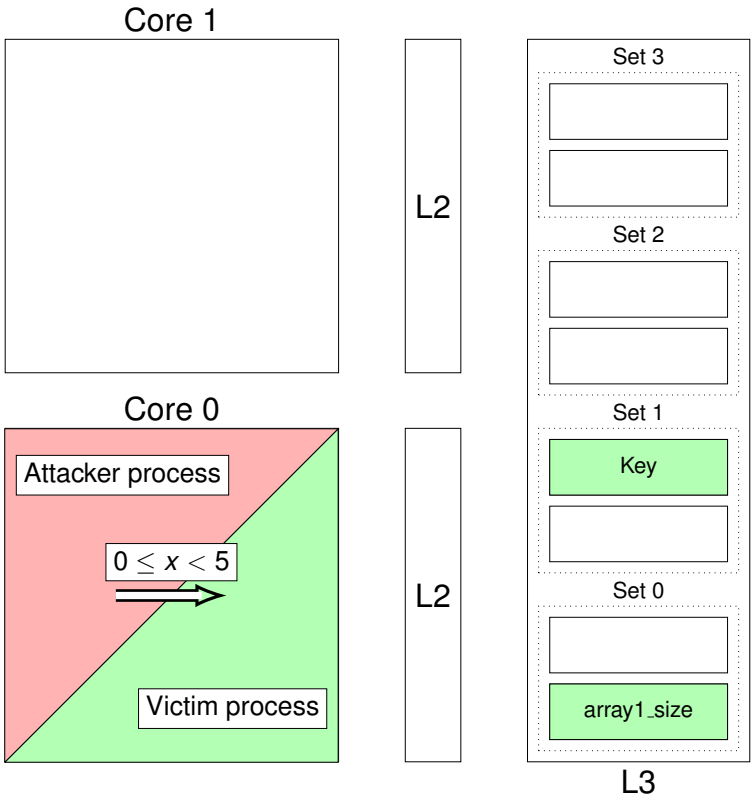


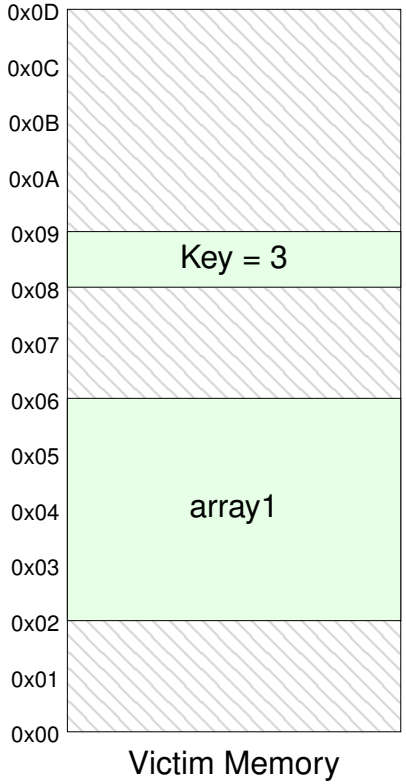



```
if( x < array1_size) {
out = array2[array1[x] * 32];
}
```

Introduction

The attacker wants to get the value of the secret key at address 0x08 (outside of array1 bounds).

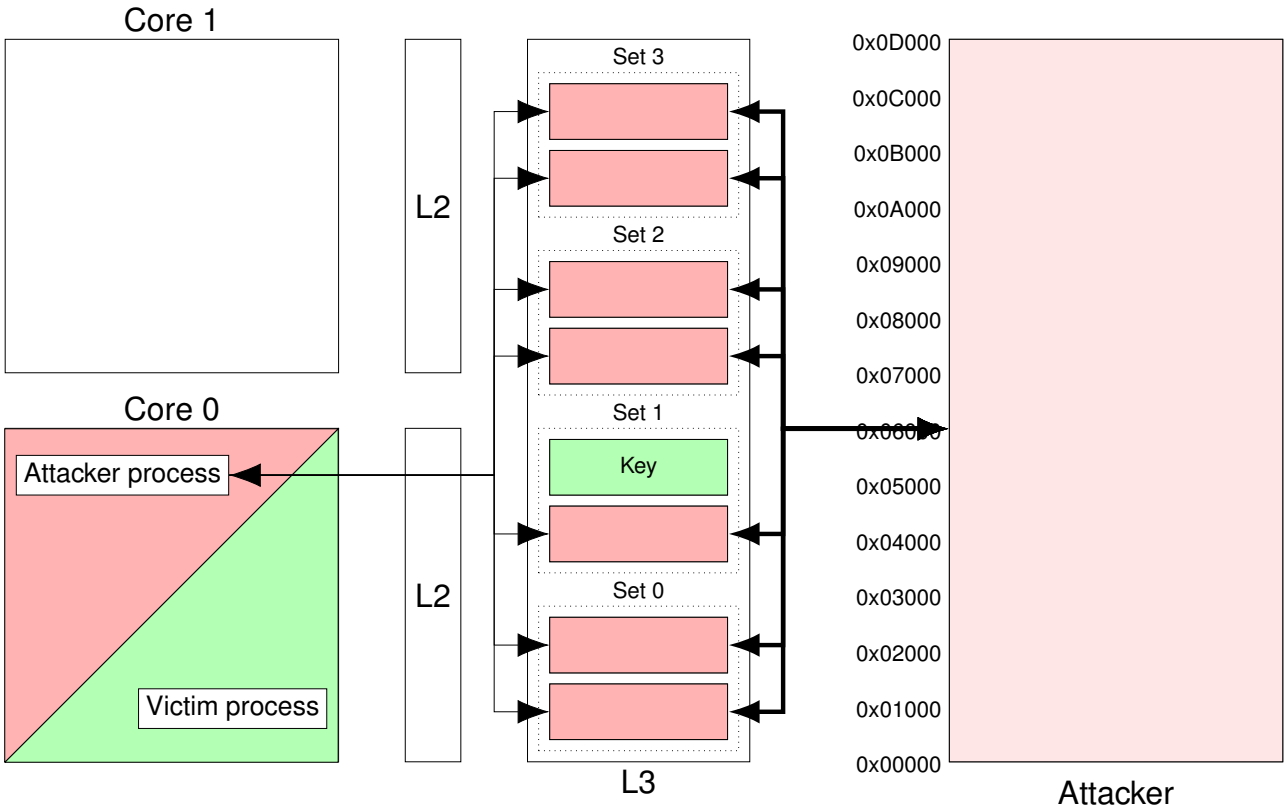





```
if( x < array1_size) {
out = array2[array1[x] * 32];
}
```

First phase: Branch Prediction Training

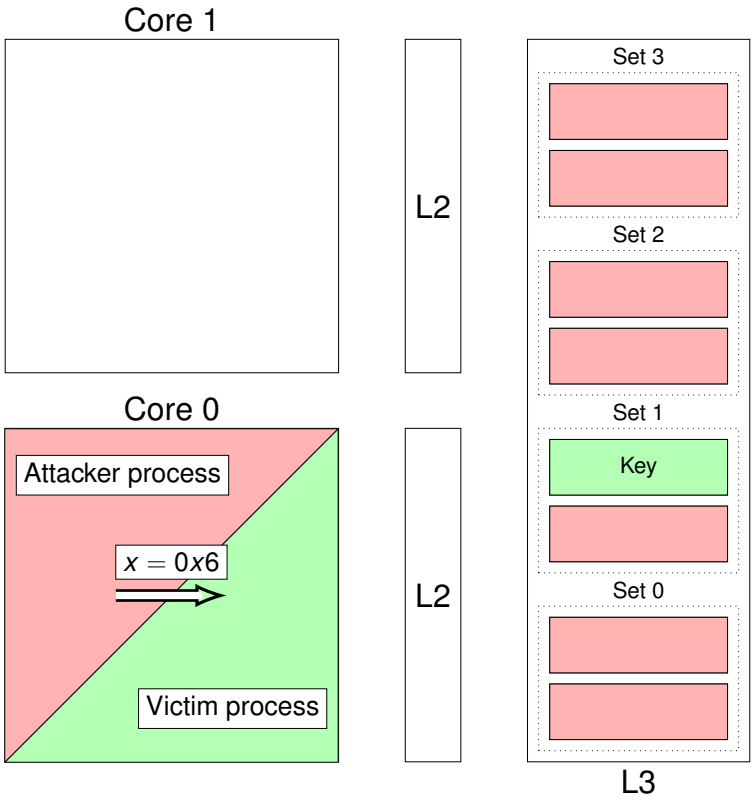
The attacker will train the branch prediction by repeatedly call Victim process to execute the code with $0 \leq x < 5$. He will also call a function that use the *key* to have it in the cache.

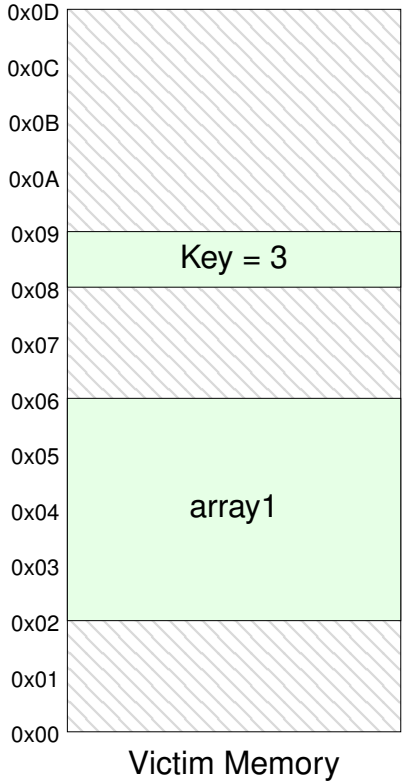



```
if( x < array1_size) {
out = array2[array1[x] * 32];
}
```

Second phase: Attacker Prime

The attacker will prime each sets with its own data except the *key*. array1_size is no longer in the cache.

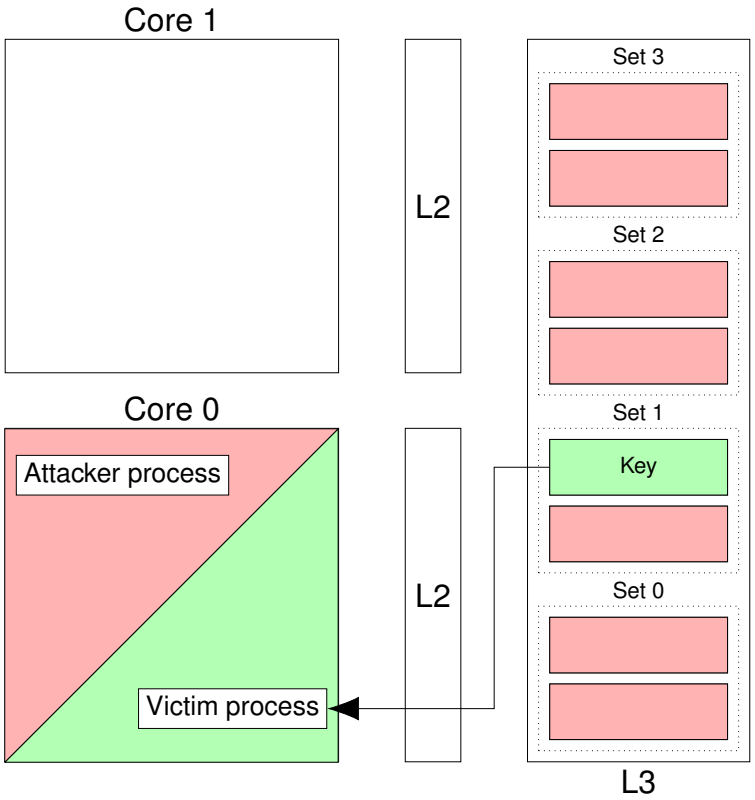


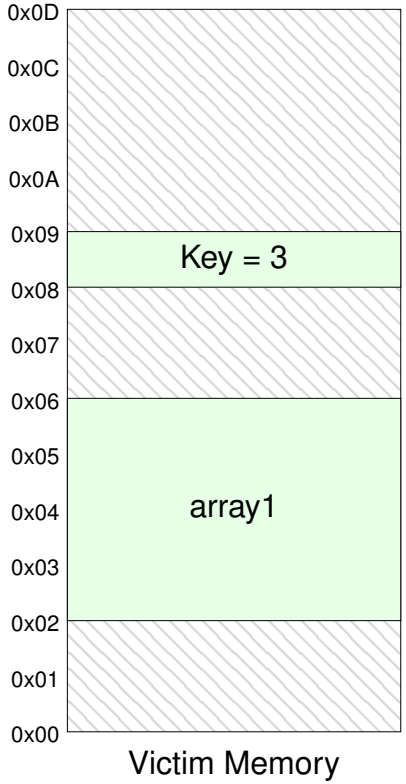



```
if( x < array1_size) {
out = array2[array1[x] * 32];
}
```

Third phase: Speculative execution and branch misprediction [1]

The attacker will now call the Victim with $x = 0x6$, which is outside of the array1 bounds. Since array1_size is no longer in the cache, the processor can't evaluate if the bound check is valid and will consider to execute (out = array2[array1[x] * 32]) speculatively since branch predictor has been trained with valid bound checks.

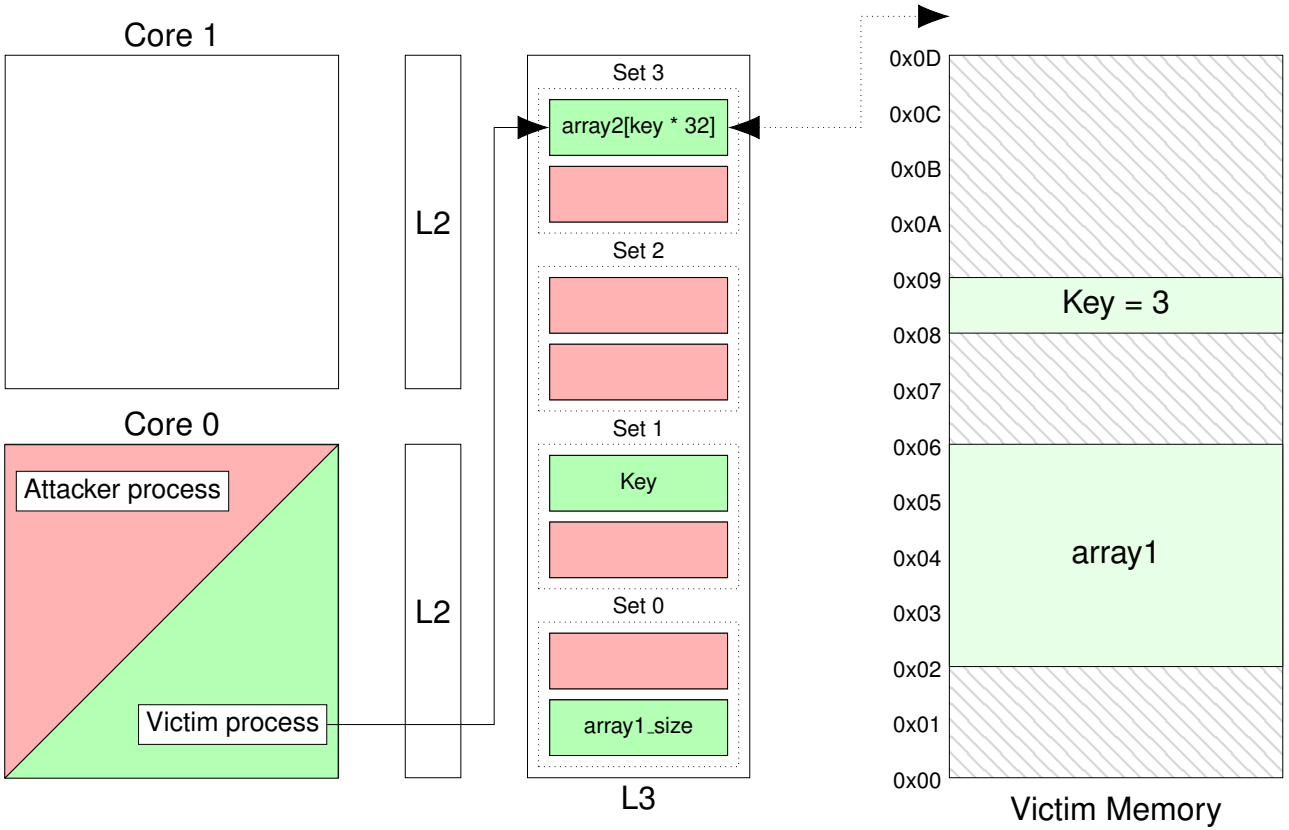





```
if( x < array1_size) {
out = array2[array1[x] * 32];
}
```

Third phase: Speculative execution and branch misprediction [2]

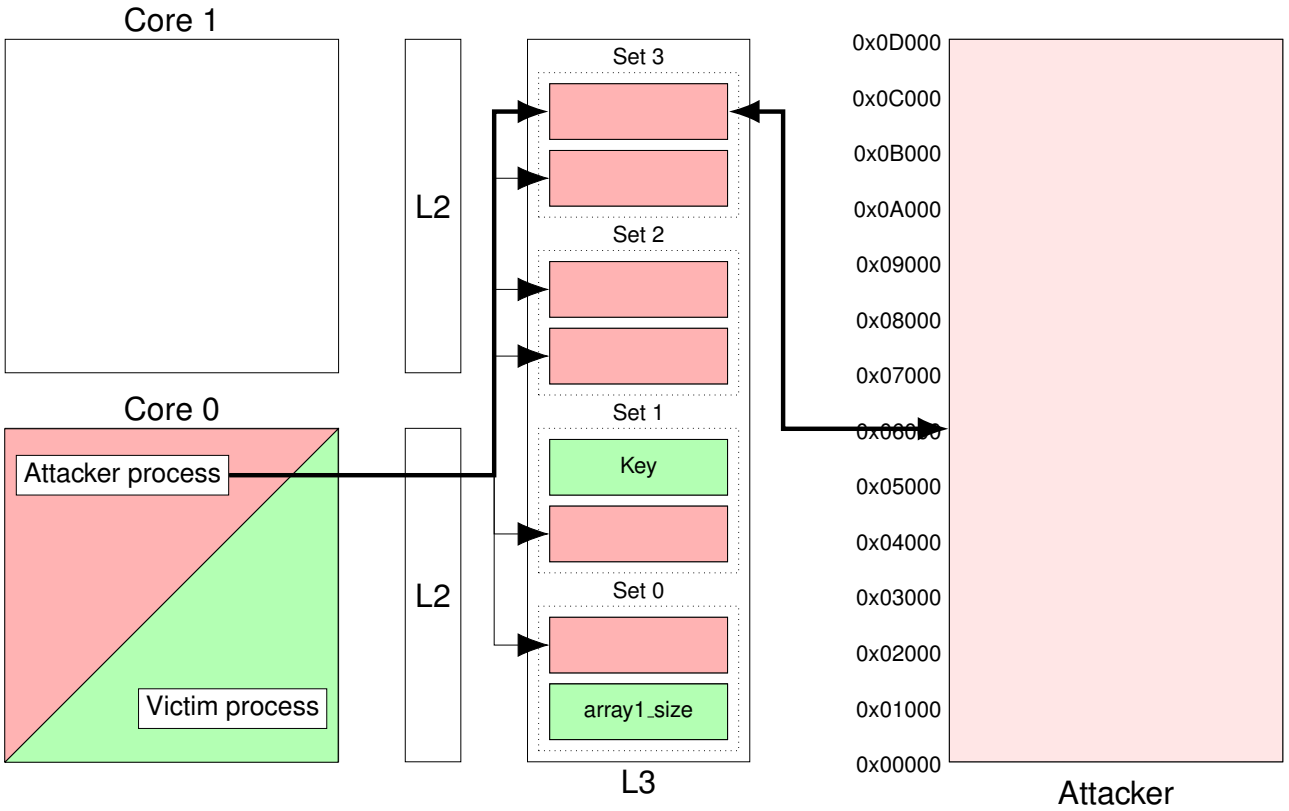
key is in the cache so processor can quickly evaluate the address of array2[*key**32] = array2[3 * 32].




```
if( x < array1_size) {
out = array2[array1[x] * 32];
}
```

Third phase: Speculative execution and branch misprediction [3]

The processor will load array2[key * 32] into the cache. In parallel, array1_size has been load into the cache, the processor knows it has misspredicted and revert the register state. However, array2[key * 32] remains in the cache.




```
if( x < array1_size) {
out = array2[array1[x] * 32];
}
```

Fourth phase: Attacker Probe

Now attacker just has to probe the cache. The cache miss appears in set 3, thus the Atacker nows the key was 3.

Out-of-Order Execution

Out-of-order execution is an optimization technique that allows maximizing the utilization of all execution units of a CPU core. Instead of processing instructions strictly in the sequential order, CPU executes them as soon as all required resources are available. While the execution unit of the current operation is occupied, other execution units can run ahead. Hence, instructions can be run in parallel as long as their results follow the architectural definition.

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

In this example, the last instruction should never be reached since an exception is raised beforehand. However, it is executed out-of-order since there is no data dependencies with raise_exception().

Exploiting Out-of-Order Execution: Meltdown

Meltdown use Out-of-Order Execution to read any address on the Kernel space from user space (as a reminder, Kernel is mapped into the user address space, but not accessible).

Child Process:

```
data = read(kernel_adress)
read(probe_array[data * 4096]);
// Child crashes due to the read into illegal address
```

Parent Process:

```
for(int i = 0; i < 256; ++i) clfush(probe_array[i * 4096])
run_child_process();
for(int i = 0; i < 256; ++i) eval_cache_hit(probe_array[i
// A cache hit occurs when i = data
```

Child process will try to access kernel at address kernel_address, then a memory access is performed into the array probe_array[data * 4096]. The process crashes and register state is reverted, but not the cache line accesses.
The the parent process just has to perform a cache attack on the prob_array addresses to recover data.

Meldown countermeasures exist. KAISER is a kernel patch that only map to the user address space part of the kernel really usefull for syscalls. Coupled with address virtualisation on the Kernel, Meltdown cannot be applied as is.