Introduction

The aim of this project was to execute Rowhammer and Rowpress on an unknown DDR3 DIMM and define concrete steps to reproduce it. I was able to produce bitflips using both rowhammer and rowpress on some of the DDR3 DIMMS. This report describes the methodology used to do so as well as describes the attempts made and shortcomings in actually getting a bit flip.

Prior Work

MIT Labs [1] has a step by step lab on executing Rowhammer on a particular DIMM. However, the lab does not go into how to reverse engineer the DRAM address mappings. They in fact just provide the mappings as well as the physical addresses of their DIMM on which Rowhammer should be executed.

Similarly, Google's rowhammer-test [2] does not even reverse engineer the DRAM address mapping and takes a probabilistic approach by picking random pairs of addresses. If the machine has 16 banks of DRAM, there is a 1/16 chance that the two addresses picked belong to the same bank. However, this approach is not effective for DDR3 memory since we need the rows to not only lie in the same bank but to also be close to each other.

Furthermore, Yoongu Kim et al's paper[3] also uses DRAM address mappings that are either already given by the vendor or uses an algorithm that needs to be run during OS boot [4]

So, in all the previous work, an important step required to execute Rowhammer has been "obfuscated". This project reproduces and defines more concrete "simpler" steps required to practically reverse engineer these DRAM address mappings.

Rowpress is a very new vulnerability and so I couldn't find any more work apart from the original paper [8].

Methodology & Results: Reverse Engineering DRAM address mappings

1. Timing memory access of addresses mapped to the same bank but different row (Row buffer conflict latency):

We need to find out approximately how many cycles it takes to access two addresses that not only lie in the same bank but also different rows. To do so we will fix a random address say X and pick multiple different random addresses and measure the time to access both of the addresses together. Plotting the data as a histogram tells us the approximate row buffer conflict latency. The smaller spike will indicate the latency of accessing addresses in the same bank and different row since the number of addresses lying in different banks is much more than the number of addresses lying in the same bank for a fixed address X.

For the DDR3 used in this project I found the row buffer conflict latency to be 390 cycles. Moreover, the latency of accessing addresses in different banks is 270 cycles (Figure A).

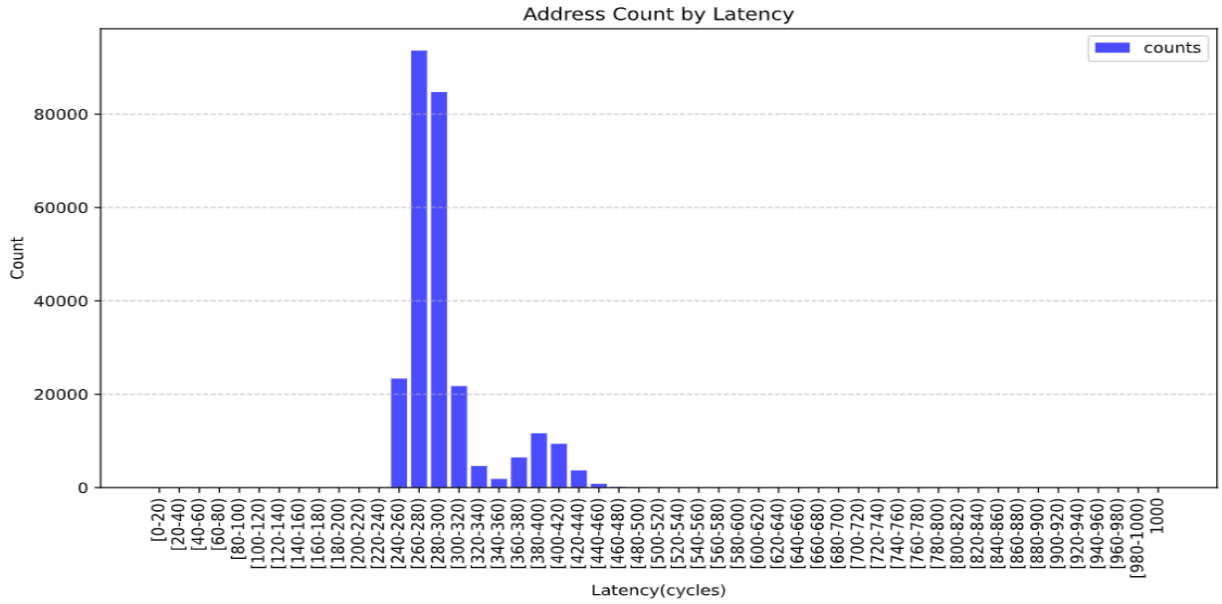


Figure A: Histogram of latency of accessing different addresses against a fixed address X

2. Generating a set of addresses that have a row buffer conflict with a fixed address

Pick a fixed address X and generate a set of addresses such that accessing X along with any one of the addresses in the set has a latency around the one we found for row buffer conflict in Step 1. From the generated set, pick 2 addresses say A and B. Record the latency of accessing X, A and accessing X, A and B. If all three X, A and B lie in the same bank and different row, then their latency will differ by tRC time (the minimum time between ACTs to the same bank).

3. Producing a potential DRAM address mapping function

First determine the number of bits required to represent a row, column, bank, rank, and channel id. Typically, there is some utility tool available like dmidecode [5] that can give you this information. For the DRAM used in this project, the configuration was:

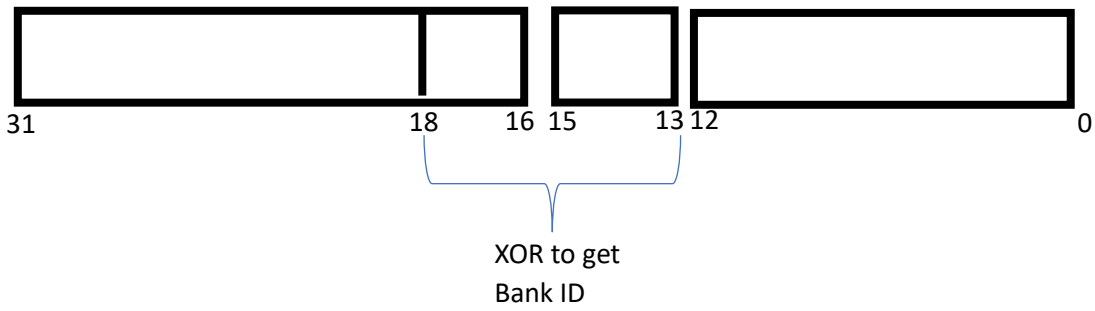
Model	SK Hynix HMT451U6AFR8C-PB
Size	4096 MB
Banks x Rows x Columns x Bits	8 x 16 x 10 x64
Ranks	1
Channel	1

This configuration implied 16 bits are used for rows, 13 bits for column (data is stored as 64B cache line so need 6 bits to index into the cache line. Since row size is 8KB, there are 128 columns for each row, so need another 7 bits to represent that), 3 bits for bank, 0 bits for rank and channel (there is only 1 channel and 1 rank).

We can generate a large set of conflicting addresses from step 2 and from the information above use a SAT solver to guess the DRAM address mapping function. Alternatively, we can analyze the dataset and use OSINT (open source intelligence) to guess a function.

For this project's DRAM, I constructed a function using the latter [6]. I came up with the following hypothesis:

- Row bits are represented by bit 16 to 31 in the physical address.
- Column bits are represented by bit 0 to 12 in the physical address.
- Bank ID is represented by xoring bits 13 to 15 and bits 16 to 18 in the physical address.



4. Confirming hypothesis

To confirm the hypothesis produced in step 3, pick a random virtual address, and construct two other virtual addresses such that their physical address maps to the same bank by changing the row bits in the address. To do the virtual to physical and back to virtual address translations, I used "/proc/self/pagemap" [7] (although you might need sudo privileges to access it during program execution).

Furthermore, the function can be confirmed using the experiment from Yoongu Kim et al's paper [4]. This involves picking a random physical address and for each bit position in the address we construct two addresses by setting that bit to 1 in one address and 0 in the other

address while keeping all the other bits the same. We measure the latency it takes to access both these addresses together. Unfortunately, since a program does not have access to the entire physical memory address space, some physical addresses generated might not be usable and so cannot be used to reliably confirm the guessed function in step 3.

Methodology: Executing Rowhammer

Once the DRAM address mapping function has been reversed, we can pick a victim address and construct addresses that are 1 row about it and 1 row below it. Then we need to activate these constructed addresses at least 130K times each (Rowhammer threshold for DDR3)[3]. To ensure that these addresses are activated we need to flush them from the cache before accessing them (Figure B)

```
asm volatile(
    "mov (%0), %%rax\n\t"
    "mov (%1), %%rax\n\t"
    "clflush (%0)\n\t"
    "clflush (%1)\n\t"
    "mfence\n\t"
    :
    : "r" (attacker_virt_addr_1), "r" (attacker_virt_addr_2)
    : "rax"
);
```

Figure B: Critical section of the hammering code. attacker_virt_addr_1 and attacker_virt_addr_2 are the constructed addresses one row up and one row below the victim address

Moreover, Yoongu Kim et al's paper [3] found that a striped data pattern produced the most number of bit flips. So, setting the victim address's entire row to 0xAA and the attacker addresses' entire row to 0x55 will increase the chance to observe a bit flip.

Results: Executing Rowhammer

I was able to produce bitflips in some DDR3 DIMMS using the above methodology. I noticed bitflips in SK Hynix (part number HMT451U6AFR8C-PB) and Samsung's 1 rank DDR3 (part number M378B5773DH0-CH9). The number of bits flipped seemed random across different runs. Bit flips were noticeable on multiple physical addresses. Notably, Samsung's 1 rank DDR3 produced bit flips at a drastically faster rate than SK Hynix. Some sample vulnerable addresses are mentioned in the vulnerable-addresses file in the hammer directory. I wasn't able to reproduce bitflips on Micron (part number 16JTF51264AZ-1G6M1) and other manufacturers

that only had 2 rank DDR3 dimms. This might be due to an inaccurate DRAM address mapping function (which I tried to fix to account for rank, but it didn't seem to work). For more details about the configuration of the dimms used refer to the dram-specs file in this repository.

Methodology: Executing Rowpress

Once the DRAM address mapping function has been reversed, we can pick a victim address and construct addresses that are 1 row about it and 1 row below it. Then we need to ensure that the selected rows remain active for the maximum duration in a given refresh period. The critical section of the code is exactly the same as described in Haocong Luo et al's Rowpress paper [8].

```
for (int iter = 0; iter < PRESS_PER_ITER; iter++) {
    for (int i = 0; i < PRESS_NUM_AGGR_ACTS; i++) {
        for (int j = 0; j < PRESS_NUM_READS; j++) {
            maccess((uint64_t)(attacker_virt_addr_1_ptr + j));
        }

        for (int j = 0; j < PRESS_NUM_READS; j++) {
            maccess((uint64_t)(attacker_virt_addr_2_ptr + j));
        }

        for (int j = 0; j < PRESS_NUM_READS; j++) {
            clflush((uint64_t)(attacker_virt_addr_1_ptr + j));
            clflush((uint64_t)(attacker_virt_addr_2_ptr + j));
        }
        mfence();
    }
}
```

Figure C: Critical section of the pressing code. attacker_virt_addr_1 and attacker_virt_addr_2 are the constructed addresses one row up and one row below the victim address

Once we have randomly pressed different physical addresses, we try to verify the vulnerability of the addresses in which we observed bitflips by pressing them again. We also verified the vulnerability of such identified rows by specifically trying to press them instead of picking random addresses.

Results: Executing Rowpress

I was able to produce bitflips in some DDR3 DIMMS using the above methodology. I noticed bitflips in SK Hynix (part number HMT451U6AFR8C-PB) and Samsung's 1 rank DDR3 (part number M378B5773DH0-CH9). The number of bits flipped seemed consistent across different runs. Bit flips were noticeable on multiple physical addresses. Not all vulnerable addresses were verifiable by the approach mentioned in the methodology. Some sample vulnerable addresses are mentioned in the vulnerable-addresses file in the press directory. I wasn't able to reproduce bitflips on Micron (part number 16JTF51264AZ-1G6M1) and other manufacturers that only had 2 rank DDR3 dimms. This might be due to an inaccurate DRAM address mapping function (which I tried to fix to account for rank, but it didn't seem to work). For more details about the configuration of the dimms used refer to the dram-specs file in this repository.

Future Work

This project showed promising results regarding the prevalence of the rowpress vulnerability in DDR3 dimms which has not yet been explored. The next step in this project should be to explore the presence of rowpress in DDR3 dimms with other configurations (like rank more than 1 rank).

References

[1] MIT Labs http://csg.csail.mit.edu/6.S983/labs/rowhammer/

[2] Google rowhammer-test https://github.com/google/rowhammer-test

[3] Yoongu Kim et al's Rowhammer paper https://users.ece.cmu.edu/~yoonguk/papers/kim-isca14.pdf

[4] Algorithm for reverse engineering DRAM address mapping function used in [3] https://ieeexplore.ieee.org/document/7842949

[5] dmidecode https://linux.die.net/man/8/dmidecode

[6] Existing reverse engineering DRAM address mapping functions used as OSINT https://lackingrhoticity.blogspot.com/2015/05/how-physical-addresses-map-to-rows-and-banks.html

[7] linux pagemap https://www.kernel.org/doc/Documentation/vm/pagemap.txt

[8] Haocong Luo et al's Rowpress paper https://arxiv.org/pdf/2306.17061