Due date: Wednesday, April 20th 11:59:59 PM ET

Points: This lab is worth 14 points (out of 100 points in 6.888).

Help: If you have any questions, please ask on Piazza.

Collaboration policy: Our full Academic Honesty policy can be found on the Course Information page of our website. As a reminder, **all 6.888 labs should be completed individually**. You may discuss the lab at a high level with a classmate, but you may not work on code together or share any of your code.

Introduction

You have learnt the theory of how Rowhammer attacks [1] work. However, there are several challenges that need to be tackled before you can observe bitflips on a real DRAM chip and conduct a practical Rowhammer attack. One of the key challenges lies in finding memory addresses that map to neighboring rows in the targeted DRAM.

In this lab, we will guide you to tackle this challenge step by step, as shown in Figure 1. In Part 1, you will examine how virtual to physical address mapping works in Linux, and use this knowledge to observe bitflips in the wild. In Part 2, you will extend your knowledge of DRAM geometry fundamentals and identify the bank mapping function used by the architecture under evaluation. In the last part, you will be able to find vulnerable rows by yourself and conduct an end-to-end Rowhammer attack.

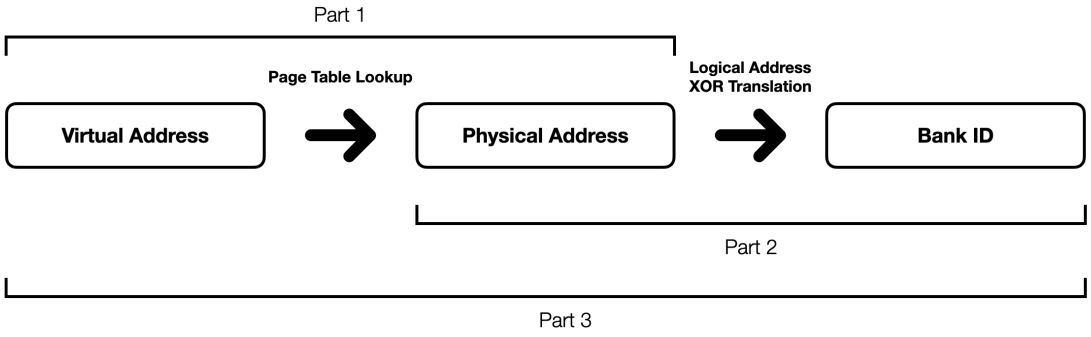

Figure 1: Lab 4 Overview.

C++: In this lab we'll be using C++, which is a slight departure from previous labs (which used C). C++ is an *object-oriented* programming language which shares a majority of its syntax with C, and is backwards compatible with C (apart from some *minor* exceptions). All of the C syntax you've learned and applied in previous labs (ex. pointer referencing, bit manipulation, inline assembly etc.) equally applies in C++. We use C++ in this lab to utilize its standard library data structure support (i.e. `std::vector` and `std:map`), which removes the typical C burden to manage these structures at a low-level (ex. using malloc).

Getting Started With the Lab Infrastructure

Computing Resources: Due to the intrinsic physical requirements of the Rowhammer vulnerability, your experiments must be run on the 6.888 lab machines. These machines have been verified to exhibit the vulnerability, and your answers to some questions will be specific to their architecture/DRAM configuration. There are fewer machines than students, so machines will be time shared.

We don't recommend running any of these experiments locally on your own machine. Recent architectures (which employ DDR4 or later) have some basic mitigations which will defeat the simple hammering approach

we'll use in this lab. Hammering can also occasionally corrupt memory outside of your program's memory space (ex. OS page tables), which may also result in system crashes/stability issues.

As there are fewer available machines than students, and rowhammer experiments can interfere with each other if run simultaneously, the lab machines are managed using HTCondor, a high-throughput job management platform. You will use **arch-sec-1.csail.mit.edu** to develop and build your code, and use HTCondor to remotely launch your attack code on one of the vulnerable machines (`csg-exp[6-8].csail.mit.edu`). Note that you *should not* access these vulnerable machines directly, as you may interfere with another student's experiments. Also note that a vast majority of this lab requires your code to be run under elevated privileges (in order to handle address translation in Linux), which is handled automatically (and only possible) while running your code under HTCondor.

IMPORTANT: Before starting the lab, edit `launch.condor` and update the `'Requirements'` line to reflect the specific machine assigned to you.

Using HTCondor: Since we use HTCondor to manage programs running on vulnerable machines, the process where you launch your code and check its output is different from previous labs. We have created multiple bash scripts to simplify the process. Here is a list of commands that you will use to interact HTcondor:

- `bash launch.sh [BINARY FILE]`: run the specified binary file in your assigned remote machine. The output will be placed in `[BINARY FILE].out`, and any errors generated by your code will be placed in `[BINARY FILE].error`.
- `condor_q`: Check the status of your job, including its place in the queue. This command also prints the id of your job. If your job is held, you can see more debugging information by issuing the `condor_q -hold` command.
- `condor_rm [ID]`: Kill your job specified by the id. Note that Condor will kill your job after 10 minutes of execution, to allow for usage fairness.
- `cat [BINARY FILE].out`: Check the output of your job by opening the corresponding output file. Feel free to use any other editor commands.

Here is an example of using condor to run a program which prints Hello World (after running `make`).

```
1: $ bash launch.sh part0
2:   Submitting job(s).
3:   1 job(s) submitted to cluster XX.
4:   All jobs done.

5: $ condor_q
6:   -- Schedd: arch-sec-1.csail.mit.edu : <127.0.0.1:9618?... @ 01/27/22 13:45:13
7:   OWNER    BATCH_NAME    SUBMITTED   DONE   RUN    IDLE  TOTAL JOB_IDS
8:   pwd      ID: 56       1/27 13:45      _      _      1      1 56.0

5: $ cat part0.out
6:   Hello World!
```

Exercise 1: Try condor commands above and ensure that you can interact with your assigned machine properly.

1 Observing Bitflips in the Wild (40%)

1.1 Virtual and Physical Addresses (20%)

In this part, you will implement the basic Rowhammer attack and observe bitflips in the wild. We have calibrated the DRAM chips in our lab machine and we will give you several vulnerable *physical addresses* to hammer. However, before you can hammer these physical addresses, you have to find the corresponding *virtual addresses*.

Code skeleton: The source code (that you will modify) can be found in `src/`, and is separated into folders corresponding to different parts of the lab (`src/part[0-3]`). In Part 1 in particular, we provide the following files:

- `src/params.hh`: Defines several key system parameters, including the size of a hugepage (i.e. 2MB), the size of a DRAM row, etc.
- `src/verif.hh`: Contains declarations for functions which will help you check your work, and will be used for grading.
- `src/shared.hh` and `src/shared.cc`: Contains functions which are used across the whole lab - in each part of the lab you will be asked to complete a subset of the functions.
- `src/part1/part1.cc`: Contains setup and test code for part 1. Do not modify this file!
- `src/part1/hammertime.cc`: Contains `hammer_addresses`, which you will complete in Section 1.2.

You can compile the code by running the command `make` at the root of the repository, which will create four executable files (`part0`, `part1`, `part2`, and `part3`) in the `bin/` folder. Note that you will need to re-run `make` *each time* you change your code to recompile it. You can test your program by running the command `./part1` and should observe the following output if the compilation works properly.

```
1: $ ./bin/part1
2:    ERROR: Root permissions required!

3: $ bash launch.sh part1
4:   Submitting job(s).
5:   1 job(s) submitted to cluster XX.
6:   All jobs done.

7: $ cat part1.out
8:  Part 1 Test FAILED: Incorrect number of pages in PPN_VPN map (current number of entries: 0)
```

Exercise 2: Run `make`, and make sure the code compiles successfully. Running the commands above should result in the provided output.

Linux `pagemap` interface: Linux and the Intel machines in this lab use *paged virtual memory*. In this lab, we assume that you know the basics of address translation – if you need a refresher, check out the lab help page, or read 6.823's lecture slides on virtual memory.

Linux provides the `pagemap` interface that allows userspace programs to examine page tables and related information. A userspace process can find out which physical page each virtual page is mapped to from a file called `/proc/pid/pagemap`, where `pid` is the process id of the process being examined. The file `/proc/self/pagemap` conveniently maps to the `pagemap` file corresponding to the process reading it.

`/proc/pid/pagemap` is a binary-encoded file containing one 64-bit value for each virtual page *assuming a page size of 4KB*. Effectively, `/proc/pid/pagemap` can be viewed as array which you can index using a *virtual page number* to obtain an 64-bit value, which we call a page table entry.

A single 64-bit page table entry, corresponding to a single virtual page number, contains the following information. First, bits 0-54 of the entry is the physical page number, if the page is present in memory. Second, bit 63 indicates whether the page presents in memory or not. Third, other bits have their own meaning, which are not relevant for this lab. The complete information can be found in Linux kernel documentation page.

In function `virt_to_phys` in `shared.cc`, we have provided you the code to perform file operations on `/proc/pid/pagemap`. The provided file I/O code first opens the pagemap for reading, *seeks* through the file until it reaches an offset corresponding to the virtual page number under inspection. It then reads the page table entry into the variable `entry`. Note that `/proc/pid/pagemap` assumes a page size of 4KB (which is the system default), and thus you should perform all manipulations in `virt_to_phys` as if we were dealing with 4KB pages (rather than hugepages).

To complete `virt_to_phys`, you will need to complete the following high-level tasks outlined in the skeleton code:

1. Given a virtual address, derive its virtual page number (used to calculate where in the file to look).
2. Given a page table entry (`entry`), derive the physical page number.
3. Finally, compute the physical address for the input virtual address

Exercise 3: Complete the `virt_to_phys` function in `src/shared.cc`.

Translating the other way: We now have a function which converts virtual to physical addresses, but we'll also need a way to translate physical addresses to virtual ones. Later in this section, you'll be hammering specific physical addresses which we have been pre-determined to be especially vulnerable. Since you can only access memory through virtual addresses, you'll need to be able to lookup the virtual address for a given physical address.

To do this, your next goal is to populate the `PPN_VPN_map` data structure, a map with physical page numbers as keys and virtual page numbers as values. The `PPN_VPN_map` is an instance of a C++'s `std::map` object. C++ maps are very similar to python dictionaries, with key-value pairs being assigned in a very similar way. One quirk with C++ maps is that, when using standard indexing operators (i.e. []), if you try to retrieve a value which isn't in the map C++ will return 0 (i.e. NULL) instead of throwing an error. [1] You are also free to use other map functions if desired – a full description of the data structure can be found here.

```
std::map<uint64_t, uint64_t> cpp_map;
uint64_t key = 0xDEAD;
uint64_t value = 0xBEEF;

// Add or modify a key-value pair
cpp_map[key] = value;

// Retrieve a value for a key
uint64_t key2 = 0xBAAD;
uint64_t value2 = cpp_map[key2]
if (value2 == 0){
    assert("Key does not exist!\n");
}
```

Throughout this lab we will exclusively use 2MB *hugepages*, as they simplify the manipulation of most physical bits relevant to DRAM. In `part1.cc` (and other parts of this lab) we allocate a large (2GB) block of hugepage-backed memory, which will be used for hammering addresses later in this section. In

[1] This shouldn't pose a problem, however, since you won't encounter any non-zero virtual page numbers.

`setup_PPN_VPN_map` (in `shared.cc`), you will need to populate the `PPN_VPN_map` with corresponding (PPN, VPN) pairs for *each hugepage* in the 2GB region. *Note:* You are asked to populate the map with page numbers assuming a 2MB page size. Be careful here, while you indexed the `/proc/self/pagemap` assuming 4KB page sizes, you will need to use the full 2MB here (your virt_to_phys function will still work for huge pages!).

Discussion Question 1: In a 64-bit system using 4KB pages, which bits are used to represent the page offset and which are used to represent the page number? How about for a 64-bit system with 2MB pages? In a 2GB buffer, how many 2MB hugepages are there?

Exercise 4: Complete the `setup_PPN_VPN_map` function in `shared.cc`. The output of your `part1` (launched on Condor using the `bash launch.sh part1` command) will tell you whether the PPN to VPN map is constructed correctly or not.

Once you're able to configure the `PPN_VPN_map`, you can now use it to translate physical addresses into virtual ones!

Exercise 5: Complete the `phys_to_virt` function in `shared.cc`.

1.2 Hammer Time (20%)

Now that we're done with memory management, it's time to actually write attack code! You will implement the double-sided rowhammer attack, report the probability of observing bit flips, and evaluate how the choices of addresses affect the effectiveness of the attack.

We've pre-profiled the lab machines for rows vulnerable to rowhammer. Note that vulnerable rows don't necessarily correspond to adjacent physical addresses, as we'll further discuss in Part 2. To simplify the hammering process in this section, we directly provide you with physical addresses corresponding to specific rows you should target during the hammering process, as seen in Table 1. You will use these physical addresses to activate the rows in question. In the `main` function (in `part1.cc`), we have provided code to collect statistics and report the observed bitflips by calling a function `hammer_addresses` (in `hammertime.cc`), which will be completed by you.

The main function will test different combinations of rows specified in Table 1 and print the resulted number of bit flips for each combination that you can use to fill Table 2.

Machine	Victim	Row Above (A)	Row Below (B)	Distant Row (C)	Same Row ID, Diff. Bank (D)
csg-exp6.csail.mit.edu	`0x753C1000`	`0x753E3000`	`0x753A7000`	`0x75349000`	`0x753C3000`
csg-exp7.csail.mit.edu	`0x75381000`	`0x753A3000`	`0x7536F000`	`0x75309000`	`0x75383000`
csg-exp8.csail.mit.edu	`0x75360000`	`0x7538E000`	`0x75342000`	`0x753E8000`	`0x75362000`

Table 1: Pre-Profiled Physical Addresses (Relative to the Victim)

Exercise 6: Fill in the victim address (`addr_victim`) and address A-D (`addr_[A-D]`) in `part1.hh` using the addresses for your assigned machine.

Attack outline: Your attack should work with three rows, one victim row and two hammer rows. The *physical addresses* of the three rows will be passed as arguments of the function `hammer_phys`. We describe the high level attack overview below with hints that can help when you get stuck.

1. **Prime** the victim row to set the content of the row, which will be used for comparison later in step 3. For our specific DRAM configuration, a row is of size $2^{13} = 8KB$.

   Hint: Make sure that you align your priming correctly to row boundaries.

2. **Hammer** two rows. Repeat alternatively accessing two rows from DRAM 5 million times.

   Hint: When accessing an address from DRAM, the address will be fetched into cache and subsequent accesses will be cache hits. You need to use `clflush` to evict the two address (which are read in quick succession) to ensure you always access rows in DRAM.

   Hint: To access a row, you only need to access one address from that row. Make sure to access exactly the addresses in Table 1, otherwise the accesses might go to different DRAM banks (discussed in more detail i n Section 2).

3. **Probe** the victim row, compare with the primed results, and check whether any bit has been flipped.

When you working on your code, you may need to convert between integers (`uint64_t`) and pointers (`uint8_t *`) and vice-versa. You will find C++'s `reinterpret_cast` useful in performing such conversion:

```
uint64_t addr = 0xDEADBEEF; // A 64bit integer

// Cast the 64bit integer to a pointer
volatile uint8_t * addr_ptr = reinterpret_cast<volatile uint8_t *>(addr);

uint8_t tmp = *addr_ptr; // Read using the casted pointer
```

Exercise 7: Complete the `hammer_addresses function`. The `part1.cc` code will use your function to hammer each aggressor pair in Table 1 100×, and report how often the attack succeeds (at least one bit flip). Report your results in Table 2. You should (at least) see some bitflips using pair A/B (i.e. double-sided rowhammering).

Hammer Pairs	**A/B**	**A/C**	**A/D**
Number of Successes (out of 100 trials)			

Table 2: Success rate of Rowhammer for different combinations of hammering rows.

Discussion Question 2: Do your results match your expectations? Why might some attacker pairings result in more flips than others? Do you expect any of the pairs to *never* cause a flip?

Submission and Grading: There is no checkoff for Part 1. This part is graded manually based on the following submitted materials: code and a pdf file.

- Code. You will need to submit `part1/hammertime.cc` to your assigned Github repository. You should not modify other files. We will use the Github commit message to track your submission time. We give partial credit (10% of the whole lab) if your code pass the test 1.1 for address translation.
- Discussion Questions. A pdf named as `solutions.pdf` at the root of your assigned Github repository. You will create a pdf file to provide your answers to the discussion questions above. Note that this pdf file will also be used by you to provide answers for the questions in Part 2 and Part 3.

2 Reverse-Engineering DRAM Bank Mapping Functions (40%)

In Part 1, we provided you with physical addresses to hammer and told you which rows are adjacent to the victim row. You may have noticed that adjacent rows are not always consecutive in physical memory, such as the victim address and the address B. This is because DRAM is physically organized into DIMMs, Channels, Ranks, and Banks, necessitating its own addressing scheme. Recall that rowhammer effects are

strongest when considering *adjacent* rows that exist in the *same bank*. You can refer to the detailed address mapping of DRAM in the lecture slides.

As part of the DRAM addressing scheme, the DRAM *bank mapping function* dictates which DRAM bank a physical address is assigned to. It takes a physical address as input, and outputs a bank ID – since there are 16 banks in our DRAM configuration, the bank ID is thus a 4-bit binary value. Each of these 4 bank ID bits is computed by XORing a selection of bits from the physical address.

In Part 2, you will use a timing side channel to reverse engineer the DRAM bank mapping function for our particular architecture, which will assist you to locate adjacent rows and find your own vulnerable bitflips in Part 3.

In this part, we'll guide you through a reverse engineering process which follows these three steps:

1. Implement a function to determine whether two physical addresses map to the same bank
2. Collect a large number of addresses and bin them into multiple groups, such that all the addresses in a group map to the same bank
3. Try different bank mapping functions to find one that does not violate your binning results

2.1 Detecting Bank Collisions (10%)

As the first step of reverse engineering the bank mapping function, you will first implement a function which will help us determine whether two given physical addresses are mapped to the same bank. If two addresses are mapped to the same bank, when accessed, there will be bank collisions which result in longer observable access latencies. Therefore, bank collisions can be detected via timing side channel attacks, which you should be very familiar with from Lab 1. This time, however, instead of attacking the caches you will exploit either memory bus contention, row buffer contention, or both.

Here are two possible timing strategies you might consider employing:

- **Detect row buffer conflicts:** Given two physical addresses x and y, make sure both of the addresses are not cached. First, access address x from DRAM (filling the row buffer with x's row). Next, access address y from DRAM, which will close the row buffer opened by x *if it maps to the same bank*. Finally, access address x again from DRAM and measure its access latency. If x and y are mapped to the same bank but different rows, then this access will result in a row buffer miss and should take a longer time to complete.
- **Detect bus contention and row buffer conflicts:** Given two physical addresses x and y, again make sure both of the addresses are not cached. Access the two addresses back-to-back without memory fences in between and measure their collective access latency. If the two addresses are mapped to the same bank and different rows, they will cause memory bus contention in addition to row buffer conflicts, and thus will result in even longer latency.

Implement an approach of your choice in `measure_bank_latency` (in the file `shared.cc`). The function takes two physical addresses as arguments and measure the access latency using one of the above approaches. We have provided two reference implementations and python scripts for you to check whether your code works correctly or not.

Code skeleton (you do not need to modify any of these files!):

- `src/part2/part2_1.cc`: This file contains the main function which calls the `measure_bank_latency`, populates two arrays `same_bank_latency` and `diff_bank_latency`, and outputs latency results that can be processed by the python scripts.
- `bin/part2_reference1`: a reference implementation that detects only row buffer conflicts.
- `bin/part2_reference2`: a reference implementation that detects both bus contention and row buffer conflicts.

- `src/part2/run.py`: similar to Lab 1, it launches a target binary program 100 times to collect statistics in a folder `data`. The folder `data` will be overwritten each time you run the script.
- `src/part2/graph.py`: similar to Lab 1, it takes the statistics in folder `data` and generate a histogram in the folder `graph`.

Once you've completed your implementation of `measure_bank_latency`, collect and generate the timing histogram by running `bash launch.sh part2_1`. The final timing histogram will be written to the `graphs/` folder. To view the histogram, either copy it to your own machine via SCP, or run `gv graphs/graph_name.pdf` to view the histogram over the network (you may need to set up x-forwarding/add the forwarding flag when connecting via ssh, i.e. `ssh arch-sec-1.csail.mit.edu -X`).

Exercise 8: Complete the function `measure_bank_latency`, and use the provided python scripts to generate a histogram. You can use the reference implementations to see whether your code works correctly or not.

Discussion Question 3: According to the histogram, what is the appropriate threshold to determine whether two addresses are mapped to the same bank or not?

2.2 Find the Correct Bank Mapping Function (30%)

Now you can determine whether two addresses are mapped to the same bank, you can reverse engineer the mapping function by completing the last two steps: 2) collecting a large number of addresses and binning them into multiple groups so that the addresses in the same group are mapped to the same bank; 3) trying all possible bank mapping functions to find one that does not violate your binning results.

To make the lab easier, instead of asking you to try all possible functions via brute force, we will give you 3 candidate functions and ask you to figure out which one is correct. The 3 candidate functions are listed below (listed as $\{B_0, B_1, B_2, B_3\}$, where B is the output bank ID:

Function 0: $\{a_{13} \oplus a_{14}, a_{15} \oplus a_{16}, a_{17} \oplus a_{18}, a_{19} \oplus a_{20}\}$
Function 1: $\{a_{13} \oplus a_{15}, a_{14} \oplus a_{16}, a_{17} \oplus a_{19}, a_{18} \oplus a_{20}\}$
Function 2: $\{a_{13} \oplus a_{17}, a_{14} \oplus a_{18}, a_{16} \oplus a_{20}, a_{15} \oplus a_{19}\}$

First, you will need to write a function (in `shared.cc`) which takes a physical address and a candidate ID from 0-2 (corresponding to the three mapping functions above), and computes the bank ID corresponding to that candidate function.

Exercise 9: Complete the `phys_to_bankid` function.

Now to identify the correct function for our architecture, you should complete the following steps (in `part2/part2_2.cc`, as always launching with `bash launch.sh part2_2`).

1. Allocate a large memory region by calling function `allocate_pages`.
2. Select many addresses that are mapped uniformly across different rows (usually about 1000 addresses is enough to produce clear, correct bins).
3. Bin these addresses into 16 groups where each group corresponds to one bank. You will need to use the function `measure_bank_latency` and the threshold you have derived.

   Hint: It may be useful to instantiate an array of 16 C++ vectors, with each vector representing a bin.
4. Compute the bank id of all the addresses using each of the candidate functions and check whether the computed bank ids for the addresses in the same group are consistent or not.

If the correct function is chosen, the computed bank ids for the addresses in the same group should be the same. However, there may exist some noise in the binning process, and you may observe that the binning results are not 100% accurate. Therefore, you should check the ratio of addresses that have a consistent bank id in each group.

We define the *consistency rate* as the proportion of addresses in a bin which map to the *most common bank ID* in that bin. For instance, if a 4-element bin has addresses mapping to banks [1,1,3,1], the consistency rate of that bin is 75%.

Generally, when using the correct mapping function, the consistency rate should be at or above 98%, as around 98% of the addresses in the same group will have the same bank id, with the remaining 2% having different bank ids. It's okay if your consistency rates fluctuate slightly across tests – it is sufficient to report the results from an arbitrary iteration.

Exercise 10: Use the binning methods described above to compute the consistency rate (minimum, maximum, and average across all of the 16 groups) when using the 3 candidate mapping functions and fill the table below.

	Function 0	Function 1	Function 2
Minimum			
Average			
Maximum			

Table 3: Consistency rate across each of the 16 bank groups

Discussion Question 4: Which function is the correct bank mapping function?

Submission and Grading: There is no checkoff for Part 2. This part is graded manually based on the following submitted materials: code and a pdf file.

- Code. You will need to submit file `part2/shared.cc` and `part2/part2_2.cc` to your assigned Github repository. You should not modify other files. We will use the Github commit message to track your submission time.
- `solutions.pdf`. You will include the generated histogram in the pdf file with the answers to the discussion questions.

3 Finding Vulnerable Rows (20%)

Using our newfound DRAM bank mapping function, you can easily find adjacent rows. Let's now try to find your own vulnerable addresses! You can start by hammering addresses around an arbitrary victim row of your choosing. Recall that the row index of an address is the bits [31:17] (inclusive) of the physical address – also make sure to select your addresses such they target the same bank! If no flips are found in that row, continue trying different victim rows until a vulnerability is found.

Exercise 11: Complete the `phys_to_bankid` function with the mapping function we discovered from part 2. Hammer around to find your own vulnerable addresses. Feel free to re-use your Part 1 implementation to check that your vulnerable addresses exhibit repeatable flips!

Discussion Question 5: What rows did you find bit flips in (give 2-3 examples)? Report the physical addresses. Approximately how many aggressor page pairs did you try to find each vulnerable row?[a]

[a] This isn't a competition! There's inherent randomness involved here, so your numbers may be very different than someone else's!

Discussion Question 6: In this lab, you make Rowhammer attack work by flipping a bit in the DRAM. What kind of attack can you do with such a bitflip? Given an example.

Submission and Grading: You need to submit your code (`shared.cc` and `part3/part3.cc`) to the Github repository and include your answers to the discussion question in `solutions.pdf`. Besides, we will appreciate it if you could help improve this lab by including a description of (a) what challenges you ran into, (b) what worked/didn't work, and (c) any feedback you have for this lab for future years.

There *will be* a checkoff for this part. You will meet with the course staff and reproduce a bit flip using your reported physical address. We understand the Rowhammer phenomenon may not always be reproducible. We will give full credits if you can run the attack code for part 3 and find another bit flip.

Acknowledgments

Contributors: Peter Deutsch, Miguel Gomez-Garcia, Mengjia Yan.

References

[1] Yoongu Kim et al. "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors". In: *ACM SIGARCH Computer Architecture News* 42.3 (2014), pp. 361–372.