Universidad Zaragoza

Trabajo Fin de Grado

Análisis de vulnerabilidades hardware basadas en la ejecución especulativa

Analysis of hardware vulnerabilities based on speculative execution

AUTOR

Miguel Santiago Moniente Pannocchia

DIRECTORES

Rubén Gran Tejero

Darío Suárez Gracia

ESCUELA DE INGENIERÍA Y ARQUITECTURA

2019

RESUMEN

Hoy en día, prácticamente la totalidad de los datos son procesados por un computador, siendo de vital importancia asegurar su integridad y confidencialidad. Dado que el valor de estos datos es muy elevado, los ataques informáticos que buscan extraerlos se han vuelto cada vez más frecuentes.

Como causa de este interés por la extracción de datos y el aumento en la seguridad de los sistemas, cada vez se deben desarrollar técnicas más complejas para conseguir comprometer estos datos. Durante un tiempo, la mayor parte de estos ataques estaban enfocados en las vulnerabilidades presentes en el software, tanto aplicaciones o sistemas operativos. Sin embargo, en los últimos dos años han aparecido ataques hardware cuyo objetivo es la implementación (micro-arquitectura) de los procesadores, y que comprometen la integridad de todos los niveles superiores [8-10, 12, 13].

Este trabajo se centra en el estudio de estos últimos, abriendo un nuevo foco de investigación en un ámbito de información limitada a causa de su novedad y la falta de documentación por parte de los fabricantes. Por este motivo, gran parte de este trabajo fin de grado se ha dedicado al estudio en detalle de estos ataques, su clasificación y sus implicaciones, en un constante esfuerzo para mantenerse al día con los últimos ataques publicados este mismo año.

Para el estudio de estos ataques, ha sido necesaria la preparación de una plataforma vulnerable para la experimentación. Se ha aprovechado esta plataforma para estudiar en detalle los ataques Meltdown [9] y Spectre [8]. Como fruto de este estudio, se ha podido mejorar el rendimiento de Meltdown a través de la introducción de 2 mejoras en el código fuente del ataque, obteniendo una tasa de extracción 50 veces mayor que en su versión original. Por otro lado, este estudio ha permitido proponer una posible mitigación nueva del mismo, reduciendo la eficacia del ataque.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE CUADROS

LISTINGS

ACRÓNIMOS

PoC Proof-of-Concept

CPU Central Processing Unit (Unidad Central de Proceso)

SO Sistema Operativo

ROB Reorder Buffer

ILP Instruction-Level Parallelism (Paralelismo a nivel de instrucción)

ALU Arithmetic Logic Unit (Unidad aritmético-lógica)

LLC Last-Level Cache

TSX Transactional Synchronization eXtensions

TSC Time-Stamp Counter

CDB Common Data Bus

BTB Branch Target Buffer

TLB Translation-Lookaside Buffer

IPC Instrucciones por ciclo

KPTI Kernel Page Table Isolation

SGX Software Guard eXtensions

KASLR Kernel Adress Space Layout Randomization

GLOSARIO

Arquitectura Conjunto de reglas que debe conocer el programador para programar un computador. Eso es la ISA, mapa de memoria, gestión de interrupciones, etc.

Especulación Deducción que realiza el procesador cuando no tiene plena certeza del futuro comportamiento de una instrucción. Puede fallar o acertar.

Estado micro-arquitectónico Estado almacenado en estructuras de almacenamiento interno del procesador que no son visibles al programador. Por ejemplo. caches, predictores de salto, *fill-buffers*...

Estado arquitectónico Estado almacenado en estructuras de almacenamiento que son visibles al programador, como los registros y la memoria..

Micro-arquitectura Diseño interno de un procesador. También llamado organización de un computador.

Predicción Deducción que realiza el procesador cuando tiene plenza certeza del futuro comportamiento de una instrucción. Siempre acierta.

INTRODUCCIÓN

1.1 MOTIVACIÓN

En los últimos años, gran parte de los ataques informáticos se han centrado en encontrar y explotar vulnerabilidades en el *software* que permitan a los atacantes acceder a información sensible, escalado de privilegios e incluso el control total de la máquina. Estos ataques, por lo general, pueden ser mitigados a través de actualizaciones de la aplicación o del Sistema Operativo sin afectar demasiado al rendimiento. Sin embargo, recientemente se han desarrollado ataques capaces de aprovechar el diseño (micro-arquitectura) de algunos de los mecanismos que mejoran el rendimiento de los procesadores modernos para extraer información sensible sin dejar ninguna traza en el sistema [8-10, 12, 13]. Los procesadores afectados van desde los ARM Cortex [1] presentes en la gran mayoría de los teléfonos móviles hasta la familia de CPUs POWER8 [7] en todos los IBM Power Systems, afectando también a toda la gama de procesadores utilizados en ordenadores personales.

Estos ataques a nivel micro-arquitectónico en procesadores son especialmente peligrosos ya que es muy difícil su solución vía *software*, y cuando es posible, la penalización en el rendimiento es reseñable. Además, dado que la vulnerabilidad se encuentra en el nivel más bajo del computador, algunos de estos ataques son capaces de romper barreras de aislamiento como la virtualización o las *Software Guard eXtensions* (SGX) de Intel [2]. Esto es especialmente importante en las infraestructuras *cloud*, en las que la confidencialidad de una máquina virtual se puede ver comprometida por máquinas virtuales ajenas funcionando sobre el mismo *hardware*.

Debido a su criticidad, es muy importante entender estos ataques y sus contra-medidas. Solo así se podrán proponer micro-arquitecturas más robustas y mejorar las contra-medidas ante futuros ataques de este tipo.

1.2 OBJETIVOS

El principal objetivo de este trabajo es estudiar, analizar y clasificar algunos de los ataques basados en especulación más importantes, entre ellos Meltdown [9], Spectre [8], Foreshadow [12, 13] o ZombieLoad [10], así como caracterizar y comprender mejor alguno de ellos. Por otro lado, este trabajo busca profundizar en el estudio de los factores implicados para que estos ataques funcionen, tanto desde el punto de vista del *hardware* (micro-arquitectura y arquitectura) como desde el punto de vista del *software* (características del Sistema Operativo, uso de bibliotecas compartidas o análisis de binarios vulnerables mediante *reverse-engineering*).

1.3 ALCANCE

Para la realización de este trabajo se parte de una base de conocimientos adquiridos en el grado que no ha sido suficiente para cubrir la complejidad presentada en éste, lo que implica una gran fase de estudio e investigación en diversos ámbitos antes de poder comenzar la fase de experimentación. Este estudio ha comprendido extender conceptos de administración de sistemas y seguridad informática, así como la necesidad de asimilar conceptos avanzados de las asignaturas de sistemas operativos, procesadores comerciales, multiprocesadores y arquitectura y organización de computadores, todo ello aplicado en ámbitos virtualizados y no-virtualizados.

La consecución de los objetivos requiere, en primer lugar, realizar un profundo estudio de los ataques y todos los mecanismos explotados por éstos para comprender su funcionamiento. Esto implica comprender en gran detalle aspectos como la micro-arquitectura y la arquitectura del procesador en el que se realizarán las pruebas. En este caso, los experimentos serán realizados sobre un Intel i7-7700 (Kaby Lake H), por lo que es imprescindible comprender su *pipeline* de ejecución de instrucciones, jerarquía de memoria y estructuras de almacenamiento interno del procesador. Al mismo tiempo, también es necesario estudiar las características *software* que son aprovechadas por los ataques, como el mapeo directo de memoria física en espacio de kernel, el uso de bibliotecas enlazadas dinámicamente (mapeadas entre varios usuarios y el SO), y la posibilidad de analizar binarios con el fin de encontrar secciones de código vulnerables.

Una vez alcanzados los conocimientos necesarios para comenzar a experimentar, se ha preparado una máquina con un entorno controlado en el que poder realizar las pruebas de dichos ataques. Dado que para la mayoría de estos ataques ya se han desplegado parches de microcódigo y actualizaciones del kernel Linux, es imprescindible realizar un *downgrade* tanto de kernel como del parche de microcódigo a una versión vulnerable contra estos ataques.

Tras esto, se ha analizado el comportamiento de los *Proof-of-Concept* (PoC) aportados en los artículos originales y obtenido resultados experimentales de varios de los ataques presentados. Posteriormente, se ha programado una versión funcional de Meltdown desde cero, con el fin de estudiar en más detalle los mecanismos implicados en su funcionamiento. A partir de este código, junto con el estudio realizado sobre la micro-arquitectura del procesador, se ha conseguido mejorar la eficiencia del ataque con respecto a la versión original analizando los elementos críticos y experimentando con diferentes configuraciones, lo que a su vez permite proponer nuevas mitigaciones.

1.4 DESCRIPCIÓN DEL DOCUMENTO

En el Capítulo 2 se describen los factores del procesador que hacen posible estos ataques, tanto desde el punto de vista de la ARQUITECTURA (Sección 2.1) como desde el de la MICRO-ARQUITECTURA (Sección 2.2). En este capítulo también se presentan los diferentes ataques estudiados (Sección 2.4) y se propone una TAXONOMÍA (Sección 2.5). En el Capítulo 3 se ahonda en los ataques Meltdown (Sección 3.1) y Spectre (Sección 3.3). En el Capítulo 4, se describe la metodología de trabajo y la

preparación de la PLATAFORMA (Sección 4.1) y HERRAMIENTAS (Sección 4.2). En el Capítulo 5 se presentan las pruebas realizadas y los resultados obtenidos, así como las mejoras realizadas sobre el ataque Meltdown. Finalmente, en el Capítulo 6 se presentan las conclusiones y el posible trabajo futuro de este trabajo.

ESTADO DEL ARTE

2.1 MAPA DE MEMORIA DEL SISTEMA OPERATIVO

En este apartado se expone el mecanismo de mapeo de memoria que utilizan los sistemas operativos modernos para la gestión paginada de memoria. Este mecanismo es fundamental para los ataques objeto de este trabajo.

Con la necesidad de implementar multi-tarea y multi-usuario en los sistemas operativos, se ideó el concepto de memoria virtual paginada, una característica que permite la división del espacio de direccionamiento virtual (o lógico) de los programas en fragmentos más pequeños y manejables denominados páginas. A su vez, toda la memoria física de un computador es dividida en fragmentos del mismo tamaño que las páginas, llamados marcos de página, que albergarán en su interior las páginas pertenecientes a los procesos.

En la creación de un proceso, el sistema operativo buscará marcos de páginas libres y les asignará las páginas del proceso, anotando esta asociación en una tabla de páginas propia del mismo proceso. Asimismo, en Linux, el sistema operativo provee para cada uno de los procesos, un interfaz que contiene la información de la tabla de páginas del mismo, ubicada en /proc/self/pagemap.

Cada proceso tiene un espacio de direccionamiento lógico (o virtual), en el que el proceso cree que se ejecuta en solitario en un sistema con toda la memoria virtual que puede direccionar. Puesto que es imposible tener un sistema con suficiente memoria física como para mapear el espacio virtual de todos los procesos que se ejecutan, es necesario que en memoria física, solo se alberguen algunas páginas lógicas (las más usadas). Para acceder a estas páginas, existe un mecanismo de traducción de direcciones virtuales a físicas, el *Translation-Lookaside Buffer* (TLB).

Con el fin de optimizar algunas llamadas al sistema, dentro del espacio lógico del proceso se encuentra mapeado el propio kernel, de este modo las llamadas al sistema ahorran cambios de contexto y vaciados del TLB. Estas páginas del kernel mapeadas en el espacio de memoria del proceso, son protegidas con los permisos de acceso presentes en la tabla de páginas.

Una de las tareas del kernel es gestionar el movimiento de marcos de página entre memoria principal y almacenamiento no volátil. Así pues, el kernel es el encargado de cargar en memoria las páginas de cada uno de los procesos. Para llevar a cabo esta labor, el kernel, en su espacio de direccionamiento, tiene mapeada toda la memoria física del computador a partir de un cierto offset, que por defecto es la dirección `0xffff880000000000`. Esto es lo que se conoce como *direct-physical mapping*.

2.2 MICRO-ARQUITECTURA DE UN PROCESADOR MODERNO

Los procesadores de hoy en día llevan una serie de mecanismos *hardware* que permiten acelerar la ejecución de las instrucciones, permitiendo así un mayor rendimiento en instrucciones por ciclo.

A continuación se detalla el funcionamiento de algunos de estos mecanismos que fundamentan los ataques presentados.

2.2.1 *Ejecución fuera de orden (Out-of-order execution)*

La ejecución fuera de orden es la capacidad del procesador para ejecutar instrucciones en un orden distinto al orden secuencial descrito por el programador, permitiendo romper la dependencia de control implícita entre instrucciones (secuencialidad). De este modo, un procesador superescalar (que es capaz de lanzar a ejecución múltiples instrucciones en un mismo ciclo) puede aprovechar en mayor medida el paralelismo a nivel de instrucción presente en los programas. El paralelismo a nivel de instrucción de un programa (*Instruction-Level Parallelism* o ILP) corresponde al número de instrucciones que pueden ser ejecutadas en paralelo. Dos instrucciones pueden ser ejecutadas en paralelo siempre y cuando sean independientes, es decir, que no existe una dependencia de datos entre ellas. Estas dependencias entre instrucciones pueden ser: RAW (*read-after-write*), WAR (*write-after-read*) y WAW (*write-after-write*) [5].

Con el fin de explotar al máximo el ILP, es necesario eliminar las dependencias existentes entre las instrucciones de un programa. Para ello, en 1967 Tomasulo propuso un algoritmo [11] capaz de eliminar las dependencias WAR y WAW (o dependencias falsas), quedando únicamente las dependencias RAW (o dependencias verdaderas).

Dado que se desea maximizar el ILP explotado aún cuando existen dependencias verdaderas, los procesadores hacen uso de diversas técnicas para exponer más ILP. Muchas de estas técnicas se basan en la predicción del comportamiento de una cierta instrucción del programa, y de este modo poder explotar el ILP presente en el programa más allá de dicha instrucción. En este trabajo, se habla de **predicción** cuando se tiene plena certeza del futuro comportamiento de la instrucción. Un ejemplo de ésta sería la predicción de latencia de ejecución de instrucciones de latencia constante (operaciones de ALU). Por otro lado, cuando no existe plena certeza se habla de **especulación**, la cual será acierto o fallo dependiendo de la coincidencia del comportamiento observado con el comportamiento predicho. Esto ocurre, por ejemplo, con la predicción de latencia en instrucciones de latencia variable como el load.

2.2.2 *Predictor de saltos*

Otra optimización micro-arquitectónica añade estructuras para la predicción del comportamiento de instrucciones de salto: dirección destino de salto y decisión sobre tomar o no tomar el salto. Estas estructuras se conocen como predictores de saltos, y su objetivo es realizar una predicción o especulación cada vez que aparece

un salto en el programa.

En el caso de los retornos de subrutina, las direcciones destino de salto suelen ser bien conocidas con anterioridad. Para estas instrucciones un sencillo predictor puede ser capaz de acertar siempre la dirección de la siguiente instrucción a ejecutar.

Por otro lado, cuando la decisión de tomar el salto o no depende de una instrucción de salto condicional, el procesador debe especular con el futuro comportamiento de dicha instrucción de salto para poder seguir extrayendo ILP más allá de dicha instrucción (Subsección 2.2.1). De esta manera, el procesador decidirá si el salto es tomado o no, y continuará la ejecución del flujo elegido de manera especulativa, sin esperar a que la condición se resuelva. Una vez resuelta la condición, si el flujo de ejecución tomado era el correcto, la ejecución continuará desde ese punto. De lo contrario, se hará un *rollback* de todas las instrucciones ejecutadas y se continuará por el otro flujo.

Esta decisión será tomada basándose en los comportamientos anteriores para dicha instrucción, haciendo uso de técnicas que combinan estos comportamientos almacenados en estructuras de almacenamiento interno como el *Branch Target Buffer* (BTB), que indexa la predicción con los bits de menor peso del contador de programa.

En el ejemplo del Listado 1, cuando llega el salto condicional jne, el procesador debe decir, en base a ejecuciones anteriores, si seguir ejecutando la resta (sub, línea 4) posterior, o la suma (add, línea 6) en la dirección del salto. En ambos casos, el procesador está calculando en base a una suposición, siendo una ejecución especulativa.

```
1         add ...
2         cmp ...
3         jne eti0
4         sub
5     eti0:
6         add ...
```

Listado 1: Ejemplo de especulación

2.2.3 *Tratamiento de excepciones*

La ejecución fuera de orden permite mejorar el rendimiento del procesador a base de exponer un mayor ILP, pero existe la posibilidad de que ocurra una excepción (fallo de página, instrucción ilegal, etc.) mientras instrucciones posteriores ya se encuentran en ejecución o han sido ejecutadas, que deben ser revertidas.

En todos los procesadores con posibilidad de ejecutar instrucciones fuera de orden se hace uso de una estructura comúnmente llamada Reorder Buffer (ROB). Dicha estructura mantiene las instrucciones con el mismo orden con el que llegaron al *pipeline* del procesador, es decir, en orden de programa. En la fase del *pipeline* de *retirement* se utilizará esta estructura para retirar la instrucción mas vieja, y así, permitir que el estado arquitectónico modificado por esa instrucción sea visible.

Así pues, para que una instrucción pueda ser retirada, deben haberse retirado todas aquellas instrucciones que son anteriores (más viejas) en el orden de programa.

Los mecanismos para tratamiento de una excepción y los mecanismos para el *rollback* de una especulación fallida hacen uso del ROB para asegurar un estado arquitectónico correcto. Así pues, si una instrucción provoca una excepción, será marcada en su entrada del ROB y cuando llegue el momento de ser retirada, se invalidarán todas las instrucciones posteriores a ésta. Aunque la ejecución de una instrucción no modifique el estado arquitectónico, las modificaciones realizadas en el estado micro-arquitectónico persistirán y serán visibles.

2.3 ATAQUES DE CANAL LATERAL

De manera sucinta, los ataques de canal lateral son un tipo de ataques capaces de inferir información de la víctima a través de la monitorización de canales en el sistema atacado. Es decir, requieren conocer en profundidad la implementación física del hardware. Por ejemplo, en procesadores es conocido que el consumo depende de los datos de entrada y de las operaciones realizadas, por lo tanto mediante su observación se puede determinar si, por ejemplo, un algoritmo criptográfico está operando con un 1 o un 0 y en qué fase está.

Para este TFG se han estudiado ataques de canal lateral que emplean las estructuras micro-arquitectónicas del procesador como el predictor de saltos o la memoria cache para robar información, ya que la ejecución especulativa no es vista por el usuario pero si por dichas estructuras. Además de ataques de canal lateral basados en especulación, también hay electromagnéticos, acústicos, ...

2.4 ATAQUES BASADOS EN ESPECULACIÓN

En esta sección se presentan los ataques estudiados para este trabajo, destacando las características principales de cada uno de ellos para posteriormente proponer una posible taxonomía de éstos.

2.4.1 *Flush+Reload*

El primero de estos ataques, aunque no es un ataque basado en la ejecución especulativa, tiene especial importancia para los siguientes ataques presentados, ya que es utilizado para extraer los secretos robados desde el estado micro-arquitectónico al estado arquitectónico y concluir el ataque.

Flush+Reload [14] es un ataque de canal lateral que permite inferir el comportamiento del programa atacado a través de la monitorización del tiempo de acceso a una dirección de memoria compartida entre ambos. Un caso de uso de este ataque sería, por ejemplo, el uso de bibliotecas compartidas entre la víctima y el atacante.

Para realizar a cabo este ataque, el atacante fuerza la expulsión de la línea de cache monitorizada, y esperará a que el proceso víctima acceda a dicha dirección. Pasado un tiempo, el atacante accede a la dirección monitorizando el tiempo (en ciclos) que cuesta acceder. Si el tiempo ha sido menor que un *threshold* estipulado, quiere decir que el proceso víctima ha accedido a dicha dirección. De lo contrario,

se asumirá que la víctima no ha accedido.

Esta información se puede utilizar para inferir el comportamiento de la víctima, o en combinación con otros ataques, extraer un dato perteneciente al espacio de memoria privilegiada, sin necesidad de permisos de superusuario para llevar a cabo ninguna de las operaciones.

2.4.2 *Meltdown*

Meltdown [9] es un ataque basado en especulación capaz de acceder a direcciones de memoria arbitrarias, privilegiadas o no privilegiadas, desde espacio de usuario. Esto lo convierte en uno de los ataques más peligrosos, pero tiene fácil mitigación.

Este ataque se aprovecha principalmente de dos vulnerabilidades. La primera de ellas se basa en la implementación del mapa de memoria en el kernel Linux. Como se explica en la Sección 2.1, por cuestiones de eficiencia todos los procesos tienen mapeado en su espacio de direccionamiento una copia del kernel, que a su vez éste tiene mapeado toda la memoria física en uso para minimizar los cambios de contexto en algunas llamadas al sistema. Esto se hace confiando en que el hardware bloqueará el acceso a dicho espacio de memoria a un usuario no privilegiado. Sin embargo, la segunda vulnerabilidad explotada por Meltdown es la característica de especulación que permite a un proceso acceder a una dirección protegida mientras espera la validación de los permisos.

De esta manera, el atacante accede a la dirección de memoria que contiene el secreto del proceso víctima, mapeado dentro de su propio espacio de direccionamiento. Tras acceder, el atacante utiliza el valor del secreto al que ha accedido para indexar un vector de su propiedad antes de que la excepción sea procesada. Mientras tanto, este vector es monitorizado con un Flush+Reload, lo que permite inferir qué elemento del vector ha sido accedido, y por lo tanto obtener el secreto, que se corresponde con el valor de índice de dicho elemento.

2.4.3 *Spectre*

De todos los ataques basados en especulación, Spectre [8] es el ataque que más procesadores afecta, pues no solo afecta a los procesadores de Intel, si no que afecta a la gran mayoría de procesadores que hacen uso de la especulación para decidir el camino de ejecución de un salto.

Al igual que Meltdown, Spectre permite leer direcciones de memoria de otros procesos sin tener los privilegios para ello, pero lo hace a través de la manipulación de los predictores de salto (Subsección 2.2.2).

Spectre comparte con Meltdown el mecanismo básico para extraer un secreto, que consiste en un acceso a la dirección de memoria con el secreto seguido de un ataque de canal lateral Flush+Reload para exponer el valor del secreto al estado arquitectónico del atacante. Lo que diferencia a Spectre de Meltdown, es que esta secuencia de ataque va a ser ejecutada bajo un fallo de especulación de salto pendiente de resolver. Frente a lo que ocurre en Meltdown, recuperarse de un fallo de

especulación de salto es menos costoso que tratar una excepción por acceder a una dirección de memoria sin permiso (secreto), lo que acelera la filtración del secreto. Lo que Spectre aporta es la manipulación del predictor de salto para que el ataque sea ejecutado al igual que Meltdown, pero pendiente de una predicción de salto que el atacante sabe que va a ser fallo de especulación.

En la primera variante de Spectre, es el propio atacante que codifica un ataque Meltdown (acceso a una dirección protegida) dentro del ámbito de una instrucción de control tipo *if*. A continuación, el atacante entrena al predictor de salto para que se prediga la condición del *if* como verdadera. Antes de ejecutar el ataque, modifica la condición del *if* para que sea resuelta como falsa. Así pues, cuando se ejecuta el código, el predictor de salto indicará que se ejecute especulativamente el código del ataque. Sin embargo, cuando se resuelva el salto, el ataque sera invalidado por un fallo de predicción de salto y no por una excepción.

Por otro lado, la segunda variante de Spectre es mucho más compleja, pues requiere hacer uso de ingeniería inversa en el binario que se desea atacar, con el fin de encontrar un segmento de código que realice la misma función que la variante 1, pero cumpliendo una serie de condiciones. La primera de ellas es que este segmento tiene que estar dentro del espacio de direccionamiento de la víctima, para poder acceder a su información. Por otro lado, el atacante debe tener poder influir en los valores que recibe como parámetro. Esto podría ser, por ejemplo, una función de biblioteca compartida entre el atacante y la víctima.

Una vez encontrado este código, el atacante deberá replicar el comportamiento del código de la víctima, mimetizando las direcciones utilizadas por la víctima con el fin de entrenar el BTB accediendo a dicho segmento. Esto provocará que cuando entre en ejecución la víctima, se forzará el salto indirecto al segmento y así realizar el ataque, de manera similar a la variante 1.

2.4.4 *Foreshadow*

Foreshadow [12, 13] es otro de los ataques especulativos que, a causa de la lenta comprobación de permisos de acceso a una página, ciertas instrucciones ejecutadas especulativamente serán capaces de acceder a un byte secreto. Estas instrucciones, al igual que los demás ataques, reflejarán los cambios en el estado microarquitectónico de manera que puedan ser extraídos mediante un canal lateral como Flush+Reload.

La peculiaridad de Foreshadow es que, en su variante SGX, es capaz de romper las barreras de aislamiento que provee SGX. Estas extensiones, permiten al programador definir unos almacenes llamados enclaves, en los que se almacenarán los secretos del programa. Ni siquiera el kernel o un usuario privilegiado es capaz de ver el contenido almacenado en estos enclaves, pudiendo acceder únicamente a través de una previa verificación mediante clave pública con un hash del enclave. La posibilidad de romper esta barrera de aislamiento es lo que hace que este ataque sea particularmente peligroso.

Para llevar a cabo este ataque, un atacante debe ejecutar el enclave víctima de manera que los secretos se almacenen en cache. Dado que los datos en cache no se

encuentran cifrados, el atacante realiza un ataque Meltdown sobre la dirección en la que se almacena el secreto, exponiendo los secretos al estado arquitectónico con un canal lateral.

Además, Foreshadow permite la extracción de las claves necesarias para la creación de un enclave malicioso, verificando el acceso por clave y así recibir directamente los secretos de la víctima.

2.4.5 *ZombieLoad*

El último de los ataques presentados en este trabajo, al igual que Meltdown y Foreshadow, explota la especulación realizada durante la comprobación de permisos de acceso, y permite extraer información desde espacio de usuario.

ZombieLoad [10] aprovecha la existencia de unas estructuras de almacenamiento interno del procesador denominadas *fill-buffers*, que son utilizadas para rellenar la cache con datos de loads anteriores que han sido expulsados, por ejemplo, por una excepción. Estos *buffers*, además, son compartidos entre los procesos lógicos de un mismo núcleo del procesador.

Este ataque permite a un atacante inducir fallos de página mientras realiza un acceso a memoria. De esta manera, cuando se recupera de la excepción el *fill-buffer* cargará en cache los datos utilizados por loads anteriores, independientemente del proceso lógico que los haya ejecutado. Esto permite, a través de un vector de muestreo, la extracción de estos datos mediante un canal lateral como Flush+Reload.

Este ataque, a diferencia de los anteriores, no tiene la capacidad de decidir la dirección de memoria de la que se extrae un dato, pero sí puede saber qué datos están siendo utilizados por otros procesos, lo que rompe las barreras de aislamiento entre procesos.

2.5 TAXONOMÍA

Tras el análisis realizado de los ataques presentados en la sección anterior, en la Tabla 1 se resumen una serie de características de todos ellos. Los ataques referenciados son los siguientes: Meltdown (M), Spectre Variante 1 (S1), Spectre Variante 2 (S2), Foreshadow (F) y ZombieLoad (Z). En el margen izquierdo, se encuentran las características estudiadas, y para cada uno de los ataques se indica en una casilla si la cumple o no. En caso de cumplirse, será indicado con un *check* verde. Si se cumple pero bajo ciertas restricciones, se indicará con un *check* amarillo. De lo contrario, no se cumple, se indicará con una cruz roja. Si se desconoce la característica, se indica con una interrogación.

Observando este resumen de las características de cada uno de los ataques, se propone una posible taxonomía, que permite clasificar los ataques estudiados en este trabajo a través de un diagrama de Venn (Figura 1). Esta clasificación divide los ataques estudiados en base a tres criterios. Estos criterios son, el mecanismo de especulación explotado, la necesidad de forzar la ejecución de un código en la víctima y la posibilidad de especificar la dirección de memoria atacada.

	Ataque				
	M	S1	S2	F	Z
Funciona desde espacio de usuario	✓	✓	✓	✓	✓
Funciona en Máquina Virtual	✓	✓	✓	?	?
Funciona de MV a host	✘	✘	✘	✘	✓
Funciona de MV a MV	✘	✘	✘	✘	✓
Afecta sólo a arquitectura Intel	✓	✘	✘	✓	✓
Afecta a varias arquitecturas (Intel, ARM, AMD, IBM..)	✘	✓	✓	✘	✘
Basado en predicción de saltos	✘	✓	✓	✘	✘
Basado en excepciones	✓	✘	✘	✓	✓
Extracción por canal lateral (Flush+Reload)	✓	✓	✓	✓	✓
Tiene control de la @ atacada	✓	✓	✓	✓	✘
Accede a @ memoria arbitrarias	✓	✓	✓	✓	✘
Permite romper SGX	✘	✘	✘	✓	✓
Necesita interacción de la víctima	✘	✘	✓	✓	✓

Cuadro 1: Resumen de características de los ataques basados en especulación estudiados

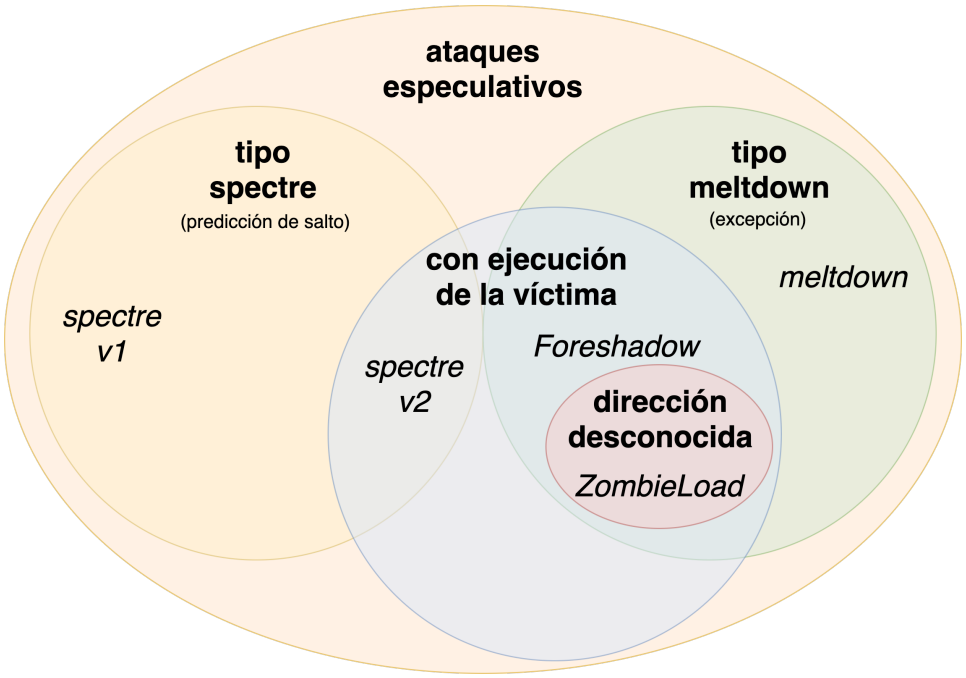

Figura 1: Posible taxonomía de los ataques estudiados. Todos los ataques excepto ZombieLoad conocen la dirección que están atacando

MELTDOWN Y SPECTRE

3.1 MELTDOWN

Como se explica en la Subsección 2.4.2, Meltdown permite a un atacante leer secretos pertenecientes a la memoria de otros usuarios o incluso del kernel. A continuación se detallan todos los aspectos implicados en el funcionamiento de este ataque, cuya comprensión es necesaria para entender las modificaciones propuestas en los apartados 3.2.2 y 3.2.3.

3.1.1 *Obtención de la dirección de memoria del secreto*

Como se comenta en Sección 2.1, con el fin de poder acceder y escribir las páginas con datos para el usuario, se aplica la técnica de *direct-physical mapping*, que mapea todas las direcciones físicas en espacio de kernel. Por otro lado, con el fin de optimizar el uso de algunas llamadas al sistema, todas las páginas del kernel son mapeadas en espacio de usuario. Esto provoca que todos los procesos tienen en su espacio lógico una copia de toda la memoria física utilizada por otros procesos, lo que aprovecha Meltdown para acceder a direcciones arbitrarias de memoria.

Aunque el ataque puede leer posiciones de memoria arbitraria y realizar un volcado de toda la memoria física en uso, para este trabajo se asume que se conoce la dirección de memoria en la que se encuentra el secreto. Para facilitar la experimentación, el proceso víctima proveerá la traducción de dirección virtual a física a través de su mapa de memoria ubicado en /proc/self/pagemap. Posteriormente, se ejecutará Meltdown sobre la dirección resultante de concatenar la dirección física con `0xffff880000000000` (dirección por defecto del *direct-physical mapping*), pues esa es la dirección en la que el atacante tendrá mapeado el secreto de la víctima.

3.1.2 *Preparación del canal lateral*

Tal y como se comenta en la Sección 2.4, Meltdown hace uso de un canal lateral para extraer el byte (secreto) del estado micro-arquitectónico al estado arquitectónico. De los diferentes ataques de canal lateral, Meltdown hace uso de Flush+Reload, que a través de la observación del tiempo de acceso a un vector de muestreo, es capaz de inferir el secreto robado por Meltdown.

Para poder realizar el Flush+Reload, el atacante debe disponer de un vector que será utilizado para muestrear los datos que se encuentran en la cache. Este vector (*probe_array*), es un vector formado por 256 * 4096 elementos de tamaño de un byte. Cada uno de los 256 elementos a muestrear se encuentra separado del siguiente por 4096 elementos, es decir, 4 KB. Se aplica esta separación aprovechando que el *prefetcher* no funciona a través de páginas, ya que éste podría traer elementos contiguos impidiendo detectar de manera clara el byte accedido.

Antes de cada iteración del ataque, el atacante debe asegurarse de que este vector no se encuentra en cache, de lo contrario podrían ser inferidos los resultados de manera incorrecta.

3.1.3 *Fases del ataque Meltdown*

El funcionamiento de Meltdown consta de tres fases bien diferenciadas, tal y como describen la Figura 2 y el Listado 2. Este código lo ejecuta exclusivamente el atacante sin intervención de la víctima.

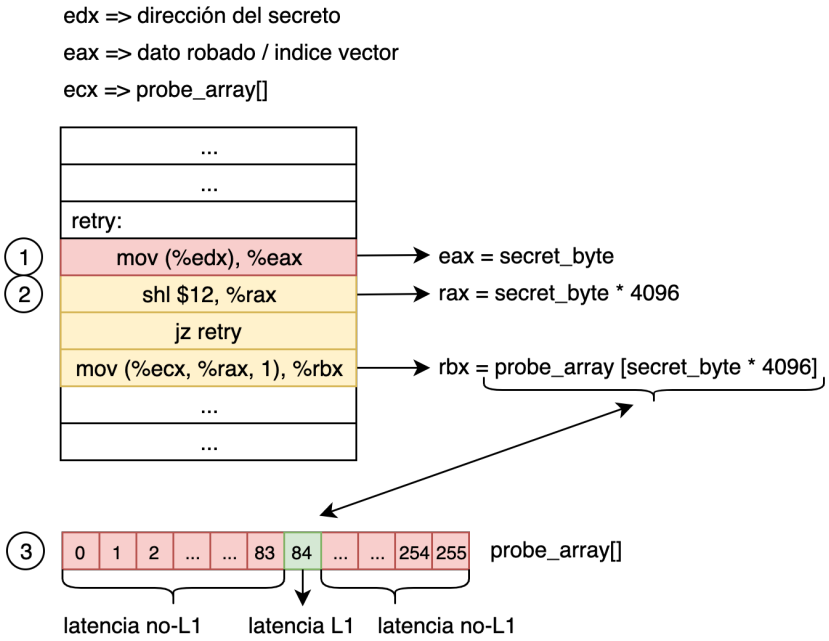

Figura 2: Diagrama de funcionamiento de Meltdown

```
1    retry:
2    mov (addr), %eax
3    shl $12, %rax
4    jz retry
5    mov (probe_array, %rax, 1), %rbx
```

Listado 2: Fragmento de código de Meltdown que realiza el robo de información

Fase 1. Durante la primera fase del ataque, el atacante accede a la dirección del secreto de la víctima (línea 2), que se encuentra mapeado en el espacio del atacante, como se indica en la Subsección 3.1.1. Dado que la comprobación de que el atacante tiene permisos para acceder a la dirección `addr` es un proceso lento, el procesador especula asumiendo que tiene permisos, y por lo tanto accede a `@addr` y carga el dato obtenido en el registro `eax`. Sin embargo, este acceso provocará una excepción que forzará el *rollback* de todas las instrucciones posteriores a la última instrucción retirada correctamente que se encuentren en el ROB, pero como se comenta en Subsección 2.2.3, esta excepción no será tratada hasta la fase de *retirement*. Esto produce una condición de carrera desde que se accede al dato hasta que se anula la ejecución, siendo ésta la ventana del ataque.

Fase 2. Durante esta fase, tras el acceso de memoria, las instrucciones de las líneas 3 y 5 serán ejecutadas especulativamente, antes de que sean revertidas por el tratamiento de la excepción. Una vez el secreto de la víctima está cargado en `rax` (o por lo menos disponible mediante encadenamiento de instrucciones), se multiplica su valor por 4096, realizando un *shift* de 12 bits y almacenándolo de nuevo en `rax`. Tras ésto, la instrucción de la línea 5 será la encargada de acceder al *probe_array* utilizando `rax` como índice. Aquí hay que recordar que *probe_array* ha sido expulsado de la cache antes del ataque, y `rax` contiene el valor del secreto multiplicado por 4096. Esto provocará que el elemento `probe_array[secreto * 4096]` sea llevado a memoria cache. La instrucción de la línea 4 reintenta el ataque si el byte leído se corresponde con un `0`, con el fin de asegurar que no ha ocurrido un fallo en la lectura.

Fase 3. Finalmente, durante la tercera fase del ataque, el atacante aplica las técnicas utilizadas por Flush+Reload sobre su propio vector de muestreo (*probe_array*) para inferir el dato robado. Para ello, haciendo uso del contador de ciclos de alta precisión de Intel (TSC), se monitoriza el tiempo de acceso a cada uno de los 256 elementos a muestrear. Cuando el tiempo de acceso de alguno de ellos es similar al tiempo de latencia de cache L1, quiere decir que ese elemento se encontraba en la cache, a diferencia de los demás elementos del vector. Si la fase 2 ha concluido satisfactoriamente, el elemento `probe_array[secreto * 4096]` debería estar en cache, resultando en un tiempo de acceso mucho menor que los demás. Dado que el índice de dicho elemento es el valor del secreto multiplicado por 4096, deshaciendo la multiplicación se obtiene, en estado arquitectónico, el valor del secreto extraído de la víctima.

Para mayor certeza de que el byte extraído es el correcto, estas tres fases pueden ser repetidas continuamente tantas veces como se considere necesario, siempre y cuando se use un método para manejar la excepción y evitar la finalización del programa atacante.

3.1.4 *Tratamiento de la excepción*

Dado que la ejecución de este ataque se basa en un acceso a una dirección inaccesible para el usuario, se provocará una excepción que finalizará el proceso cada vez que se ejecute. Sin embargo, la posibilidad de extraer un dato es mayor cuantos más ataques se realizan, siendo necesario tratar la excepción para poder continuar la ejecución y reintentar el ataque sobre el mismo byte nuevamente.

Para tratar la excepción existen dos alternativas, hacer uso de las extensiones de sincronización transaccional (TSX) o el uso de un manejador de la señal (*signal handler*). En este trabajo solo se abarca el tratamiento de la excepción mediante un *signal handler*. Para ello, se programa el manejador de manera que cuando se detecte una señal SIGSEGV (violación de segmento), éste la atrape y la suprima, para posteriormente continuar con una nueva ejecución del ataque.

3.2 ESTUDIO Y CARACTERIZACIÓN DE MELTDOWN

3.2.1 *Meltdown M1*

En esta sección se define Meltdown M1, una variante del ataque Meltdown original desarrollada para este trabajo, que permite explotar algunas de sus características para mejorar, en términos generales, el rendimiento y la eficacia de Meltdown.

3.2.2 *Llenado de estructuras internas del procesador*

Durante las pruebas realizadas con el PoC original [6], los resultados carecían de consistencia, dado que el ataque sólo funcionaba en ocasiones puntuales, y obteniendo resultados erróneos la mayor parte de las veces que se conseguía ejecutar.

Dado que la biblioteca del PoC añade una sobrecarga para poder generalizar el código en distintas arquitecturas (x64 y x86), detectar automáticamente la utilización de TSX, y otras configuraciones, se han programado un código víctima y una versión mínima de Meltdown con la que empezar a trabajar, sin dicha sobrecarga. El código resultante se encuentra en el anexo Sección A.4.

Sin embargo, esta primera versión simplificada no funciona, por lo que se propone hacer un estudio más detallado del funcionamiento de las instrucciones del ataque. Una intuición es que las instrucciones ya presentes en la ventana de instrucciones podían interferir con la adecuada secuenciación de las instrucciones que conforman el ataque. Se pensó entonces en proveer al ataque un escenario en el que disponía de vía libre para ejecutarse sin interferencias. Por este motivo, se procede a aislar las instrucciones de Meltdown mediante la introducción de una cadena de dependencias verdaderas justo antes de la ejecución del ataque. Con ello, estructuras internas del procesador como la estación de reserva o el ROB se llenan de instrucciones que deben ser ejecutadas secuencialmente, vaciando así el *pipeline* de ejecución fuera de orden. Con el *pipeline* vacío, para el momento en el que el ataque entre en la ventana de lanzamiento (o *Unified reservation station*), se encontrará todas las unidades funcionales disponibles, y podrá ejecutarse inmediatamente. Esta cadena de dependencias debe ser una instrucción que no interfiera con los puertos utilizados por el ataque, como podrían ser instrucciones del tipo `ADD $1, %rax` (ver Figura 3).

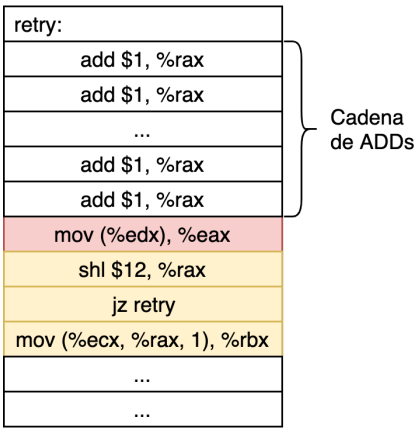

Figura 3: Código de Meltdown con cadena de dependencias

Como se verá en el Capítulo 5, la cadena de ADDs mejora el rendimiento del ataque gracias al aislamiento de las instrucciones del mismo.

3.2.3 *Optimización de la temporización de Flush+Reload*

Con el fin de detectar por qué no funciona el ataque el 100% de las veces, se decide analizar en profundidad el binario generado haciendo un desensamblado del mismo. Observando la función intrínseca `__rdtscp()`, ésta incluye mucha sobrecarga a la hora de leer el TSC, lo que provoca que a veces la lectura no sea precisa, y por lo tanto dando por no presente el dato. Por este motivo, se reescribe el código ensamblador mínimo necesario para realizar un `rdtscp`, eliminando la sobrecarga. Con este cambio, la lectura del TSC es más precisa y constante, pero siguen existiendo instrucciones que pueden retrasar el acceso al dato para su muestreo (por ejemplo, otros loads en ejecución ocupando el puerto). Para solucionar este problema, se plantea la introducción de instrucciones `NOP`, que se encargarán de vaciar el *pipeline*, pero que serán retiradas antes de entrar en la ventana de lanzamiento (o *Unified reservation station*). Esto provoca que, para cuando se desea monitorizar el tiempo de acceso a memoria mediante Flush+Reload, el acceso al elemento del vector sea el único load en ejecución, y por lo tanto mejorando la precisión del conteo de ciclos. De esta manera, se consigue optimizar tanto la eficacia del ataque como su tasa de éxito, tal y como se explica en el Capítulo 5.

3.2.4 *Aplicación de estas variaciones sobre el ataque original*

Tras haber optimizado la variante Meltdown M1, se vuelve a intentar la ejecución del PoC original, pero ésta vez aplicando las mismas técnicas de optimización que se han desarrollado para Meltdown M1. Tras aplicarlos, la inconsistencia de funcionamiento del ataque original desaparece, y se consiguen unos resultados muy aproximados a los alcanzados con el Meltdown M1.

Analizando más en detalle la biblioteca original, en ésta se hace uso de varios *threads* que se dedican a introducir NOPs constantemente. Sin embargo, estos *threads* añadían demasiada sobrecarga, y el ataque estaba siendo expulsado del hilo de ejecución continuamente. Por ello, tras eliminar los *threads*, e incluir las mejoras comentadas anteriormente, la eficacia del Meltdown original mejora considerablemente.

3.3 SPECTRE

Aunque existen distintas variantes de Spectre, en esta sección se profundiza en el funcionamiento de la variante 1 de este ataque.

3.3.1 *Preparación del ataque y el canal lateral*

Como se comenta en la Sección 2.4, Meltdown y Spectre tienen un comportamiento muy similar, con la principal diferencia de que Spectre aprovecha la ejecución especulativa del salto para ocultar la excepción. Por este motivo, este ataque debe prepararse tal y como se comenta en la Subsección 3.1.2.

Esta preparación incluye, además del vector de muestreo de 256 elementos (*probe_array*) para la extracción del dato mediante Flush+Reload, la creación de un nuevo vector que será el encargado de acceder al dato durante la especulación del salto. Tanto este vector (*array1*), como el vector de muestreo (*probe_array*), son internos del código del atacante.

Además, para facilitar la experimentación con Spectre, también se parte de conocer la dirección en la que se encuentra el secreto de la víctima, de la misma manera que se realiza para las pruebas de Meltdown (ver Subsección 3.1.1).

3.3.2 *Fases del ataque*

El ataque Spectre en su variante 1 tiene tres fases diferenciadas. Con el fin de entender mejor el ataque, se incluye el código necesario para el ataque en el Listado 3. Al igual que Meltdown, este ataque no necesita que la víctima realice nada, ya que todo el ataque se realiza desde el espacio de usuario del atacante.

```
1 if (x < array1_size)
2       temp = probe_array[array1[x] * 4096];
```

Listado 3: Spectre Variant 1

Fase 1. Durante la primera fase del ataque, el atacante ejecutará continuamente el segmento de código del Listado 3 con un valor de `x` dentro de los límites del vector `array1`. De esta manera la condición de la línea 1 siempre será cierta, y el salto no será tomado. Este comportamiento, indirectamente, está entrenando la estructura de almacenamiento del procesador encargada de predecir los saltos.

Fase 2. En la segunda fase, el atacante sustituirá el valor de `x` por una `x` maliciosa, que apunta a una dirección fuera de los índices del vector, la dirección del secreto de la víctima. Dado que el valor de la variable `array1_size` se encuentra en cache, la comprobación de la condición sería demasiado rápida y bloquearía el ataque. Es por esto que el atacante expulsará de cache la variable de comprobación del límite del vector (`array1_size`). Esto provocará que la siguiente vez que se ejecute instrucción de la línea 1, el dato `array1_size` tenga que ser traído desde memoria principal, obligando al procesador a especular en los distintos caminos de ejecución disponibles: salto tomado o no tomado.

En este momento, el predictor de salto especula que el salto no será tomado, en base al entrenamiento introducido en la fase 1. Esto provocará el acceso de manera especulativa a la dirección del secreto fuera de los índices de `array1`. Al igual que Meltdown, el valor obtenido por el acceso es utilizado para indexar el vector de muestreo (que se encuentra fuera de cache), de manera que el elemento `probe_array[secreto * 4096]` es traído a cache.

En condiciones normales, el acceso de memoria especulativo provocaría una excepción, forzando el *rollback* de todo el ROB. Sin embargo, al haberse producido en el interior de un camino especulativo de un salto, estos cambios son invalidados y continúa la ejecución por el camino correcto, sin tratar la excepción. Por este motivo, el rendimiento de Spectre es muy superior al de Meltdown, ya que no es necesario atrapar la excepción en cada uno de los intentos, lo que implica muchos

ciclos perdidos.

Fase 3. Finalmente, dado que durante la fase 2, el elemento `probe_array[secreto * 4096]` fue accedido especulativamente, los rastros de este acceso aún son visibles micro-arquitectónicamente. De igual manera que en Meltdown, se monitoriza el tiempo de acceso a cada uno de los elementos del *probe_array*, y si alguno de ellos tiene latencia L1 a diferencia de los demás, el secreto robado a la víctima corresponde al resultado de deshacer la multiplicación del índice.

Este ataque también se puede repetir tantas veces como sea necesario, y en este caso no es necesario el tratamiento de excepciones, ya que es el propio error de la especulación quien se encarga de deshacer los cambios.

METODOLOGÍA

En este capítulo se detalla la plataforma utilizada para la experimentación junto con sus características más significativas para el caso de estudio de este trabajo. Además, se comenta su preparación y las herramientas utilizadas para llevar a cabo la experimentación.

4.1 PLATAFORMA

La máquina utilizada para las pruebas es un Intel i7-7700 @ 3.6 GHz, de cuatro núcleos y ocho procesos lógicos (*threads*) y 32 GB de memoria RAM, y ejecuta un sistema Operativo CentOS Linux RHEL 7.0, actualizado en su versión 3.10.0-862.11.6.el7.x86_64. Dado que para este trabajo es relevante, el procesador parte de tener la versión de microcódigo `0x8e`, en la que Meltdown y Spectre se encuentran parcheados.

Este procesador pertenece a la familia de procesadores Kaby Lake H, que mantiene la misma micro-arquitectura que la familia anterior, Skylake. Esta micro-arquitectura dispone de un *Frontend* capaz de buscar 16 B por ciclo e introducir 6 µOPs al *Backend*(ROB) por ciclo. Este ROB cuenta con 224 entradas, y está comunicado con la ventana de lanzamiento (96 entradas), que dispone de 8 puertos para el lanzamiento de instrucciones fuera de orden. Las unidades funcionales, la ventana y el ROB están comunicados a través de un CDB. Este procesador es capaz de retirar instrucciones a 4 µOPs por ciclos por hilo. En la Sección A.3 se encuentra un diagrama detallado de la micro-arquitectura de este procesador.

Este procesador cuenta con caches L1I y L1D de 32KB compartidas por núcleo, a latencia de 5 ciclos para loads, 256 KB de cache L2 no-inclusiva, con latencia de acceso 12 ciclos y 2 MB por núcleo de LLC no-inclusiva compartida entre todos los núcleos, a latencia 42 ciclos.

4.2 HERRAMIENTAS

Para la realización de este trabajo, tan solo se han utilizado tres herramientas externas, puesto que el grueso del trabajo ha sido la experimentación con programas propios desarrollados en lenguaje C.

La primera de estas herramientas Radare2 [15], un software de desensamblado de binarios y análisis y depuración de código, utilizado comúnmente para *reverse-engineering* y hallar vulnerabilidades en binarios. Se ha hecho uso de esta herramienta para poder analizar el binario de Meltdown y ejecutarlo de manera controlada para probar comportamientos.

La siguiente herramienta utilizada ha sido el software de *benchmarking* stress-ng, utilizado para medir el impacto en el rendimiento de los parches publicados para

estos ataques.

Finalmente, se han realizado pruebas en máquinas virtuales corriendo en Qemu-KVM.

4.3 PREPARACIÓN DEL ENTORNO

Para preparar el entorno para la ejecución de estos ataques, existen dos alternativas. La primera de ellas consiste en desactivar las mitigaciones del kernel con los flags `nopti` y `nokaslr` en la secuencia de arranque de grub, dejando el sistema sin las protecciones activadas. Para hacer estos cambios persistentes, existe la posibilidad de editar el fichero /etc/default/grub indicándole estos flags, y regenerar el grub.cfg para que se aplique en todas las secuencias de arranque.

Sin embargo, con el fin de exponer más el kernel a estos ataques, se ha realizado la segunda alternativa, que consiste en realizar un downgrade el kernel a una versión vulnerable, como podría ser la 3.10.0-327.el7.x86_64. Además, también se ha forzado la carga de una versión de microcódigo vulnerable en el arranque, la versión `0x5e`.

Una vez realizado esto, la máquina queda totalmente vulnerable a los todos ataques basados en ejecución especulativa desde la aparición de Meltdown y Spectre.

4.4 MÉTRICAS DE EXPERIMENTACIÓN

Para los experimentos realizados en el Capítulo 5, se han definido las siguientes métricas:

- **Rendimiento**: Medido en B/s, corresponde a la cantidad de bytes que pueden ser extraídos por el ataque. Estos bytes pueden ser correctos o incorrectos, dado que el atacante nunca tendrá una forma de validarlo.
- **Tasa de éxito**: Se define como tasa de éxito el porcentaje del total de ataques sobre un mismo byte que han finalizado con éxito. Esto es, que un byte ha llegado a ser extraído mediante un canal lateral y se ha filtrado al estado arquitectónico. El byte extraído puede ser correcto o incorrecto.
- **Eficacia**: Expresa el porcentaje de los bytes que se han extraído que coinciden con el secreto esperado. Es decir, el total de bytes secretos válidos que se han extraído en un ataque completo.

EXPERIMENTACIÓN Y ANÁLISIS

5.1 EXPERIMENTACIÓN

En este capítulo se presentan y analizan los resultados de las diferentes variantes y modificaciones realizadas sobre los *Proof-of-Concept*(PoC), buscando mejorar el rendimiento de los ataques lo máximo posible, y estudiando los mecanismos implicados en estas mejoras.

Debido a la limitación temporal de un Trabajo de Fin de Grado, el estudio se ha centrado únicamente en la validación de los dos ataques presentados en el Capítulo 3, Meltdown y Spectre, principalmente en la caracterización de Meltdown.

Los resultados comentados en este capítulo se basan en las 3 métricas definidas en la Sección 4.4: **rendimiento** del ataque en bytes extraídos por segundo, la **eficacia** del ataque mediante el porcentaje de bytes correctos extraídos (respecto al secreto original), y el porcentaje de veces que el ataque se ha realizado con éxito sobre cada byte, es decir, el número de veces que ha sido capaz de detectar el byte en la cache (**tasa de éxito**).

5.1.1 *Flush+Reload*

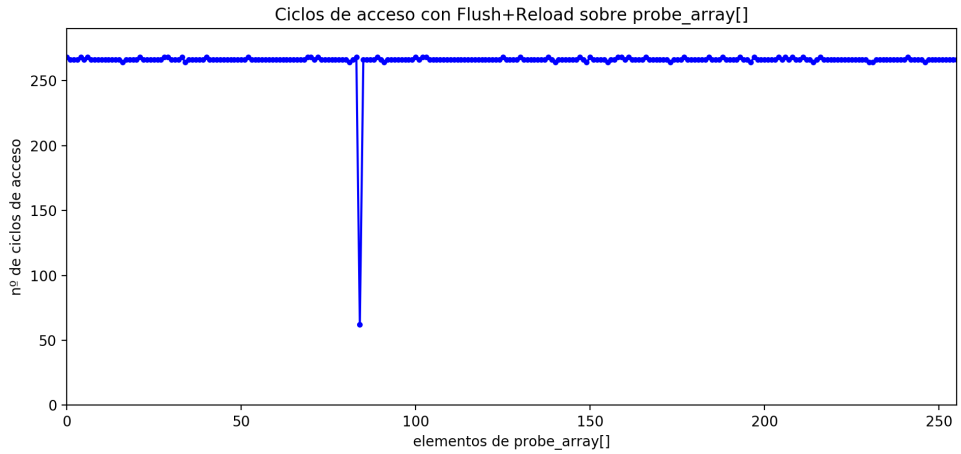

Figura 4: Tiempo de acceso (en ciclos) a cada uno de los datos de *probe_array* medido por FLUSH+RELOAD.

En el Capítulo 2 se indica que gran parte de los ataques basados en especulación hacen uso de Flush+Reload como canal lateral para extraer al estado arquitectónico los secretos. Este ataque de canal lateral permite inferir el secreto robado a través de la observación del tiempo de acceso a un array de muestreo del atacante, previamente accedido durante el ataque. La Figura 4 presenta los resultados de realizar Flush+Reload sobre los 256 bytes del vector de muestreo (*probe_array*). Para cada valor del vector, se obtienen los ciclos que ha tardado el dato en estar

disponible, tardando 59 ciclos para el byte 84 (carácter "T"), mientras que el resto se encuentran por encima de los 250 ciclos. Aunque la latencia del dato en L1 se indique como 59 ciclos, ésta incluye el tiempo de sobrecarga aportado por las instrucciones que realizan el rdtscp, que son aproximadamente entre 45 y 50 ciclos (en la arquitectura de la máquina testeada).

5.1.2 *Llenado de estructuras internas del procesador*

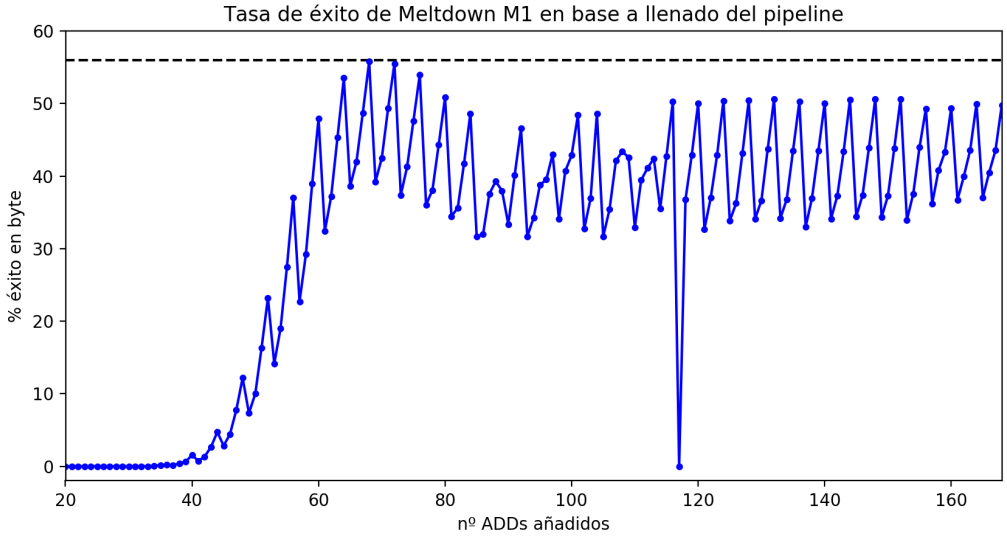

Figura 5: Tasa de éxito (ataque completado) sobre un byte con distintas configuraciones de llenado de la ventana de lanzamiento con ADDs antes de la ejecución de Meltdown.

Como se comenta en la Subsección 3.2.2, se ha estudiado el efecto que provoca añadir diferentes longitudes de cadenas de dependencias antes del ataque. La Figura 5 representa, en el eje X, el número de ADDs dependientes introducidos antes del ataque. El eje Y representa la mediana de 50 ensayos de la tasa de éxito del ataque. Cada ensayo intenta extraer un byte del secreto 1000 veces consecutivas.

A la vista de los resultados, la realización del ataque aplicando la técnica descrita influye directamente en la cantidad de veces que el ataque es capaz de extraer el dato. Con cadenas inferiores a 40 instrucciones la tasa de éxito es nula, pues no consigue leer prácticamente ningún byte. Para cadenas entre 40 y 70 instrucciones, a medida que se va llenando la estación de reserva, los resultados mejoran considerablemente, llegando a leer un byte correctamente el 56% de los intentos con 68 ADDs antes del ataque. Para cadenas superiores a este número, este porcentaje decae un poco y se mantiene constante a ese valor. Sin embargo, en esta gráfica destacan varios aspectos. El primero de ellos es que ocurre un patrón muy regular, en la que el ataque funciona mejor con las cadenas que son múltiplos de 4, mientras que en los demás valores decae el resultado. Aunque se han estudiado varias alternativas capaces de causar de este comportamiento, todavía no se ha podido validar ninguna de ellas, por lo que se propone como trabajo futuro. El Capítulo 6 detalla en mayor profundidad las posibles causas. Por otro lado, destacan un pico negativo en el valor 117 así como una sección irregular desde 84 hasta 117 ADDs, descrito en las siguientes secciones.

5.1.3 *Número de intentos por byte*

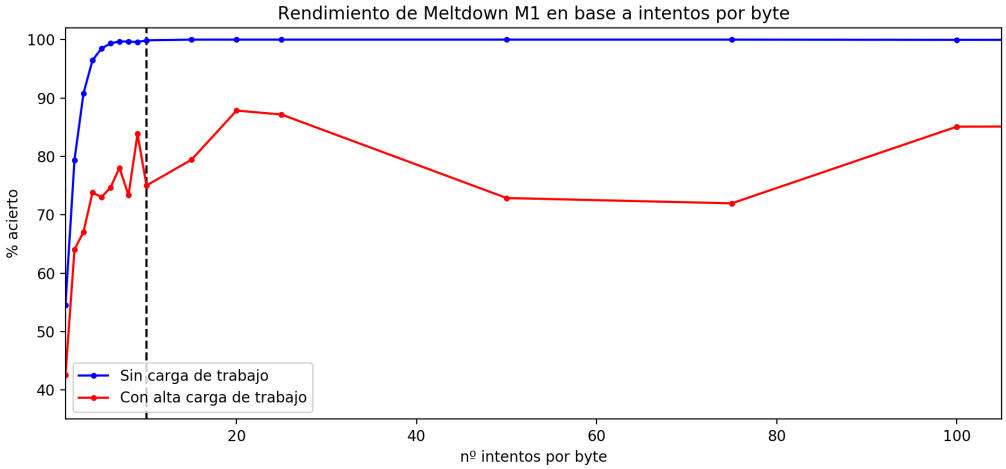

Figura 6: Porcentaje de bytes de la cadena final leídos correctamente en base al número de intentos (*probes*), sin aislamiento de lectura (NOPs).

En este apartado se intenta caracterizar cuántos intentos son necesarios para tener una buena eficacia en el ataque, y la impacto que tiene una fuerte carga de trabajo en el sistema en este número de intentos.

La Figura 6 representa el porcentaje de bytes leídos correctamente (eje Y) de una cadena secreta arbitraria de otro proceso en base al número de intentos (*probes*) realizados sobre cada byte. La línea azul representa las pruebas sin carga de trabajo, mientras que la roja representa las pruebas con carga de trabajo en todos los núcleos del procesador. Todas las pruebas se han realizado con 68 ADDs antes del ataque, dado que es el valor óptimo de tasa de éxito observado en el apartado anterior.

Sin carga de trabajo, Meltdown M1 extrae correctamente aproximadamente el 55% de la cadena realizando la lectura tan solo *una* vez. De esta manera, se puede volcar la memoria de un proceso hasta 30 KB/s. A medida que se aumenta el número de lecturas sobre cada byte, la eficacia del ataque aumenta, obteniendo un 100% de acierto a partir de las 10 lecturas por byte a una velocidad de 3 KB/s.

Por otro lado, cuando el procesador está sometido a una carga de trabajo intensa la eficacia del ataque se ve afectada, ya que al haber múltiples procesos accediendo a memoria es más difícil muestrear la cache y detectar el dato que se quiere extraer, lo que implica en un menor porcentaje de bytes correctos extraídos. Esto es así especialmente cuando se realiza un número pequeño de intentos sobre cada byte. En este caso, aumentar el número de intentos no asegura al 100% una mayor eficacia del ataque, pues sigue existiendo la posibilidad de leer datos basura a causa del ruido generado por los datos de otros procesos en el muestreo del vector.

5.1.4 *Precisión de Flush+Reload*

Tras optimizar el método de temporización de Flush+Reload (Subsección 3.2.3), se han repetido los experimentos realizados en las secciones anteriores, dando como resultado las Figuras 7 y 8.

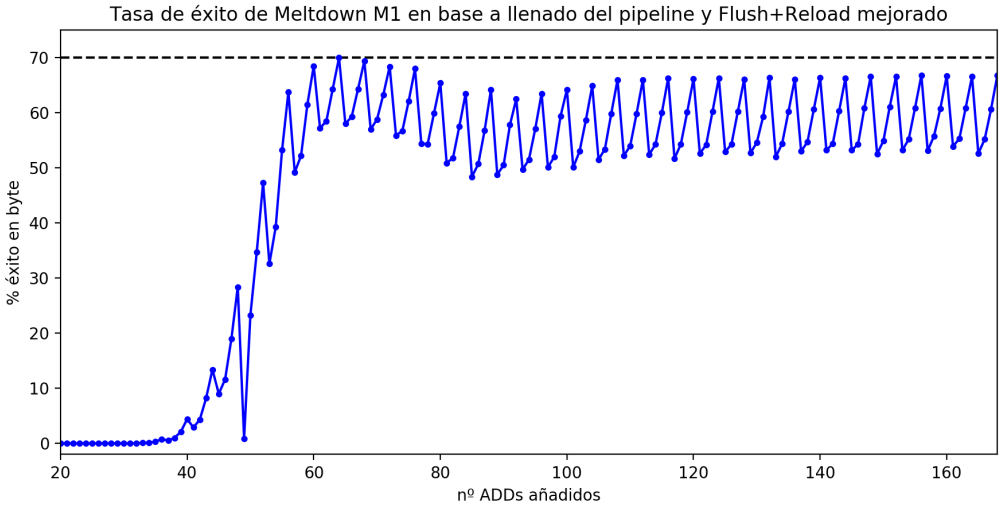

Figura 7: Tasa de éxito (ataque completado) sobre un byte con distintas configuraciones de llenado de la ventana de lanzamiento con ADDs antes de la ejecución de Meltdown, con aislamiento de lectura (NOPs).

La Figura 7 corresponde a la Figura 5, pero incluyendo un aislamiento en la temporización de la cache al incluir 500 NOPs antes de la realización del canal lateral Flush+Reload, junto con el valor óptimo de instrucciones encadenadas comentadas en el apartado anterior para cada uno de los intentos realizados.

Se puede ver que en las dos figuras se comparte el mismo patrón de funcionamiento hasta la cadena de 80 ADDs, aunque con una mayor tasa de éxito con esta optimización. A diferencia del experimento anterior, esta figura presenta mayor regularidad ya que al optimizar la temporización de Flush+Reload incluyendo un número suficiente de NOPs en el *pipeline* es posible aislar cada una de las lecturas, reduciendo el ruido, y aumentando consecuentemente la precisión del ataque y el porcentaje total de ataques llevados a cabo correctamente. Por otro lado, en esta figura se puede observar un pico negativo en el 49, al igual que aparece en el 117 en la Figura 5. Al igual que ocurre con la tasa de éxito, no se puede asegurar que a qué se deben estos picos, y no se han podido validar las alternativas propuestas. El estudio de este comportamiento se deja como trabajo futuro (Capítulo 6).

La Figura 8 corresponde a la Figura 6, pero incluyendo un aislamiento en la temporización de la cache al incluir 500 NOPs antes de realizar cada lectura mediante el canal lateral Flush+Reload. Al optimizar la temporización de Flush+Reload se percibe un aumento considerable en la eficacia del ataque, dado que con sólo 4 intentos lectura, Meltdown M1 es capaz de leer el 100% de los bytes correctamente. Sin embargo, esta sobrecarga de instrucciones tiene un impacto considerable en la eficiencia del mismo, siendo capaz de leer datos a una velocidad de 7,6 KB/s.

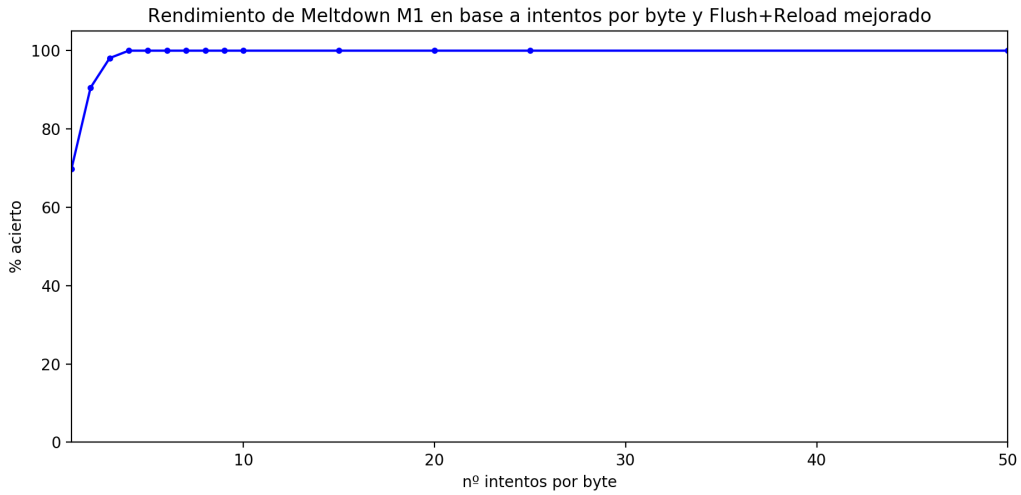

Figura 8: Porcentaje de bytes de la cadena final leídos correctamente en base al número de intentos (*probes*), con aislamiento de lectura (NOPs).

5.1.5 *Tabla comparativa de Meltdown y Meltdown M1*

Variante	intentos	Rendimiento (B/s)	Eficacia (% bytes Icorrectos)	Tasa de funcionamiento (%)
Meltdown M1 (ADDs)	1	30804,34	55,66	100
	5	6253,765	98,11	60
	50	625,705	100	58
	100	315,355	100	57
	1000	31,81	100	57
Meltdown M1 (ADDs + NOPs)	1	26071,31	69,81	100
	5	5259,05	100	80
	50	529,09	100	72
	100	265,03	100	72
	1000	26,69	100	71,4
Meltdown Original	-	96,796	100	-
Meltdown Original (ADDs)	-	2078,37	100	-
Meltdown Original (ADDs + nothread)	-	4324,49	100	-
Meltdown Original (ADDs + NOPs)	-	2707,2	100	-
Meltdown Original (ADDs + NOPs + nothread)	-	5362,08	100	-

Cuadro 2: Comparación de resultados en las diferentes variantes de Meltdown

La Tabla 2 resume todos los resultados comentados en las secciones anteriores de este capítulo. La primera de las variantes de esta tabla es la variante propia de Meltdown M1, que incluye la cadena de dependencias que llenan las estructuras internas del procesador (Subsección 3.2.2). El segundo ataque, corresponde al mismo Meltdown M1, pero incluyendo la mejora que permite aumentar la precisión de Flush+Reload (Subsección 3.2.3). La entrada clasificada como "Meltdown Original", corresponde al código PoC proporcionado por los investigadores [6]. Las entradas siguientes corresponden a este PoC, pero incluyendo progresivamente las mejoras desarrolladas en Meltdown M1, como los ADDs (llenado de estructuras) y los NOPs (optimización de temporización). Además, se incluyen también las variantes que, como se comenta en la Subsección 3.2.4, eliminan los *threads* utilizados por el PoC.

En esta tabla se puede ver que al aplicar las técnicas desarrolladas, Meltdown M1 con ADDs y NOPs, con 5 intentos es capaz de extraer datos 50 veces más rápido que la versión original sin optimizaciones, con la misma eficacia del 100%. Esta mejora también se puede observar en la última variante, resultado de aplicar las mejoras directamente sobre el PoC. Por otro lado, usando Meltdown M1 con 1 intento, es posible conseguir tasas de extracción de datos muy altas, de entre 26 a 30 KB/s, a cambio de perder parte de la precisión en los datos.

Cabe destacar que en el artículo oficial de Meltdown [9], ellos mencionan que utilizando *signal handling*, consiguen una eficacia de 103 KB/s, con un error del 0,03% de los datos. Es posible que no se hayan podido alcanzar estos resultados ya que estos ataques van fuertemente ligados a la micro-arquitectura de la máquina, y no especifican en cuál de las máquinas que prueban obtienen estos resultados.

5.1.6 *Resultados de la prueba de concepto de Spectre*

intentos	rendimiento (B/s)	eficacia (% bytes correctos)
1	26572,66	90,56
5	5353,92	96,22
10	2675,55	100
15	1784,19	100

Cuadro 3: Resultados de la ejecución de Spectre

Partiendo del PoC disponible en un gist anónimo [3], se han introducido en el código las modificaciones necesarias para obtener las métricas de rendimiento y eficacia para Spectre. La Tabla 3 representa la mediana de 50 ejecuciones de Spectre, repitiendo el ataque 1, 5, 10 y 15 veces sobre cada byte. Como es de esperar, el rendimiento es mucho mayor con menos intentos, pudiendo volcar memoria a 26 KB/s en el mejor de los casos. Por otro lado, cuando se aumenta el número de intentos, la eficacia aumenta a costa de una ligera caída de rendimiento.

5.2 MITIGACIONES

En esta sección se comentan los parches oficiales desplegados para mitigar Meltdown y Spectre, así como una posible nueva manera de reducir el impacto del ataque, propuesta en este trabajo.

5.2.1 *Kernel*

Los dos ataques presentados en esta sección se aprovechan de dos optimizaciones hechas por el sistema operativo: el mapeo del kernel dentro de un proceso en espacio de usuario, y a su vez, el mapeo de toda la memoria física en el kernel (Sección 2.1). Desde hace tiempo, el kernel de Linux dispone de una técnica denominada *Kernel Address Space Layout Randomization* (KASLR), que en cada arranque del sistema aleatoriza las direcciones del espacio de direccionamiento del kernel. Esto permite colocar la dirección base del *direct-physical mapping* en una dirección diferente cada vez, pero no es suficiente ya que con unos pocos barridos de memoria, un atacante puede encontrar la nueva dirección base de este mapeo.

Para mitigar estos ataques, Gruss et al. [4] proponen una solución denominada KAISER , posteriormente bautizada por el parche como *Kernel Page Table Isolation*(KPTI), que reduce el mapeo del kernel en espacio de usuario a únicamente lo necesario, y elimina el *direct-physical mapping* del kernel. Este parche es capaz de mitigar completamente Meltdown, y la parte de Spectre que permite acceder a la memoria de otros procesos utilizando la misma técnica.

Sin embargo, este parche tiene un impacto considerable en el rendimiento, especialmente en los escenarios en los que hay múltiples llamadas al sistema virtuales implicadas.

Otra de las mitigaciones proporcionadas por el kernel, es la inclusión de máscaras de comprobación de validez en el índice de un vector, antes de realizar un acceso. Estas máscaras bit a bit son capaces de comprobar rápidamente si el valor con el que se indexa un vector es válido o no. Esto consigue mitigar completamente la variante 1 de Spectre, sin afectar demasiado al rendimiento del procesador.

Tras ejecutar de nuevo Meltdown y Spectre con las mitigaciones del kernel activadas, la eficacia del ataque se reduce al 0%, ya que es ahora es imposible disponer del secreto de otro proceso mapeado en el espacio de usuario del atacante.

5.2.2 *Firmware*

A la par que las mitigaciones software, se han publicado parches de microcódigo que actualizan el firmware del procesador, y la forma en la que las instrucciones son decodificadas en µOPs. Estos parches, principalmente han sido enfocados a la protección de ataques como Spectre V2, Foreshadow y ZombieLoad.

Tras ejecutar Meltdown y Spectre con este parche, el resultado obtenido ha sido muy similar al de la ejecución con microcódigo vulnerable. Esto se debe principalmente a que la principal vulnerabilidad que explotan Meltdown y Spectre es vía software. Este resultado es el esperado, dado que este parche va enfocado a mitigar los ataques que no se han probado en este trabajo.

5.2.3 *Paralelismo del procesador*

A la vista de los resultados de la Figura 6, en los que tener el sistema con alta carga de trabajo parece aplacar la eficacia del ataque, una posible mitigación o contra-medida que se propone en este trabajo es la explotación del paralelismo del procesador. Si se fuerza al procesador a tener todos los puertos ocupados, las posibilidades de que la temporización del ataque se den de la manera idónea para que funcione se reducen, además del ruido generado en cache que dificulta la detección del dato robado.

Esta contra-medida no pretende bloquear ni proteger contra el ataque, pero podría reducir los daños provocados sin sufrir las impactos de las mitigaciones anteriores.

5.3 IMPACTO DE LAS MITIGACIONES

Para finalizar la fase de experimentación, con el fin de medir el impacto que tienen en el rendimiento del sistema los parches desplegados para mitigar estos ataques, se ha hecho uso de la herramienta *stress-ng* (Sección 4.2) para ejecutar los diferentes *benchmarks*. Estas pruebas tienen como objetivo caracterizar la pérdida de rendimiento que supone tener activadas la mitigaciones de la Sección 5.2.

Para la primera de las pruebas se valorará cómo afectan estas mitigaciones al rendimiento en aplicaciones intensivas en cálculo. La Tabla 4 indica el rendimiento del procesador, en instrucciones por ciclo (IPC) para los diferentes estados de protección del sistema.

benchmark - CPU	Vulnerable	Parche microcodigo	Parche kernel	Ambos parches
IPC obtenido	1,267	1,27	1,26	1,265

Cuadro 4: Resultados del *benchmark* de CPU en cálculo

En este caso, ninguno de los parches provoca una caída de rendimiento notable en el procesador.

Para la siguiente prueba, se estudiará como afectan las mitigaciones, especialmente el KPTI, a la ejecución de las llamadas al sistema como getclock, gettimeofday o getpid. Dado que el KPTI elimina la característica de mapeo de kernel en espacio de usuario, se espera que el rendimiento en esta prueba se vea afectado considerablemente.

benchmark - Syscall	Vulnerable	Parche microcodigo	Parche kernel	Ambos parches
IPC obtenido	0,736	0,734	0,243	0,243

Cuadro 5: Resultados del *benchmark* de CPU en llamadas al sistema

Como se puede ver en la Tabla 5, el impacto que sufre el rendimiento en el uso de estas llamadas al sistema a raíz de esta mitigación es considerable.

CONCLUSIONES Y TRABAJO FUTURO

6.1 CONCLUSIONES

La cantidad y la importancia de la información que procesamos con computadoras requiere garantizar su seguridad y confidencialidad. Estudiar el cumplimiento de estas propiedades es arduo debido a la gran complejidad de los computadores. Este TFG analiza varios ataques especulativos a procesadores para poder reproducir su comportamiento, y ha requerido un profundo estudio previo de todos los mecanismos implicados en el funcionamiento de los procesadores así como los ataques mismos, además de la anterior preparación de un entorno vulnerable en que puedan ser reproducidos. Para facilitar la comprensión de los ataques, este TFG agrupa buena parte de la literatura reciente sobre los mismos y aporta una sencilla taxonomía sobre ellos.

Tras la puesta en marcha del entorno experimental, se han comprendido varios ataques en profundidad lo que ha permitido caracterizar y mejorar una implementación del ataque Meltdown. Estas mejoras permiten al ataque rendir del orden de 50 veces más rápido con la misma eficacia, gracias al aumento de la tasa de éxito proporcionado por otra de las mejoras propuestas. Las mejoras están orientadas a asegurar que las instrucciones especulativas que roban la información se ejecuten lo antes posible y que la excepción que producen ocurra lo más tarde posible. Además, este trabajo vislumbra la posibilidad de otra mitigación basada en la explotación del paralelismo del procesador, rompiendo la temporización del ataque y reduciendo así su impacto.

6.2 TRABAJO FUTURO

En la Sección 5.1, se comenta la existencia de comportamientos que afectan negativamente al ataque, pero no está claro el por qué debido a la gran complejidad del procesador y la falta de documentación sobre ellos. Una de las hipótesis que se baraja es que el ataque es muy dependiente de la alineación de los paquetes que salen del ROB y su planificación en los puertos, que podría coincidir con los picos de sierra cada 4 ADDs. Observar más de cerca este comportamiento está fuera del alcance de este TFG y se estudiará con posterioridad así como la experimentación con otros ataques similares y el diseño de arquitecturas seguras por construcción que sigan permitiendo la especulación.

BIBLIOGRAFÍA

[1] ARM. (2018). Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism, dirección: https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability.

[2] I. Anati, S. Gueron, S. Johnson y V. Scarlata, «Innovative technology for CPU based attestation and sealing», en *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, ACM New York, NY, USA, vol. 13, 2013.

[3] Anonymous. (2018). The proof-of-concept code for "Spectre Attacks: Exploiting Speculative Execution"., dirección: https://gist.github.com/anonymous/99a72c9c1003f8ae0707b4927ec1bd8a.

[4] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice y S. Mangard, «KASLR is Dead: Long Live KASLR», en *Engineering Secure Software and Systems*, E. Bodden, M. Payer y E. Athanasopoulos, eds., Springer International Publishing, 2017, págs. 161-176.

[5] J. L. Hennessy y D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[6] IAIK. (2018). Meltdown Proof-of-Concept, dirección: https://github.com/IAIK/meltdown.

[7] IBM. (2018). Potential Impact on Processors in the POWER Family, dirección: https://www.ibm.com/blogs/psirt/potential-impact-processors-power-family/.

[8] P. Kocher, J. Horn, A. Fogh y col., «Spectre Attacks: Exploiting Speculative Execution», en *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[9] M. Lipp, M. Schwarz, D. Gruss y col., «Meltdown: Reading Kernel Memory from User Space», en *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[10] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher y D. Gruss, «ZombieLoad: Cross-Privilege-Boundary Data Sampling», *arXiv:1905.05726*, 2019.

[11] R. M. Tomasulo, «An efficient algorithm for exploiting multiple arithmetic units», *IBM JOURNAL OF RESEARCH AND DEVELOPMENT*, 1967.

[12] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom y R. Strackx, «Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution», en *Proceedings of the 27th USENIX Security Symposium*, See also technical report Foreshadow-NG [13], USENIX Association, 2018.

[13] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch e Y. Yarom, «Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution», *Technical report*, 2018, See also USENIX Security paper Foreshadow [12].

[14] Y. Yarom y K. Falkner, «FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack», en *Proceedings of the 23rd USENIX Conference on Security Symposium*, ép. SEC'14, USENIX Association, 2014, págs. 719-732.

[15] P. aka @trufae. (). Radare: The Unix-Friendly Framework for Reverse Engineering, dirección: https://www.radare.org/n/.