UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DISCA
DEPARTAMENTO DE INFORMÁTICA DE SISTEMAS Y COMPUTADORES

Máster en Ingeniería de Computadores y Redes

Trabajo Fin de Máster

Estudio mediante ingeniería inversa sobre la estructura de memorias DRAM

Autor: Javier Sales Moliner

Director: José Ismael Ripoll Ripoll

Septiembre 2021

Agradecimientos

Ante todo, quiero agradecer a Ismael su apoyo constante y su papel de guía, aportando información vital sobre cómo orientar el trabajo y afrontar los problemas encontrados. El presente proyecto no podría haberse logrado sin su colaboración.

Por otra parte, debo agradecer a la universidad, al DISCA y a su profesorado, el enseñarme los conocimientos necesarios para desarrollar este proyecto.

También, gracias a mis padres, a Carol, y a Carlos.

Resumen

La vulnerabilidad en los módulos DRAM ante ataques RowHammer presenta una amenaza a la privacidad e integridad de la información. Aunque actualmente los fabricantes integran mecanismos de mitigación en sus chips que requieren de ataques mucho más sofisticados, siguen sin ser totalmente infalibles, y con el avance de las futuras generaciones DDR la vulnerabilidad será aún mayor.

En torno a esta problemática, para la efectividad del ataque resulta clave conocer con la mayor exactitud posible cómo las direcciones del módulo están organizadas internamente. Este trabajo recopila toda la información útil para tal propósito, y aporta un programa de análisis que combina métodos nuevos, junto a optimizaciones en métodos existentes, para determinar datos sobre la estructura del módulo.

Palabras clave: Vulneración de seguridad, DRAM, RowHammer.

Abstract

The vulnerability in DRAM modules to RowHammer attacks poses a threat to data privacy and integrity. Although manufacturers currently integrate mitigation mechanisms in their chips that require much more sophisticated attacks, they are still not totally infallible, and with the advance of future DDR generations the vulnerability will be even greater.

In this regard, knowing as accurately as possible how the module addresses are mapped internally is key to the effectiveness of the attack.This work gathers all the information useful for this purpose, and provides an analysis program that combines new methods, together with optimizations in existing methods, to determine information about the module structure.

Keywords: Security vulnerability, DRAM, RowHammer.

Tabla de contenido

Índice de figuras

Introducción

RowHammer ha sido la primera vulnerabilidad que afecta al hardware, en concreto a las memorias RAM dinámicas (DRAM), que puede explotarse mediante la ejecución de un programa consistente en la sencilla lectura de sus valores almacenados. Representó un punto de inflexión en lo que hasta entonces se consideraban los límites de los ataques informáticos, y tras su descubrimiento, la integridad de la información en memorias DRAM se ha convertido en objeto de múltiples estudios.

Desde entonces, gracias a la integración de mecanismos de mitigación centrados en esta vulnerabilidad. la industria ha conseguido mantener las memorias DRAM relativamente libres de peligro. Sin embargo, se han descubierto variantes del ataque que, conscientes de estas medidas, modifican su estrategia para seguir aprovechando la vulnerabilidad.

Motivación

Durante la búsqueda de un tema sobre el que realizar el presente trabajo, la naturaleza singular de la vulnerabilidad RowHammer generó inmediato interés. Sin conocer nada relativo a él, aparte de la premisa del ataque, surgían multitud de cuestiones cuya respuesta normalmente traía otras nuevas.

Sumado a esto, el tema involucraba múltiples áreas de conocimiento dentro de la informática, mostrándose como una buena elección para desarrollar y profundizar su entendimiento. En el siguiente estudio confluyen aspectos sobre el funcionamiento de dispositivos a nivel electrónico, teoría de procesadores y sistemas operativos, y tareas de programación de bajo nivel.

Además, como se expondrá en el documento, el futuro de esta vulnerabilidad apunta a ser un problema persistente para los fabricantes de memoria que busquen continuar mejorando las prestaciones de sus productos al tiempo que los mantienen protegidos contra el RowHammer, por lo que cualquier aportación puede acabar resultando de utilidad para estudios posteriores.

Objetivos

El presente trabajo es el producto de las siguientes metas:

En primer lugar, realizar una tarea de documentación y formación en profundidad sobre el estado del arte de la tecnología de la memoria DRAM y las distintas variantes de RowHammer, a través de documentos técnicos, publicaciones científicas, y otras fuentes, y consolidar la información con pruebas que contrasten lo adquirido y permitan ganar familiaridad.

Una vez adquirida una visión global de la situación, considerar la mejor dirección en la que desarrollar un trabajo de investigación, que aporte nueva información o metodología de estudio sobre el marco del proyecto, y llevarlo a cabo.

Estructura

De acuerdo con los objetivos descritos, el resto del documento sigue el siguiente orden:

A continuación, se presenta un capítulo que resume de forma esquematizada toda la información útil del marco teórico del proyecto, recopilada durante el proceso de documentación. En este capítulo se encuentran distintas secciones que hablan acerca de la estructura y funcionamiento de un módulo DRAM, la evolución de su tecnología durante el tiempo, qué es un ataque RowHammer, cúales son sus variantes y cómo han evolucionado junto a los mecanismos de mitigación del RowHamer integrados en memoria, y cómo ésta es gestionada por el procesador. Para cerrar el capítulo, se habla de la base sobre la que se apoyan los métodos principales de ingeniería inversa en DRAM, describiendo un programa ya existente que los utiliza para determinar información de la DRAM que favorece la eficacia del RowHammer. A continuación, se describe el trabajo realizado, divididos en tres secciones. En primer lugar, un capítulo describe un proceso de documentación y experimentación para tratar de acceder a todos los datos que un sistema puede facilitar sobre su memoria y que resulten útiles para incrementar la eficacia del RowHammer, seguido de un capítulo donde se desarrolla una nueva versión del mencionado programa de ingeniería inversa que incluya la implementación de mejoras. En la sección final, se aportan nuevos enfoques de aplicación para la ingeniería inversa aplicada a la DRAM, los cuales se podrán utilizar para obtener información sobre la estructura física de la memoria DRAM y el aceceso del procesador a ella.

Estado del arte

Para evitar interrumpir la descripción del trabajo realizado más adelante con explicaciones puntuales, se incluye como un primer capítulo la explicación de todo lo necesario sobre:
·La estructura electrónica de la memoria DRAM, su funcionamiento y evolución tecnológica.
·La gestión de la memoria DRAM por parte del procesador.
·El descubrimiento y evolución del ataque RowHammer, hasta hoy.
·Cómo obtener de forma indirecta información sobre la estructura interna de la memoria DRAM.

La memoria DRAM

La RAM (Random Access Memory) es un tipo de memoria volátil (sin alimentación, los datos se pierden) basada en el acceso directo a celdas de memoria dispuestas en una matriz 2D. Cada una de las celdas guarda un único bit. Como sabemos, el término 'Random' no se refiere a accesos aleatorios, sino que busca diferenciarla de los dispositivos de almacenamiento secuencial (p. ej, las cintas magnéticas).

Dentro de las memorias RAM encontramos dos tipos principales: la SRAM (Static Random Access Memory) y la DRAM (Dynamic Random Access Memory). La primera forma cada una de sus celdas con un total de 6 transistores MOSFET. Cuatro transistores son utilizados para formar un biestable que almacena dos estados distintos (interpretados por el sistema como '0' o '1'), y los otros dos transistores controlan su acceso. Esta variante se utiliza en cachés y buffers.

La memoria DRAM, popularmente conocida simplemente como memoria RAM, compone la memoria principal de los equipos actuales. En comparación, es más barata y permite mayor densidad de almacenamiento que la SRAM, pero sus accesos son más lentos. Además, padece de otro gran inconveniente: todas sus celdas requieren de un "refresco continuado".

Figura 1: Módulo DRAM DDR4 de 8GB, fabricado por Micron. Fuente: Amazon

Las celdas DRAM están formadas únicamente por un único transistor MOSFET y un condensador (como se ha mencionado, una construcción mucho más sencilla y económica que la celda SRAM). El condensador retiene una pequeña carga, que luego es amplificada e interpretada (si no detecta suficiente voltaje, bit = '0' -LOW-, y si se supera cierto umbral, bit = '1' -HIGH-), pero solo durante un breve periodo de tiempo. Por la naturaleza del condensador, la carga se va fugando continuamente, obligando a realizar operaciones de refresco antes de que el voltaje sea inferior al mencionado umbral (noise margin) y se pierda su información. Para ello, se accede a la celda, se lee su valor, y se reintroduce en el condensador.

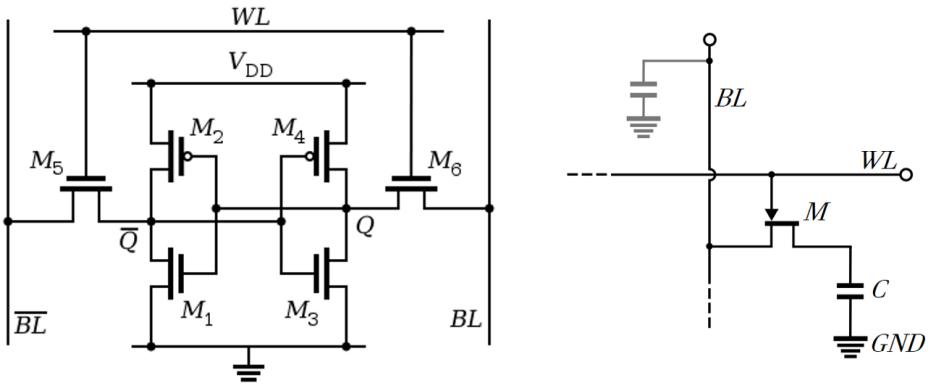

Figura 2: Diagrama electrónico de las celdas DRAM (SRAM, a la izquierda, y DRAM a la derecha), conectadas a la matriz de datos por su wordline (WL) y bitline (BL). Fuente: propia

Como resulta aparente, la diferenciación entre los términos "Static/Dynamic" de la memoria SRAM/DRAM hace referencia a la persistencia de los datos en las celdas. Esta diferencia en el diseño electrónico hace además que, en la DRAM, cualquier acceso a memoria sea destructivo, eliminando el dato almacenado de los condensadores y obligando a reintroducir el dato. Se puede entender mejor describiendo el proceso de acceso a los datos de la matriz (Bhati, 2015) (Kim J. S., 2020, Mayo).

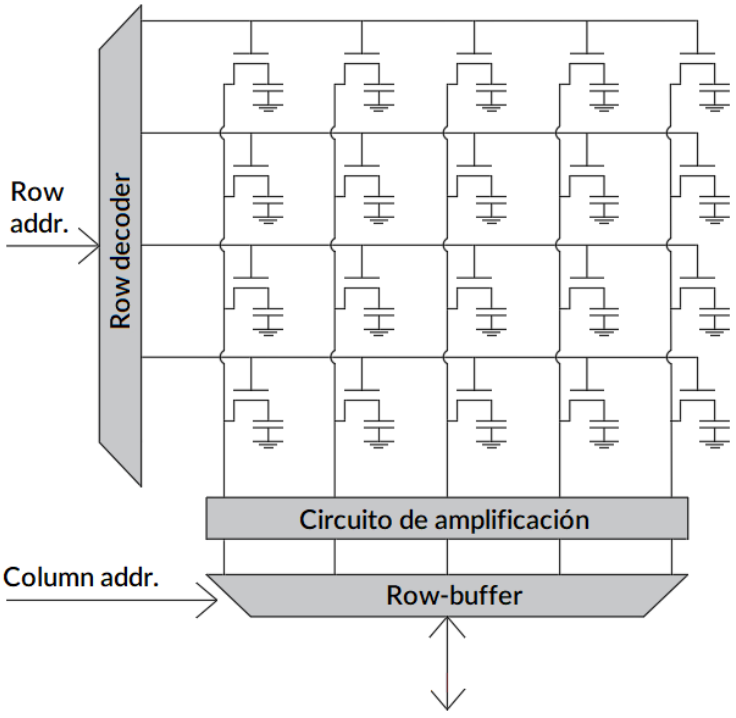

Figura 3: Diagrama electrónico de una matriz DRAM. Fuente: propia

Cuando la CPU ejecuta un acceso a memoria, ésta le transmite la instrucción al controlador de memoria integrado en el procesador (en adelante MC, Memory Controller). Este se comunica con

la DRAM a través de un bus de datos dedicado, activando los pines precisos para apuntar al dato concreto.

Se parte de una matriz DRAM cuyas celdas pueden o no contener carga, según el valor bit almacenado (Figura 4.1). Primero, se aplica una tensión a la fila (wordline) seleccionada con un demultiplexor (row decoder) (Figura 4.2). Esto activa todos los transistores conectados a ella, que cortocircuitan el condensador de esa celda con su columna (bitline), transfiriendo la tensión a toda la línea. En su extremo, las bitlines están conectadas a un circuito de amplificación, que hace la señal de cada bit perceptible para poder ser volcada en un buffer (row-buffer) (Figura 4.3). Con un multiplexor, se accede al bit del row-buffer que corresponde a la columna que contenía el dato.

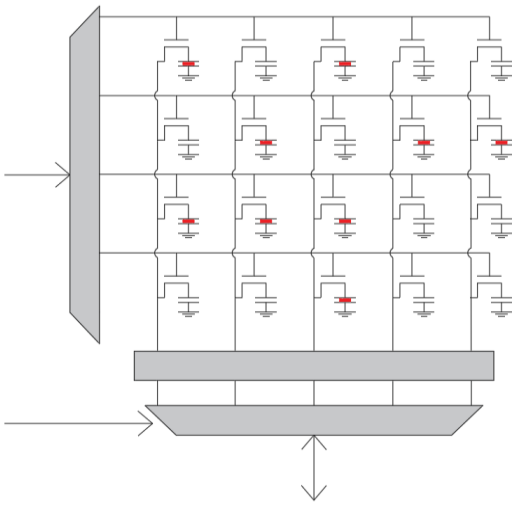

Fig. 4.1

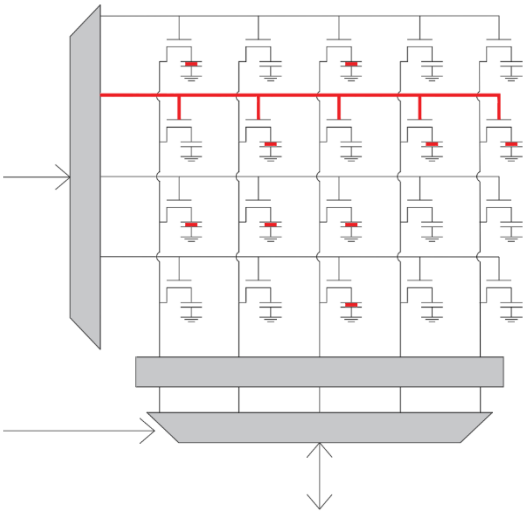

Fig. 4.2

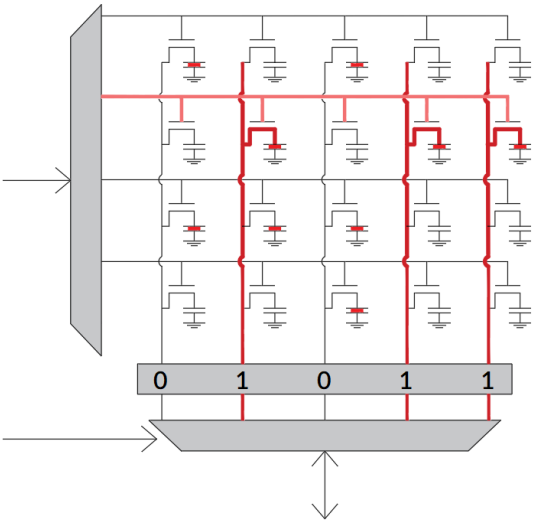

Fig. 4.3

Figura 4: Pasos para acceder datos en una matriz DRAM. Fuente: propia

Para acceder a otra fila, primero se aplica la tensión a los condensadores a través de las bitlines acorde a los datos leídos (como se ha mencionado, el acceso a los condensadores vacía su carga y requiere que se les reintroduzca la información) y desactiva la wordline, aislando así los condensadores de las bitlines.

Desde la perspectiva del MC, los accesos se realizan a través de 3 pasos. Para activar la fila correspondiente, transmite la instrucción 'ACT' junto a la matriz y fila deseada, volcándola a su row-buffer, como se ha descrito. Para interactuar con el buffer, lanza un 'READ' o un 'WRITE', según sea una lectura o escritura, indicando la matriz y la columna precisa. Las celdas permanecen conectadas a los amplificadores del row-buffer hasta que el MC lance un 'PRE' (precharge), para cerrar la fila y preparar la matriz para abrir una diferente.

Por cuestiones de optimización, el MC no desvincula la fila cargada en el row-buffer hasta que no se lance un acceso a una dirección en una fila distinta para esta matriz (es decir, si se accede a la fila A de una matriz y posteriormente se ejecuta una serie de accesos a otras matrices, y luego un acceso a la fila A de esa misma matriz, es probable (aunque el MC no acceda a esa fila, las operaciones de refresco del módulo acaban requiriendo en cuestión de ms que se utilice el row-buffer, como se explica a continuación) que en su row-buffer siga cargada la fila A). Por ello, leer datos que se encuentran en una misma matriz y fila es más rápido, ya que no hace falta vaciar y recargar el row-buffer. El tiempo tomado para realizar estas acciones es del orden de nanosegundos.

En cuanto a los comandos lanzados por el MC, simplemente se lanzaría una primera activación de la wordline (ACT), seguida de los accesos continuados a la fila (READ, WRITE, READ...), sin tener que intercalar cierres y nuevas activaciones entre cada uno. Para aprovechar esta dinámica, el MC puede reordenar los accesos a memoria, lanzando accesos consecutivos a una misma fila si los detecta en cola.

Las celdas DRAM pueden tener tiempos de retención diferentes, desde milisegundos a horas. Para prevenir la alteración de datos, el MC lanza instrucciones 'REF' (refresh) regulares para asegurar el mantenimiento de datos en intervalos regulares. El módulo de memoria recibe estas instrucciones y las administra por su cuenta, refrescando unos miles de filas a la recepción de cada 'REF'.

Operation	*Command*	*Address(es)*
1. Open Row	`ACTIVATE (ACT)`	Bank, Row
2. Read/Write Column	`READ/WRITE`	Bank, Column
3. Close Row	`PRECHARGE (PRE)`	Bank
Refresh	`REFRESH (REF)`	—

Figura 5: Tabla de comandos de control de memorias DRAM. Fuente: *(Kim Y. D., 2014)*.

Desde una perspectiva electrónica, refrescar una fila (REF) y activarla y desactivarla (ACT + PRE) son operaciones idénticas. Cuando una matriz recibe esta instrucción, refresca múltiples filas generando internamente pares 'ACT' y 'PRE'.

La especificación DDR3 DRAM define un tiempo de retención mínimo de 64 ms. En el caso de DDR4, varía entre 32 o 64 ms. El MC lanza una cantidad suficiente de 'REF's para garantizar que toda celda es refrescada al menos una vez en estos tiempos.

En adelante, por practicidad, nos referiremos a la matriz de celdas como ‘bank’, o ‘set’. Es importante recordar que un bank es un conjunto de direcciones de memoria que comparten un mismo row-buffer. Actualmente, lo habitual es encontrar 8 banks en cada módulo DDR3 y 16 en los DDR4. Tener múltiples ‘banks’ incrementa el paralelismo, porque los accesos a diferentes unidades pueden ser servidos de forma concurrente. El tamaño de fila típico (y por tanto el tamaño del row-buffer) en los banks es de 8 kB.

Los banks se alojan en chips, los cuales de por sí tienen prestaciones limitadas, por lo que se comercializan módulos con múltiples chips, soldados a ambas caras del módulo DRAM. Cada una de las caras se denomina 'rank', y es independiente de la otra. Actualmente, en el mercado de consumo, podemos adquirir módulos DRAM de entre 2 y 32 GB.

Los chips de un rank operan en modo bloqueo, o paso a paso (lockstep), y ejecutan el mismo conjunto de operaciones al mismo tiempo en paralelo (reciben la misma información), ignorando la instrucción si señala a otro chip distinto. Los pines combinados de todos los chips forman el bus de datos del módulo DRAM, de 64 bits. Este bus de datos, también conocido como canal, es un enlace directo del módulo al MC. Cada uno de los canales (más adelante [punto] se habla en detalle de sus configuraciones) es independiente y puede ser accedido en paralelo.

La terminología descrita es diferente según donde se lea, optando en este trabajo por la considerada como más común. En conclusión, la memoria DRAM está jerarquizada en una serie de niveles, y para realizar accesos a una dirección específica, se debe especificar canal, módulo (puede haber más de uno instalado), rank, bank, fila y columna.

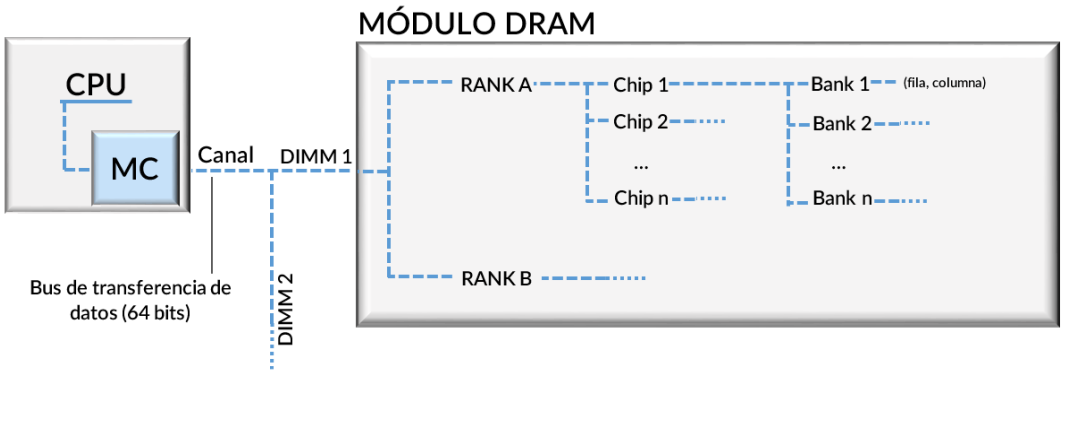

Figura 6: Jerarquía interna en módulos DRAM. Fuente: propia

Evolución de la DDR DRAM

La DDR DRAM (Double Data Rate DRAM), utilizada hoy en día, se introdujo en el año 2000 para reemplazar la SDR DRAM (Single Data Rate DRAM). De esta tecnología, que trajo la capacidad de transferir el doble de datos en único ciclo de reloj (utilizando el flanco de subida y bajada de la señal) y pines independientes para cada una de las caras del módulo, se han lanzado 4 actualizaciones. Actualmente, lo común es encontrar equipos con memorias DDR3 (2011) o DDR4 (2014) que permiten respectivamente módulos de hasta 16GB y 64GB. Respecto a la tecnología DDR5, su llegada al mercado se encuentra próxima.

La LPDDR (Low Power DDR, también llamada Mobile DDR o mDDR), es una variante de DDR -no será objeto de estudio en este trabajo, pero se describe brevemente- centrada en mantener un consumo más reducido, enfocada al mercado de la telefonía móvil y dispositivos de pequeñas dimensiones en general. Esta condición hace que los módulos sean más susceptibles a perturbaciones electromagnéticas, ya que, aunque por una parte la densidad de almacenamiento sea menor (lo que distancia entre sí las celdas y mitiga más el poder de interferencia entre ellas), se reduce también la sensibilidad umbral de las celdas de memoria. Además, los módulos instalados en móviles suelen ser de una calidad inferior al grueso de lo que ofrece el mercado para instalar en PCs.

Existen memorias DRAM con ECC (Error Correction Code) integrados por un precio más elevado, pero estos módulos no son comunes en el mercado general, al estar más orientados a sistemas de servidores.

Desde hace años, el mercado de los módulos DRAM está disputado casi en su totalidad por 3 fabricantes (Samsung, Hynix, y Micron).

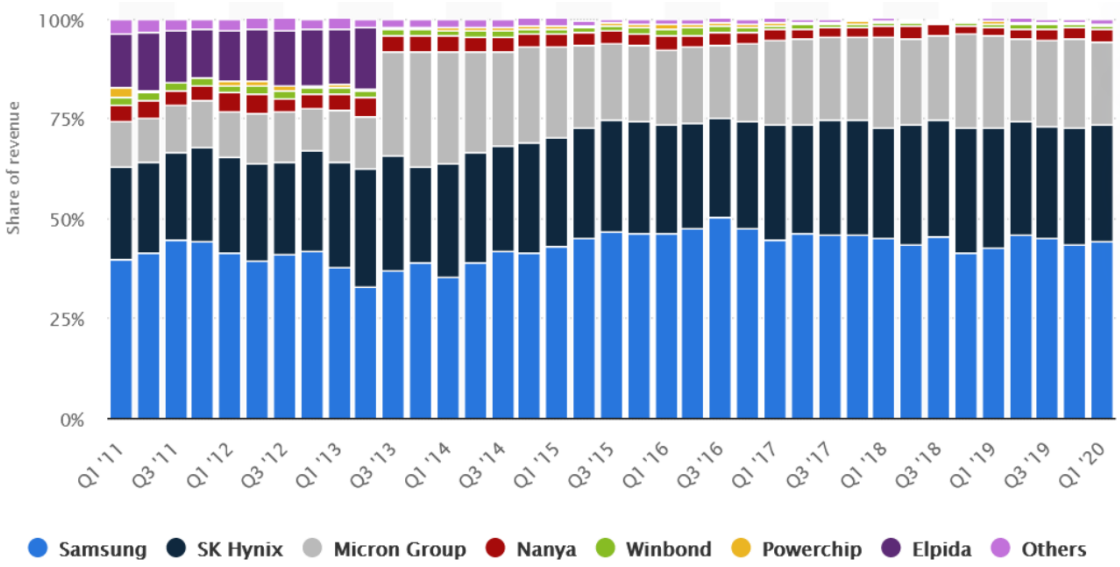

Figura 7: Predominancia de los fabricantes de DRAM en el mercado (Periodo 2011-2020). Fuente: statista.com

El interés de los fabricantes es el de aumentar la densidad de memoria tanto como la tecnología lo permita (es decir, aumentar la densidad de celdas), para lo cual se requieren componentes más pequeños, en particular, el condensador que almacena la carga.

Un condensador de menor tamaño (y que por tanto puede almacenar una carga inferior) consume menos energía, pero por otra parte, el tiempo de persistencia de la información es menor. La reducción de las distancias entre los componentes hace que los condensadores estén más próximos tanto a otras celdas como a los buses de acceso, lo cual acentúa las posibles interacciones electromagnéticas entre los componentes.

RowHammer

En 2014, en el marco del ISCA (International Symposium on Computer Architecture), se presentó el paper (Yoongu Kim, Ross Daly, et al.) "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors" (Kim Y. D., 2014), que revelaba la vulnerabilidad de módulos DRAM ante un ataque definido como "RowHammer". Años atrás ya se había mencionado este problema sin ahondar en detalles, pero este paper señala y le da importancia por primera vez.

El ataque RowHammer consiste sencillamente en el constante acceso a alta frecuencia, a dos direcciones alternas de memoria que se encuentren en un mismo bank, pero en distintas filas (Filas agresoras). Al compartir row-buffer, esta interacción provoca la activación y desactivación de ambas filas a gran velocidad, lo que se descubrió, reducía los tiempos de retención de algunas de las celdas en las filas físicamente próximas (filas víctima) lo suficiente como para que el siguiente comando de refresco no llegara a tiempo para salvar la información.

La explicación del fenómeno reside en el proceso de activación de una fila, que genera un ‘estrés’ eléctrico sobre las filas adyacentes. Durante el uso común de la memoria, esta pequeña perturbación no provoca ningún efecto y pasa inadvertida, pero realizando activaciones a una frecuencia elevada, este se potencia hasta lograr perturbar celdas en filas cercanas, las cuales pierden su carga.

Así, simplemente realizando accesos legítimos a memoria desde un programa sin permisos elevados se pueden modificar datos en memoria principal que pertenezcan a otros procesos.

El ataque fue diseñado para ejecutarse en la arquitectura x86 de Intel, al depender de las instrucciones de ensamblador que a continuación se comentan. Dicho esto, es importante recordar que RowHammer no es un fallo del procesador, sino del diseño de la memoria.

Para que el ataque tenga éxito, se debe realizar un borrado (instrucción de ensamblador “clflush”) de la caché para eliminar los datos almacenados en las dos direcciones, forzando al sistema a volver a acceder a la memoria principal para conseguirlos en cada lectura.

Además, con la instrucción “mfence” se indica al compilador que no debe optimizar el programa reordenado los accesos a memoria, ya que es necesario que éstos se realicen en el orden descrito para asegurar que los datos están completamente desalojados de la caché antes de realizar la siguiente solicitud de memoria.

```
1 code1a:
2   mov (X), %eax
3   mov (Y), %ebx
4   clflush (X)
5   clflush (Y)
6   mfence
7   jmp code1a
```

Figura 8: Código en ensamblador de un ataque RowHammer. Fuente: *(Kim Y. D., 2014)*.

Respecto a las causas de la vulnerabilidad, la publicación incluso apunta a la existencia de celdas en las filas agresora/víctima, que pueden tanto evitar como contribuir a la existencia de fallos de retención durante el ataque, concluyendo en que estas alteraciones son producto de un complejo fenómeno colectivo.

Tras describir el ataque e hipotetizar sobre los posibles fenómenos físicos causantes de la pérdida de carga, se detallan los resultados de una serie de pruebas en distintos módulos del mercado, no positivos. Los módulos se testaron instalándolos en una plataforma FPGA, ejecutando distintos códigos de ataque, modificando los intervalos de activación (la agresividad del ataque) pero manteniendo los intervalos de refresco de filas dentro de lo que demanda la especificación DDR3. Se poblaba toda la memoria con distintos patrones de datos, y se inspeccionaba tras ejecutar los códigos en busca de fallos. De 129 módulos sometidos al ataque, se logró producir una pérdida de datos de al menos un bit en 110 de ellos, siendo vulnerables todos los fabricados en los últimos dos años (2012 y 2013).

De media, en los módulos vulnerables no se necesitaban más de 139k accesos para inducir fallos en memoria, siendo una de cada 1.7k celdas susceptible a ellos y predominando los fallos ubicados a dos filas o menos de la agresora, especialmente de las filas adyacentes (+1/-1). Además, los errores eran repetibles en un 75% de las celdas vulnerables de forma constante, en cada una de las iteraciones del ataque.

Hasta entonces, la percepción del fenómeno era que mediante un fallo de diseño, se podía hacer que uno o varios bits perdieran su carga. Sin embargo, el fallo se convirtió en vulnerabilidad cuando se demostró que podía aprovecharse para escalar privilegios u obtener información sensible. El 9 de marzo de 2015, Mark Seaborn y Thomas Dullien, miembros del equipo Google Project Zero, lanzaron otro trabajo, "Exploiting the DRAM RowHammer bug to gain kernel privileges" (Seaborn M. &., 2015), que introdujo el Double-Sided RowHammer, un patrón de ataque mucho más efectivo que el presentado un año atrás en el ISCA, además de describir cómo gracias a él se conseguía escapar de un entorno virtual u obtener permisos de superusuario en un sistema Linux.

En el Double-Sided RowHammer, en vez de tomar dos filas aleatorias, se fija como objetivo una única fila víctima N, haciendo agresoras a sus vecinas (N1/N+1) para combinar el efecto perturbador. En algunos módulos, resulta el único patrón con el que obtener fallos en celdas. En otros donde el patrón original ya producía fallos, la efectividad del ataque se incrementa ampliamente.

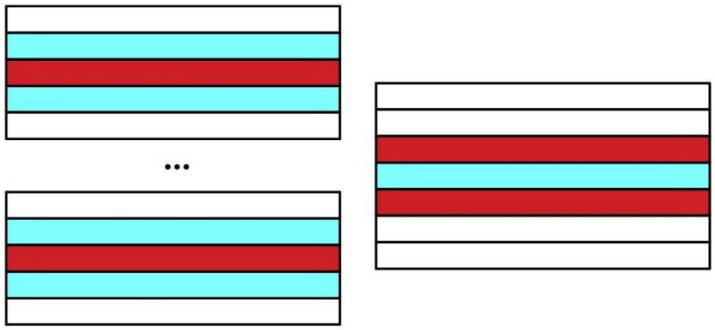

Figura 9: Relación física entre filas en un ataque Single-Sided (izquierda) y Double Sided (derecha). Las filas rojas representan las agresoras, las azules las víctimas principales. Fuente: propia

Por supuesto, el tener que acceder a un par de filas específicas dificulta mucho que el ataque se lleve a cabo, pues como veremos más adelante, un sistema informático común gestiona su memoria física de forma opaca de cara al usuario. Los analistas de Google salvaron el inconveniente realizando ataques RowHammer aleatorios en uno de sus equipos e interpretando el offset entre los fallos de datos producidos y las direcciones de ataque escogidas. Se lanzaron ataques contra otros equipos del mismo fabricante manteniendo ese mismo offset manteniendo la efectividad del mismo.

En el marco del USENIX Security Symposium celebrado en agosto de 2016, se presentó el paper (Peter Pessl, Daniel Gruss, et al.) "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks" (Pessl, 2016). La motivación del trabajo se centra en el estudio de vulnerabilidades en sistemas multiprocesadores enfocados al Cloud Computing, y en como dos procesos ejecutados por CPUs distintas pueden interferir u obtener información sobre el otro al compartir la memoria principal del sistema.

La publicación incluye el desarrollo de un programa que realizaba este tipo de ataques, 'DRAMA', el cual se apoyaba en otro programa secundario escrito en C++ (también desarrollado para la ocasión), y que, mediante ingeniería inversa, podía encontrar grupos de direcciones que pertenecieran a un mismo set.

Consciente de que este programa secundario resulta útil en un ataque RowHammer, ya que permite lanzar patrones de ataque más precisos, que siempre van a utilizar pares de direcciones de memoria que pertenecen a un mismo set (al obtener éstas de un conjunto de direcciones que el programa previamente ha confirmado, comparten esta característica), el paper comenta que se realizaron algunas pruebas de ataque que confirmaron una mejora en la efectividad.

En la fase de desarrollo, se volverá a este trabajo para realizar un análisis en profundidad de las técnicas de ingeniería inversa empleadas.

Técnicas de mitigación del RowHammer

Desde que el problema del RH está sobre la mesa, los fabricantes de DRAM han trabajado para mejorar el aislamiento entre celdas y monitorizando errores relativos a la sensibilidad por perturbaciones electromagnéticas durante el testeo de post-producción, aunque dichos esfuerzos son insuficientes de por sí.

Tras la publicación de la publicación del ISCA, éstos no integrarían en sus nuevos modelos mecanismos de mitigación contra el ataque inmediatamente, dando Intel el primer paso en su familia de procesadores para servidores Ivy Bridge-EP, los cuales podían monitorizar los accesos a memoria para tratar de aliviar el impacto del ataque. Sin embargo, a partir de 2016, los módulos comenzaron a incorporar técnicas de mitigación del RowHammer, liberando al procesador de dicha tarea.

Lógicamente, las marcas de DRAM no han facilitado detalles sobre los mecanismos de mitigación utilizados, pero tal y como apunta la publicación de (Frigo, P., Vannacc, E et al.) "TRRespass: Exploiting the Many Sides of Target Row Refresh" (Frigo, 2020), aunque cada uno haya utilizado soluciones distintas, el principio es el mismo, y se componen de 2 partes esenciales: un contador, que monitoriza qué filas son accedidas con más frecuencia, y un inhibidor, que refresca las filas vecinas a éstas. Se sospecha que el inhibidor podría no crear refrescos nuevos, sino redirigir algunos de los comandos de refresco intencionadamente a las filas consideradas bajo ataque.

Tras la integración de los mecanismos de mitigación, los patrones de ataque de RowHammer conocidos dejaron de producir efecto. Sin embargo, este paper también aporta nuevos patrones de ataque, que conscientes de la naturaleza de la protección, utilizan múltiples filas como señuelo para despistar el contador, haciéndole creer que éstas son los objetivos del ataque, y hacer que la verdadera fila víctima no reciba un refresco a tiempo. Estos patrones se obtuvieron con un fuzzer, que iba escogiendo aleatoriamente conjuntos de filas agresoras. La complejidad de los patrones obtenidos y el hecho de que cada uno funcionara únicamente para un modelo y fabricante concreto, hace que no se puedan deducir conclusiones claras de ellos.

El pasado mayo, el equipo Google Project Zero ha publicado una nueva publicación que introduce un nuevo patrón de ataque (llamado Half-Double) (Salman Qazi, 2021), que consciente de la presencia del inhibidor, utiliza su capacidad independiente de refresco a su favor. Fijada una fila víctima N, utiliza como agresoras las filas N+2 / N-2 (es decir, las dos filas que tiene a distancia 2). El inhibidor interpreta como el objetivo del ataque a las filas vecinas a éstas, refrescándolas a una frecuencia superior a la habitual. Este incremento de refrescos preventivos hace de las filas N+1 / N-1 filas pseudo-agresoras, que, aunque de por sí no generan suficiente efecto sobre la fila N para producir un fallo, combinadas con las filas agresoras a distancia 2 sí logran afectarla.

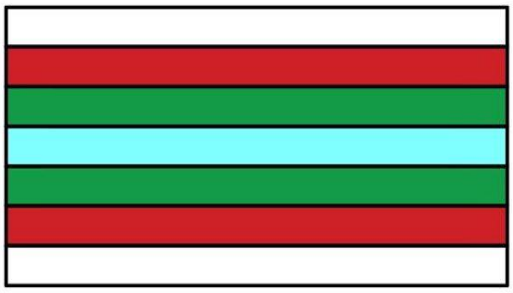

Figura 10: Relación física entre filas en un ataque Half-Double. Las filas rojas representan las agresoras, las verdes las pseudo-agresoras refrescadas por el inhibidor y la azul la fila perturbada (víctima). Fuente: propia

En conclusión, considerando la dirección en la que avanza la industria, cada nueva generación de DDR es más vulnerable al RowHammer, y aunque los mecanismos de mitigación integrados en la DRAM sean suficientes para que este ataque no sea motivo de preocupación urgente hoy día, se sabe que conforme la tecnología se desarrolle y previsiblemente, sea más vulnerable, éstos necesitarán ser mucho más agresivos, y realizar refrescos preventivos con más antelación, limitando las prestaciones del módulo, provocando en un punto del futuro por determinar que se requerirá afrontar la vulnerabilidad con una solución distinta (Mutlu, 2019).

Fundamentos de paginación y gestión de memoria en Linux

En Linux, el sistema gestiona la memoria principal en segmentos de un mismo tamaño, llamados páginas. La paginación permite, por ejemplo, mantener la privacidad entre distintos procesos y que varios procesos compartan memoria y puedan acceder a una misma página, optimizando su uso. Además, con este sistema, si un nuevo proceso solicita una cantidad de memoria solo disponible en fragmentos dispersos de la memoria física, la demanda puede ser satisfecha sin problemas.

La paginación es un método utilizado comúnmente por los sistemas operativos modernos, pero algunos (por ejemplo, Windows) se muestran extremadamente opacos en cuanto a su gestión, dando el mínimo de información y complicando el proceso intencionalmente con el fin de minimizar posibles riesgos. En su lugar, el SO Linux otorga mayor transparencia, permitiendo estudiar la paginación.

Se aclara que, en adelante, se considera que cada dirección de memoria contiene un total de 8 bits (=1 byte) (tamaño de palabra). Esto significa que, por ejemplo, cuando un programa en C almacena un valor float (=32 bits), se necesitan 4 direcciones, y cuando almacena un valor double (=64 bits), un total de 8 direcciones distintas es requerido.

Cuando desde un proceso se consulta una dirección de memoria, se está observando en realidad una dirección virtual. A sus ojos, toda la memoria disponible es un único segmento continuo (espacio de direcciones virtuales) formado por páginas virtuales, cuando en realidad, cada una de

éstas representa a una página física (del espacio de direcciones físicas), que no tiene por qué guardar continuidad con el resto de las utilizadas por el proceso (Rusling, 1999).

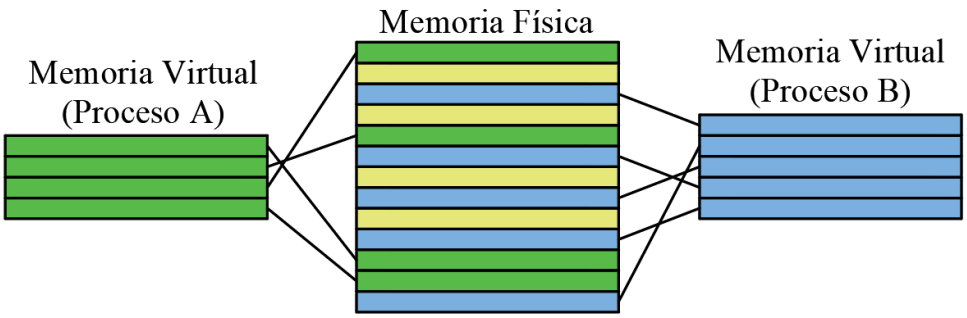

Figura 11: Representación del reparto de páginas físicas de memoria entre dos procesos, para formar sus respectivas regiones de memoria virtual. En amarillo se representan páginas utilizadas por otros procesos. Fuente: propia

Cuando este proceso solicita acceder a una dirección virtual, el procesador recibe la demanda, traduciendo primero la dirección virtual a física y lanzando el acceso a memoria después. Para llevar a cabo la traducción utiliza una tabla de páginas (Page Table), generada por el sistema operativo en el momento de creación del mismo proceso, y accesible a través de la ruta /proc/<PID del proceso>/pagemap. La tabla de páginas está formada por entradas de 64 bits, y enlazan una página virtual con una física. Para consultarla, se requieren permisos de superusuario.

Suponiendo un tamaño de página de 4kB, cuando un proceso solicita el valor almacenado en una dirección virtual X, la CPU guarda por una parte los 12 bits menos significantes (offset) de esa dirección, y utiliza el resto (13-64) como un índice de la tabla de páginas, que apunta a la entrada de la página correspondiente. La entrada, además de algunos flags sobre la página, contiene el número de página física (Page Frame Number). Para acabar la conversión, combina los bits del número de página obtenido con el offset reservado, obteniendo una dirección física que apunta a una página física, y dentro de ella, a un byte concreto (mediante el offset) (kernel.org) (Seaborn M. , 2015).

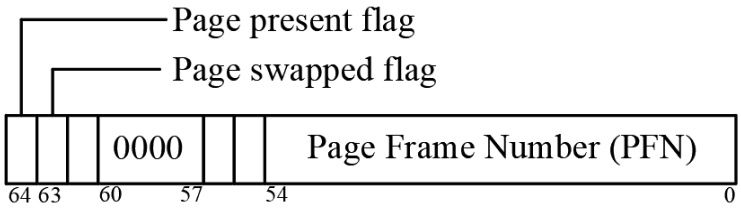

Figura 12: Formato de las entradas en la tabla Page Table. Fuente: propia

El espacio de direcciones virtuales tiene un tamaño igual a toda la memoria disponible. Físicamente, presenta mayor contigüidad que el espacio de direcciones virtuales, pero como se expone a continuación, tampoco garantiza que dos direcciones físicas consecutivas se encuentren físicamente próximas en el módulo de memoria.

Una vez el procesador sabe la dirección física a la que se quiere acceder, se la transmite al MC para que gestione la comunicación necesaria con el módulo de memoria. Para determinar cómo acceder a los niveles internos de la DRAM, el controlador de memoria aplica una serie de funciones XOR a la dirección física (Pessl, 2016). Por ejemplo, para escoger a qué canal dirigirse, tomará uno o varios bits específicos, aplicará un XOR al conjunto, y según el resultado sea '0', o '1', se comunicará con el canal A o B. Este procedimiento se aplica con distintos grupos de bits para acabar obteniendo un canal, DIMM, rank, bank, fila y columna precisos.

El fabricante AMD tiene documentos públicos sobre las funciones de direccionamiento utilizadas en sus productos. Por contra, Intel se reserva esta información, que además cambia según la configuración de memoria utilizada (por ejemplo, según el número de DIMMs instalados).

En conclusión, una dirección de memoria, desde que su acceso es solicitado por un proceso hasta que este finalmente ocurre, experimenta dos transformaciones:

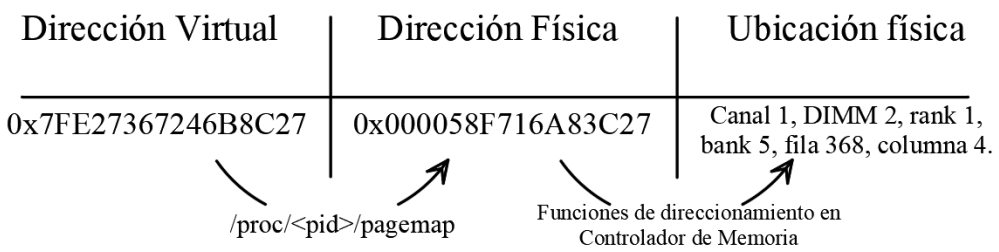

Figura 13: Transformaciones aplicadas a una dirección de memoria durante su acceso. Fuente: propia

El tamaño por defecto de una página en Linux es de 4kB, significando que los 12 bits menos significantes de una dirección virtual (offset) son los mismos que en su respectiva dirección física. Sin embargo, se permite cambiar el tamaño de página por uno mayor (Huge Page), para reducir el tiempo de accesos.

Esta reducción en el tiempo de accesos se explica con la existencia del búfer de traducción anticipada (Translation Lookaside Buffer, TLB), una memoria caché donde se guardan dinámicamente las entradas de la tabla de páginas más utilizadas recientemente, y cuyo acceso es más rápido que la consulta a la tabla de páginas. Si la memoria de un programa está repartida en 50 páginas, en vez de 50.000, la probabilidad de que un acceso a memoria pueda aprovechar la ventaja que brinda el TLB para obtener la página física a la que hace referencia la página virtual es mucho mayor. En sistemas de bases de datos, contar con páginas del mayor tamaño posible marca una diferencia de rendimiento enorme.

Linux permite utilizar Huge Pages de 2MB y de 1GB. Ambas opciones no están disponibles en todos los equipos, siendo poco común poder utilizar las de 1GB. Para habilitar el uso de las HugePages, se puede utilizar el comando 'sysctl', indicando la cantidad de HugePages que el sistema tendrá disponibles para los procesos que las soliciten. Se puede consultar información sobre las HugePages (cantidad disponible, su tamaño...) desde el archivo /proc/meminfo.

Una Huge Page es simplemente una página de mayor tamaño. Cuando un proceso la utiliza, todo el segmento continuo de direcciones virtuales que hay en la página virtual (offset) es un segmento verdaderamente continuo en el espacio de direcciones físicas. Visto de otro modo, permite a un proceso, acceder a un mayor número de bits de las direcciones físicas.

A mayor número de bits que podemos manipular directamente en la dirección física, más se facilita el estudio para comprender cómo ésta es manipulada por las funciones XOR del MC (Pessl, 2016). Con páginas de 2MB, se pueden modificar libremente los 21 bits menos significantes.

Casuística en latencias de acceso

El siguiente planteamiento conforma la base sobre la que se apoyan parte de las conclusiones obtenidas por el presente trabajo realizado, será mencionado en múltiples ocasiones más adelante (Pessl, 2016).

Entendiendo el funcionamiento de un módulo DRAM, del repetido acceso alternado a dos direcciones distintas, podemos ubicar la latencia entre estos accesos en dos rangos de tiempo diferenciables:

·Tiempo de acceso BAJO: (Row Hit)

-Si ambas direcciones pertenecen a sets distintos, y alternamos repetidamente lecturas entre ambos, estaremos leyendo de dos row-buffers distintos, ya cargados y sin requerir el vaciado y posterior activación de una nueva fila sobre ellos.

-Si ambas direcciones pertenecen a un mismo set, pero se encuentran en una misma fila, también estaremos leyendo únicamente de un row-buffer ya cargado, sin provocar nuevas activaciones.

·Tiempo de acceso ALTO: (Row Conflict)

-Si ambas direcciones pertenecen a un mismo set y a una fila distinta, obligaremos al row-buffer a vaciar la fila almacenada previamente tras cada lectura, y a precargar la solicitada. Esta gestión se realiza entre cada acceso, ralentizando el proceso.

Funcionamiento del programa de ingeniería inversa utilizado en DRAMA

Tal y como se comentó en el capítulo 2, 'Estado del arte', DRAMA es un programa desarrollado durante un trabajo de investigación de la Universidad Tecnológica de Graz, que realiza ataques en entornos de Cloud Computing (Pessl, 2016). Éstos parten de una premisa diferente a la del RowHammer, pero los dos ataques tienen un fuerte punto en común: se benefician del acceso preciso a sets en la memoria física. Junto al desarrollo del programa principal, el estudio contribuyó también con un programa que, mediante ingeniería inversa, permitía diferenciar conjuntos de direcciones por el set al que pertenecían. A continuación se describe el funcionamiento del programa de ingeniería inversa, evitando entrar en detalles secundarios para facilitar su comprensión:

En primer lugar, el programa reserva un porcentaje importante de la memoria del sistema. Después, del espacio reservado se eligen aleatoriamente una gran cantidad de direcciones (aproximadamente, 1.000), y además, una dirección aparte (base), también obtenida aleatoriamente.

A continuación, se realizan múltiples iteraciones para medir la latencia de los tiempos de acceso entre la dirección base, y cada una de las mil direcciones reservadas. En cada iteración, el programa realiza muchos accesos alternos (por defecto, 5.000) entre la dirección base y la correspondiente al grupo de mil direcciones. Toma los ciclos de reloj transcurridos antes y después de comenzar, obtiene la diferencia, y calcula la cantidad media de ciclos que ha tomado realizar un acceso entre ambas direcciones.

Transcurridas las iteraciones, se han recopilado mil tiempos de acceso a memoria, correspondientes al acceso alternado entre las direcciones de memoria escogidas y la dirección base. Aplicando el planteamiento descrito en el capítulo anterior, 'Casuística en las latencias de acceso', al observar en un histograma la distribución de estas latencias y siendo capaces de reconocer que están distribuidas en dos grupos diferenciados, se puede deducir la relación física entre cada una de las direcciones seleccionadas y la dirección base.

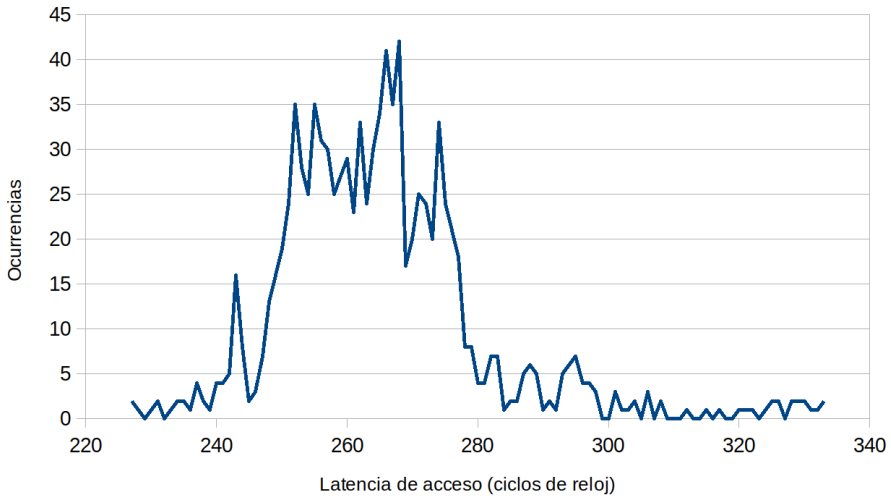

Figura 14:Histograma obtenido por el programa de ingeniería inversa desarrollado para DRAMA. Fuente: propia

Si una dirección de las seleccionadas pertenece al grupo con los tiempos de acceso comparativamente más bajos, significa que, o bien ésta se encuentra en un set distinto al que contiene la dirección base, o bien está ubicada en su misma fila física. El segundo caso es

extremadamente improbable, por lo que el programa siempre supondrá que esta dirección se encuentra en cualquiera de los otros sets. Lógicamente, las direcciones que caen en esta clasificación son la gran mayoría.

En cambio, el criterio con las direcciones cuyo tiempo de acceso es considerablemente más elevado que el resto, lleva a una única conclusión posible: que éstas se encuentren en el mismo set y distinta fila que la dirección base. Es por tanto, una dirección de interés para DRAMA, y este programa la almacena.

Por probabilidad, al haber tomado un número tan grande de direcciones aleatorias, lo más probable es que siempre haya entre ellas alguna dirección que almacenada en el mismo set que la dirección base.

Tras clasificar todas las direcciones, se almacena en una misma estructura de datos las direcciones que pertenecen al mismo set que la base (y por tanto, comparten el mismo set entre sí), descartando el resto.

El programa repite el proceso descrito para conseguir tantos grupos de direcciones ubicadas en un mismo set como sets tenga el sistema (se suponen 8 por defecto, pero el valor puede modificarse con un argumento al lanzar la ejecución). Concluida esta parte del programa, se realiza una traducción de todas las direcciones virtuales almacenadas para obtener sus respectivas direcciones físicas. Finalmente, una serie de funciones con base matemática analizan todos los grupos de direcciones obtenidos para señalar qué función XOR ha aplicado el controlador de memoria para escoger entre cada uno de los sets, indicando los bits utilizados para ello.

Propuesta

A la hora de realizar estudios sobre el RowHammer y someter a pruebas un módulo DRAM, en general, los sistemas informáticos convencionales no permiten realizar accesos a banks y filas concretas de forma transparente, o a conocer como estas están distribuidas físicamente (como se ha descrito en el capítulo ‘Fundamentos de paginación y gestión de memoria en Linux’).

Los investigadores recurren a desactivar los mecanismos de mitigación y realizar ataques RowHammer aleatorios, buscando que los bitflips generados den indicios sobre la localidad de las filas (opción no siempre posible, pues éstos vienen integrados en la memoria) o a crear entornos de estudio, insertando las memorias en una placa FPGA, (con un controlador de memoria que permite observar los comandos específicos que envía a la DRAM) (Frigo, 2020) o monitorizando los voltajes de los pines del puerto DIMM de la placa base (Pessl, 2016).

El resto del documento describe, teniendo en cuenta todo lo expuesto, un estudio sobre la DRAM que busca aportar nuevos métodos -o mejorar existentes- de ingeniería inversa para definir su estructura física e interpretar la relación entre ésta y las direcciones físicas con las que el sistema accede a ella. Se espera que las conclusiones puedan resultar útiles en el marco de estudio de la vulnerabilidad RowHammer, así como otras vías de investigación relacionadas con este tipo de memoria.

Solución Propuesta

Plan de trabajo

En primer lugar, se propone reunir toda la información útil accesible de los SPD en los módulos DRAM, chips de memoria persistente integrados en la electrónica de cada unidad que contienen información útil para que el procesador pueda inicializarlo y gestionarlo correctamente. Se tratará de acceder a los datos que contienen los SPD, y se ponderará si alguno de ellos resulta de utilidad en el contexto del resto del proyecto.

A continuación, se estudiará en profundidad el programa de ingeniería inversa que utiliza DRAMA, para luego buscar formas de mejorar su rendimiento, su eficacia, o su facilidad de uso. También se estudiará la corrección de posibles errores de diseño en el programa.

Por último, se buscará aplicar ampliaciones al programa, o métodos independientes, que puedan arrojar nuevos detalles sobre la estructura de memoria DRAM en la que este se ejecute o cómo se interpreta una dirección física en dicho sistema para lanzar un acceso dentro de memoria. Para realizar este último paso, se utilizará la información que puede deducirse de la diferenciación entre latencias, probando distintas parejas de direcciones para llegar a nuevas conclusiones.

Presupuesto

Para valorar económicamente el coste del presente proyecto, se tiene únicamente en cuenta el tiempo invertido por una persona para llevarlo a cabo. Ya que, aunque se han utilizado múltiples computadores para pruebas, se trataban de equipos comunes fácilmente accesibles. Además, todas las pruebas realizadas a lo largo del proyecto son completamente inofensivas hacia cualquiera de sus componentes.

MANO DE OBRA

	Ud. (h)	€/Ud.	Precio
INGENIERO	400	20	8000
TOTAL (€)			8000

Desarrollo: Acceso al SPD

Con el fin de tener un medio fiable con el que contrastar información obtenida mediante métodos de ingeniería inversa, se trata de recabar toda la información que un equipo maneja y puede facilitar sobre sus módulos DRAM.

Para ser correctamente reconocidos y utilizados por la BIOS y otras herramientas, todos los módulos incorporan el SPD (Serial Presence Detect), una unidad de memoria persistente 'read-only', con datos que el procesador lee al arranque para poder inicializar el controlador de memoria correctamente (capacidad, velocidad de transferencia, fabricante...) (Frigo, 2020).

Tras el descubrimiento del RowHammer, algunos módulos incluían el valor máximo de activaciones de una fila entre refrescos en el SPD, para que los procesadores con un sistema de mitigación integrado (algunos modelos Intel orientados a servidores, como se ha mencionado) no permitieran exceder dicho número, una medida provisional ya que actualmente los sistemas de mitigación están integrados en el propio módulo.

A través del SMBus (SMBus, 2018), un derivado del bus I2C presente en los chips de Intel, se pueden monitorizar parámetros clave de la placa base y sus sistemas integrados, incluyendo la decodificación de la información en el SPD (Guard, 2020). El comando 'decode-dimms' permite listar los parámetros que contiene el SPD, incluyendo, además de los datos mencionados, información acerca de la estructura interna del módulo (número de banks, filas en cada bank, el ancho de la fila (kB)…).

Sin embargo, en la mayoría de portátiles la comunicación con el SMBus / I2C se dirige directamente con el MC (sin exponerse a nivel software), puesto que estos equipos no están preparados para ser re ensamblados con otras configuraciones. Aunque en menor cantidad, se ha conseguido obtener datos sobre los módulos en portátiles con la herramienta 'lshw' (List Hardware), que consulta información en archivos de configuración del sistema, pero sin llegar a conocer la mencionada información sobre la organización interna del módulo.

Con la llegada del estándar DDR4, el SPD pasó de almacenarse de una unidad eeprom a una unidad flash, y en Linux algunas herramientas para decodificar su contenido cuentan con un soporte incompleto. Por ello, según el equipo la información recabada puede ser bastante escasa.

Desarrollo: Mejoras aplicadas a la ingeniería inversa utilizada en DRAMA

Como se ha descrito, el programa DRAMA (Pessl, 2016), desarrollado por un grupo de investigadores de la Universidad Tecnológica de Graz, y que realiza ataques en entornos de Cloud Computing entre distintos procesos que comparten una misma memoria principal, se apoya en un subprograma para conocer grupos de direcciones que pertenecen a un mismo set en la memoria física.

Tras estudiar este programa, se consideró que contaba con potencial para mejorar sus capacidades y posteriormente, recibir nuevas ampliaciones que permitieran conocer más detalles sobre la memoria DRAM mediante ingeniería inversa. A continuación se detallan las mejoras aplicadas:

IMPORTANTE: Para poder ejecutar tanto el método original como las versiones modificadas descritas en el resto del trabajo, se necesita un sistema:

· Con procesador Intel x86 posterior a Pentium IV (al ser necesario utilizar su función de ensamblador 'clflush'). Dicha generación es lo suficientemente vieja como para suponer que cualquier equipo con procesador Intel que se encuentre actualmente cumple el requisito.

· Con SO Linux (el programa se apoya en la transparencia y control que Linux otorga sobre paginación y su gestión de memoria) NO virtualizado (en equipos virtualizados el programa no ha sido capaz de funcionar). Se ha probado su funcionamiento en equipos con las distribuciones Ubuntu, Arch y Manjaro, sin diferencias en la experiencia de uso apreciables.

· Además, necesitará ejecutarse con permisos de superusuario para poder acceder a la tabla de páginas mediante la ruta /proc/<pid>/pagemap (para traducir direcciones virtuales a físicas), así como habilitar el uso de Huge Pages (La instrucción 'clflush' no requiere de privilegios).

1-Mejor exposición de los datos, añadidas opciones de verbose (incluyendo ocultar o no el histograma) y guía de uso con ‘help’ (argumento 'h').

2-Se ha sustituido la función que interpreta el histograma para diferenciar correctamente direcciones de un mismo bank por una versión más compleja. Antes se desechaban muchas direcciones válidas.

3-Para tomar las direcciones aleatorias, el método original simplemente añadía un offset aleatorio al comienzo de la memoria virtual reservada, con un tamaño máximo para escoger una dirección dentro del límite. Esto provocaba que hubieran direcciones cuya latencia se volviera a medir, aunque estuvieran ya guardadas como parte de un set. Ahora, una vez se guarda, se asegura que no vuelva a formar parte del conjunto de direcciones analizadas. Esta aproximación, aunque aun guarda utilidad, fue concebida especialmente para una ejecución del programa durante un largo periodo de tiempo, para que identificara prácticamente la totalidad de la memoria disponible y la agrupara en sus respectivos sets, información que volcaría en un archivo para su posterior análisis.

4-En el programa utilizado con DRAMA, el número de sets es un parámetro por defecto (presupuesto) que el usuario puede introducir como argumento al ejecutar el programa. El programa finaliza una vez encuentra la cantidad de grupos de direcciones indicado, sin comprobar si alguno de esos corresponde al mismo grupo. Ahora, una vez se identifica un grupo de direcciones pertenecientes a un mismo bank, este se compara con cada uno de los ya almacenados para ver si se trata de uno encontrado anteriormente. Si es el caso, las direcciones se añaden, y solo en caso contrario se guardan en un nuevo grupo. El programa continúa hasta que ha transcurrido una cantidad prudente de iteraciones sin encontrar un nuevo set. En resumen, el método nuevo permite determinar la cantidad de sets de memoria con los que cuenta el sistema, mientras que la versión original dependía de un supuesto.

En los sistemas donde se pudo acceder al SPD y obtener la cantidad de banks en memoria, se contrastó que efectivamente coincidía con la que ahora deduce el programa.

Como apunte, y como la lógica podría sugerir, se observó que si comparamos la latencia entre la dirección X y un total de n direcciones aleatorias, siendo z el número de direcciones encontradas que pertenecen al mismo set que X, el resultado de n/z es un número considerablemente próximo a la cantidad de sets totales en el sistema. Por supuesto, este método está sujeto a la aleatoriedad de la toma de direcciones inicial y no se utiliza en el programa, pero contribuye para confirmar la cifra de sets calculada.

5-En muchas de las iteraciones que realizaba el programa, la toma de latencias producía un histograma carente de un “valle” de separación, necesario para diferenciar entre ocurrencias a latencia alta y a latencia baja.

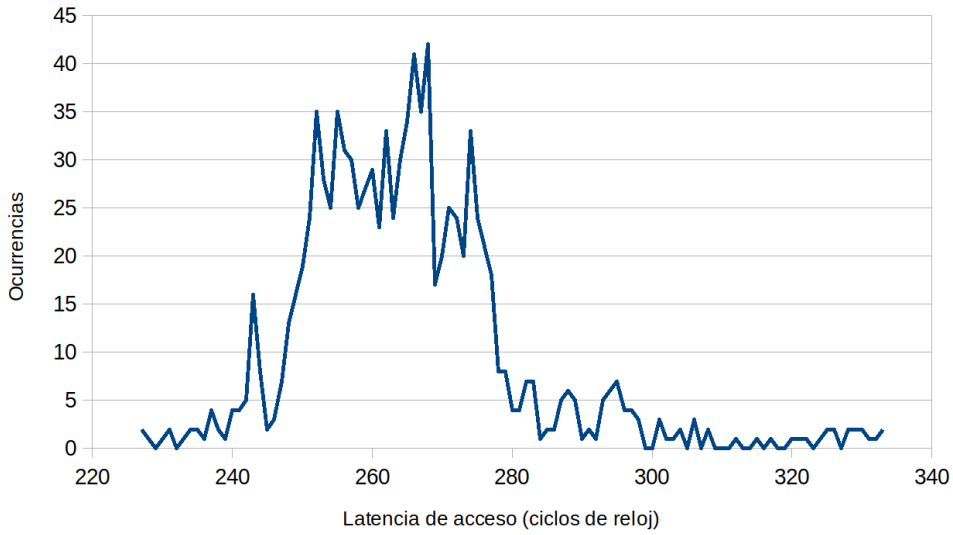

Figura 15: Histograma producido con el programa original, con una dispersión tal que no permite ser utilizado para el propósito del programa. Fuente: propia

Esta situación echaba a perder la medición realizada, obligando a repetir el proceso. El método original integraba una sencilla función para comprobar si esto ocurría y automáticamente relanzaba un nuevo intento de generar el histograma, pero en la mayoría de casos, tras una primera toma fallida, las consecutivas también lo eran. Tras 10 intentos, el programa finalizaba, señalando que no ha sido capaz de determinar suficientes direcciones de un mismo set.

Como ignorar las “manzanas podridas” que aparecían durante las constantes mediciones no era una solución efectiva (pues tras una, las siguientes no mejoran), se trató de buscar la causa del problema y corregirlo.

En un primer lugar se consideró simplemente realizar un corte a partir de un umbral, marcado por iteraciones anteriores en la misma ejecución del programa. Es decir: que si hasta ese momento todas las direcciones con una latencia de acceso superior a 480 ciclos de reloj pertenecían claramente al grupo de direcciones de latencia de acceso alta, en esa medición y en las consecutivas, independientemente de la forma del histograma, se considera que todas las direcciones que excedan ese valor (en una misma medición) se encuentran en el mismo set.

Sin embargo, la dispersión que presentaba el histograma hacía imposible confiar en que no se fuera a acabar tomando direcciones que en realidad forman parte de sets distintos.

Al comprobar que este problema se daba solo en algunas ejecuciones del programa de forma aleatoria, se sospechó que algunos de los accesos estaban siendo ralentizados por la presencia de otros procesos en ejecución y sus propios accesos a memoria. Tras lanzar el programa de ingeniería inversa con el equipo arrancado en modo 'single', (con el programa telinit), es decir, con el mínimo imprescindible de procesos ejecutándose a la vez, los resultados no variaban en absoluto.

La atención pasó a la instrucción 'clflush', que como se ha explicado, elimina de las cachés el contenido de las direcciones accedidas antes de cada acceso para obligar al procesador a buscar dicha información en la memoria principal. Se sospechó que el procesador podría lanzar la orden de anular la validez de los datos en cachés, y sin esperar que esta gestión finalice, realizar los siguientes accesos a memoria. Sin embargo, si 'clflush' no llegara a causar efecto a tiempo, la latencia medida al acceder a una caché sería de muchos menos ciclos a la visible en el histograma.

También se probó el uso de la instrucción de ensamblador ‘CPUID’ por su propiedad serializante (es decir, que serializa la ejecución de instrucciones, garantizando que cualquier transacción en memoria será completada antes de su 'fetching' y ejecución) (Intel, 2016) (Intel, 2010). Aparte del tremendo incremento en la latencia que conllevaba su ejecución cada pocos ciclos, no se contemplaron cambios en los resultados.

Finalmente, la solución al problema resultó más simple de lo esperado: El programa tomaba los ciclos transcurridos hasta ese punto, ejecutaba todos los accesos al par de direcciones, borrando la memoria de las cachés entre cada iteración, y luego consultaba de nuevo los ciclos transcurridos, interpretando la latencia de accesos como la diferencia de ciclos entre las dos tomas. Es decir, además de la latencia en los accesos se incluía en la medición el tiempo de ejecución de 'clflush', que añade una latencia variable que perjudica la toma de datos en algunas ejecuciones. Cambiando el código para que calcule los ciclos exclusivamente entre el acceso a las direcciones, las medidas pasan a ser siempre claras.

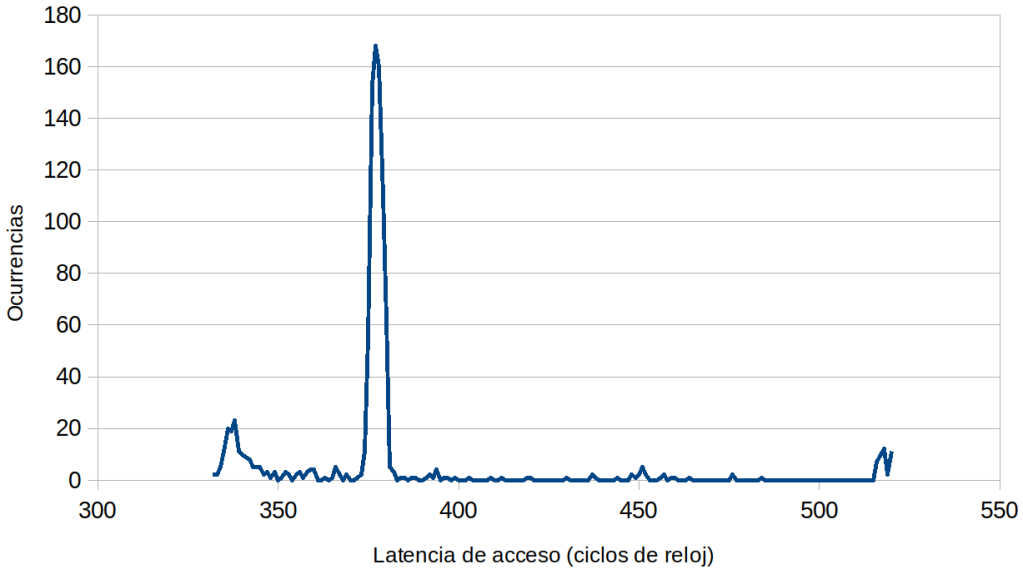

Figura 16: Histograma obtenido tras la corrección del error. Se puede diferenciar perfectamente un grupo de direcciones con latencias de acceso mucho mayores que el resto. Fuente: propia

A favor de los investigadores que desarrollaron el programa original, hay que señalar que el problema descrito se encontró al ejecutarlo en máquinas posteriores a 2016, fecha de su publicación, y no al ejecutarlo en máquinas anteriores (fabricadas en 2011 y 2012). Aunque la corrección sí reduce la dispersión entre los dos grupos de direcciones en todos los casos, en máquinas antiguas no llegan a mezclarse entre sí. En caso de que este inconveniente no se produjera nunca durante su estudio, es comprensible que no percibieran el error.

6-Las anteriores mejoras del programa permitieron reducir el número de lecturas durante cada medición de latencia. Originalmente establecido por defecto en 5000, el valor se redujo a 400, manteniendo unos resultados totalmente válidos en la mayoría de equipos testados (En algunos equipos más sensibles, se requiere incrementar el número unos cuantos cientos de lecturas).

Aplicadas todas las mejoras sobre el método de ingeniería inversa que utilizaba DRAMA, los bits indicados al final de cada ejecución se mantenían ahora estables, salvo alguna excepción. Sin embargo, la información aportada era difícilmente interpretable:

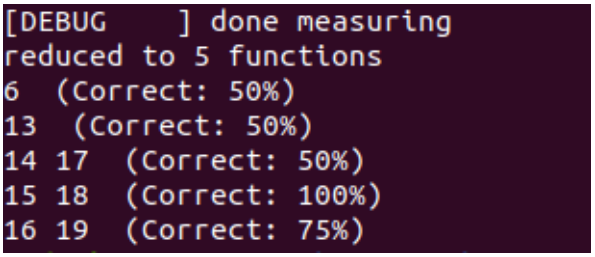

Figura 17: Salida en la ventana de comandos del programa de ingeniería inversa que utilizaba DRAMA. La información hace referencia a los bits de la dirección física con los que se selecciona el bank. Fuente: propia

Aunque obviamente la salida indica las funciones XOR aplicadas a las direcciones físicas para escoger a qué set se refieren, faltaba información sobre qué diferencia a cada función, y la documentación tampoco aportaba información útil para entenderlo. Hay que recordar que el programa que se ha manipulado es una herramienta desarrollada por el equipo de investigadores y para su propio uso, los cuales sí debían saber estas cuestiones.

Ante esta situación, se estudiaron formas independientes para definir las funciones XOR que el MC aplicaba a las direcciones físicas para seleccionar la ubicación física (canal, DIMM, rank, etc.) en la memoria DRAM.

La descripción de los siguientes métodos se fundamenta en la combinación del uso de Huge Pages de 2MB, para poder estudiar la influencia de los 21 bits menos significantes de la dirección, con el ya conocido método de clasificación de direcciones mediante las latencias de acceso, que se describe en el capítulo (‘Casuística en latencias de acceso’).

A través de la función de C++, 'system( )', se añadió al programa mejorado una función que reserva por si sola una cantidad de HugePages al inicio, para no tener que ejecutar los comandos de sysctl manualmente tras cada arranque.

Mínimo segmento común en una fila

Tras un periodo de experimentación, se determinó un método con el que determinar el tamaño mínimo de un segmento de memoria que fuera contiguo tanto lógicamente (dirección física) como físicamente (es decir, mismo bank, misma fila). A partir de los equipos donde se permitía el acceso al SPD, se podía confirmar como estándar un tamaño de fila en los sets de los módulos de memoria de 8kB, por lo que este suponía el segmento continuo de máximo tamaño posible.

Como apunte, si el tamaño del segmento es inferior a 4kB, ésta puede detectarse aún sin utilizar Huge Pages, ya que al modificar los primeros 12 de una dirección virtual utilizando páginas de 4kB, estamos modificando los mismos bits en la dirección física.

El método consiste en tres pasos, que combinados, permiten despejar toda clasificación en cuanto a la relación entre las direcciones de ese segmento, excepto a una contigüidad física:

Teniendo reservada una región de memoria virtual, se toma una dirección X dentro de la misma, que tenga sus 13 bits menos significantes a 0. Se aplica un offset (el utilizado para el método fue de 4kB, pero no es importante) a esta dirección virtual para obtener una segunda dirección Y, a la que se le seguirá añadiendo el offset una vez tras otra hasta que el tiempo de acceso entre ambas direcciones alternadas sea alto [(X vs X + (1*4096)), (X vs X + (2*4096)), …]. Esto significará que del par de direcciones obtenidas X e Y, ambas se encuentran físicamente en un mismo set, pero diferente fila.

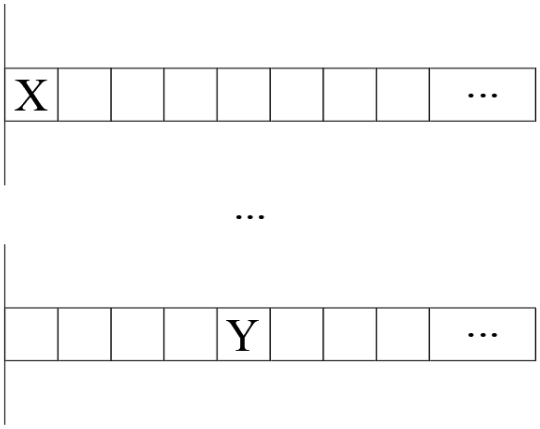

Figura 18: Relación física entre las direcciones X e Y. Fuente: propia

Ahora, comenzando por X e Y, se vuelven a medir las latencias de acceso, incrementando en una unidad la dirección X y repitiendo las lecturas hasta que los tiempos de acceso pasan a ser bajos, conservando las direcciones utilizadas hasta ese momento para obtener un rango de direcciones que sigue manteniendo la relación con Y. [(Y vs X + 1), (Y vs X + 2), … , (Y vs X + N)]

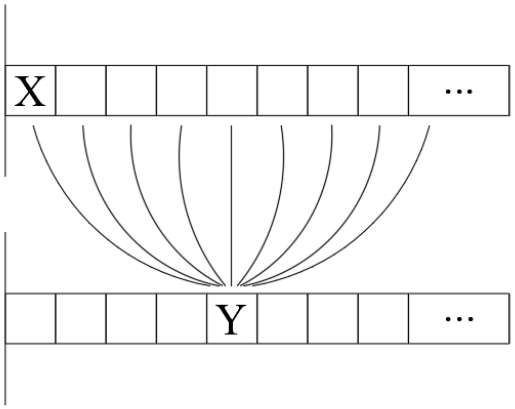

Figura 19: Proceso de medición en las latencias de acceso entre la dirección Y y una sucesión de direcciones físicamente contiguas. Fuente: propia

Se cuenta por tanto con un rango de direcciones [X – X + N] que se puede asegurar, forma parte del mismo set. Para corroborar que se trata de un segmento que además se encuentra en la misma fila, se mide la latencia entre la dirección X y cada una del resto de direcciones del mismo

[(X vs X + 1), (X vs X + 2), … , (X vs X + N)]. Si los tiempos de acceso son bajos, sabemos que es porque todas las direcciones están accediendo a la misma fila, al no provocar activaciones múltiples filas aun accediendo a un mismo set.

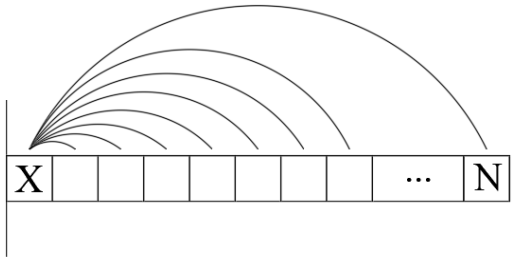

Figura 20: Proceso de comprobación para deterimnar que una serie de direcciónes con bank común también comparten misma fila en el módulo DRAM. Fuente: propia

Este método se probó en los equipos de test disponibles, obteniendo segmentos mínimos de hasta 64 bytes en algunos equipos, y máximos de 8kB. Siguiendo con las pruebas, se comprobó que en equipos con un segmento inferior a 4kB, la mayoría de módulos estaban instalados en modo Dual Channel.

El Dual Channel utiliza dos buses de datos independientes de 64 bits para comunicarse con las ranuras DIMM. Para soportar este modo, tanto el procesador como la placa base (según el modelo, puede permitir conectar uno o dos DIMMs a cada uno de los canales, a través de ranuras conectadas internamente que comparten el canal) deben ser compatibles (Crucial, s.f.).

Es posible encontrar configuraciones de 1, 2, 4, 6 y 8 canales (las dos últimas, enfocadas únicamente a servidores), pero las más comunes en la informática de consumo a día de hoy son las dos primeras. Esto es, Single Channel y Dual Channel.

Se sospechó que el uso del doble canal tenía relación directa con el tamaño del segmento mínimo, y se volvió a ejecutar el programa de prueba tras cambiar la configuración de los módulos DRAM en los equipos de sobremesa, instalando primero un solo módulo, y luego los dos módulos en DIMMs internamente conectados a un mismo canal. En ambos casos el modo de funcionamiento corresponde a Single Channel, y la segmentación mencionada ya no estaba presente, alcanzando todos un mínimo tamaño de memoria físicamente continua de 4kB.

Se demuestra así, que para aprovechar de forma eficiente el doble canal, el MC distribuye entre ambos módulos segmentos de memoria de tamaño reducido que el sistema cree contiguo (visto desde el espacio físico de direcciones), consiguiendo así que por pequeña que sea la cantidad de memoria utilizada por un proceso, este pueda beneficiarse del mayor ancho de banda de transferencia de datos, ya que un 50% de su memoria estará alojada en uno de los módulos, siendo accesible a través de un bus de datos, y el otro 50% será accesible a través del otro bus.

Se deduce que el bit que marca el fin del segmento cuando se activa a ‘1’ en el caso de Dual Channel forma parte de la función XOR que utiliza el MC para seleccionar el canal. En la informática de consumo, lo habitual no es encontrar equipos que soporten más de dos canales, por lo que podría parecer lógico que solo fuera necesario ese único bit para elegir entre ambos. Sin embargo, no se ha confirmado que así sea (especialmente, porque se recuerda que solo se tiene acceso a los primeros 21 bits de la dirección física), y al tratarse de una medida para proteger esta función de direccionamiento,

De hecho, en el único caso donde un equipo tenía su memoria física segmentada en regiones inferiores a 4kB, y al mismo tiempo funcionaba en modo Single Channel, se trataba de un equipo portátil, cuyo módulo estaba ensamblado y no permitía modificaciones. La configuración en estos equipos suele ser relativamente particular y este caso podría ignorarse a la hora de tomar conclusiones en equipos de sobremesa, pero teniendo ya una discrepancia, se considera que hace falta una cantidad mayor de equipos testados para hacer absolutos.

Como conclusión, los resultados obtenidos indican que si tras ejecutar el código de ampliación aquí descrito se obtiene que el mínimo tamaño del segmento de direcciones virtuales que guarda continuidad física es de al menos 4kB, el equipo en cuestión funciona en configuración Single Channel.

Finalmente, el método se añadió como ampliación de la versión mejorada del programa utilizado en DRAMA. Tras determinar el número de banks que tiene el sistema, se realiza el proceso descrito, mostrando a continuación el tamaño del segmento físico continuo, y el bit de la dirección física que ha marcado el fin del segmento. Si el segmento es menor a 4kB, y el equipo funciona en modo Dual Channel, el estudio realizado indica que ese bit se utiliza en la función del MC para determinar el canal al que se accede.

Localización de bits utilizados en las funciones de selección de bank y fila

Con las páginas de 2MB ya configuradas y en uso, se estudia la localidad física de este segmento. A partir del método ya descrito, se obtiene una dirección X del mismo set que el inicio de una página (sea una dirección cualquiera del espacio reservado, cuyos primeros 21 bits sean 0), y se comprueba la latencia de accesos entre X y toda la región de 2MB, comenzando por el inicio de la página y aplicando offsets de 8kB (este experimento se llevó a cabo en un equipo funcionando en Single Channel, cuyo segmento de memoria física se había comprobado, abarcaba 8 kB, toda la fila física, lo que permite descartar el análisis de los bits utilizados para seleccionar canal y columna). Es decir, tomamos latencias de (X vs N, X vs N+8.192, X vs N+16.384, ... , X vs N+2.093.056), en un total de 256 iteraciones distintas.

Tras este proceso, se almacenan las direcciones correspondientes con las que se han obtenido tiempos de acceso alto, y se vuelve a empezar, cambiando X por (X + 8.192), para volver a recorrer de nuevo toda la región de 2MB. Desplazando poco a poco la dirección X por toda la Huge Page, acabamos obteniendo grupos de direcciones que sólo varían en sus 21 bits menos significativos, y cuyas variaciones sólo tienen el propósito de apuntar a sets y filas distintos.

Agrupando todas las direcciones de un mismo set, se obtenía un conjunto de direcciones cuyos bits sólo cambiaban para señalar filas distintas, indicando que eran utilizados por el MC para sus funciones de direccionamiento. Tras diferenciar estos bits, si se comparan direcciones físicas correspondientes a distintos banks, y se ignoran los ‘bits de fila’, podemos aislar también los bits utilizados en ese espacio de 21 bits que se utilizan para seleccionar el bank.

Por supuesto, en la región de bits que no podemos manipular (del bit 22 en adelante) siguen habiendo bits que se utilizan para la selección de fila, junto a los que se ha podido aislar, ya que con sólo estos, no es posible abarcar todo el rango de filas que hay en un mismo set. Aunque no se comprueba (en el apartado ‘Trabajos futuros’, se señala como propuesta de ampliación), lo más probable es que los bits a los que tenemos acceso no lleven a filas contiguas, con tal de proteger la memoria del RowHammer. Se supone así porque no es una medida que requiera un coste o elaboración altos, y reduce mucho la posibilidad de éxito del ataque.

La existencia de estos bits que no pueden manipularse, sumado al hecho de que un mismo bit en algunas ocasiones es utilizado en varias funciones XOR distintas, hace de la determinación de las funciones una tarea difícil y enrevesada.

Conclusiones

El presente trabajo ha servido como un puente excelente para profundizar en muchas áreas de la informática y la electrónica, aprendiendo nuevos conceptos y ampliando conocimientos ya adquiridos a lo largo del curso, durante las aproximadamente 400 horas que han sido necesarias para su realización.

Sobre el desarrollo y cumplimiento del primer objetivo, es decir, el proceso de documentación y formación sobre todo el marco conceptual que rodea el tema del trabajo, se considera que la labor realizada cumple perfectamente lo pautado: Se ha realizado un exhaustivo trabajo de investigación y recopilación de datos, obtenidos poco a poco a partir de publicaciones científicas, fichas técnicas, documentos formativos, foros de discusión, etc.

En muchos casos la información entre fuentes se comprobaba contradictoria, (nomenclaturas distintas, valores típicos…) contrastando cada pequeño concepto con múltiples fuentes y si era posible, con los propios equipos disponibles, desarrollando pequeños programas específicos (tamaño de palabra utilizado, tamaños de páginas, distribución de los espacios de direcciones…). Como es lógico, y desde una perspectiva general, se ha hecho frente también al secretismo de los fabricantes sobre la memoria DRAM

Cabe reseñar el esfuerzo dedicado a entregar, en el momento de la presentación, una documentación sobre el estado del arte completamente actualizada, incluyendo publicaciones científicas con importantes aportaciones de este mismo año, lanzados después de que la fase de documentación se diera por concluida, ya durante la fase de desarrollo.

En cuanto a la fase de desarrollo, en lo personal, era consciente de que no habiendo cursado el grado de Ingeniería Informática, encontraría ciertas dificultades y necesitaría formarme sobre bastantes áreas de forma independiente. Detrás del proceso de comprensión de cada uno de los detalles del programa de ingeniería inverso utilizado por DRAMA hay decenas y decenas de horas, además de la elaboración y manipulación de mucho código propio, modificado una vez tras otra para conseguir ganar familiaridad. Digna de mención es el proceso de corrección del error que creaba una distorsión en las lecturas del histograma, que como describe la memoria, llevó a deducir múltiples causas erróneas antes de llegar a la solución correcta.

La misma cantidad de reflexión demandó el proceso de ampliaciones. Al establecer como objetivo el desarrollo de algún añadido útil para la ingeniería inversa de las direcciones de memoria de la DRAM, sin más especificaciones, se avanzó por muchos callejones sin salida hasta encontrar resultados, pero al final, se considera que la dirección tomada a partir del proceso de documentación fue la correcta, ya que permitió encontrar planteamientos, que aunque no resuelvan un gran problema sobre el tema, sí aportan nuevas formas de afrontarlo. Teniendo en cuenta los recursos y formación de la que se partía, representa un buen resultado.

En conclusión y tras lo expuesto, las impresiones sobre los resultados obtenidos con el proyecto general son satisfactorias.

Relación del trabajo desarrollado con los estudios cursados

Los conocimientos asimilados durante el Master en Ingeniería de Computadores y Redes impartido por el DISCA han probado ser de un gran valor durante el desarrollo del presente trabajo. En primer lugar, la familiaridad adquirida para sintetizar artículos académicos y documentación técnica durante el curso ha facilitado notablemente el proceso de documentación.

Además, las prácticas de programación realizadas con lenguaje C durante el Máster, que se suman a la experiencia previamente adquirida, también han resultado una ayuda fundamental para obtener la desenvoltura necesaria para afrontar un programa que aprovecha en gran medida el bajo nivel que lo caracteriza (es decir, la precisión con la que podemos realizar accesos a memoria, tipar y manipular variables a nivel de bit, etc.).

Esta capacidad para realizar instrucciones a bajo nivel es inútil si no va acompañada de conocimientos sobre la estructura de un procesador, cómo gestiona las instrucciones recibidas y los accesos a memoria solicitados, los cuales también comprendió el temario del curso.

Trabajos futuros

A continuación se señalan algunas direcciones que podrían tomar nuevos estudios, a partir de los principios que el presente expone:

En primer lugar, resultaría interesante estudiar si los bits de selección de fila que podemos modificar utilizando páginas de 2MB permiten acceder a filas contiguas, permitiendo un nivel de transparencia suficiente para ejecutar ataques RowHammer. Para ello, habría que ejecutar aleatoriamente entre pares de filas distintas de un mismo set el patrón de ataque Half-Double (Salman Qazi, 2021), que actualmente sí produce fallos de memoria pese a los mecanismos de mitigación, y comprobar si alguna combinación produce algún efecto.

Además, otro trabajo podría centrarse en la determinación de diferencias entre la gestión de memoria en sistemas integrados (véase, la mayoría de portátiles, móviles…) o modulares(ordenadores de sobremesa, principalmente). Es decir, si hay alguna generalidad por parte de alguno de los dos grupos que hasta ahora haya pasado inadvertida.

Como pequeña mejora posible a implementar, el código desarrolado podría ser capaz de ajustar por sí mismo el número de lecturas en cada medición de latencia, probando primero una cantidad reducida (100, por ejemplo) e ir incrementandola hasta que el histograma obtenido permita identificar los dos rangos de latencias. Al ser éstos algo distintos en cada equipo, no se puede utilizar siempre el mismo valor, y ahora mismo es el usuario el que debe modificar el parámetro manualmente.

Glosario de términos y acrónimos

Términos

Celda	Estructura electrónica capaz de almacenar 1 bit de memoria.
Bank / set	Matriz de datos DRAM.
Chip DRAM.	Conjunto de banks.
Rank	Conjunto de chips DRAM.
Canal	Bus de datos a través del cual se realiza la comunicación entre la memoria y la CPU.
Page Table	Tabla de páginas, se utiliza para traducir direcciones virtuales a físicas.
HugeTable	Formato de página de un mayor tamaño al de la página por defecto.
SPD	Chip de memoria de poco tamaño integrado en la memoria DRAM y otros componentes que contiene información sobre ella.
Dual Channel	Configuración hardware en un equipo que permite a la CPU realizar accesos a memoria a través de dos buses de datos individuales.
Single Channel	Configuración hardware en un equipo donde la CPU realiza accesos a memoria a través un único bus de datos.

Acrónimos

CPU	Core Processing Unit
MOSFET	Metal-Oxide-Semiconductor Field-Effect Transistor
RAM	Random Access Memory
DRAM	Dynamic RAM
SRAM	Static RAM
LPRAM	Low-Power RAM
DDR RAM	Double Data Rate RAM
EEC	Error correction code
MC	Memory Controller
SPD	Serial Presence Detect
PID	Process Identifier
DRAMA	DRAM Addressing

Referencias

Bhati, I. C. (2015). DRAM refresh mechanisms, penalties, and trade-offs. *IEEE Transactions on Computers*, 2-3.

Crucial. (s.f.). *¿Qué es la memoria de doble canal?* Obtenido de https://www.crucial.es/articles/about-memory/what-is-dual-channel-memory

Guard, D. (Febrero de 2020). *DDR4 memory information in Linux.* Obtenido de https://damieng.com/blog/2020/02/08/ddr4-ram-spd-linux

Intel. (2010). Benchmark C Code Execution I32-64. 9.

Intel. (2016). *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1.*

kernel.org. (s.f.). Obtenido de Pagemap, from the userspace perspective: https://www.kernel.org/doc/Documentation/vm/pagemap.txt

Kim, J. S. (2020, Mayo). Revisiting RowHammer: An Experimental Analysis of Modern DRAM Devices and Mitigation Techniques. *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2-3.

Kim, Y. D. (2014). Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. *ACM SIGARCH Computer Architecture News*, 3-11.

Mutlu, O. &. (2019). Rowhammer: A retrospective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, (págs. 1-4).

Pessl, P. G. (2016). DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. *25th USENIX security symposium*, (págs. 565-579).

Rusling, D. A. (1999). *The Linux Documentation Project*. Obtenido de https://tldp.org/LDP/tlk/mm/memory.html

Salman Qazi, Y. K. (25 de Mayo de 2021). *https:/security.googleblog.com.* Obtenido de Introducing Half-Double: New hammering technique for DRAM Rowhammer bug: https://security.googleblog.com/2021/05/introducing-half-double-new-hammering.html?m=1

Seaborn, M. &. (2015). Exploiting the DRAM rowhammer bug to gain kernel privileges. *Black Hat.*

Seaborn, M. (2015). *How Physical addresses map to rows and banks in DRAM.* Obtenido de Lacking Rhoticity: http://lackingrhoticity.blogspot.com/2015/05/how-physical-addresses-map-to-rows-and-banks.html

SMBus. (2018). *System Management Bus (SMBus) Specification.*

Anexo: Programa de Ingeniería Inversa Mejorado

```
#include <bitset>  //std::bitset, arrays de bits de acceso independiente

#include <iostream>//Funciones cin & cout

#include <stdio.h> //printf(), scanf()

#include <assert.h>//assert() function (return 1 if true)

#include <fcntl.h> //fcntl() [Acts over an open file] y open() [Gets file descriptor]

#include <linux/kernel-page-flags.h>

#include <stdint.h>//std library for int types

#include <sys/sysinfo.h>//sysinfo() returns certain memory stats, load avg & swap usage

#include <sys/mman.h> //Memory management declarations

#include <unistd.h> //Reúne la declaración de varias ctes, tipos y funciones del marco

            //de la API del estándar POSIX. (P.ej, size_t, lo trae de otro .h)

#include <string.h> //Definición de tipos, macros y funciones relativas a strings.

#include <time.h>   //var types, macros & functions to manipulate date & time (like timeval struct)

#include <stdlib.h> //var types, macros & functions of general purpose

#include <map>      //map< , > structure

#include <list>     //Type of container

#include <utility>  //Includes tuple objects, amongst others
```

```
#include <fstream>  //File stream classes

#include <set>      //Tipo de contenedor asociativo. Cada elemento debe ser único, es

                //su forma de identificarse, y no puede modificarse una vez añadido.

#include <algorithm>//Funciones para usar en rangos de elementos, directamente en los valores.

#include <sys/time.h>//Contiene estructuras que miden intervalos de tiempo (timeval...)

#include <sys/resource.h>//Define constantes a usar en funciones getpriority()/setpriority()

                    //aplicadas a procesos, usuarios...

#include <sstream>  //Stream containing Strings, useful for stream-style manipulation on them.

#include <iterator> //needed to read from map< , >

#include <math.h>   //Funciones trigonométricas, logarítmicas, exponenciales...



#include "measure.h"//Define LOG macros y printBinary(x). Define el uint64_t 'pointer'

                //y el pair < pointer, pointer > 'addrpair'

                //"" Para buscar primero en la misma ubicación que este código



                // ------------ global settings ----------------

                //int verbosity = 0;

                // default values

                size_t num_reads = 400;
```

```
double fraction_of_physical_memory = 0.6;//% DRAM a consumir, mucha matará el proceso

size_t expected_sets = 8;//Número total de banks (=#ranks*#banks/rank)

int set_comparative;



#define MEMORY_DIVIDER     4096 //Used to prevent the selection of add. of same page

#define POINTER_SIZE      (sizeof(void*) * 8)//Tamaño del puntero en bits (nºbytes * 8)

#define ADDRESS_ALIGNMENT  6

#define MAX_XOR_BITS      7

// ---------------------------------------------



#define ETA_BUFFER 5

#define MAX_HIST_SIZE 1500



std::map<int, std::vector<pointer> > functions;//Map int - vector de uint64_t

std::vector <std::vector<addrpair>> sets;//Vector de vectores de uint64_t

std::vector <std::vector<pointer>> phys_sets;//Vector de vectores de uint64_t

int g_pagemap_fd = -1;//FileDescriptor del Pagemap (-1 para luego com+çprobar si se modificó
o <0)

size_t mapping_size;
```

```
        void *mapping;

        std::vector <int> mem_offsets;//Will contain all different

        int separation_delay = 2000;

        int lines_to_erase = 1;

        // ---------------------------------------------

        size_t getPhysicalMemorySize() {

           struct sysinfo info;//Genera instancia de struct sysinfo 'info'

           sysinfo(&info);//Populates sysinfo struct

           return (size_t) info.totalram * (size_t) info.mem_unit; //8.229.638.144 B
(Mem/modulo*#Módulos)

        }

        // ---------------------------------------------

        const char *getCPUModel() {

           static char model[64];

           char *buffer = NULL;

           size_t n, idx;

           FILE *f = fopen("/proc/cpuinfo", "r");

           while (getline(&buffer, &n, f)) {

              idx = 0;
```

```
        if (strncmp(buffer, "model name", 10) == 0) {
            while (buffer[idx] != ':')
                idx++;
            idx += 2;
            strcpy(model, &buffer[idx]);
            idx = 0;
            while (model[idx] != '\n')
                idx++;
            model[idx] = 0;
            break;
        }
    }
    fclose(f);
    return model;
}
// ------------------------------------------------
void setupMapping() {
  /*Calcula, reserva, y puebla de datos el rango de memoria en RAM*/
    mapping_size = //(Bytes)
```

```
        static_cast<size_t>((static_cast<double>(getPhysicalMemorySize())
                        * fraction_of_physical_memory));
//Si la fracción es aprox 0, toma por defecto 2GB
if (fraction_of_physical_memory < 0.01)
    mapping_size = 2048 * 1024 * 1024u;//u-unsigned to avoid overflow
//-Reservamos rango de memoria (void* mapping)
mapping = mmap(NULL, mapping_size, PROT_READ | PROT_WRITE,
            MAP_POPULATE | MAP_ANONYMOUS | MAP_PRIVATE | MAP_LOCKED, -1, 0);
assert(mapping != (void *) -1);//Check for MAP_FAILED (=(void *)-1)


//-Poblamos de datos el rango de memoria
logDebug("%s", "Initialize large memory block...\n");
for (size_t index = 0; index < mapping_size; index += 0x1000) {
    pointer *temporary = //uint64_t*
          reinterpret_cast<pointer *>(static_cast<uint8_t *>(mapping)
                                + index);
    temporary[0] = index;//1 uint64 cada 4096 B (0x1000)
/*Maybe a marker value at the start of every 4K page(?)*/
}
```

```
    logDebug("%s", " done!\n");
}

// ------------------------------------------
size_t frameNumberFromPagemap(size_t value) {
  /*Aplica máscara de bits a la PTE, devolviendo solo los primeros 54 bits (pPFN)*/
    return value & ((1ULL << 54) - 1);
}

// ------------------------------------------
pointer getPhysicalAddr(pointer virtual_addr) {
    //off_t es un signed int de 64 bits.
    pointer value; //uint64_t para almacenar una entrada del PageMap

    /*Quita los primeros 12 bits (offset) para obtener el índice (vPFN), y
      multiplica para que el índice coincida con el comienzo de cada entrada (64b)*/
    off_t offset = (virtual_addr / 4096) * sizeof(value);//(*8(B))

    //Extrae PTE del pagemap
```

```
    int got = pread(g_pagemap_fd, &value, sizeof(value), offset);
    assert(got == 8);
    assert(value & (1ULL << 63));// Check the "page present" flag.
    /*Ejecutando sin permisos de superusuario, los asserts se superan pero
    el fpn obtenido será 0*/

    /*Extrae el campo 'frame number' de la entrada del PageMap, bits [0-54]*/
    pointer frame_num = frameNumberFromPagemap(value);
    /*Pageframe*pf.size (0x1000) + offset (máscara OR con primeros 12 bits de v.addr)*/

    //printf("Dirección virtual: 0x%012lX\n", virtual_addr);
    //printf("Frame Number:      0x%012lX\n", frame_num);
    //printf("Dirección física:  0x%012lX\n\n", (frame_num * 4096) | (virtual_addr & (4095)));
    return (frame_num * 4096) | (virtual_addr & (4095));//0x0FFF
}

// ------------------------------------------------
void initPagemap() {
    /*Obtiene el File Descriptor del pagemap de este proceso (self)*/
```

```
    g_pagemap_fd = open("/proc/self/pagemap", O_RDONLY);
    assert(g_pagemap_fd >= 0);
}

// ------------------------------------------------
long utime() {
    struct timeval tv; //To represent a time interval
    gettimeofday(&tv, NULL);
    /*Segundos (tv_sec) y micro s. restantes (tv_usec) desde Unix Epoch*/
    return (tv.tv_sec) * 1000 + (tv.tv_usec) / 1000; //Devuelve total (ms)
}

// ------------------------------------------------
uint64_t rdtsc() {
    uint64_t a, d;
    /*Usage of CPUID as it is a serializing function that finishes all previous
    tasks on execution, to get a precise benchmark on the timing test*/
    /*Sets rax (64 bits) to 0, neccesary to get manufacturer on cpuid*/
```

```
    asm volatile ("xor %%rax, %%rax\n" "cpuid"::: "rax", "rbx", "rcx", "rdx");
    asm volatile ("rdtscp" : "=a" (a), "=d" (d) : : "rcx");
    a = (d << 32) | a;
    return a;
}

// ------------------------------------------------
uint64_t rdtsc2() {
    uint64_t a, d;
    asm volatile ("cpuid"::: "rax", "rbx", "rcx", "rdx");
    asm volatile ("rdtscp" : "=a" (a), "=d" (d) : : "rcx");
    //asm volatile ("cpuid"::: "rax", "rbx", "rcx", "rdx");
    a = (d << 32) | a;
    return a;
}

// ------------------------------------------------
uint64_t getTiming(pointer first, pointer second) {
```

```
//Toma las dos direcciones virt. almacenadas en first y second, y accede a ambas
//durante 'num_reads'. Con rdtsc antes y después, calcula el número de ciclos
//al que le lleva realizar todas las lecturas. Repite el proceso 4 veces,
//devuelve el menor número de ciclos
 size_t min_res = (-1ull);//Max of 64 b value
 for (int i = 0; i < 4; i++) {

    size_t number_of_reads = num_reads;// 400 by default
    volatile size_t *f = (volatile size_t *) first;
    volatile size_t *s = (volatile size_t *) second;

    for (int j = 0; j < 10; j++) sched_yield();
      //sched_yield - fuerza al hilo en ejecución a ceder el procesador a otro
      //hilo de misma prioridad, y se retira a la última posición de la cola
    size_t t0, t1 = 0;
    uint64_t res, difference;

    while (number_of_reads-- > 0) {//1º comprueba, luego resta si TRUE
       //Vacía las líneas de caché de las direcciones f y s
```

```
                    //(r = constraint, permite al comp. ejercer sobre todos los registros)
                    asm volatile("clflush (%0)" : : "r" (f) : "memory");
                    asm volatile("clflush (%0)" : : "r" (s) : "memory");
                    //asm volatile ("cpuid"::: "rax", "rbx", "rcx", "rdx");//Serializing function, really slows
execution
                    t0 = rdtsc();

                    *f;//Menciona el contenido en la primera dirección
                    *(f + number_of_reads);//Se accede a otro valor aleatorio, necesario por qué(?)----------
----------duda
                    *s;
                    *(s + number_of_reads);
                    difference = rdtsc2() - t0;
                    t1 = t1 + difference;
                }
                res = t1 / num_reads;
                for (int j = 0; j < 10; j++)
                    sched_yield();
                if (res < min_res)
                    min_res = res;
```

```
    }

    return min_res;

}

// ------------------------------------------------

void getRandomAddress(pointer *virt, pointer *phys, int erase) {

  /*Offset aleatorio en rango de memoria V. reservada, múltiplo de 128B (1024b)*/

    //size_t offset = (size_t)(rand() % (mapping_size / 128)) * 128; //Old approach

    auto it = mem_offsets.begin();

    std::advance(it,  (size_t)(rand() % mem_offsets.size()));

    size_t offset = (size_t)(*it) * MEMORY_DIVIDER;

    if (erase) mem_offsets.erase(it);



    *virt = (pointer) mapping + offset;//Dereference of value in virt

    *phys = getPhysicalAddr(*virt);//Dereference of value in phys

}



// ------------------------------------------------

void clearLine() {

    printf("\033[2K\r"); //"\033[2K" elimina la linea actual. \r lleva el cursor al incio
```

```
}



// ------------------------------------------------

char *formatTime(long ms) {

 /*Expresa ms en minutos y segundos, devuelve el array de chars*/

   static char buffer[64];

   long minutes = ms / 60000;

   if (minutes == 0) {

      sprintf(buffer, "%.1fs", ms / 1000.0);

   } else {

      sprintf(buffer, "%lum %.1lfs", minutes, //(lf = double)

            (ms - minutes * 60000) / 1000.0);

   }

   return buffer;

}




// ------------------------------------------------
```

```
int findSeparation ( size_t hist[MAX_HIST_SIZE] , int min, int max){

 int empty = 0, found = 0, void_valid = 0;

 int peak_diff_bank_pos = 0, peak_diff_bank_value = 0;

 int main_percent_min = 0, main_percent_max = 0;

 int diff_bank_centre = 0;

 int ok_zone_start = 0;

 int peak_same_bank = 0, peak_same_bank_pos = 0;

 int main_void = 0, acceptable = 0;

 //Look for biggest peak (will belong to diff_bank)

 for (int i = min; i <= max; i++){

    if (hist[i] > peak_diff_bank_value) {

      peak_diff_bank_value = hist[i];

      peak_diff_bank_pos = i;

    }

 }

 //Find limits where main body of addresses are (min 1/3 of peak value)

 for (int i = peak_diff_bank_pos; i > min; i--){

    if (hist[i] > (peak_diff_bank_value / 3))

      main_percent_min = i;
```

```
   if (hist[i] <= 1)
     break;
}

//In case peak is in (min), avoids some weird behaviour
if (peak_diff_bank_pos == min){
  main_percent_min = peak_diff_bank_pos;

}

for (int i = peak_diff_bank_pos; i < max; i++){
   if (hist[i] > (peak_diff_bank_value / 3))
     main_percent_max = i;
   if (hist[i] <= 1)
     break;
}
//Calculate center of main body of addresses
diff_bank_centre = (main_percent_max + main_percent_min) / 2;
```

```
/*          //Calculate where is ok to start considering same bank addresses
          ok_zone_start = (diff_bank_centre - min) + diff_bank_centre;


          for (int i = ok_zone_start; i > min; i--){
           if (hist[i] = 0){
             ok_zone_start--;
           }else{
             break;
           }
          }
 */


          //Find start of addr of same bank
          for (int i = diff_bank_centre; i < max; i++){
              if (hist[i] <= 1)
                 empty++;
              else
                 empty = 0;
```

```
    if (empty > 40) {
      acceptable = 1;
    }
   /* Finds biggest void in all histogram */
 //  if (empty > main_void) {
 //    main_void = empty;
 //    found = i;
 //  }
    /* Finds first void of minimum size 30 */
    if ((hist[i] > 1) && acceptable){
      found = i;
      break;
    }

}
//Establishing a minimum distance from same bank measurements
//if (main_void < 15) found = 0;

//In case the program 'scrambles' latencies, the program can be
```

```
//a little bit less restrictive in order to success.



if(!found && (separation_delay!=2000)){

 for (int i = max; i > ok_zone_start; i--){

  if(hist[i]=0) found = i;

 }

}




logDebug("Found peak of %i, main body between %i and %i, center in %i\n",
peak_diff_bank_value, main_percent_min, main_percent_max, diff_bank_centre);

logDebug("Addresses between %i and %i are considered non-valid\n", min, diff_bank_centre);

if (found) logDebug("Found a valid area between %i and %i\n", found, max);

return found;

}



// ---------------------------------------------

// ---------------------------------------------

// ---------------------------------------------
```

```
// -----------------------------------------------

// -----------------------------------------------

// -----------------------------------------------

// -----------------------------------------------

// -----------------------------------------------

// -----------------------------------------------

// -----------------------------------------------

// -----------------------------------------------


int main(int argc, char *argv[]) {

   size_t tries, t;

   std::set <addrpair> addr_pool;

   std::map<int, std::list<addrpair> > timing;

   size_t hist[MAX_HIST_SIZE]; //Defined as 1500

   int c;


   while ((c = getopt(argc, argv, "p:n:s:v:h")) != -1) {//Cambiado EOF por -1

      switch (c) {

         case 'p':
```

```
                fraction_of_physical_memory = atof(optarg);//Default: 0.6
        //      if ( fraction_of_physical_memory >= 0.9 ){
        //       printf("\n[*] Too much memory usage, kernel won't allow. Try 0.75 or
below.\n\n");
        //       return(0);
        //      }
                break;
            case 'n':
                num_reads = atol(optarg);//Default: 400
                break;
            case 's':
                expected_sets = atoi(optarg);//Default: 8
                break;
            case 'v':
                verbosity = atoi(optarg);//Default: 0
                break;
            case 'h':
                printf("\n[*] Usage: \n\n\n  sudo %s [-p <memory percentage>] [-n <number of
reads>] [-s <expected sets>] [-v <verbosity>]\n",
                    argv[0]);
```

```
            printf("\n  ·Defaults:   -p = 0.6   |   -n = 400   |   -s = 8   |   -v = 0");

            printf("\n  ·Program requires to run with superuser privileges (for /pagemap access)");

            printf("\n  ·Verbosity: 0 (results only) - 1 (log. messages - 2 (displays histogram)\n\n");

            exit(0);

            break;

        case ':'://Como optstr no empieza por ':', nunca ocurrirá

            printf("Missing option.\n");

            exit(1);

            break;

        default:

            printf("\n[*] Error - Unspecified parameter. '-h' for help.\n\n");

            exit(0);

            break;

    }

}



if(geteuid() != 0){

  printf("\n[*] Error - %s needs to be run as root!\n\n", argv[0]);

  exit(0);
```

```
        }


        tries = expected_sets * 125;


        logInfo("CPU: %s\n", getCPUModel());

        uint64_t available_ram =
(static_cast<uint64_t>(static_cast<double>(getPhysicalMemorySize())))/(1000*1000*1000);

        printf("RAM size: %lu\n",
(static_cast<uint64_t>(static_cast<double>(getPhysicalMemorySize()))));

        logInfo("RAM size: %lu GB\n", available_ram);

        logInfo("Memory percentage: %.0f %%\n", (fraction_of_physical_memory*100));

        logInfo("Number of reads: %lu\n", num_reads);

        //logInfo("Expected sets: %lu\n", expected_sets);


        srand(time(NULL));//Randomiza rand() en cada ejecución

        initPagemap();

        setupMapping();


        logInfo("Mapping has %zu MB\n\n", mapping_size / 1024 / 1024);//B > MB
```

```
pointer first, second;

pointer first_phys, second_phys;

pointer base, base_phys;



size_t remaining_tries;



//Generamos vector con offset para los espacios de memoria reservada

for (int i = 0; i < (mapping_size/MEMORY_DIVIDER); i++) mem_offsets.push_back(i);



//          printf("\n Mem offsets tiene un tamaño de %li", mem_offsets.size());



getRandomAddress(&base, &base_phys, 0);//Why(?)



long times[ETA_BUFFER], time_start; //long times[5], time_start;

int time_ptr = 0;

int time_valid = 0;



int found_sets = 0;
```

```
// choose a random base address

getRandomAddress(&base, &base_phys, 0);//En &base_phys se almacena la dir.fís de base



/*PRIO_PROCESS indica que el segundo argumento es un PID al que establecer prioridad.

0 es el PID 'Self'

El 3er arg. es la prioridad, en un rango [-20 - 19]. Lower gets better scheduling*/

setpriority(PRIO_PROCESS, 0, -20);



int failed;

//while (1) {

//while (found_sets < 10) {

while (mem_offsets.size() > 1000){

   for (size_t i = 0; i < MAX_HIST_SIZE; ++i)

     hist[i] = 0;//Inicializa todo el histograma a 0

   failed = 0;

   search_set:

   failed++;

   if (failed > 10) {

      logWarning("%s\n", "Couldn't find set after 10 tries, giving up, sorry!");
```

```
    exit(0);
}



// build address pool
//logInfo("%s\n", "Filling set with random addresses"); //1.000 addresses each time
addr_pool.clear();
while (addr_pool.size() < tries) { //tries = expected_sets * 125
    getRandomAddress(&second, &second_phys, 1);
    addr_pool.insert(std::make_pair(second, second_phys));
}


logLog("Searching for set %d (try %d)\n", found_sets + 1, failed);
timing.clear();
remaining_tries = tries;


// measure access times------------------------------------------------------
sched_yield();
auto pool_front = addr_pool.begin();//iterator to move on set
```

```
while (--remaining_tries) {//expected_sets*125

    sched_yield();

    time_start = utime();//(ms)



    // get sequentialy all address from address pool (prevents any prefetch or something)

    first = pool_front->first;

    first_phys = pool_front->second;

    std::advance(pool_front, 1);



    // measure timing

    sched_yield();

    t = getTiming(base, first);

    sched_yield();

    timing[t].push_back(std::make_pair(first, first_phys));



    times[time_ptr] = utime() - time_start;

    time_ptr++;

    if (time_ptr == ETA_BUFFER) {

        time_ptr = 0;
```

```
    time_valid = 1;
}

sched_yield();
clearLine();
if (time_valid) {
    long mean = 0;
    for (int i = 0; i < ETA_BUFFER; i++) {
        mean += times[i];
    }
    mean /= ETA_BUFFER;
    printf("%lu%% (ETA: %s) %c",
        (tries - remaining_tries + 1) * 100 / tries,
        formatTime(mean * remaining_tries), "|/-\\"[time_ptr % 4]);;
} else {
    printf("%lu%% (ETA: %c)",
        (tries - remaining_tries + 1) * 100 / tries,
        "|/-\\"[time_ptr % 4]);
}
```

```
    fflush(stdout);

}



clearLine();

printf("\033[1A"); //clear progress line

// time measure end ----------------------------------------------------




printf("\n");



// identify sets -> must be on the right, separated in the histogram

std::vector < addrpair > new_set;

std::map < int, std::list < addrpair > > ::iterator hit;

int min = MAX_HIST_SIZE;

int max = 0;

int max_v = 0;

for (hit = timing.begin(); hit != timing.end(); hit++) {

    hist[hit->first] = hit->second.size();

    if (hit->first > max)
```

```
      max = hit->first;
    if (hit->first < min)
      min = hit->first;
    if (hit->second.size() > max_v)//Max amount of add. with same cycles
      max_v = hit->second.size();
}

// scale histogram
double scale_v = (double) (100.0)
       / (max_v > 0 ? (double) max_v : 100.0);//If > 0, =max_v, otherwise, =100
assert(scale_v >= 0);
while (hist[++min] <= 1);
while (hist[--max] <= 1);

// print histogram
if (verbosity == 2){
  for (size_t i = min; i <= max; i++) {
     printf("%03zu: %4zu ", i, hist[i]);
```

```
      assert(hist[i] >= 0);
      for (size_t j = 0; j < hist[i] * scale_v && j < 80; j++) {
         printf("#");
      }
      printf("\n");
   }
}


 // find separation
 int found = findSeparation(hist, min, max);

 //If failed finding a set of add. from same bank, returns all
 //in 'addr_pool' to 'mem_offsets' and retries.
 if (!found) {
    logLog("%s\n", "No set found, trying again...");

    uint64_t offset;
    for (auto it = addr_pool.begin(); it != addr_pool.end(); it++){
```

```
    offset = it->first - (pointer)mapping;
    mem_offsets.push_back((int)(offset/MEMORY_DIVIDER));
  }
  goto search_set;
}

logDebug("Found same set addr. between the cycles %i and %i\n", found, max);

// save physical of same bank addresses in new_set
int returned = 0; //Diff. bank add. returned counter
int useful = 0;   //Same bank add. added counter

for (hit = timing.begin(); hit != timing.end(); hit++) {
  if (hit->first >= found && hit->first <= max) { //Same bank add.

    for (std::list<addrpair>::iterator it = hit->second.begin();
      it != hit->second.end(); it++) {
        new_set.push_back(std::make_pair(it->first, it->second));
        //printf("Virt. 0x%012lX, Phys. 0x%012lX \n", it->first, it->second);
```

```
            useful++;

        }



    } else { //Address from another bank, must be returned to mem_offsets

      for (std::list<addrpair>::iterator it = hit->second.begin();

        it != hit->second.end(); it++) {

        uint64_t offset;

        offset = it->first - (pointer)mapping;

        mem_offsets.push_back((int)(offset/MEMORY_DIVIDER));

        returned++;

      }



    }

}

if(verbosity >= 1) printf("\n");

logLog("[*] Returned to mem_offsets: %i. In new_set: %i\n", returned, useful);



if (new_set.size() <= 10) {
```

```
                    logLog("Set must be wrong, contains too few addresses (%lu). Try again...\n",
new_set.size());
                    goto search_set;
                }
                if (new_set.size() > addr_pool.size() / expected_sets) {
                    logLog("%s\n", "Set must be wrong, contains too many addresses. Try again...");
                    goto search_set;
                }

                // Updating separation delay point
                if (separation_delay > found){
            //          printf("%i > %i, el 10 tiene el valor del 20", separation_delay, found);
                  separation_delay = found;
                }
                // check if current set is part of a bank already identified
                // VIRTUAL addresses needed! virtual (first) y física (second).
                // "expected_sets" afecta a la cantidad de elementos en el mapa 'timing'

                if (found_sets != 0) {
```

```
//printf("[*] Separation delay point: %i (if greater, same bank)\n", separation_delay);

uint64_t new_set_addr, saved_set_addr;



 // Recorremos todos los sets guardados. Comparamos 10 direcciones

 // aleatorias de cada uno con 10 dire. aleatorias del nuevo. Si su

 // tiempo de lectuar es muy elevado (> separation_delay), son del

 // mismo bank.



 set_comparative = 1;

 int time_average = 0;

 int same_set = 0;

 if(verbosity >= 1) printf("\n");

 logLog("%s\n","[*] Comparing with existing sets...");

 for (auto it = sets.begin(); it != sets.end(); it ++) {



  //printf("\n\n[*] Comparing with set %i...\n", set_comparative);

  time_average = 0;

        for (int i = 0; i < 10; i++){
```

```
                auto ait = new_set.begin();

                std::advance(ait, rand() % new_set.size());



                auto nit = it->begin();

                std::advance(nit, rand() % it->size());



                new_set_addr = ait->first; //Random V.add from new_set

                saved_set_addr = nit->first;//Random V.add from already stored set



                sched_yield();

                t = getTiming(new_set_addr, saved_set_addr);

                sched_yield();



                time_average += t;

               }

        time_average = (int)time_average/10;

//      printf("\nTime average: %i, Separation delay: %i", time_average, separation_delay);

        if (time_average > (separation_delay-50)){

                same_set = 1;
```

```
            it->insert(it->end(), new_set.begin(), new_set.end());

            break;

   }

   set_comparative ++;

 }



if (same_set){

 logLog("****** SAME SET FOUND ****** (%i)    \n\n", set_comparative);

} else {

 sets.push_back(new_set);

 logLog ("%s\n\n","[*] Unique new set. Added to the rest.");

 found_sets++;

}



//Clear previous lines with data to update

if (verbosity == 0){

 for (int i = lines_to_erase + 4; i != 0; i --){

   printf("\033[1A");

   //printf("hehe");
```

```
 }
}

printf("Total sets: %i\n", (int) sets.size());
printf("\n ---------- \n\n");

set_comparative = 1;

lines_to_erase = 0;
for (auto it = sets.begin(); it != sets.end(); it ++) {
 if (set_comparative < 10){
   printf("Addresses in set %i:  %lu\n", set_comparative, it->size());
 }else{
   printf("Addresses in set %i: %lu\n", set_comparative, it->size());
 }
 set_comparative++;
 lines_to_erase++;
}
```

```
    } else {
      // save identified without inspection (only first)
      printf("Total sets: 1\n");
      printf("\n ---------- \n\n");
      printf("Addresses in set 1:  %lu\n", new_set.size());
      sets.push_back(new_set);
      found_sets++;
    }

  // choose base address from remaining addresses
  auto ait = addr_pool.begin();
  std::advance(ait, rand() % addr_pool.size());
  base = ait->first;
  base_phys = ait->second;

}

//Pasamos dir. fisicas de sets (second) a phys_sets
```

```
int q = 0;

for (auto it = sets.begin() ; it != sets.end(); it++) {

 phys_sets.push_back( std::vector<uint64_t>() ); //Avoids segfault

 for (auto nit = it->begin(); nit != it->end(); nit++) {

  phys_sets[q].push_back(nit->second);

 }

 q++;

}

logDebug("%s\n", "done measuring");




logDebug("%s\n", "Calculating same set-addr. delays...");



//Aplicar puntero al set si el escaneo se detiene tras un tamaño marcado

/*

auto set_pointer = sets.begin();

for (;set_comparative != 1; set_comparative--){

 set_pointer++;

}
```

```
*/

//Aplicar puntero al mayor set. (Escaneo detenido al llegar a un num. de sets)

int max_set_size = 0, current_set = 1, max_set;

auto set_pointer = sets.begin();

for (auto wit = sets.begin(); wit != sets.end(); wit++){

 if ( wit->size() > max_set_size){

   max_set_size = wit->size();

   set_pointer = wit;

   max_set = current_set;

 }

 current_set++;



}



printf("\nSelected biggest set (%i), size: %lu\n", max_set, set_pointer->size());



int tiempo_average = 0;

std::list <addrpair> same_addr_timings;

auto witi = same_addr_timings.begin();
```

```
FILE *fpt;

fpt = fopen("Resultados_Same_Add.csv", "w+");




//for (int x = 0; x != 10; x++){

   for(auto zit = set_pointer->begin(); zit != set_pointer->end(); zit++){



     sched_yield();

     t = getTiming(zit->first, zit->first);

     sched_yield();



     //same_addr_timings.insert(witi, std::make_pair(zit->first, (pointer) t));

     //witi++;

//    }

     //printf("Dirección 0x%012lX por debajo de t con %lu ciclos\n", y->first, y->second);

//}

//for(auto y = same_addr_timings.begin(); y != same_addr_timings.end(); y++){

     //Escribir en excel
```

```
            fprintf(fpt,"0x%012lX,%lu\n", zit->first, t);



        }

        fprintf(fpt,"\n\n\n\n\n");

//.....................................................................................

        //for (int x = 0; x != 10; x++){

          for(auto zit = set_pointer->begin(); zit != set_pointer->end(); zit++){



            pointer same_page_addr = (zit->first)+40;



            sched_yield();

            t = getTiming(zit->first, same_page_addr);

            sched_yield();



            //same_addr_timings.insert(witi, std::make_pair(zit->first, (pointer) t));

            //witi++;

        //    }

            //printf("Dirección 0x%012lX por debajo de t con %lu ciclos\n", y->first, y->second);

        //}
```

```
            //for(auto y = same_addr_timings.begin(); y != same_addr_timings.end(); y++){

                //Escribir en excel

                fprintf(fpt,"0x%012lX,%lu\n", same_page_addr, t);

            }




/*

            auto wit = set_pointer->begin();

            std::advance(wit, rand() % set_pointer->size());



            printf("\nVirt. 0x%012lX, Phys. 0x%012lX \n", zit->first, zit->second);

            printf("Transmitimos las direcciones 0x%012lX y 0x%012lX\n", zit->first, wit->first);



            sched_yield();

            t = getTiming(zit->first, wit->first);

            sched_yield();

*/

    /*

            printf("Ciclos entre zit y wit: %li\n", t);
```

```
        pointer same_page_addr = (zit->first);



        sched_yield();

        t = getTiming(wit->first, same_page_addr);

        sched_yield();



        printf("Ciclos entre wit y otra dirección de la misma página de zit: %li\n", t);



        sched_yield();

        t = getTiming(zit->first, zit->first);

        sched_yield();



        printf("Ciclos entre zit y otra dirección de su misma página: %li\n", t);

*/

        //fclose(fpt);

        return 0;
```

```
}
```