Vulnerabilidades en arquitecturas modernas

M. B. García Lapegna, M. U. Sáenz Valiente, T. Bursztyn

***Resumen*— En los procesadores modernos, existen técnicas de optimización como la ejecución fuera de orden o los branch-predictors, que permiten acelerar la velocidad de procesamiento. Sin embargo, estos dejan efectos laterales en el estado interno de la microarquitectura. Meltdown y Spectre son dos vulnerabilidades que, mediante estos efectos laterales, logran leer memoria a la que no tienen acceso normalmente. En este trabajo se evaluará el funcionamiento de estas vulnerabilidades, así como las formas de mitigarlas y sus consecuencias.**

***Términos clave*— Ataques de canal lateral, canales encubiertos, ejecución especulativa, ejecución fuera de orden.**

I. INTRODUCCION

Los procesadores modernos frecuentemente incorporan cambios específicos en la arquitectura para mejorar su rendimiento. La ejecución fuera de orden, por ejemplo, permite que mientras se termina de realizar una instrucción específica, sus instrucciones siguientes también puedan resolverse en simultáneo. Los branch-predictors, por otro lado, predicen cómo continuará el flujo del programa basándose en información sobre los resultados de instrucciones anteriores. Esto permite que la ejecución continúe realizándose especulativamente mientras se evalúa el flujo correcto del programa. Todos estos mecanismos logran un aumento significativo de la velocidad de procesamiento. Por esta razón, fueron adoptados rápidamente por los fabricantes. [1]

Cuando un procedimiento produce cambios no contemplados ni explicitados, se dice que este genera efectos laterales o *side-effects*. Los mecanismos antes mencionados generan *side-effects* en el estado de la microarquitectura, por la forma en la que están implementados. Estos cambios no son visibles directamente para un usuario, por lo que inicialmente no son un problema. Sin embargo, se pueden causar estos *side-effects* intencionadamente para obtener información protegida, y luego filtrar la información mediante un canal encubierto.

Meltdown es una vulnerabilidad que aprovecha la ejecución fuera de orden para enviar instrucciones maliciosas a la cola de ejecución. En primer lugar, se ejecuta una instrucción que toma un tiempo en ejecutarse. Normalmente en esta instrucción se accede a una dirección de memoria inválida.

Luego, mientras se termina de resolver esta instrucción, se inicia la ejecución de instrucciones sin dependencias, que serán resueltas antes de que la excepción sea enviada.

Estas últimas instrucciones son llamadas instrucciones transitorias. Si bien sus resultados son descartados, la realización de estas instrucciones genera side-effects que luego se pueden filtrar usando *Flush+Reload*[6] o algún otro canal encubierto. [1]

Por otro lado, un ataque Spectre consiste en entrenar a un branch-predictor para que su predicción sea una determinada. Luego, se evalúa la condición con un valor erróneo. Mientras la condición es evaluada, las instrucciones siguientes se ejecutan especulativamente siguiendo la predicción obtenida (la que el atacante indujo). Una vez se termina de evaluar la condición las instrucciones realizadas especulativamente son descartadas. Sin embargo, los *side-effects* generados pueden ser filtrados usando canales encubiertos, de una forma similar a la de Meltdown. [2]

En la sección 2 se detallan las herramientas necesarias para entender y poder replicar cualquiera de estos ataques.

II. TÉCNICAS DE OPTIMIZACIÓN

En esta sección se detalla el funcionamiento de algunas técnicas de optimización utilizadas por las vulnerabilidades que se analizarán más adelante.

2.1 Ejecución fuera de orden

La ejecución fuera de orden es una técnica de optimización que permite maximizar la utilización de las unidades de ejecución del procesador. Su funcionamiento consiste en realizar las instrucciones que no dependan de las unidades de ejecución utilizadas por la instrucción actual. Para lograrlo, las instrucciones se ejecutan siguiendo la idea del Algoritmo de Tomasulo.

En la arquitectura Intel, las instrucciones son decodificadas y convertidas en varias microinstrucciones. Luego se envían a un buffer llamado Reorder Buffer. Estas microinstrucciones son enviadas luego a la estación de reserva o Unified Reservation Station, que se encarga de encolarlas en su respectiva unidad de ejecución correspondiente. Una ALU (*Arithmetic Logic Unit*) o una AGU (*Address Generation Unit*) son ejemplos de unidades de ejecución. [3]

2.2 Ejecución especulativa

Cuando el flujo de ejecución de un programa depende del resultado de una instrucción específica, evaluarla puede ser muy costoso, ya que la ejecución estará detenida hasta que la instrucción finalmente se resuelva. Por ejemplo, si una condición depende de una lectura en memoria, su evaluación

Melanie Belen Garcia Lapegna, Estudiante de Ingeniería Informática de la Universidad de Buenos Aires (email: mgarcial@fi.uba.ar)

Mirko Uriel Sáenz Valiente, Estudiante de Ingeniería Informática de la Universidad de Buenos Aires (email: msaenz@fi.uba.ar)

Tomas Bursztyn, estudiante de Ingeniería Informática de la Universidad de Buenos Aires (email: tbursztyn@fi.uba.ar)

puede ser muy costosa. Para solucionar este problema, los procesadores guardan los resultados de evaluaciones anteriores a esa misma evaluación, y asocian a cada evaluación un valor de predicción. Entonces, mientras se evalúa cual es la rama correcta por la que el programa debe ir, la ejecución continua siguiendo el valor que el procesador predice como correcto. A las unidades que se encargan de almacenar y devolver las predicciones se los llama *branch-predictors* o predictores de salto.

Llamamos ejecución especulativa a las realización de instrucciones basándose en el resultado de una predicción. Cuando se realiza una predicción, hay dos escenarios posibles: si la predicción fue correcta, las instrucciones realizadas especulativamente se terminaran de realizar y serán finalmente ejecutadas. Cuando esto sucede, la velocidad de procesamiento es usada de la manera más eficiente posible, ya que en ningún momento se detuvo la ejecución. Si la predicción es incorrecta, en cambio, las instrucciones ejecutadas especulativamente son las incorrectas, y por lo tanto no se debe devolver su resultado. Por eso, los resultados de estas instrucciones son descartados y el programa continúa ejecutándose por el rumbo correcto. En este segundo caso, el procesador debió descartar instrucciones ya realizadas, y por lo tanto su rendimiento no es el máximo.

Es deseable que los *branch-predictors* tengan una alta tasa de acierto en sus predicciones. Para lograr eso, los procesadores asocian un resultado a una evaluación recién cuando se cumplen condiciones determinadas, asociadas a la cantidad de veces que se obtuvo un resultado, entre otras cosas. [1]

III. VULNERABILIDADES

Cuando se crea una microarquitectura, se pretende que el acceso a cada uno de sus componentes esté definido y contemplado. Es decir, solo se debería poder obtener información siguiendo el procedimiento que brinda la arquitectura. Sin embargo, los componentes de una microarquitectura pueden producir efectos no deseados o side-effects que permitan obtener información de una forma alternativa. De esta manera, se crean huecos de seguridad en la microarquitectura, dando lugar a nuevas vulnerabilidades.

3.1 Ataques de canal lateral

Cuando surge un *side-effect*, también se genera una nueva forma de acceder a nueva información mediante un canal no definido por la arquitectura. A ese nuevo canal se lo llama canal lateral o *side-channel*. Una vulnerabilidad puede aprovechar estos canales para obtener información a la que no debería tener acceso. A este tipo de vulnerabilidades se las conoce como *side-channel attacks* [5]). Estas vulnerabilidades suelen producirse por descuidos en la implementación, que luego son descubiertos y aprovechados. Algunos ejemplos comunes son variaciones de temperatura específicas, Cuando el side-effect solo se refleja en el estado de la microarquitectura, se puede transmitir esta información al estado de la arquitectura mediante un canal encubierto como *Flush+Reload*. [3] [6]

3.2 Canales encubiertos

Cuando se ejecuta una instrucción, antes de que el resultado se envíe al estado de la arquitectura, se pueden generar cambios en la microarquitectura que se reflejan en su estado interno. Cuando se lleva a cabo un ataque de canal lateral, estos cambios son producidos intencionadamente. Para transmitir estos cambios y obtener el resultado de la filtración, se debe usar un canal encubierto. Estos canales permiten enviar información creando cambios en la microarquitectura que pueden consultarse con instrucciones desde la arquitectura. *Flush+Reload*[6] es un canal encubierto que permite enviar la información a través de los tiempos de carga de la memoria caché.

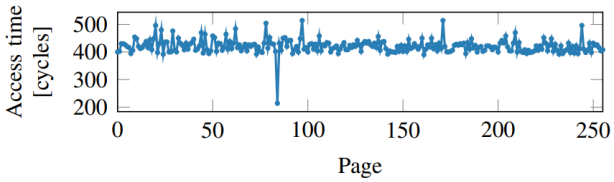

Fig. 1. Mediciones de los tiempos de respuesta de la caché para cada página. En este caso, el valor enviado mediante *Flush+Reload[6] era 84.*

Para implementar *Flush+Reload*[6] se debe tener un arreglo mediante el cual se transmite la información. En primer lugar se debe desalojar toda la memoria del arreglo de la caché (esta sería la etapa *Flush*). Durante el ataque, el atacante accede al arreglo en una posición que depende del valor que pretende enviar. De esta manera, se guarda esa dirección en la caché. Cada acceso debe corresponder a una línea de caché para que un *hit* en una línea signifique únicamente un valor. Esto se logra, por ejemplo, multiplicando el valor que se desea transmitir por el tamaño de cada página. Así, el arreglo queda de tamaño *cantidad de páginas* * *tamaño de cada página*. El último paso (la etapa *Reload*) consiste en leer cada línea de cache y medir el tiempo en que se obtiene el resultado. Un tiempo de carga menor indica entonces un hit en la memoria caché, y por lo tanto, ese es el dato que se envió desde la microarquitectura. Por ejemplo, con una cache de 256 líneas y un arreglo dividido en 256 partes, un hit en la línea 81 indica que el valor filtrado es 81.

3.3 Meltdown

```
1 ; rcx = kernel address, rbx = probe array
2 xor rax, rax
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

Fig. 2. Ejemplo básico de implementación de Meltdown escrito en Assembler.

Como se explicó en la sección 2.1, la mayoría de las

arquitecturas modernas utilizan el Algoritmo de Tomasulo para optimizar su rendimiento. De esta forma, las instrucciones son separadas en microinstrucciones y enviadas a la *reservation-station*, y la instrucción se resuelve cuando la última de las microinstrucciones se resuelve. Esto da lugar a la ejecución fuera de orden, que es la clave para replicar un ataque Meltdown. [4]

Si una instrucción accede a una dirección de memoria a la que no tiene acceso, la ejecución se detiene con una excepción. Aun así, por la ejecución fuera de orden, una instrucción siguiente puede ser realizada antes de que se lance la excepción. Esto no cambia el estado de la arquitectura ya que los resultados nunca son enviados, pero si puede efectuar cambios en el estado de la microarquitectura. Por ejemplo, se podría guardar en un registro el valor de una dirección a la que no se tiene acceso. Luego, dadas las condiciones adecuadas, una instrucción realizada fuera de orden podría usar este valor y generar *side-effects*.

Un ataque Meltdown está compuesto por dos bloques: en el primero de ellos, se ejecuta una instrucción que guarda el dato inaccesible en un registro. Inmediatamente después, se ejecutan instrucciones transitorias, que utilizan el dato del registro y transmiten la información. Es esencial que las instrucciones transitorias involucren la información inaccesible que se quiere transmitir. La primera instrucción va a lanzar una excepción, pero si las instrucciones siguientes son transitorias no tendrán dependencias, y por lo tanto, sus microinstrucciones serán realizadas igualmente, reflejando sus cambios en el estado de la microarquitectura. En el segundo bloque se transmiten los cambios en el estado de la microarquitectura mediante un canal encubierto, como puede ser *Flush+Reload*[6] o cualquier otro que cumpla esta función. Al replicar el primer paso de un ataque Meltdown, el caso más común es que el programa lance una excepción y el flujo del programa se detenga de inmediato, dañando los resultados obtenidos. Para solucionar este problema se pueden utilizar diversas técnicas que evitan este comportamiento. En procesadores Intel, se puede utilizar un hardware específico llamado Intel TSX. Su funcionamiento consiste en realizar un conjunto de instrucciones “en conjunto”. Es decir, o bien todas las instrucciones se ejecutan, o ninguna lo hace, sin lanzar ninguna excepción. La ventaja de esto es que, al no lanzar excepciones en ningún caso, los *side-effects* permanecen intactos luego de intentar ejecutar las instrucciones. Por lo tanto, utilizar Intel TSX soluciona por completo el problema de la excepción. [4]

Siguiendo los pasos mencionados, con las consideraciones correspondientes, se puede enviar la información desde el estado de la microarquitectura.

En la Figura 2, las líneas 5-7 son las instrucciones transitorias, que se ejecutan fuera de orden mientras se resuelve el acceso a memoria de la línea 4.

3.4 Spectre

Cuando el flujo de ejecución de un programa depende de un valor leído en memoria, esperar a que ese dato sea leído puede demorar demasiado tiempo. Esto puede suceder, por ejemplo, cuando hay un salto condicional y el valor de esa condición implica leer un valor. Los procesadores modernos resuelven este problema usando *branch-predictors*. Cuando un *branch-predictor* acierta su predicción, las instrucciones realizadas especulativamente son ejecutadas. Por el contrario, cuando la predicción falla, las instrucciones realizadas son descartadas y nunca se ejecutan. [1] De todas formas, la realización de estas instrucciones genera *side-effects* en la arquitectura que pueden ser detectados. En particular, los ataques Spectre se centran en los cambios que se generan en la memoria caché.

```
if (x < array1_size)
    y = array2[array1[x] * 4096];
```

Fig. 3. Ejemplo básico de implementación de Meltdown escrito en Assembler.

Para preparar el ataque se requiere primero tener un canal encubierto por el cual filtrar la información. Para este ataque puede utilizarse casi cualquier canal encubierto indistintamente. Para cualquiera de estas implementaciones se debe tener un arreglo que sirva como medio para filtrar la información (en adelante *arreglo_1*). También se debe tener un arreglo desde el cual se leerán posiciones aledañas no permitidas (en adelante *arreglo_2*). Para que el ataque funcione correctamente, se deben cumplir las siguientes condiciones:

- El *conditional-branch* deberá estar entrenado para resolver como verdadera la condición hasta obtener el resultado correcto.
- Las posiciones de *arreglo_1* y el largo de *arreglo_2* deberán estar fuera de la memoria cache.
- La posición a filtrar deberá ser consultada y estar alojada en la caché previamente.

En primer lugar, para realizar el ataque es necesario entrenar al *branch-predictor*. En el caso de un *conditional-branch*, para lograr esto el atacante debe invocar una y otra vez la condición con un valor correcto para la rama que se desea acceder. De esta manera el predictor asume que la condición será verdadera en próximas invocaciones. En nuestro ejemplo, la condición que se consulta es si un valor x es menor al largo del arreglo (ver esto en Figura 3).

Luego de entrenar al predictor con valores correctos para la condición, se evalúa nuevamente con un valor invalido. Como el largo del arreglo no está en la memoria caché, se debe consultar su valor a la memoria RAM. Este proceso demora un tiempo elevado. Mientras tanto, el conditional-branch resuelve la condición como verdadera y continua la ejecución especulativamente. En las próximas instrucciones, se accede a *arreglo_2* en la posición inválida. Como el valor está alojado en la memoria caché, es devuelto inmediatamente y posteriormente guardado en un registro. A continuación se consulta *arreglo_1* en una posición dependiente del valor filtrado, para transmitir ese valor desde el estado de la microarquitectura. Mientras se consulta *arreglo_2*, la condición se resuelve finalmente como falsa (por el valor

invalido de x) y las instrucciones son descartadas. Sin embargo, el acceso al *arreglo_2* deja side-effects en la memoria caché que son filtrados mediante *Flush+Reload*. [6] [2]

Como fue explicado en la sección 3.2, y de de una forma similar a la de Meltdown, para obtener el valor basta con medir los tiempos de respuesta de *arreglo_1* al consultar direcciones asociadas a valores distintos.

IV. MITIGACIÓN DE LAS VULNERABILIDADES

El descubrimiento de Meltdown y Spectre derivó en una búsqueda desesperada por solucionar de manera rápida y eficaz las inseguridades que causaban. En esta sección, se analizarán las distintas alternativas para mitigar estas vulnerabilidades, y sus consecuencias.

4.1. Parches

Para proteger los procesadores contra Meltdown y Spectre se lanzaron parches. Se realizaron actualizaciones tanto en los microcódigos de los procesadores como en los sistemas operativos y navegadores.

Hubieron tres tipos de parches, clasificados según que parte del sistema afectan y de qué manera están implementados:

- Parches en navegadores: se debió actuar rápidamente para mitigar a Spectre en los navegadores, ya que todos estos se veían afectados por este.
- Microcódigo: todos los fabricantes de procesadores afectados publicaron parches de microcódigo para cerrar los agujeros de seguridad generados por ambas vulnerabilidades. Entre ellos se encontraban Intel, AMD, ARM e IBM. [4]

4.2. Eficacia

Se considera que un parche es eficaz si realmente es capaz de cerrar las brechas de seguridad. En su momento, todos los fabricantes lanzaron parches para mitigar las vulnerabilidades, pero no en todos los casos se logró de inmediato.

A. Meltdown

La vulnerabilidad afecta principalmente a los procesadores de Intel. Actualmente, ya ha sido completamente mitigada. Esto fue posible gracias a las actualizaciones lanzadas en los respectivos sistemas operativos. De igual manera los fabricantes de los procesadores afectados también publicaron actualizaciones de microcódigo.

B. Spectre

Existen varias soluciones para esta vulnerabilidad, una de ellas es "*Google's Retpoline*" y la otra es "*Indirect Branch Control*" (IBC). Estas se implementaron usando nuevas funciones del CPU. Igualmente, ninguna de las posibles soluciones logró cerrar por completo la brecha. Es decir que, aunque estas actualizaciones dificultan la explotación de la vulnerabilidad, no ofrecen una protección completa.

En la actualidad, existen muchos parches efectivos disponibles para protegerse de las distintas vulnerabilidades en los distintos sistemas y dispositivos. En cuanto a los problemas restantes con Spectre y otras vulnerabilidades de seguridad emergentes de este estilo, se sigue trabajando para cerrarlas de manera eficiente.

El principal problema no es la disponibilidad de parches sino su distribución y el soporte de los dispositivos. Debido a la gran cantidad de parches y las muchas actualizaciones defectuosas, toma un gran tiempo que se actualicen todos o la gran mayoría de los sistemas afectados. Los dispositivos más antiguos fueron pasados por alto con la excusa de que si el usuario quiere seguridad debe pagar por ella. [4]

4.3. Impactos negativos

La aplicación de estos parches tuvo algunas consecuencias adversas, entre estas, la más evidente fue la reducción del rendimiento del sistema. Aunque también hubo problemas relacionados con la distribución de parches falsos e incorrectos.

Respecto a los impactos en el rendimiento, se puede decir que están relacionados específicamente con los parches para la brecha de Meltdown. El número de llamadas al sistema o *system-calls* ejecutadas es el factor clave en esta pérdida. Independientemente del sistema operativo o la aplicación que se trate, la reducción de rendimiento depende exclusivamente de la cantidad de llamadas al sistema.

También existieron efectos secundarios por fuera de los parches, como se explicó anteriormente se lanzaron muchos parches para poder abordar estas vulnerabilidades. Hubo errores, la distribución fue poco clara, y hubo falta de conciencia por parte de los usuarios. Además, el "hype" mediático generó mucha confusión entre ellos. Según informes hubo una campaña que se esparció mediante correo electrónico para atraer a las víctimas a un dominio falso, este contenía un archivo comprimido descargable para procesadores Intel y AMD el cual se presentaba como un parche, pero en realidad era un archivo que instalaba "Smoke Loader" , un software que permite a los atacantes inyectar otro código malicioso en el sistema.

En síntesis, se puede decir que la brecha de Meltdown pudo cerrarse, aunque esto implica un impacto negativo en el rendimiento. La pérdida promedio es de aproximadamente el 10%, aunque esta puede ser significativamente mayor o menor dependiendo de la cantidad de entradas y salidas de las aplicaciones. Respecto a Spectre, las brecha no pudo cerrarse por completo. [4]

V. CONCLUSIÓN

En este trabajo se analizaron las vulnerabilidades de Meltdown y Spectre, las cuales son brechas de seguridad relacionadas con optimizaciones como la ejecución fuera de orden y la predicción de saltos. Estas optimizaciones dieron paso a nuevas vulnerabilidades que los atacantes pueden aprovechar para acceder a información no autorizada, utilizando debilidades de implementación en las técnicas de optimización.

Como solución a estas, se lanzaron parches que han sido efectivos en algunos casos, pero también tuvieron efectos

adversos como la mala distribución de actualizaciones o el mal rendimiento.

A día de hoy, estas vulnerabilidades siguen estando parcialmente mitigadas, es decir, no están completamente resueltas.

Referencias

[1] Hennessy, J. L., & Patterson, D. A. (2011). Computer architecture: a quantitative approach. Elsevier.

[2] Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., ... & Yarom, Y. (2020). Spectre attacks: Exploiting speculative execution. Communications of the ACM, 63(7), 93-101.

[3] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Mangard, S., ... & Hamburg, M. (2018). Meltdown. arXiv preprint arXiv:1801.01207.

[4] Löw, M. (2018). Overview of meltdown and spectre patches and their impacts. Advanced Microkernel Operating Systems, 53.

[5] Standaert, F. X. (2010). Introduction to side-channel attacks. Secure integrated circuits and systems, 27-42.

[6] Yarom, Y., & Falkner, K. (2014). {FLUSH+ RELOAD}: A high resolution, low noise, l3 cache {Side-Channel} attack. In 23rd USENIX security symposium (USENIX security 14) (pp. 719-732).