RowHammer; A Review of the Exploit Used to Access Protected, Inaccessible Memory

An MQP

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Undergraduate Degree of Engineering

In

Electrical and Computer Engineering

Andrew Adiletta

September 2021

APPROVED:

Professor Berk Sunar	Professor Donald R. Brown
ECE Department	ECE Department
MQP Adviser	Department Head-Engineering

This report represents work of WPI undergraduate students submitted to the faculty as evidence of a degree requirement. WPI routinely publishes these reports on its web site without editorial or peer review. For more information about the projects program at WPI, see http://www.wpi.edu/Academics/Projects

Abstract

This MQP explores an implementation of the Rowhammer attack [1] on a Unix server using DDR3 memory to access normally in-accessible memory through a vulnerability in the physical hardware that exploits the tendency of cells in DRAM to experience voltage leakage which leads to data errors. The setup for the attack, and organizing virtual memory to align to physical memory, and attacking cells housed in the same bank is explored using side channels. These side channel attacks focus on finding memory that appears consecutively in both physical and virtual memory, and aligning a victim row between two attacking rows, and reading from the attacking row at a high rate of speed in order to flip bits in the victim row.

Contents

Abstract ... 2

Figures ... 4

Background: Inspiration and introduction to The Rowhammer Attack ... 5

- The Memory Hierarchy ... 5
- DRAM cells and architecture ... 6
- DRAM Cells and Sense Amplifiers ... 6
- Taking Advantage of Leakage for Rowhammer ... 7
- Memory mapping and virtual to physical hardware translation. ... 8
- Finding Memory Continuity in Memory Banks ... 9

Our Approach: Using A C level program to translate virtual addresses to physical addresses.... 10

- Finding Continuity of Physical Addresses ... 11
- Designing an Iterative Function to Find Physical Memory ... 12

Determining Continuity in Banks ... 13

The Rowhammer Attack in Assembly ... 15

Timing Analysis and Optimizations ... 18

Conclusion ... 19

Bibliography ... 20

Figures

Figure 1: DRAM array architecture illustrated with bit Line, word line, and DRAM cell illustrated. Simplified to show a total of 21 DRAM cells 6

Figure 2: Graphically showing memory mapping, where a continuous physical memory is a portion of virtual memory, and bank continuity is a portion of physical memory 9

Figure 4: Finding physical memory continuity through virtual memory buffer iterations, while keeping a counter of immediate previous physical addresses that are continuous 12

Figure 5: DRAM stick with the DRAM memory bank indicated; DRAM with two ranks has one rank on each size 13

Figure 6: Percentage of successfully recorded data points based on a threshold value; data indicating an interrupt process occurring regularly disrupting the capturing of data 14

Figure 7: Determining continuous bank memory based on peaks in memory read time; due to an optimization in Intel hardware, the rowbuffer for a particular bank needs to be cleared which causes a spike in timing [22] 15

Figure 8: Experimentation with read cycles in attacking rows and the resulting number of flips; the optimal number of flips occurs between 900000 and 1100000 read cycles to get the most flips 18

Background: Inspiration and introduction to the Rowhammer Attack

The first concept for a Turing complete computer is generally attribute to an English mathematician Charles Babbage in 1837. During the industrial revolution, steam had driven innovation to the point where Babbage believed a machine could be developed that could solve complex mathematical and conceptual problems. The machine was as mechanical computer, which used punch cards to hold data, as well as physical wheels and axels to hold state, for computation. A symbolic instruction set was used to perform computation, which led to the first programs being written. The concept of data being held in a part of the machine called a *store* and being manipulated and computed on in the part of the machine known as the *mill* would eventually lead to the idea of a memory hierarchy [2].

Part of the reason this machine was never built was that it would require thousands of moving parts to function. At a time when precise manufacturing was in its infancy, this made the project infeasible. However, had a machine been built, a flaw that Babbage may not have even considered for a machine like the analytic engine was security. One might be able to reach their hand into the whirling machine (hopefully without losing a limb), remove a punch card, add an additional punch, and dangerously change information.

This is analogous to the attack described in this MQP report. The Rowhammer attack uses flaws in the design of DRAM to access protected areas in memory. It allows a user to effectively reach in and edit the “punch cards” in memory, not physically, but by using parasitic capacitance induced by fast consecutive cell reads in adjacent cells to the row of protected memory being attacked. The effect is that cells in protected areas in memory that a program shouldn’t have access to may be flipped from high to low, or low to high, depending on what the attacker wants.

The Memory Hierarchy

The memory hierarchy is a concept in computer architecture that organizes data storage to optimize latency. There is the fastest layer of memory that sits closest to the CPU known as the cache. Due to its expensive design, it is the smallest memory layer, but also the fastest, being anywhere from 10 to 100 times faster than DRAM which is the next level up [3]. Data that sits in cache has low latency access times, so operations that require data from cache are faster than operations that require data from different layers of storage [4]. The next level of memory in the memory hierarchy is the primary storage that most often takes the form of DRAM, which stands for dynamic random-access memory. DRAM storage is larger than the cache layer, but data required from DRAM for computation in the CPU requires a larger latency. Above the DRAM layer is disk storage. Unlike DRAM and cache, disk storage is nonvolatile, and can remain even after a system restart. Disk storage is the slowest storage, but often the largest and cheapest [5]. The Rowhammer attack targets the DRAM layer of memory and allows write access to sensitive information. Thus, the background will go in depth into the architecture of DRAM.

DRAM cells and architecture

DRAM stores bits in an array of memory cells. For each bit stored, there is a capacitor and a transistor that form the cell circuit. In one dimension on the grid, are word lines in rows, where each word line connects to every cell in the row. The cell itself has two-bit lines, that connect perpendicularly to the word lines to cells above and below itself, which are described as "+" or "-" bit lines [6].

DRAM Cells and Sense Amplifiers

When one bit-line is brought high and the other bit line is brought low, a sense amplifier causes positive feedback which allows the cell to remain at a high or low voltage. This means that for the cell to be written to, the sense amplifiers must be disconnected.

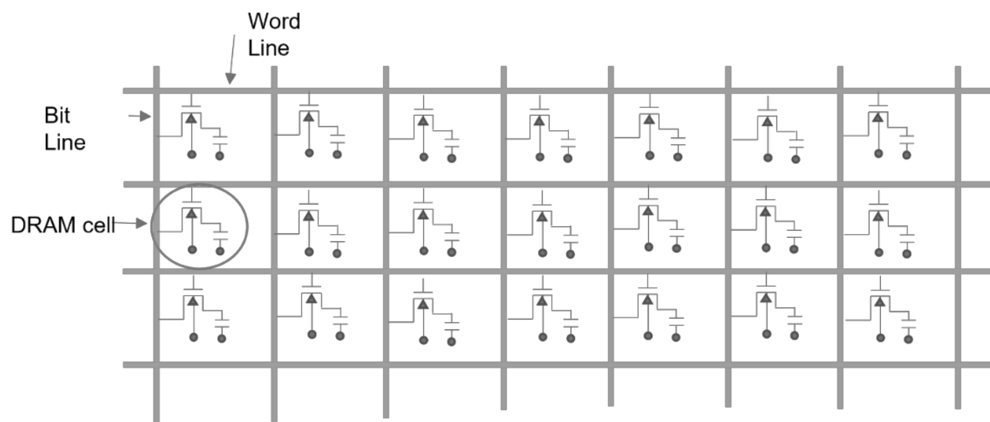

Figure 1: DRAM array architecture illustrated with bit Line, word line, and DRAM cell illustrated. Simplified to show a total of 21 DRAM cells

The Rowhammer exploit works by abusing the function of the sense amplifier, so it is important to understand how they operate. A sense amplifier is made of two cross-connected inverters between the memory cells' bit lines [7]. An inverter is also often called a not gate which is a logic gate that negates its input; a one input to an inverter would be a zero and a zero input to an inverter would be a one. Despite the stabilizing properties of the sense amplifier, the scaling of the DRAM cells has forced smaller noise margins which results in electronic noise causing errors in memory cells [8]. The scaling is a resulted in cells containing less charge, meaning the noise on the cell is a larger percentage of the cells total charge.

Reading a cell requires the sense amplifiers to be disconnected, and the target rows word-line to be brought high. Once brought high, if the target cell has a value of 1, then the charge from the capacitor in the cell charges the bit line, and after the sense amplifiers are reconnected, the entire row has their sense amplifiers outputs latched. Current is allowed to flow back up the bit lines to

recharge the storage cell. After all the reading is done from the target row, the word line is switched off [9].

Taking Advantage of Leakage for Rowhammer

Capacitors leak voltage over time, so the DRAM is required to undergo a “refresh” every 64ms or less. This is defined by the Joint Electron Tube Engineering Council (JETEC), which sets hardware standards for manufacturers [10].

The act of reading a cell can cause voltage fluctuations to adjacent cells. This has been observed in DDR3 memory, where consecutive reads can cause voltage fluctuations which can cause errors to get written to memory cells during a refresh [11]. DDR stands for double data rate, and in simple terms, it is an architecture of memory that can undergo one transfer per clock cycle. Using both the rising and falling edge of the clock signal, the data can be transferred at double the rate. DDR-200 and DDR-400 have an IO bus speed of 100Mhz and 200Mhz respectively. DDR2-800 and DDR3-1600 have an IO Bus speed of 400Mhz and 800Mhz respectively [12]. The faster the memory devices become; the more consecutive reads can occur between refresh periods to the memory. Thus, DDR3’s high speed and high density allows it to be a perfect candidate for a Rowhammer attack. Modern DRAM sticks have protections against Rowhammer attacks, the two most common protections being doubling the refresh rate of DRAM cells and preventing an attacker from executing the CLFLUSH command in assembly [13]. The CLFLUSH clears the cache and preventing an attacker from clearing the cache would force reads from the cache rather than the DRAM, which would prevent the attack. On legacy systems however, there is no current protection from Rowhammer attacks, unless special software deployed from bootloader is installed that prevents user space and kernel space memory from sharing physical locality on the DRAM stick [14].

Rowhammer Implementation and Performance Analysis

Overview of the Work

Understanding the firmware relating to memory mapping was necessary for the study of the Rowhammer attack. There is a virtual address space that programs are allocated, and a portion of that virtual address space may be continuous in physical memory. This physical continuity can be found using the page map file, which translates virtual addresses to physical addresses. After using the page map file to find physical continuity, the next step is to find continuity of addresses within the same bank. This allows rows to be targeted that are physically adjacent to each other.

Understanding the firmware relating to memory mapping was necessary for the study of the Rowhammer attack. There is a virtual address space that programs are allocated, and a portion

of that virtual address space may be continuous in physical memory. This physical continuity can be found using the page map file, which translates virtual addresses to physical addresses. After using the page map file to find physical continuity, the next step is to find continuity of addresses within the same bank. This allows rows to be targeted that are physically adjacent to each other.

The last section of the report describes the specific assembly instructions required to execute the attack. This involves clearing the cache and reading from adjacent rows as quickly and as many times as possible between row refreshes. A counter keeps track of the number of reads to stop the iteration and check for flipped bits. Analysis of the function determines an optimal number of read cycles that results in the greatest number of flipped bits.

Memory mapping and virtual to physical hardware translation.

Another important subject to understand for the Rowhammer attack is memory mapping. Bits are stored in DRAM as a grid, and programs can allocate a certain amount of the grid for data required for the program. This allocation takes the form of virtual memory, where a virtual address space is given to the program, and the program can request to store data in any address in that virtual address space [15]. Bits in a chunk of the virtual memory known as a page are all mapped to the same physical location in memory. Usually, this page size is around 4 kilobytes in size, thus requiring a program to have many pages spreading data out across the entire physical DRAM.

The idea of a virtual address space is to allow a program to operate in its own "sandbox" for security reasons. It has access only to the memory allocated to it in the virtual memory space, which maps to different places all around the physical hardware. If a program requests access to data outside its virtual address space, it will receive a segmentation fault error.

For an address in virtual memory to be mapped to physical memory, page tables are used to lookup the address. The translation operations often occur in the memory management unit (MMU). If a page is requested that does not exist in physical memory, the memory management unit will raise a page fault exception. Another component of the virtual memory mapping is the paging supervisor, that organizes the page tables [15].

Memory can be manually requested and taken of the portion of unallocated memory known as the heap. Memory can be freed and returned to the heap for use by other programs. Most modern programming languages also implement a call stack, where memory is managed automatically, and usually contains local variable to a subroutine in the program. A memory-mapped file is a portion of the virtual memory that has been written consecutively for the file. This is an important aspect of computer architecture to understand because it allows

Rowhammer to function. A memory-mapped file is created on a Unix machine with the **`mmap`** command, and after the side channel determines which cells are vulnerable to be flipped, will use the **`mumap`** command to free up the memory and allow the sensitive program to take it over [16].

Finding Memory Continuity in Memory Banks

The Rowhammer requires finding rows that are adjacent to each other physically. This requires two checks; the first is determining if virtual memory is continuous on the physical chip [17]. The second check determines if the memory is also continuous in the memory bank. The rows in DRAM continue across multiple chip packages, so generally only a portion of the continuous physical memory will be continuous in a particular bank. Figure 1 shows the steps required for memory mapping.

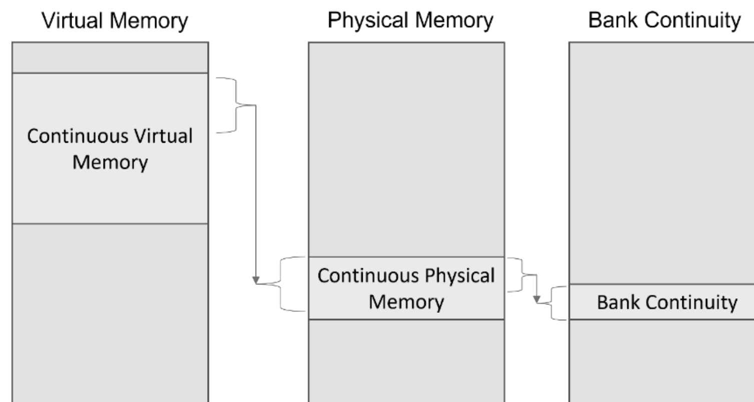

Figure 2: Graphically showing memory mapping, where a continuous physical memory is a portion of virtual memory, and bank continuity is a portion of physical memory

Using A C level program to translate virtual addresses to physical addresses

The first task of the C program is to determine a start and an end virtual address that maps to consecutive physical memory. While somewhat arbitrary, it's important that this address space is large enough to find as many flippable cells as possible. In Unix, the page map file is used to map the virtual memory to the physical memory [18]. On the Unix test environment that is being used, we find the **`pagemap`** file at

`/proc/self/pagemap`

A function to get the physical address consumes a virtual address. In C, there is special syntax of using the '&' character to get the virtual address of a variable in memory. This can get cast to a 64-bit integer.

Given a virtual address as an offset, the **`pagemap`** file the next 64-bits can be read into a 64-bit integer to obtain the physical address.

The essential lines for obtaining the value are the following [19].

```
1.  off_t offset = (virtual_addr / 4096) * sizeof(value);
2.  int got = pread(g_pagemap_fd, &value, sizeof(value), offset);
```

One thing to note about the code it that the offset is not simply the virtual_address, but the virtual address divided by 4096 [20]. 4096 is the page size, so by dividing the virtual address by the page size, the value returned is an index value of the location of the physical address. Multiplying the index value by the size of the physical address will result in an exact location in the page map file to readout the physical address from.

However, the address stored in the **value** variable isn't truly a physical address. The final two operations correct the physical address.

```
1.    // return physical address
2.    uint64_t frame_num = value & ((1ULL << 55) - 1);
3.    return (frame_num * 4096) | (virtual_addr & (4095));
```

The first line and the value bits with 1 (which is defined as an unsigned long long), that has been bit shifted 55 bits to the left, and all the bits inversed by subtracting 1. The result is retaining all the bits of the physical address except for bit 55.

Finding Continuity of Physical Addresses

First, a large portion of the virtual address space for the program is allocated using the **mmap** command. Typically, a programmer would use malloc to allocate space, however, **malloc** will not reallocate memory to the heap, which can then be used by other programs for which Rowhammer is attacking, therefore, **mmap** must be used to create a memory mapped file [21].

The **mmap** command takes several arguments to use. The first argument is the address to start the buffer. For our example, it makes sense to start the buffer at zero, so null can be passed as the address. Theoretically, the address could start anywhere in the virtual memory space but starting at zero will allow for the most amount of space to fill the buffer.

The next argument to **mmap** is the length to allocate to the buffer. The preferred length of the buffer is 256 MBs, which is enough to find at least 8 MBs of continuous memory. A page is roughly 4 Kbs in size, so that would require 65,536 pages.

After the length argument, the protection flags must be passed to the memory mapped file. For Rowhammer, the file must be read and writable, however, it is unnecessary to allow the file to be executable, so the argument **PROT_READ | PROT_WRITE** is passed.

Other flags that get passed in the next argument control the behavior of the mapped file. There are four main options for the memory mapped file behavior; **MAP_SHARED**, which allows the memory mapped file to be shared between processes, **MAP_PRIVATE**, which forces the file to be seen only by the current running process, **MAP_ANON** which doesn't actually map the memory to a file, but instead extends the heap available to the program, and finally **MAP_FIXED** which requires the exact address to be used from the first argument or an error will be thrown, and finally **MAP_POPULATE** is a flag that forces the page table to be prepopulated with the virtual to physical addresses. For Rowhammer, **MAP_POPULATE, MAP_PRIVATE**, and **MAP_ANON** is used.

The final two arguments to the **mmap** command is the file descriptor and the offset, which will be -1, and 0 respectively [16].

```
1. uint8_t * continuous_buffer = mmap(NULL, PAGE_COUNT * PAGE_SIZE, PROT_READ | PROT_WRITE,
   MAP_POPULATE | MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
```

The next step is to go through each virtual address in the continuous buffer, convert it to a physical address, and determine if it is continuous with the previous virtual address. The logic is a function that returns a start and an end address within the continuous buffer that has at least 8MBs of continuous memory.

Designing an Iterative Function to Find Physical Memory

First, the function will iterate through each page of the buffer, determining the virtual address of the page. Iterating on a page interval is necessary because the page map file only maps entire pages to different place in memory. Portions of pages cannot be split across numerous places in physical memory [22]. The function then compares this value to the physical address of the previous page in the buffer, by holding the previous physical address as a temporary file in memory. The comparison is straightforward because the addresses are stored as unsigned 64-bit integers. If these addresses are consecutive, it means the memory is continuous physically. Otherwise, the physical continuity of the buffer is broken. Below is a diagram of this function.

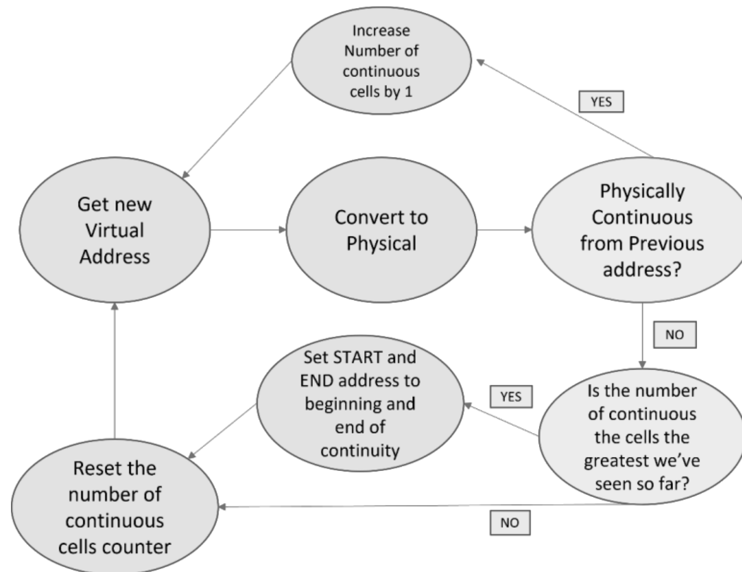

Figure 3: Finding physical memory continuity through virtual memory buffer iterations, while keeping a counter of immediate previous physical addresses that are continuous

A counter increments every time a physical address is received that is consecutive with the previous physical address. Additionally, a variable containing the current maximum number of consecutive cells found is updated every time physical continuity ends. The following is the equation used for determining the continuity of physical addresses.

$$address[n] + 1 == address[n - 1]$$

When **`address[n]+1 != address[n-1]`**, the physical continuity has ended. If the current number of physically continuous cells is greater than the current maximum, the maximum

variable is updated, and the START and END address are also updated. This requires that the START and END address of the current set of continuous memory cells are stored in temporary variables. After the START and END variables are updated, the temporary variables are reset for the next set of continuous memory cells. The number of continuous cells is reset to zero, and the process repeats with a new virtual address from the memory buffer.

At the end of the process, the function will have determined a start and an end position, of a continuous amount of physical memory, contained within the buffer created with **`mmap`**.

Determining Continuity in Banks

After determining physical continuity, the next step is to determine continuity with the memory banks. Below is a picture with the memory bank on a DRAM stick indicated.

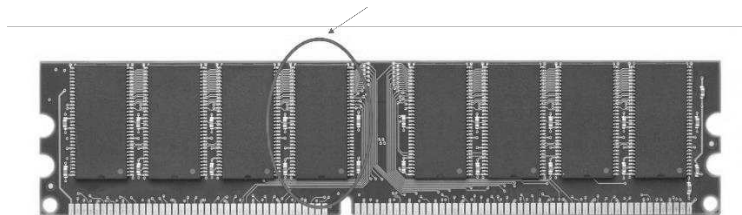

Figure 4: DRAM stick with the DRAM memory bank indicated; DRAM with two ranks has one rank on each size

The Rowhammer attack requires that memory cells in the victim row, and aggressor rows all be in the same bank. This process is like the process of finding physical continuity, however, the bank continuity can't be found within the pagemap file. Instead, a timing side channel needs to be used to estimate which cells are in a particular bank.

The function consumes the memory buffer, and the start and end address of physical continuous memory within the memory buffer, and will return an array with addresses that are all physically continuous and continuous within the memory banks

The process begins by iterating through the memory buffer, starting at the START index in continuous memory. A variable for the difference in retrieval time between the START index, and the current index in the memory buffer is determined using an assembly function called **`clfmeasure`** [23].

The **`clfmeasure`** command isn't precise enough for accurately determining bank location, so the retrieval time difference is taken 10000, and the average time is taken. Additionally, when determining the measurement time, a secondary test occurs that checks if the time difference is large enough to be an outlier in the dataset. This happens if there is a system interrupt or other anomaly that causes the error to be too imprecise. If the time difference is determined to be

within an acceptable range, it is added to a running total of time differences, and a second variable containing the quantity of within-threshold-time-retrievals is incremented. After 10000 measurements, the total is divided by the number of measurements recorded to determine the average.

Below are the percentages of successfully taken measurements.

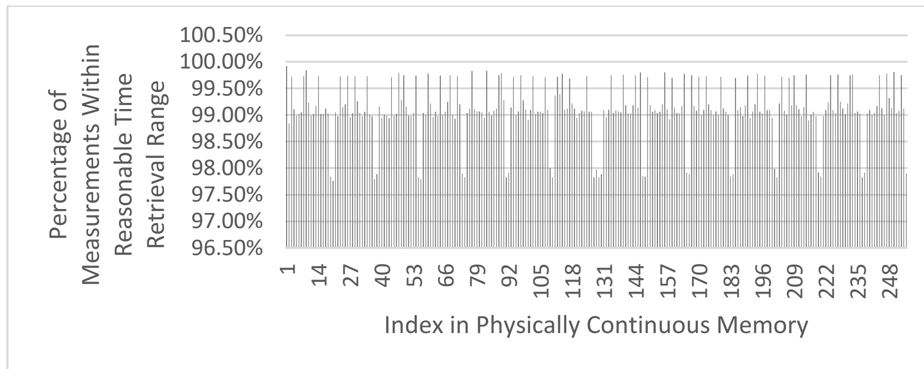

Figure 5: Percentage of successfully recorded data points based on a threshold value; data indicating an interrupt process occurring regularly disrupting the capturing of data

Based on the data presented in figure 5, most of the memory addresses in figure 5 have above a 97.5% recording accuracy. The roughly 2.5% of data points excluded are considered outliers because the retrieval time was greater than 700ns, which indicates that an external activity interrupted the retrieval process.

Roughly every 20 experiments of 10000 times retrievals, the accuracy dips. This indicates that there may be an interrupt running at that time interval which is disturbing the experiment. More precise results may be gained by finding this interrupt and disabling it, but for the purposes of Rowhammer, 97.5% is sufficient.

After removing the outliers, the next step is to determine which data points come from the same bank by reviewing the average retrieval times and looking for peaks in timing, which indicates each address is from the same bank.

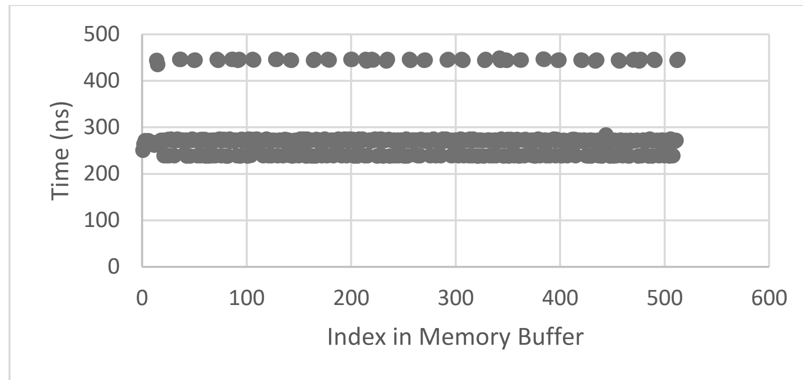

Figure 6: Determining continuous bank memory based on peaks in memory read time; due to an optimization in Intel hardware, the rowbuffer for a particular bank needs to be cleared which causes a spike in timing [23]

From figure 7, it is apparent which indices come from the same bank in memory. Essentially, due to optimizations in intel hardware the memory reads from indices in the same bank result in longer retrieval times from memory, which can be extrapolated from the graph to determine which addresses in physical memory are in the same memory bank.

This optimization is a row buffer, where pages are loaded into a cache for each bank. If a page is read from the same bank as the last page read, the row buffer must be cleared and rewritten. This is done with a pre-charge command, as the active row is closed, and the newly active row is returned with the row-active command. This is the cause of the peak in timing that allows the bank location information to be leaked. This is unique to Intel processors, as AMDs use a different row buffer and mapping function [23].

The Rowhammer Attack in Assembly

After the addresses in physical memory are determined to be in the same bank, the attack on the memory cells can commence. The buffer that contains the addresses in the same bank will be known as the conflict buffer. The Rowhammer attack will iterate through each row in the bank and check for flippy bits in the row. It determines the number of rows from the data contained in figure 5, where the total number of reasonable bank measurements were recorded.

It is also important to note that each row in a bank contains two pages of memory. Thus an aggressor row with an index of x would have a victim row of x + 2, along with a co-aggressor row of x + 4 [23].

Beginning the attack, the aggressor rows have all their bits set to zero. This is done by iterating through each index in the array for the full-page size and setting the value at that address to 0x00. In the victim row, the values are iterated through to the page size length, however, these bits are all set to one by setting the value to 0xFF, which is the hex value of 255 in decimal.

The reason the aggressor row and the victim row need to be opposites is that the aggressor rows will flip the 1s in the victim row to a zero. This can be verified afterward.

The following is the assembly code run on the two aggressor rows that forces the bits to flip.

```
1.  // Row_hammer for 1->0 flips
2.  #define hammer10(_memory, _memory2)\
3.  do{\
4.     asm volatile(\
5.     "mov $1000000, %%r11;"\
6.     "h10:"\
7.     "clflush (%%rdx);"\
8.     "clflush (%%rbx);"\
9.     "mov (%%rbx), %%r12;"\
10.    "mov (%%rdx), %%r13;"\
11.    "dec %%r11;"\
12.    "jnz h10;"\
13.    : \
14.    : "b" (_memory), "d" (_memory2)\
15.    : "r11", "r12", "r13"\
16.    );\
17. }while(0)
```

The code can be broken down line by line to understand how the attack works, starting with line 18 with the assembly **mov** command

```
5.   "mov $1000000, %%r11;"\
```

The **mov** command in assembly is used to copy data between locations [24]. The requirements for the locations are that they both be registers, or the copied value may be a constant. The **mov** command will fail if a memory address is passed instead of a register. If data need be moved from one memory address into another, it must first be copied into a register, and then out of that register. This requires two **mov** commands. In line 18, the "$" before the 1000000 indicates a constant value. The **%r11** is indicating a register that the constant value is being copied into. The r11 register is a temporary register, and the data is guaranteed to be saved between calls in. For the purposes of this code, that is fine because the r11 register is used as a counter for the number of reads of the aggressor rows run [24].

```
6.   "h10:"\
```

The next line is defining a location in the assembly code, with the prefix "h" indicating a hex value. Locations in code are useful for jump commands, where the assembly code needs to return to a specified place in the code to repeat operations.

```
7.  "clflush (%%rdx);"\
8.  "clflush (%%rbx);"\
```

The next two lines clear the cache of the CPU. Although this takes up cycles, thus reducing the effectiveness of the Rowhammer attack, it is necessary because otherwise the rows won't be read from memory, but instead read from the cache. The two addresses being cleared are **rdx** and **rbx**. These will later be defined in the code as our attacking addresses.

```
9.     "mov (%%rbx), %%r12;"\
10.    "mov (%%rdx), %%r13;"\
```

The lines are where the reading actually takes place. The **mov** command is used just like at line 5, but the instead of a constant being read into a register, the value at a specific memory address is read into register **r12** and **r13**, which are temporary registers just like **r11** which is being used as a counter [25].

```
11.     "dec %%r11;"\
12.    "jnz h10;"\
```

Line 11 in the assembly code runs a decrement command on the value store in register **r11**. Initially the value at **r11** was set to 1000000, so the first decrement would bring the value at **r11** to 999999. After that, the **jnz** command is a conditional jump, which allows the code to loop. The jump conditional is that if the value above doesn't return 0, the execution code will jump back to the line containing **h10**, located at line 6. The result is the code will loop 1000000 times, before the value at **r11** is zero, and the function will exit.

The final lines of code in the assembly assign the memory addresses into the variables used in the assembly code.

Timing Analysis and Optimizations

The first optimization that should be performed is determining that optimal number of reads that the attacker rows should undergo to maximize the number of flips. Below is a graph we generated that describes through experimentation what the optimal number should be.

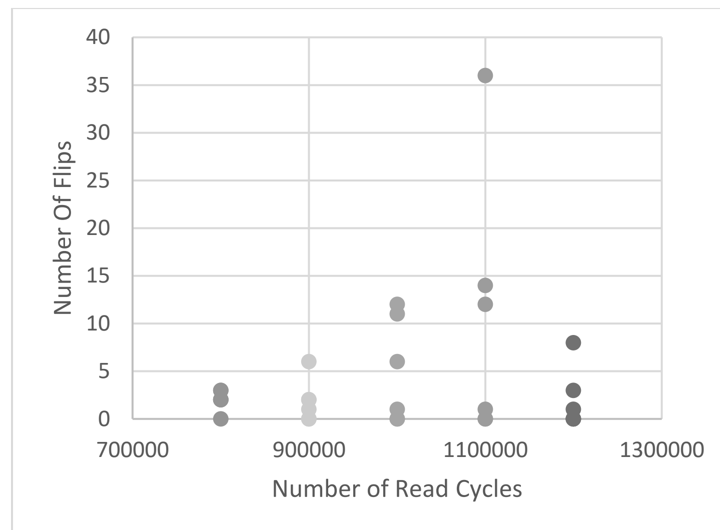

Figure 7: Experimentation with read cycles in attacking rows and the resulting number of flips; the optimal number of flips occurs between 900000 and 1100000 read cycles to get the most flips

According to JETEC memory controller standards, the rows in DRAM must refresh every 64ms. Therefore, it is important to perform as many reads to the attacker rows as possible within the timeframe before refresh. How long it takes the CPU to run a cache clear, then read from the attacker rows can help optimize the number of reads that need to be performed to optimize the attack.

Typically, to perform a timing analysis in C++, the following code is used

```
1.  cl = clock();
2.  // Code that is being timed
3.  cl = clock() - cl;
4.  timer = ((float) cl)/CLOCKS_PER_SEC;
```

The **`clock()`** function in C returns the number of clock ticks that have passed since the program started [26]. Importantly, this is not the number of cycles since the program started, but an actual timer based on the CPU clock frequency. The initial time is first read into **`cl`** before the code that requires time analysis is run. The **`cl`** variable is then reassigned to the current clock time, minus the previous clock time recorded. The result is a time difference in clock ticks that represents the number of clock ticks required to execute the code.

To translate this code into seconds, the value needs to be divided by the **CLOCKS_PER_SEC**, which is a built-in constant that represents the number of clock ticks that occur every second [26].

Running this code on a single run of the assembly code on a set of attacker rows yields about 0.1250 seconds per execution. The assembly code is set to execute in a loop 1000000 times, so the time per loop is approximately 0.125s / 1000000 = $1.25 * 10^{-7}$ seconds, or 125 nano seconds.

The system that the code is running on is an *Intel (R) Core (TM) i7-3770K CPU @ 3.50GHz*, meaning that during normal runtime is executes code at 3.50Ghz, or 3.5 billion cycles per second. Operating at 3.5 billion cycles per second means that a single cycle takes about 0.288 nano seconds to run. According to that calculation, in the period of 125 nano seconds that a single loop takes to run, approximately 434 cycles could have elapsed. In a set of code with only half a dozen assembly instructions, the necessary number of cycles should be far less than 434 cycles. Thus, for an attack with the speed that Rowhammer has, a different form of timing analysis is required.

Conclusion

The idea that protected data is vulnerable to attacks based on hardware vulnerabilities is a large part of the study of information security. To find these hardware vulnerabilities, the architecture needs to be thoroughly examined for flaws. As chips become smaller and more complex, the Rowhammer attack becomes more prevalent. Understanding and reporting on hardware vulnerabilities is the best way to keep the computing world safe.

With the Rowhammer attack, it was important to understand the memory hierarchy, and why DRAM is a useful target for an attack. Then, going deeper into the physics of DRAM cells, it was important to understand the function of sense amplifiers, and how their volatile nature leads them to be corrupted by neighboring cells.

Future work on the Rowhammer study includes methods that circumvent security measures such as Target Row Refresh (TRR) that are designed to prevent bit flipping. One such method is using non-uniform read patterns to prevent mechanisms from knowing that a cell is under attack [27]. There are a variety of hardware manufacturers with different methods of manufacturing, and a study of which methods lead cells to be more vulnerable to attack could yield valuable insight into how to prevent future attacks. Although this report touched on practical implementation, future work could describe how programs become vulnerable at the operating system level to Rowhammer attacks and could be beneficial for software-based Rowhammer security.

Bibliography

[1] K. Yoongu, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," *IEEE,* 2014.

[2] A. . Hyman, Charles Babbage, Pioneer of the Computer, ed., vol. , , : Princeton University Press, 1982, pp. 5,82–87.

[3] B. Lutkevich, "Cache Memory," 2020. [Online]. Available: https://searchstorage.techtarget.com/definition/cache-memory. [Accessed 18 11 2021].

[4] N. P. Jouppi, "Cache write policies and performance," *ACM Sigarch Computer Architecture News,* vol. 21, no. 2, pp. 191-201, 1993.

[5] J. . Meza, Q. . Wu, S. . Kumar and O. . Mutlu, "A Large-Scale Study of Flash Memory Failures in the Field," *,* vol. , no. , p. , .

[6] H. . Hidaka, Y. . Matsuda, M. . Asakura and K. . Fujishima, "The cache DRAM architecture: a DRAM with an on-chip cache memory," *IEEE Micro,* vol. 10, no. 2, pp. 14-25, 1990.

[7] K. . Sasaki, K. . Shimohigashi, K. . Ishibashi and S. . Hanamura, "Sense amplifier for a memory device," , 1990. [Online]. Available: http://freepatentsonline.com/5126974.html. [Accessed 8 12 2021].

[8] O. . Mutlu, "The RowHammer Problem and Other Issues We May Face as Memory Becomes Denser," *arXiv: Distributed, Parallel, and Cluster Computing,* vol. , no. , p. , 2017.

[9] J. . Liu, B. . Jaiyen, Y. . Kim, C. . Wilkerson and O. . Mutlu, "An experimental study of data retention behavior in modern DRAM devices: implications for retention time profiling mechanisms," *ACM Sigarch Computer Architecture News,* vol. 41, no. 3, pp. 60-71, 2013.

[10] "JEDEC History," , . [Online]. Available: https://www.jedec.org/about-jedec/jedec-history. [Accessed 2 12 2021].

[11] Y. . Kim, R. . Daly, J. S. Kim, C. . Fallin, J. H. Lee, D. . Lee, C. . Wilkerson, K. K. Lai and O. . Mutlu, "RowHammer: Reliability Analysis and Security Implications," *arXiv: Distributed, Parallel, and Cluster Computing,* vol. , no. , p. , 2016.

[12] U.-S. . Kang, H.-j. . Chung, S.-M. . Heo, S.-H. . Ahn, H. . Lee, S. . Cha, J. . Ahn, D.-M. . Kwon, J.-H. . Kim, J.-W. . Lee, H.-S. . Joo, W.-S. . Kim, H.-K. . Kim, E.-M. . Lee, S.-R. . Kim, K.-H. . Ma, D.-H. . Jang, N.-S. . Kim, M.-s. . Choi, S.-J. . Oh, J.-B. . Lee, T.-K. . Jung, J.-H. . Yoo and C.-H. . Kim, "8 Gb 3-D DDR3 DRAM Using Through-Silicon-Via Technology," *IEEE Journal of Solid-state Circuits,* vol. 45, no. 1, pp. 111-119, 2009.

[13] Z. B. Aweke, S. F. Yitbarek, R. . Qiao, R. . Das, M. . Hicks, Y. . Oren and T. . Austin, "ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks," *ACM Sigarch Computer Architecture News,* vol. 51, no. 4, pp. 743-755, 2016.

[14] F. F. Brasser, L. . Davi, D. . Gens, C. . Liebchen and A.-R. . Sadeghi, "CAn't Touch This: Practical and Generic Software-only Defenses Against Rowhammer Attacks.," *arXiv: Cryptography and Security,* vol. , no. , p. , 2016.

[15] H. G. Cragon, Memory Systems and Pipelined Processors, ed., vol. , , : Jones and Bartlett Publishers, 1996, p. 113.

[16] "mmap(2) - Linux manual page," , . [Online]. Available: https://www.kernel.org/doc/man-pages/online/pages/man2/mmap.2.html. [Accessed 8 12 2021].

[17] O. . Mutlu and J. S. Kim, "RowHammer: A Retrospective.," *arXiv: Cryptography and Security,* vol. , no. , p. , 2019.

[18] S. . Bhattacharya and D. . Mukhopadhyay, "Advanced Fault Attacks in Software: Exploiting the Rowhammer Bug," , 2018. [Online]. Available: https://link.springer.com/chapter/10.1007/978-981-10-1387-4_6. [Accessed 15 12 2021].

[19] M. S. Daniel Gruss, "Tools for "Another Flip in the Wall"," [Online]. Available: https://github.com/IAIK/flipfloyd.

[20] "Virtual Memory: pages and page frames," , . [Online]. Available: http://blog.cs.miami.edu/burt/2012/10/31/virtual-memory-pages-and-page-frames/. [Accessed 15 12 2021].

[21] "Malloc Examples," , . [Online]. Available: https://www.gnu.org/software/libc/manual/html_node/Malloc-Examples.html#Malloc-Examples. [Accessed 15 12 2021].

[22] "Page Table Management," , . [Online]. Available: https://www.kernel.org/doc/gorman/html/understand/understand006.html. [Accessed 15 12 2021].

[23] S. . Islam, A. . Moghimi, I. . Bruhns, M. . Krebbel, B. . Gulmezoglu, T. . Eisenbarth and B. . Sunar, "SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks," *arXiv: Cryptography and Security,* vol. , no. , p. , 2019.

[24] D. . Evans, "x86 Assembly Guide," , 2006. [Online]. Available: http://www.cs.virginia.edu/~evans/cs216/guides/x86.html. [Accessed 15 12 2021].

[25] K. . R. and . . Irvine, Assembly language for Intel-based computers, ed., vol. , , : Prentice Hall, , p. .

[26] C. . Ehrhardt, "CPU time accounting," , . [Online]. Available: http://www.ibm.com/developerworks/linux/linux390/perf/tuning_cputimes.html. [Accessed 15 12 2021].

[27] P. Jattke, V. v. d. Veen, P. Frigo, S. Gunter and K. Razavi, "BLACKSMITH: Scalable Rowhammering in the Frequency Domain," 2020.

[28] B. . Jacob and T. . Mudge, "Virtual memory: issues of implementation," *IEEE Computer,* vol. 31, no. 6, pp. 33-43, 1998.