Spectre y Meltdown: dos ataques basados en microarquitectura

Katzalin Olcoz Herrero
Grupo ArTeCS
Departamento de Arquitectura de Computadores y Automática

Today's CPU vulnerability: what you need to know

- https://security.googleblog.com/2018/01/todays-cpu-vulnerability-what-you-need.html January 3, 2018

- El año pasado, el equipo Google’s Project Zero (el investigador Jann Horn) descubrió importantes grietas de seguridad causadas por la “ejecución especulativa”
- Para aprovechar esta vulnerabilidad el atacante debe ser capaz de ejecutar código malicioso en el sistema objetivo.
- Los investigadores del Project Zero descubrieron tres métodos (variantes) de ataque, que son efectivos en condiciones distintas. Los tres ataques pueden permitir que un proceso con privilegios de usuario realice lecturas no autorizadas de datos de memoria.

Reading privileged memory with a side-channel (Jann Horn)

- Hemos descubierto que se puede hacer un mal uso del comportamiento temporal (timing) de la cache de datos para así filtrar information eficientemente a partir de ejecución especulativa de instrucciones con fallos de especulación (miss-speculated execution), que puede conducir (en el caso peor) a vulnerabilidades en la lectura de memoria virtual a través de fronteras de seguridad en varios contextos.
- Hay tres variantes conocidas por el momento:
  - Variant 1: bounds check bypass /conditional branch misprediction (CVE-2017-5753)
  - Variant 2: branch target injection (CVE-2017-5715)
  - Variant 3: rogue data cache load (CVE-2017-5754)
- Antes de que se revelaran estos temas, Daniel Gruss, Moritz Lipp, Yuval Yarom, Paul Kocher, Daniel Genkin, Michael Schwarz, Mike Hamburg, Stefan Mangard, Thomas Prescher y Werner Haas también informaron de los mismos, denominándolos:
  - Spectre (variantes 1 y 2). Afecta a Intel, AMD, ARM, IBM.
  - Meltdown (variante 3)

¿Cuál es el problema?

20 ↕

```
char array1[16];
int longitud1=16;
int secreto=2;
char array2[4096];

int main() {
int x;
int var1, var2;
read(x);
if (x<longitud1) {
      var1=array1[x];
      var2=array2[var1*64];
   }
return 0;
}
```

Usando Spectre un usuario malicioso puede averiguar el contenido de cualquier posición de memoria (secreto) si consigue asignarle un valor incorrecto a x (en nuestro ejemplo x=20).

Imposible: si x es incorrecto no se cumple la condición del if y no se leen los datos.

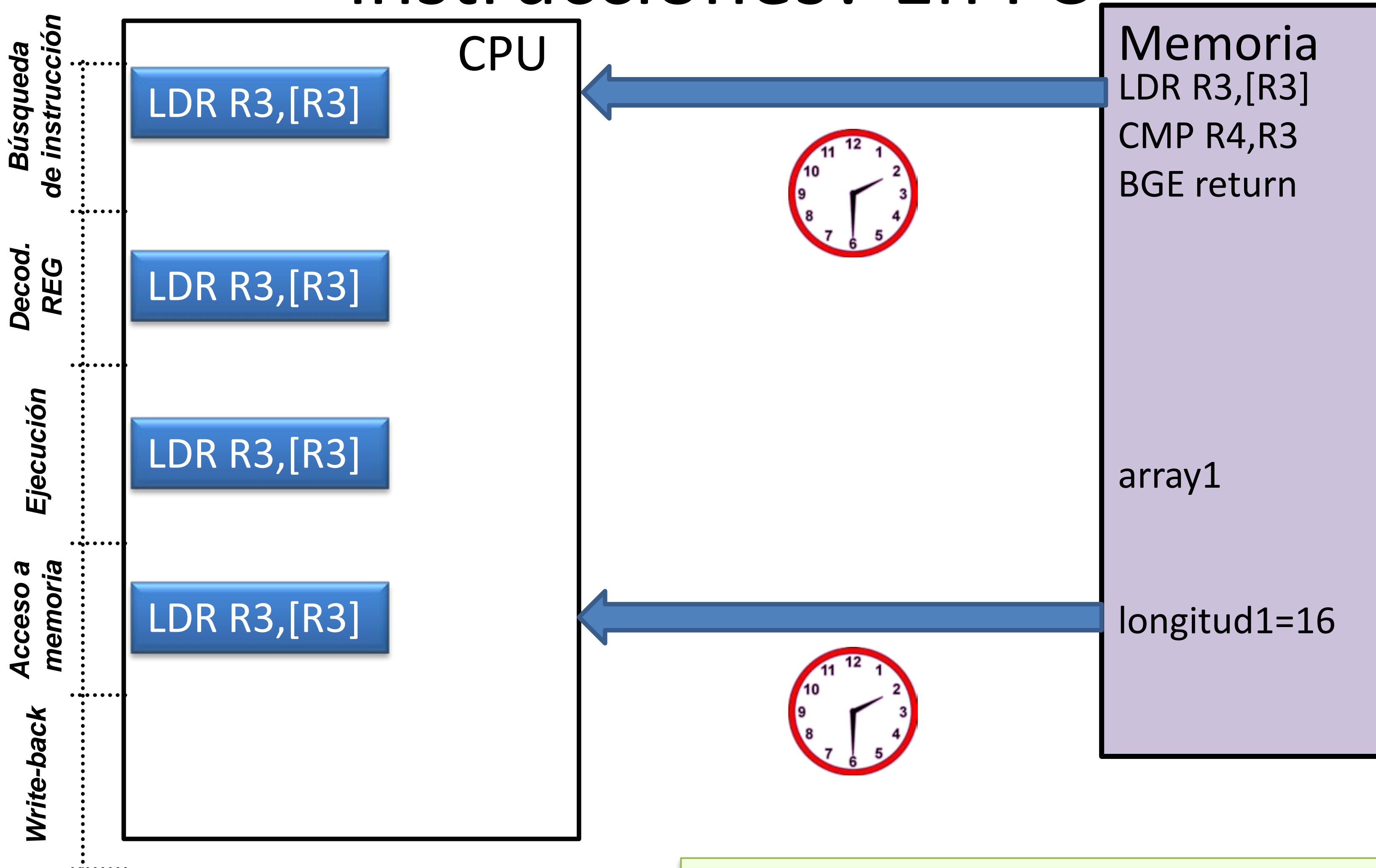
"Spectre Attacks: Exploiting Speculative Execution", P. Kocher et al. 2018

Veámoslo con más detalle: en ensamblador del ARM

```
char array1[16];
int longitud1=16;
int secreto=2;
char array2[4096];

int main() {
int x;
int var1, var2;
read(x);
if (x<longitud1) {
    var1=array1[x];
    var2=array2[var1*64];
    }
return 0;
}
```

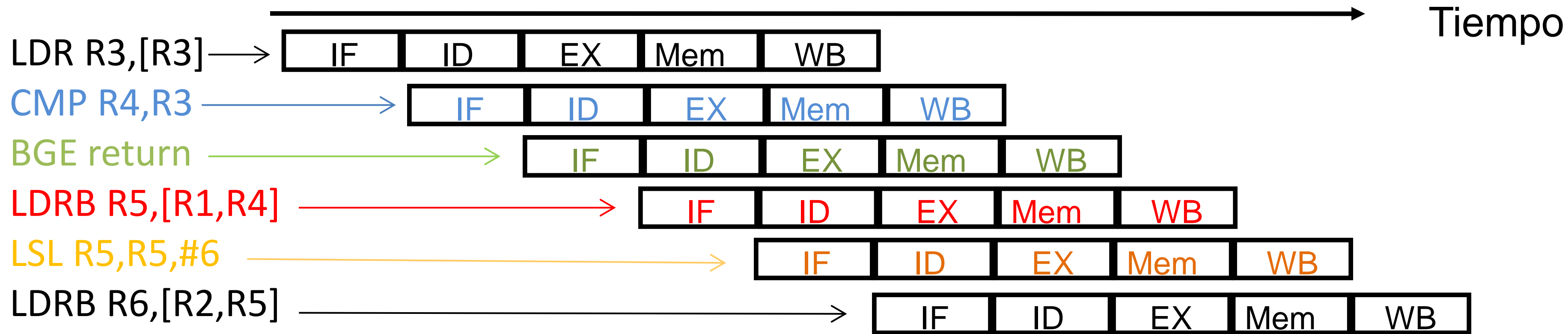
```
R1=2000 @ dirección de array1
R2=2024 @ dirección de array 2
R3= 2016 @ dirección de longitud1
R4=x @ valor introducido por usuario

LDR R3,[R3] @ R3 contiene longitud1
CMP R4,R3
BGE return

LDRB R5,[R1,R4] @ R5 contiene var1

LSL R5,R5,#6
LDRB R6,[R2,R5]   @ R6 contiene var2
```

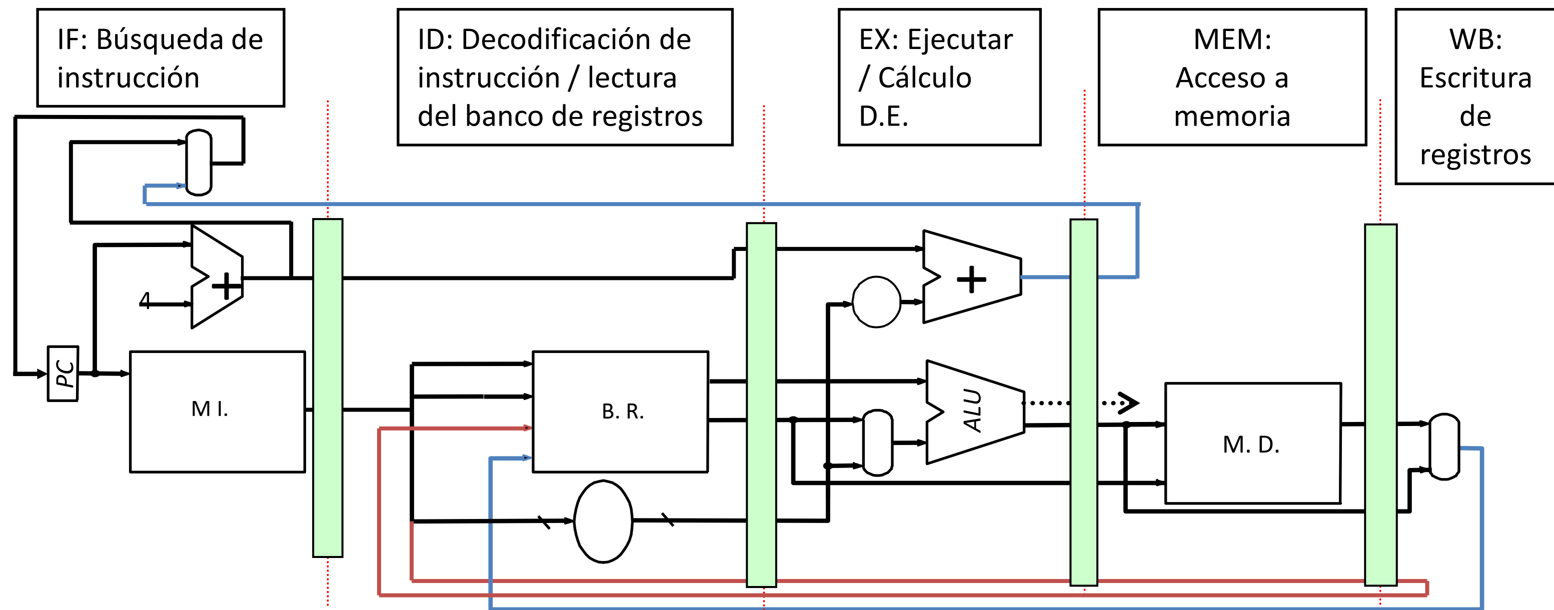
¿Cómo se ejecutan estas instrucciones? En FC



No hay problema, pero esto es lentísimo.

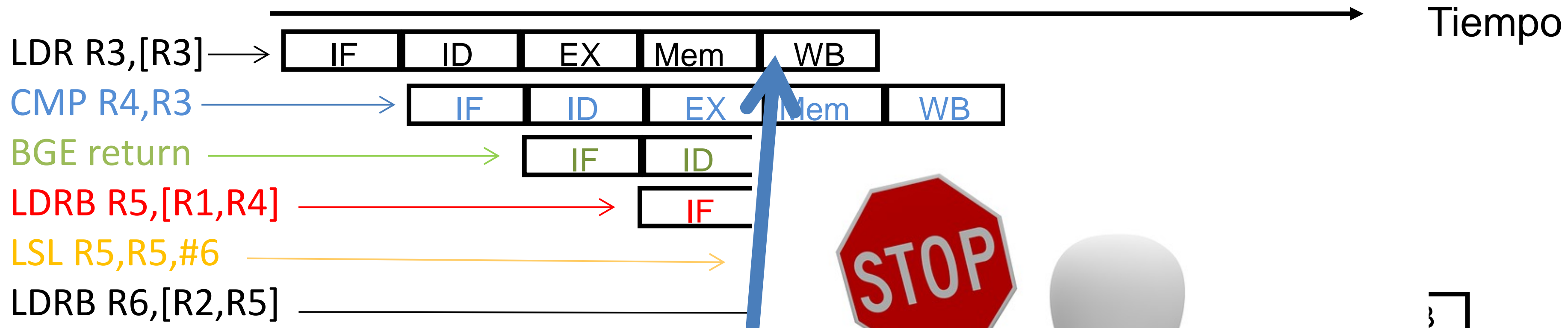
¿No podemos ir un poco más rápido?





Ejecución segmentada(EC) introducida a finales de los 60.

¿No podemos ir un poco más rápido?



Si no sé cuál es el valor de R3, hasta que no lo lea de memoria no puedo decidir si saltar o no

¿Y si voy ejecutando lo más probable?

- ## Predicción de saltos

AC Tema 2.

Cuando se detecta una instrucción de salto condicional sin resolver

Se supone o predice el camino del salto: ***tomado o no tomado (Taken - Untaken)***

Si el salto se predice como tomado se predice la dirección destino del salto

La ejecución continúa de forma ***especulativa*** a lo largo del camino supuesto

Cuando se resuelve la condición

Si la predicción fue correcta

$\Rightarrow$ La ejecución se confirma y continúa normalmente

Si la predicción fue incorrecta (fallo de predicción o *"misprediction"*)

$\Rightarrow$ Se descartan todas las instrucciones ejecutadas especulativamente

$\Rightarrow$ Se reanuda la ejecución a lo largo del camino correcto

Problemas a resolver en instrucciones de salto

1) Predecir la dirección de la instrucción destino del salto con un retardo mínimo (para saltos tomados)

2) Predecir el camino que tomará el salto

TAKEN (Tomado)

UNTAKEN (No Tomado)

El salto se predice como *Taken* si en la última ejecución fue tomado

El salto se predice como *Not Taken* si en la última ejecución no fue tomado

Predicción dinámica de saltos y ejecución especulativa(AC) introducida a mediados de los 90.

Predicción de saltos: hardware adicional en la etapa de IF

AC Tema 2.

La Branch Target Buffer (BTB) almacena

La dirección destino de los últimos saltos tomados
Los bits de predicción de ese salto

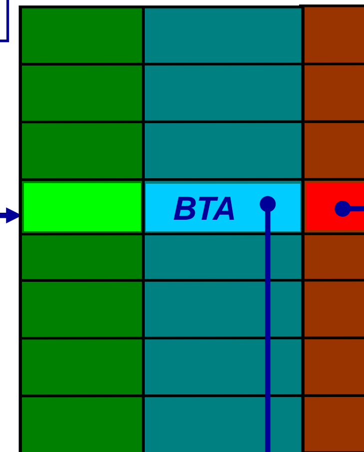
Actualización de la BTB

Los campos de la BTB se actualizan después de ejecutar el salto y se conoce:

Si el salto fue tomado o no
⇒ Actualizar bits de predicción
La dirección destino del salto
⇒ Actualizar BTA

No se usa la dirección completa de la instrucción, sino parte de los bits menos significativos.

Dirección de la instrucción de salto



Lógica de Predicción

Taken / Not Taken

+1

Dirección de la siguiente instrucción

Nuestro código con ejecución especulativa

```
LDR R3,[R3]
CMP R4,R3
BGE return
LDRB R5,[R1,R4]
LSL R5,R5,#6
LDRB R6,[R2,R5]
```

VERSIÓN SIMPLIFICADA

Suponiendo de la predicción es **SALTO NO TOMADO (x<longitud1)**

- Ejecuto especulativamente

  ```
  LDRB T1,[R1,R4]
  LSL T1,T1,#6
  LDRB T2,[R2,T1]
  ```

- Si la predicción fue correcta almaceno el resultado en los registros:

  R5 ← T1

  R6 ← T2

- Si la predicción no fue correcta descarto las instrucciones ejecutadas especulativamente.

Veámoslo otra vez

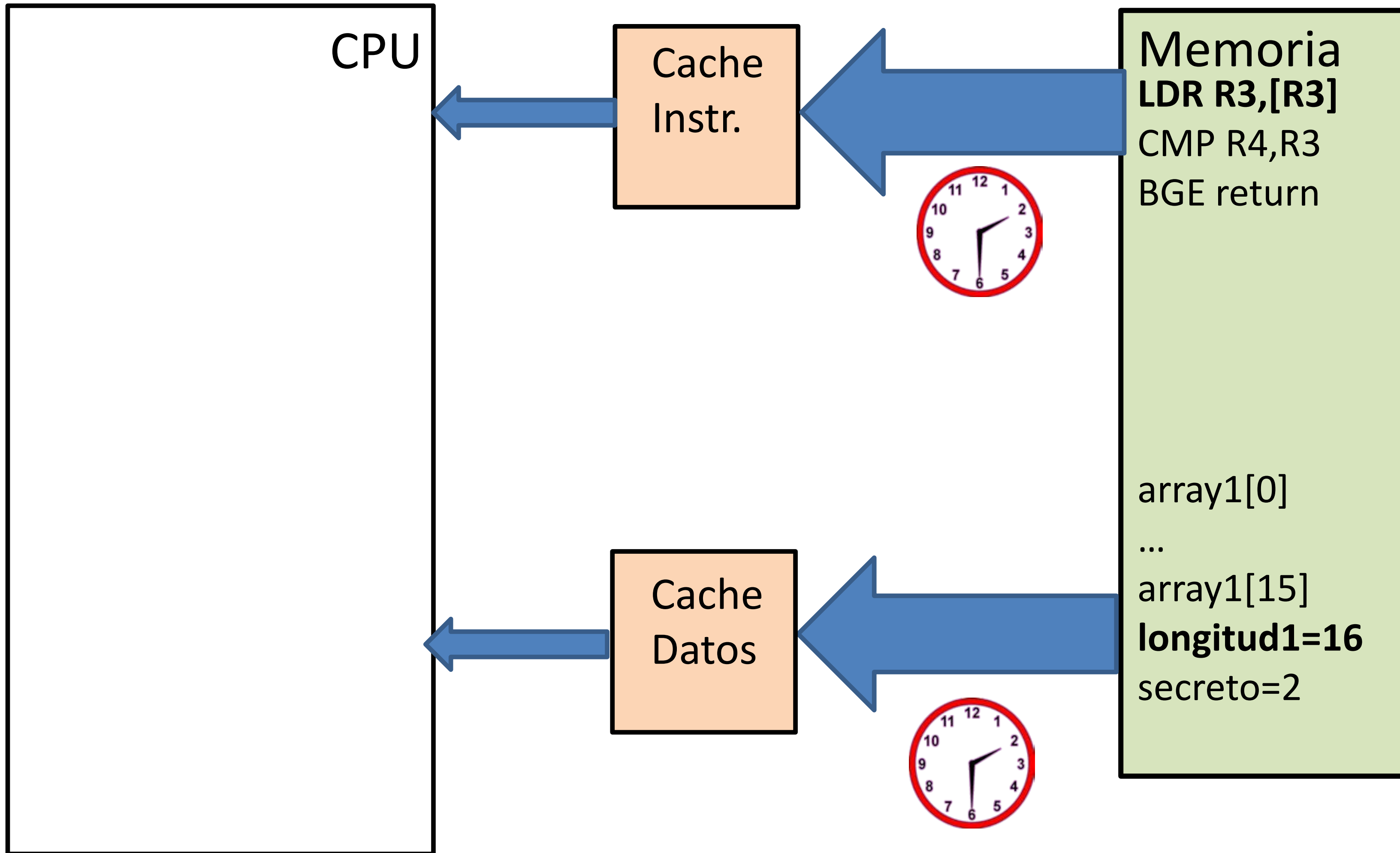
```
char array1[16];
int longitud1=16;
int secreto=2;
char array2[4096];
int main() {
int x;
int var1, var2;

read(x);
if (x<longitud1) {
    var1=array1[x];
    var2=array2[var1*64];
  }
return 0;
}
```

- Si un usuario malicioso:

1. En la fase de entrenamiento entrena al predictor para que no tome el salto (usando una instrucción con el mismo PC en los bits que usa el predictor)
2. En la fase de ataque da un valor incorrecto a x, pero **el predictor de saltos fallará y ejecutará las lecturas de memoria especulativamente, almacenando el valor secreto en var1 y accediendo al bloque de array2.**

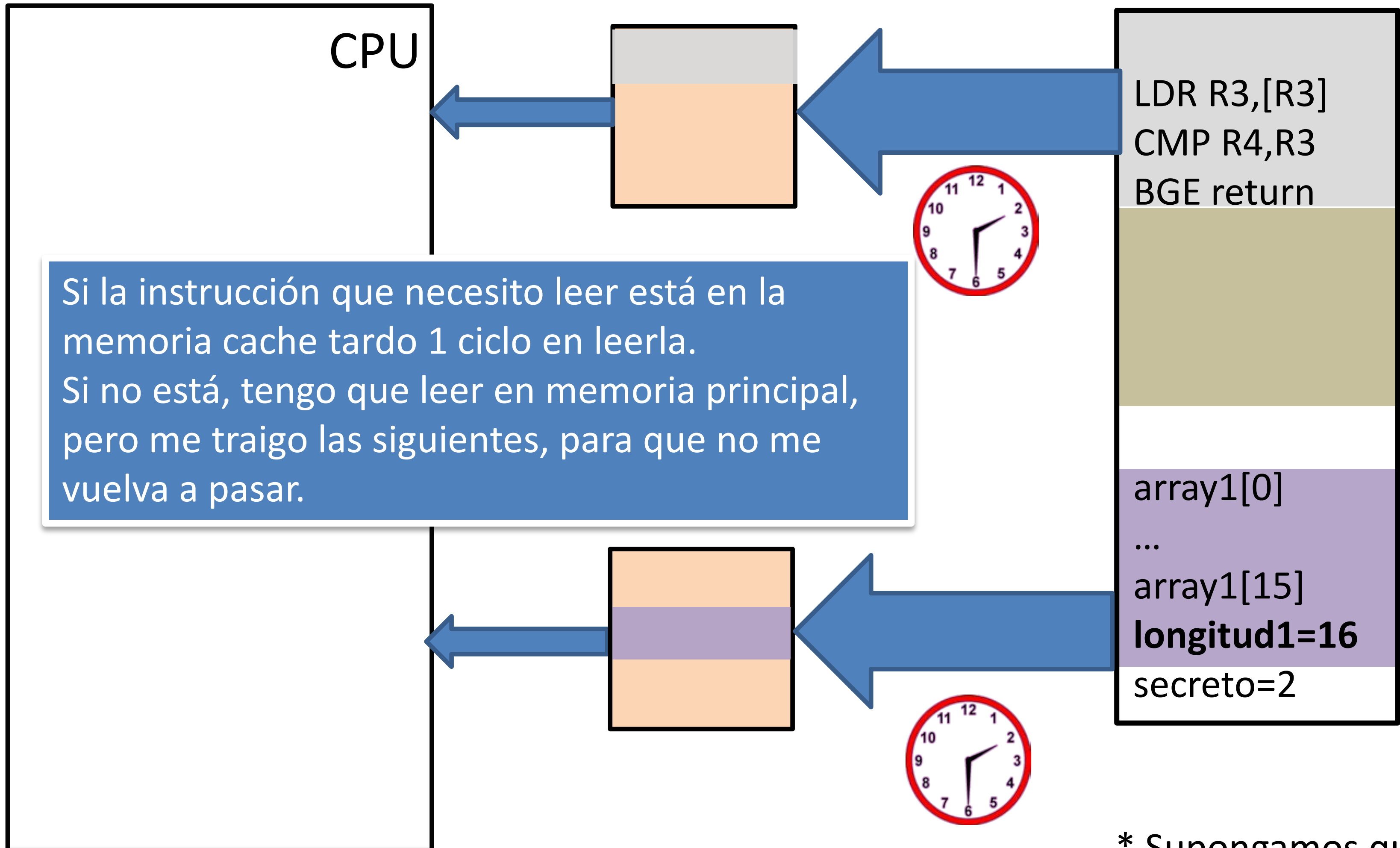
No pasa nada: los valores de var1 y var 2 están en registros especulativos (T1 y T2) y se descartan cuando se lee el valor real de longitud1.

Seguimos aumentando prestaciones



Jerarquía de memoria (EC) introducida en 1967 (IBM 360/85).
Primera caches dentro del chip a finales de los 80.

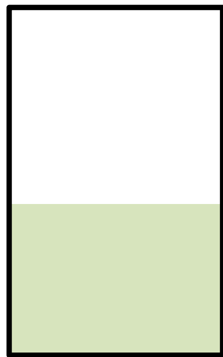
¿Cómo funciona la cache?

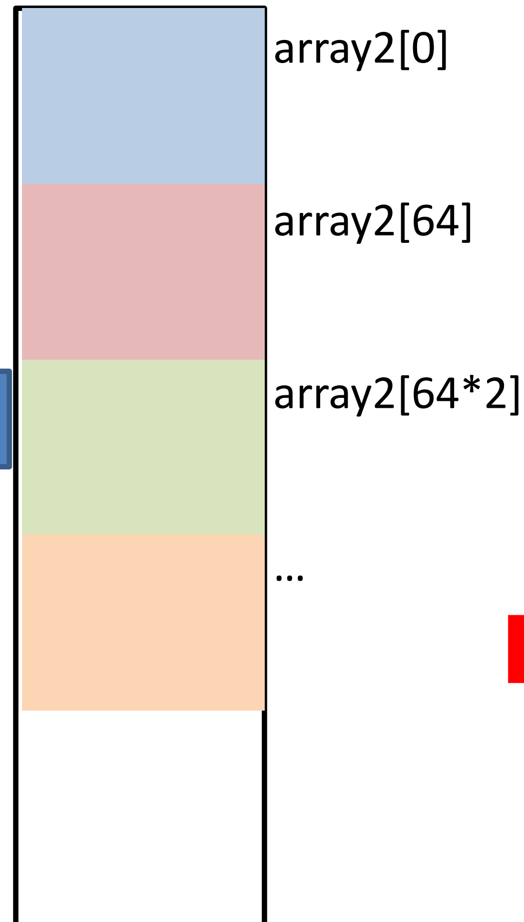


* Supongamos que cada bloque es de 64B

Aplicado a nuestro código

Si var1=2





- Ejecuto especulativamente

```
LDRB T1,[R1,R4]
LSL T1,T1,#6
LDRB T2,[R2,T1]
```

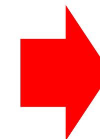
```c
char array1[16];
int longitud1=16;
int secreto=2;
char array2[4096];

int main() {
int x;
int var1, var2;
read(x);
if (x<longitud1) {
var1=array1[x];
var2=array2[var1*64];
}
return 0;
}
```

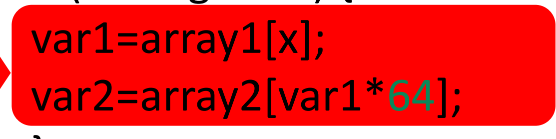
Aunque descarte las instrucciones ejecutadas por el fallo de especulación, he traído el bloque 2 a memoria cache

Cache timing attacks

- Si lo que busco está en cache tardo 1 ciclo en leerlo y si no está tardo 100.
  - Puedo saber qué datos están en cache y cuales no midiendo el tiempo que tardo en leerlos.
- Suponiendo que consiga dar un valor incorrecto a x (20) y que ejecute el if especulativamente:
  - Si ningún bloque de array2 estaba en cache antes de ejecutar el código, cuando termine la ejecución sólo estará el bloque 2.
  - Si el usuario malicioso puede leer el array2, midiendo tiempo que tarda en acceder a cada bloque averigua el valor secreto.

```
char array1[16];
int longitud1=16;
int secreto=2;
char array2[4096];

int main() {
int x;
int var1, var2;
read(x);
if (x<longitud1) {
```





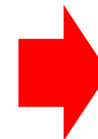
```
var1=array1[x];
var2=array2[var1*64];
}
return 0;
}
```

"Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack", Y. Yaron et al. 2014

Flush+Reload: El ataque

- Flush:
  - El espía consigue **expulsar de memoria cache todos los bloques de array2**.
- Se ejecuta la víctima y realiza accesos a memoria, en los cuales trae a cache un bloque de array2
- Reload:
  - El espía lee todos los bloques de array2, midiendo el tiempo que tarda en acceder a cada bloque.
  - Todas las lecturas tardan mucho, menos la del bloque que está en cache (el bloque 2).
  - El espía sabe que secreto=2.

```
char array1[16];
int longitud1=16;
int secreto=2;
char array2[4096];

int main() {
int x;
int var1, var2;
read(x);
if (x<longitud1) {
var1=array1[x];
var2=array2[var1*64];
}
return 0;
}
```



“Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack”, Y. Yaron et al. 2014

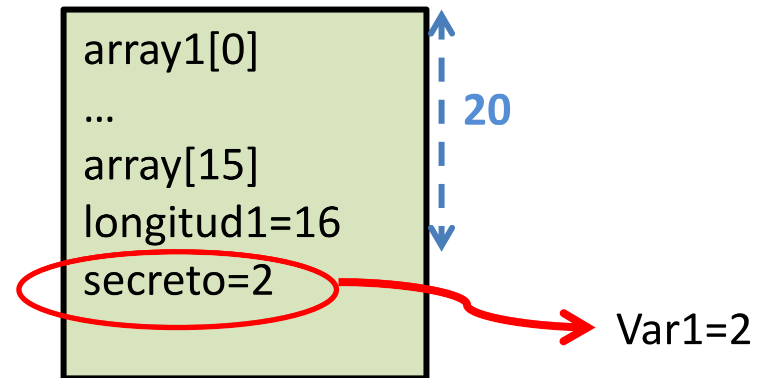
Flush+Reload: ¿Cómo funciona?

Requisitos:

1. **Flujo de información de datos sensibles al patrón de acceso a memoria**
2. Víctima y espía tienen que compartir memoria
3. El espía debe poder expulsar una línea de memoria compartida de todos los niveles de cache
4. Medidas de tiempo de alta precisión

¿Cómo lo consigue Spectre?

1. El bloque de array2 en cache me indica el valor de secreto



Si x=20
var1=array1[x];
var2=array2[var1*64];

Flush+Reload: ¿Cómo funciona?

Requisitos:

1. Flujo de información de datos sensibles al patrón de acceso a memoria
2. **Víctima y espía tienen que compartir memoria**
3. El espía debe poder expulsar una línea de memoria compartida de todos los niveles de cache
4. Medidas de tiempo de alta precisión

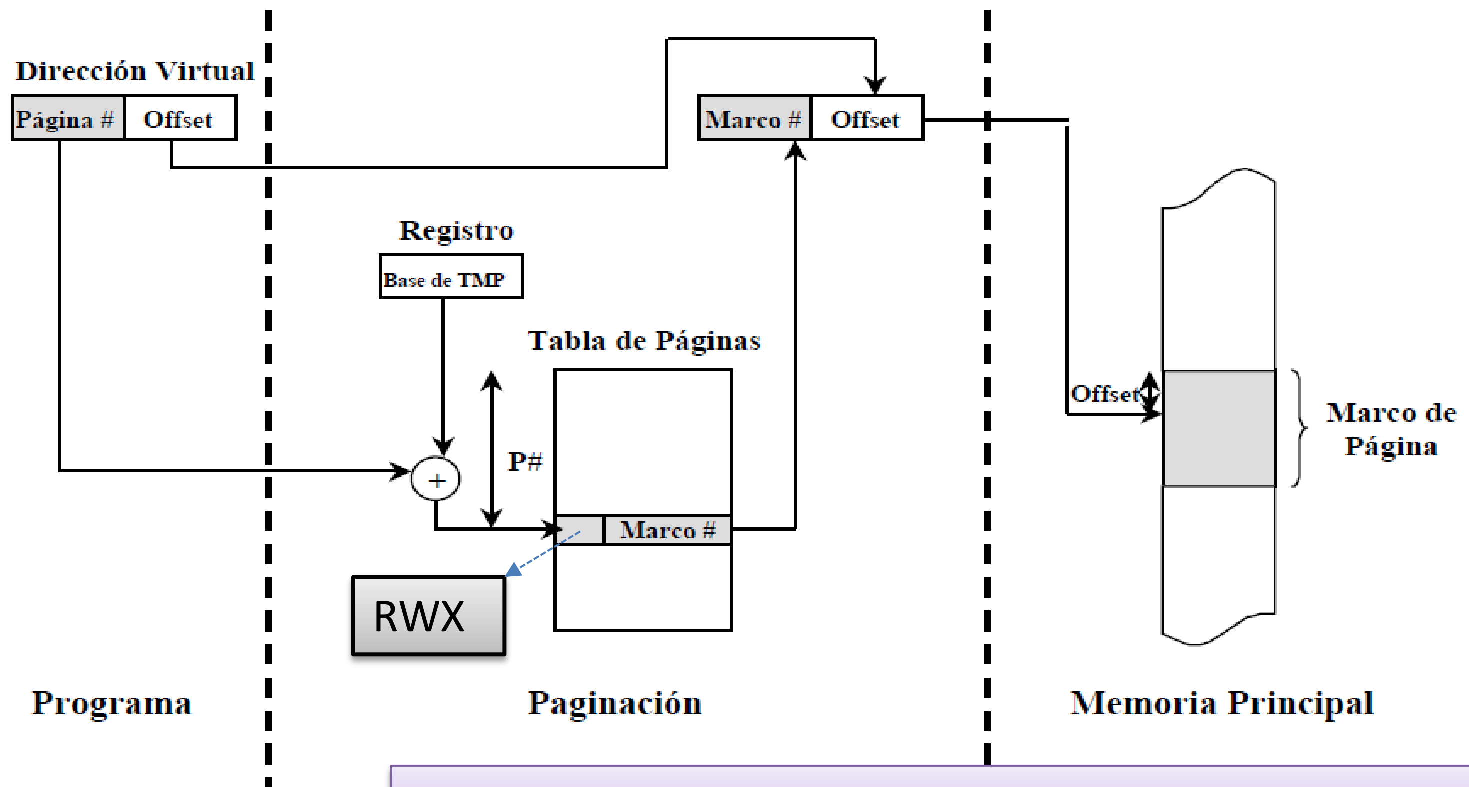
¿Cómo lo consigue Spectre?

1. El bloque de array2 en cache me indica el valor de secreto
2. Víctima y espía tienen que compartir memoria.
    1. Algún nivel de la memoria cache tiene que estar compartido
    2. **Alguna página de memoria virtual tiene que estar compartida**

“Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack”, Y. Yaron et al. 2014

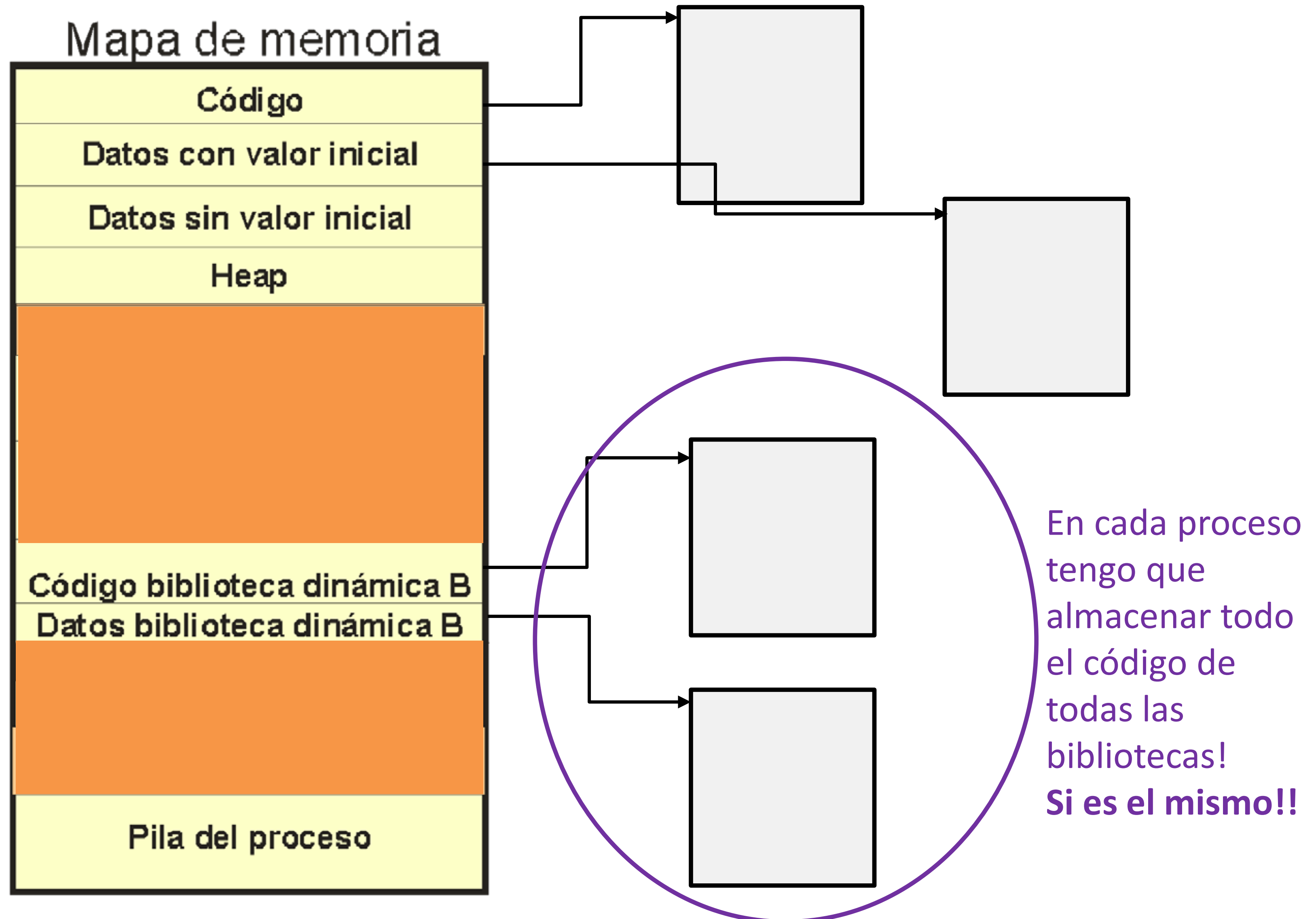
Memoria virtual

- Cada proceso tiene su propio espacio de direcciones (virtuales) completo.
- En tiempo de ejecución cada dirección virtual de memoria se traduce a la dirección física (real) usando la tabla de páginas.

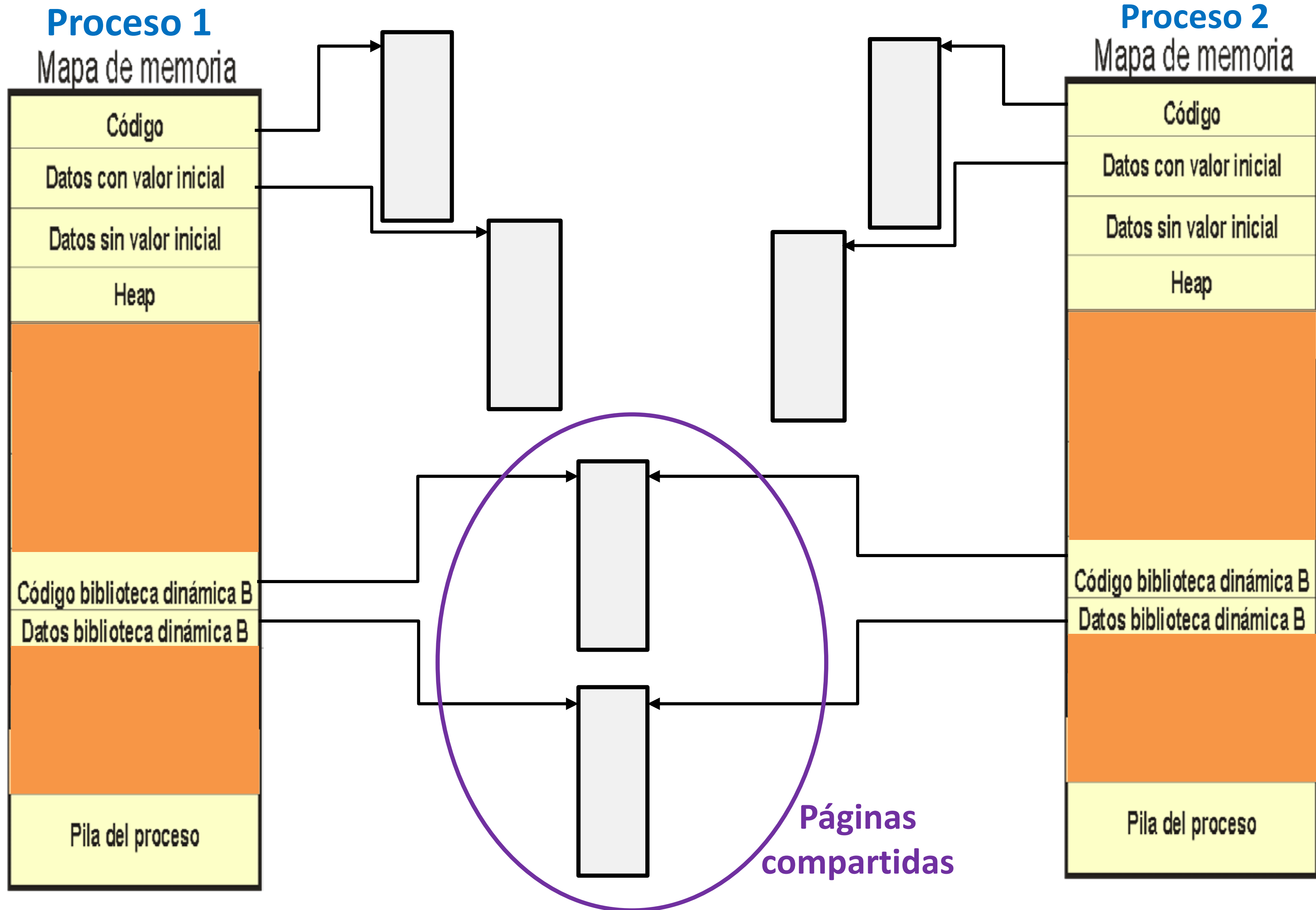


Memoria virtual (EC y SO) introducida a finales de los 50.

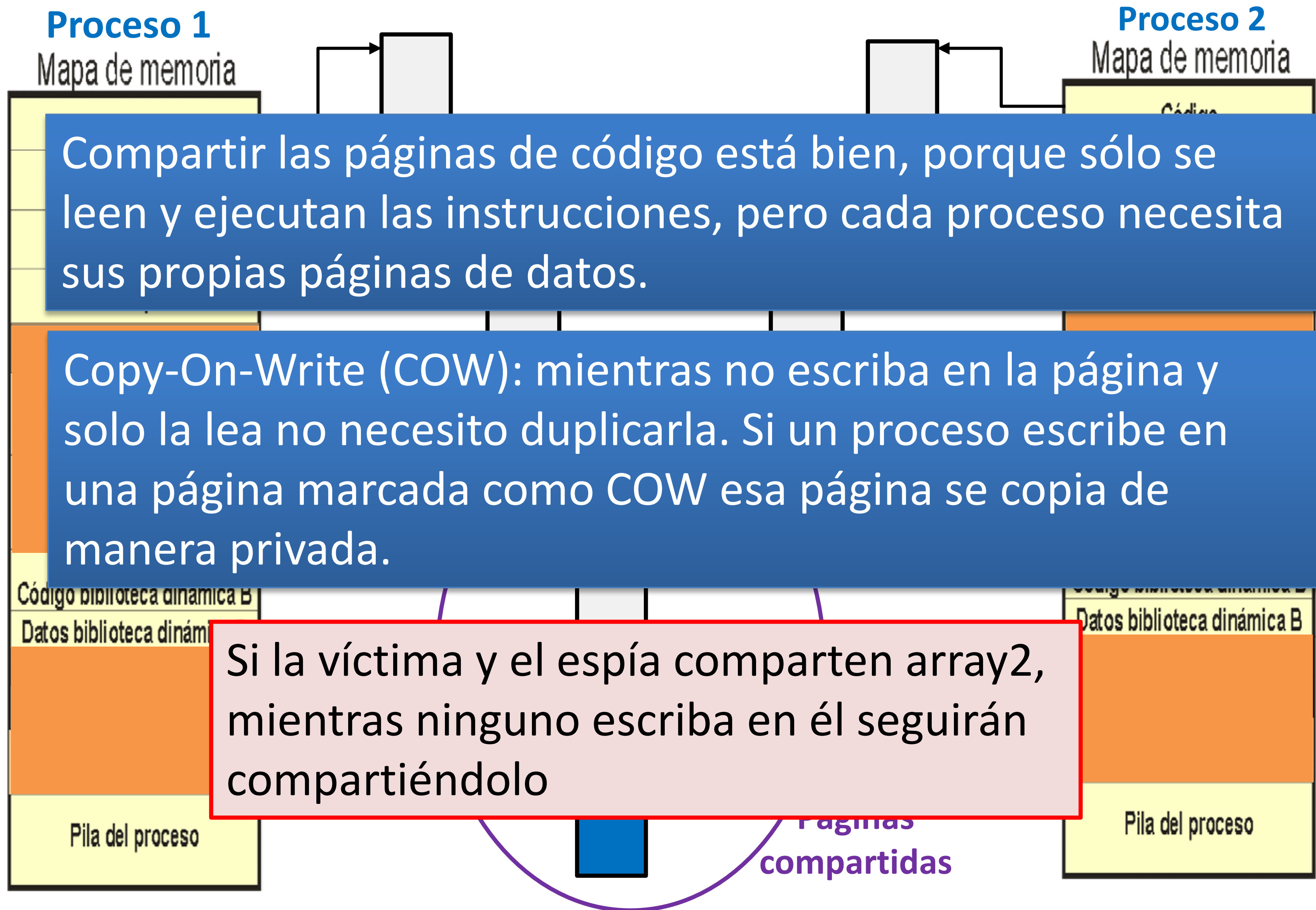
Gestión de memoria de un proceso



Seamos más eficientes



Seamos más eficientes



Compartir las páginas de código está bien, porque sólo se leen y ejecutan las instrucciones, pero cada proceso necesita sus propias páginas de datos.

Copy-On-Write (COW): mientras no escriba en la página y solo la lea no necesito duplicarla. Si un proceso escribe en una página marcada como COW esa página se copia de manera privada.

Si la víctima y el espía comparten array2, mientras ninguno escriba en él seguirán compartiéndolo

Flush+Reload: ¿Cómo funciona?

Requisitos:

1. Flujo de información de datos sensibles al patrón de acceso a memoria
2. Víctima y espía tienen que compartir memoria
3. **El espía debe poder expulsar una línea de memoria compartida de todos los niveles de cache**
4. Medidas de tiempo de alta precisión

¿Cómo lo consigue Spectre?

1. El bloque de array2 en cache me indica el valor de secreto
2. Víctima y espía tienen que compartir memoria.
   1. Algún nivel de la memoria cache tiene que estar compartido
   2. Alguna página de memoria virtual tiene que estar compartida
3. Intel:
   1. Uso de clflush en modo no privilegiado para expulsar una línea de todos los niveles de cache.
   2. Las cache son inclusivas: si expulso la línea del último nivel tiene que ser expulsada de las caches privadas de los otros cores.

En otras microarquitecturas es más complicado: el espía debe estar en el mismo núcleo para poder expulsar la línea de cache realizando muchas escrituras.

"Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack", Y. Yaron et al. 2014

Spectre: en resumen

- Fase de preparación (Setup): el atacante
  - Entrena al predictor de saltos para que falle en la predicción (if x<longitud1).
  - Expulsa de la memoria cache el valor necesario para determinar el valor correcto de la predicción (longitud1)
  - Prepara el canal-lateral (flush de array2)
- El procesador ejecuta instrucciones de la víctima.
  - Y consigue que ésta transfiera especulativamente datos sensibles desde su contexto hasta un canal-lateral basado en la microarquitectura.
- Fase final:
  - Se recuperan los datos sensibles (Flush+Reload)

En realidad, todavía falta algo…

- Para que Spectre funcione la víctima tiene que contener la secuencia de código:

```
read(x);
if (x<longitud1) {
var1=array1[x];
var2=array2[var1*64];
}
```

¿Cómo conseguir que se ejecute esta secuencia en un programa cualquiera?

Para eso están los clásicos:

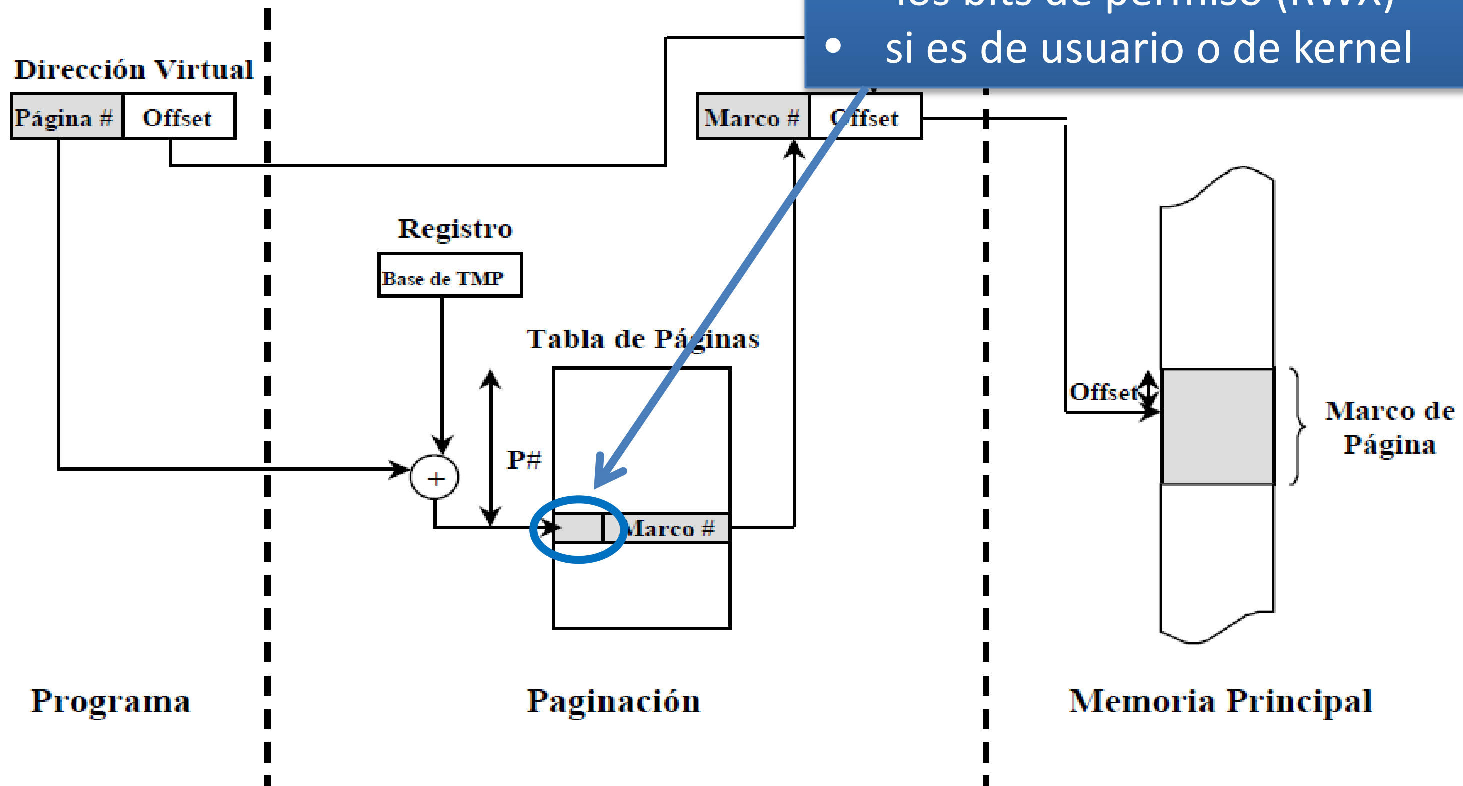
- ✓ Stack buffer overflow
- ✓ Code reuse attacks:
  - ➢ **Return-into-libc**
  - ➢ Return-oriented-programming
  - ➢ Jump-oriented-programming

1. Busco esa secuencia de código en libc
   No es difícil!
2. Consigo que la víctima la ejecute

¿Y Meltdown?

- El ataque consiste en intentar leer memoria del kernel desde espacio de usuario.

Imposible!
La tabla de páginas almacena:
- los bits de permiso (RWX)
- si es de usuario o de kernel

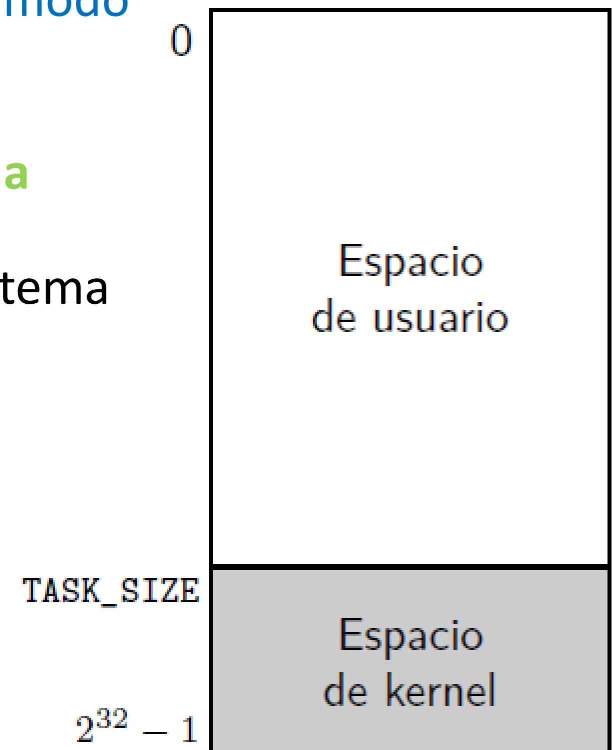


Volvemos a sistemas operativos

- Para no tener que cambiar la tabla de páginas al ejecutar llamadas al sistema, el SO divide el mapa de memoria del proceso en espacio de kernel (páginas de sistema) y espacio de usuario (páginas de usuario)
  - Cuando el proceso ejecuta instrucciones en modo usuario solo puede generar direcciones del espacio de usuario
    - **La MMU impide que el proceso acceda a direcciones del esp. de kernel**
  - Cuando el proceso invoca una llamada al sistema (paso a modo kernel) el SO accede a ambos espacios

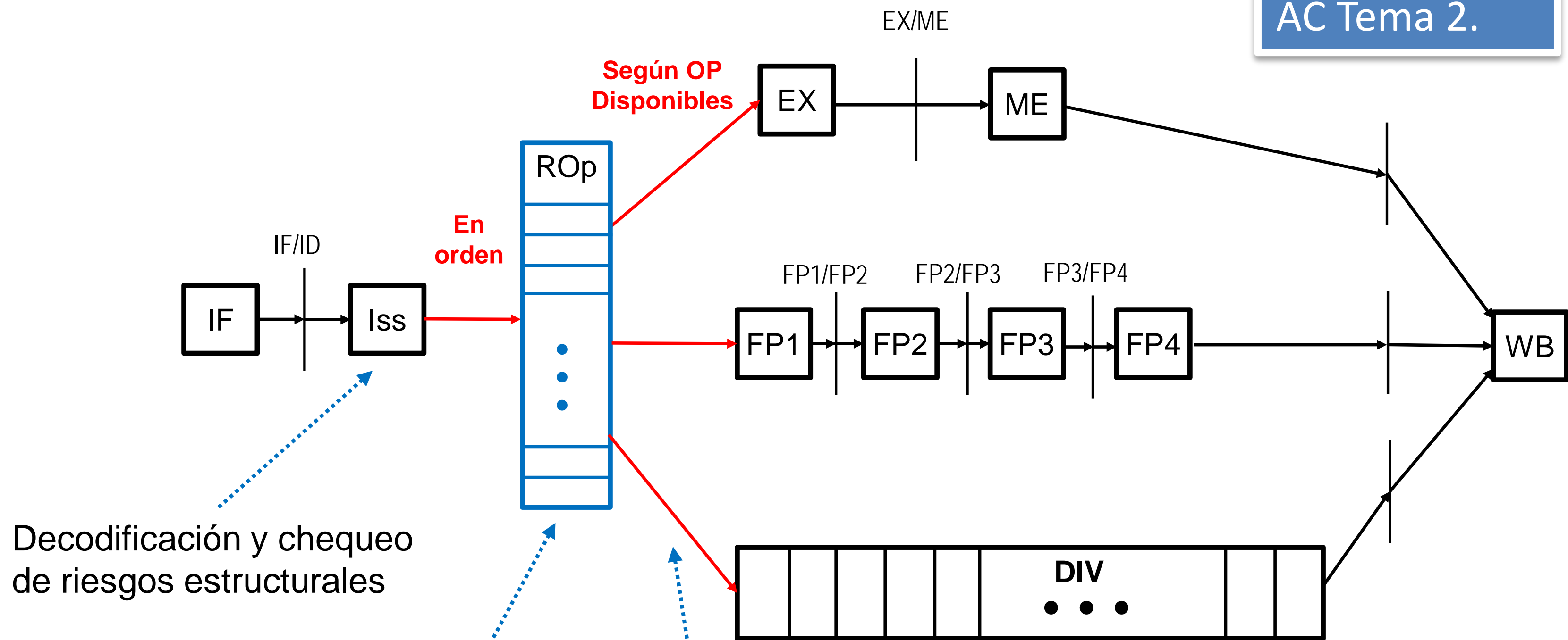
¿Eso quiere decir que las direcciones del kernel están en las tablas de página de todos los procesos? !!!!

Sí, pero si hay accesos indebidos el hardware genera una excepción



¿Cuándo se genera esa excepción?

AC Tema 2.



Las instrucciones se **ejecutan en desorden** pero **terminan en orden**.
Si la ejecución de una instrucción genera una excepción se espera a que la instrucción finalice (commit) para generar la excepción.

Meltdown: resumen del ataque

VERSIÓN SIMPLIFICADA

- Paso 1: Un proceso con permisos de usuario lee una posición de memoria del kernel.
  - Es un acceso ilegal, se producirá una excepción cuando la instrucción finalice.

Eso será así… SI finaliza…
¿recuerdas la especulación?
En cualquier caso, se puede conseguir que tarde MUCHO en finalizar.

Meltdown: resumen del ataque

VERSIÓN SIMPLIFICADA

- Paso 1: Un proceso con permisos de usuario lee una posición de memoria del kernel.
- Paso 2: Las instrucciones siguientes se ejecutan parcialmente en desorden y se descartan (transient instructions)
  - Pero pueden haber modificado el contenido de la memoria cache.
- Paso 3: Flush+Reload

Estado de los productos por fabricante

- Google https://support.google.com/faqs/answer/7622138
- IBM https://www.ibm.com/blogs/psirt/potential-impact-processors-power-family/
- Intel https://security-center.intel.com/advisory.aspx?intelid=INTEL-SA-00088&languageid=en-fr
- ARM https://developer.arm.com/support/security-update
- AMD http://www.amd.com/en/corporate/speculative-execution

Referencias

- Project Zero https://googleprojectzero.blogspot.com.es/2018/01/reading-privileged-memory-with-side.html
- A survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware, Ge et al. Journal of Cryptographic Engineering, October 2016
- Jump over ASLR: Attacking branch predictors to bypass ASLR, Evtyushkin et al. MICRO 2016.
- Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack, Y. Yaron et al. USENIX Security Symposium 2014
- Spectre Attacks: Exploiting Speculative Execution, Kocher et al.
- Meltdown, Lipp et al.
- KASRL is Dead: Long Live KASRL, Gruss et al. International Symposium on Engineering Secure Software and Systems 2017