Petite explication des failles Meltdown et Spectre

Frédéric Pétrot, Laboratoire TIMA
tima.imag.fr/sls/people/petrot
frederic.petrot@univ-grenoble-alpes.fr

Grenoble INP ENSIMAG

Préambule

Je ne connais rien à la sécurité, et mon unique objectif est d'expliquer le comportement du matériel qui mène à ces résultats.
Je ne suis en particulier pas capable d'expliquer comment on peut exploiter les failles présentées dans des systèmes informatiques opérationnels

- Mesure précise du temps sur `x86_64`
- Hiérarchie mémoire et invalidation sur `x86_64`
- Exécution spéculative
  source de l'accélération de l'exécution séquentielle depuis 20 ans
  ne jetons pas le bébé avec l'eau du bain !
- Meltdown et Spectre
  attaques utilisant ces mécanismes dévoilées le 3 janvier 2018

Mesure du temps

Instruction assembleur `rdtscp` :

> *Reads the current value of the processor's time-stamp counter (a 64-bit MSR) into the EDX :EAX registers and ... The processor monotonically increments the time-stamp counter MSR every clock cycle and resets it to 0 whenever the processor is reset.* [1]

```
1  #include <stdint.h>
2  #include <x86intrin.h>
3  register uint64_t cycle;
4  uint32_t x;
5  /*
6   * lit le cycle courant
7   */
8  cycle = __rdtscp(&x);
```

1. Source : Intel® 64 and IA-32 Architectures Software Developer's Manual Vol. 2, 4-547.

Tournons le programme `time.c` !

```
1  #include <stdio.h>
2  #include <stdint.h>
3  #include <x86intrin.h>
4  uint64_t cycle, delta;
5  uint32_t x;
6
7  int main(int argc, const char** argv)
8  {
9   cycle = __rdtscp(&x);
10  delta = __rdtscp(&x) - cycle;
11  printf("Delta en cycles : %lld\n", delta);
12  return 0;
13 }
```

```
petrot@tilleul[master|+1]% ./a.out
Delta en cycles : 58
petrot@tilleul[master|+1]% ./a.out
Delta en cycles : 165
petrot@tilleul[master|+1]% ./a.out
Delta en cycles : 185
petrot@tilleul[master|+1]% ./a.out
Delta en cycles : 165
petrot@tilleul[master|+1]% ./a.out
Delta en cycles : 183
petrot@tilleul[master|+1]% ./a.out
Delta en cycles : 194
```

Quelques variations, mais on parle ici de mesurer $\approx$ 170 cycles à plus de 3GHz!

Hiérarchie mémoire et invalidation

Intel i7-4770 (Haswell), 3.4 GHz (Turbo Boost off), 22 nm. RAM : 32 GB (PC3-12800 cl11 cr2) [2]

- L1 Data cache = 32 KB, 64 B/line, 8-WAY.
- L1 Instruction cache = 32 KB, 64 B/line, 8-WAY.
- L2 cache = 256 KB, 64 B/line, 8-WAY
- L3 cache = 8 MB, 64 B/line
- L1 Data Cache Latency = 4 cycles for simple access via pointer
- L2 Cache Latency = 12 cycles
- L3 Cache Latency = 36 cycles (3.4 GHz i7-4770)
- RAM Latency = 36 cycles + 57 ns (3.4 GHz i7-4770)

2. Source : `http://www.7-cpu.com/cpu/Haswell.html`

Instruction assembleur `clflush` :

> *Invalidates from every level of the cache hierarchy in the cache coherence domain the cache line that contains the linear address specified with the memory operand.* [3]

```
1  #include <x86intrin.h>
2  uint32_t x;
3  /*
4   * Invalide les lignes des caches cachant le contenu de l'adresse de x
5   */
6  _mm_clflush(&x);
```

3. Source : Intel® 64 and IA-32 Architectures Software Developer's Manual Vol. 2, 3-139.

Tournons le programme `inval.c`!

```c
#include <stdio.h>
#include <stdint.h>
#include <x86intrin.h>
uint32_t array[256 * 16]; uint64_t cycle, delta; uint32_t x; volatile uint8_t u;
int main(int argc, const char** argv)
{
   /* Écriture dans array pour éviter le copy-on-write de pages à zéro */
   for (size_t i = 0; i < 256; i++) array[i * 16] = 1;
   for (int i = 0; i < 256; i++)
      _mm_clflush(INVALIDATE ? &array[i * 16] : &array[0]);
   cycle = __rdtscp(&x);
   for (int i = 0; i < 256; i++) u = array[i * 16];
   delta = __rdtscp(&x) - cycle;
   printf("Delta (%s): %lld\n", INVALIDATE ? "inv" : "cached", delta);
   /* Que le compilo n'efface pas tout en optimisant ! */
   return (char)array[random()%(256 * 16)];
}
```

```
petrot@tilleul[master|+1]% gcc -O3 inval.c -DINVALIDATE=0
petrot@tilleul[master|+1]% for i in $(seq 1 8) ; ./a.out
Delta(cached):2812
Delta(cached):7772
Delta(cached):3349
Delta(cached):3323
Delta(cached):3314
Delta(cached):2991
Delta(cached):3047
Delta(cached):3056
petrot@tilleul[master|+1]% gcc -O3 inval.c -DINVALIDATE=1
petrot@tilleul[master|+1]% for i in $(seq 1 8) ; ./a.out
Delta(inv):8752
Delta(inv):8452
Delta(inv):9824
Delta(inv):8108
Delta(inv):10560
Delta(inv):8656
Delta(inv):8444
Delta(inv):8484
```

Information inattendue accessible!

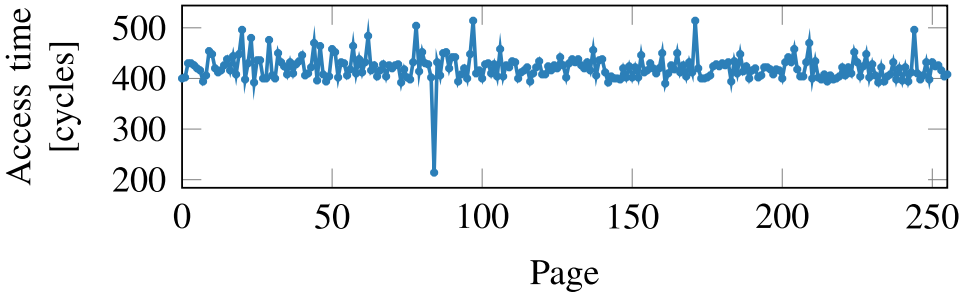
On peut savoir si une ligne de cache a déjà été lue en mesurant le temps qu'il faut pour lire une donnée qu'elle contient

C'est un canal caché qu'un attaquant peut exploiter![a]

a. *Timing (Side-channel) attack.*

Exemple avec un tableau

- invalidation tableau (modulo la taille de la ligne de cache + voies + ...)
- chargement valeur à index donné
- parcours tableau en mesurant le temps
- temps *le plus faible* indique indice *préalablement accédé*
- c'est statistique, on essaye de nombreuses fois



4. Source : papier Meltdown

Exécution spéculative

Exécution d'instructions, possiblement plusieurs centaines, « en avance »
Pas de modifications de l'état architectural de la machine
Modification de l'état micro-architectural : caches, stations de réservation, etc

Pour cacher la latence de certaines opérations

- prédiction de branchement
- prédiction des adresses de retour de fonctions
- prédiction de valeurs (plus rare)
- ...

Processeurs modernes spéculatifs par nature
Exécution O-O-O source de la performance des machines modernes !

Bugs/Feature (?) hardware

Levée d'exceptions tardive lors d'un accès interdit

Attaques !

Meltdown : exploitation de la levée tardive d'exceptions

Course entre accès mémoire et vérification des droits d'accès dans TLB Intel

```
1  ; rcx : adresse noyau
2  ; rbx : tableau espion user pointé par rbx totalement invalidé
3  mov al, byte [rcx]         ; rax chargé avec octet noyau
4  mov rdx, qword [rbx + rax] ; rdx chargé spéculativement @ rbx + rax
```

- exception levée
- gestionnaire d'exception (accessible utilisateur en C à travers `sigaction`s) du programme malfaisant parcours le tableau et mesure les temps pour chaque indice
- temps minimum pour l'indice qui vaut `rax`

Correction par logiciel

Kernel Page Table Isolation

Changement de table de pages lors des appels systèmes
Plus de mapping du kernel dans la TLB $\Rightarrow$ *page fault* immédiate sur accès kernel

Spectre v1 : exploitation de la prediction de branchement [a]

a. https://github.com/Eugnis/spectre-attack

```
1 void victim_function(size_t x)
2 {
3    if (x < array1_size) /* entrainé avec x = 0 */
4       temp &= array2[array1[x] * 512]; /* array2 invalidé au départ */
5 }
```

- `&array1 + x` : adresse du secret (`x` invalide)
- prédicteur de branchement prédit « pris »
- accès à `array1[x]` fait, récupérant `k`, puis accès à `array2[k]`
- outcome de la branche = « non-pris »
- parcours `array2`, temps minimum pour indice = `k`

Ne marche heureusement qu'à l'intérieur d'un même processus (par ex. entre les onglets d'un navigateur, ...)

Correction par logiciel

Navigateurs (interpréteurs javascript)

Plus de tampons partagés entre onglets
Plus d'accès à la mesure du temps précise (granularité 20μs) [a]

a. Source : `https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack/`.

Autres programmes

???

Spectre v2 : exploitation de la prédiction des branches indirectes

Viser une instruction de branchement indirect dans le programme victime dont la cible est en mémoire :

`jmp *%rdx` ou `call *%rdx`

- évincer ligne contenant adresse cible du cache
- processeur exécute quelques centaines de cycles spéculativement en utilisant la prédiction de « base »
- cette dernière a pu être entraînée grâce à un autre processus sur le même processeur, par ex. sur un autre thread matériel
  - faire une copie du processus (copy-on-write après un `fork` par ex.)
  - entraîner un saut indirect dans la copie à ce qu'il pointe à l'endroit du code que l'on veut exécuter dans la victime
    le code cible ne fait rien dans la copie
    marche car prédicteur de branchement partagé par tous les *thread*s matériels

- appel dans la victime ⇒ code à l'adresse apprise exécuté spéculativement
- le processeur détricotera ce qu'il a fait spéculativement, mais modification de l'état du cache => fuite d'information possible
- nécessite l'utilisation d'un *gadget* (suite d'instructions que l'on peut détourner dans le code de la victime)

Procédé plus complexe que les précédents, mais que les hackeurs connaîssent et maîtrisent par ailleurs

Correction par logiciel

Utiliser une suite d'instructions trompant le prédicteur pour les programmes « sensibles »

Changement de la génération de code dans les compilateur pour les sauts indirects [a]

- fait dans `gcc`
- bientôt `llvm`

Recompilation des programmes « sensibles » : Linux, Xen, ...

a. Source : `https://support.google.com/faqs/answer/7625886`.

Exécution spéculative

- source de la performance
- source de failles de sécurité par canal temporel caché

Corrections logicielles

- contournements plutôt que solutions définitives
- coût variable, mais mesurable, de quelques % a des dizaines de pourcents

$\Rightarrow$ sans doute pas viables sur le long terme

Corrections matérielles

- se priver de la spéculation ⇒ chute de la performance séquentielle
- invalider tous les caches en cas de mauvaise prédiction ⇒ impossible
- interdire l'accès à certaines resources en mode *user* est délicat
  - micro-noyaux avec *drivers* en *user space*
  - jeux vidéo avec des dépendences sur les durées fines pour les intéractions
  - etc, ...

⇒ coût matériel et en performance sans doute important