ECE 5770 - Resilient Computer Systems
DRAM Bit Flip Errors and Its Countermeasures

Kailin Yang, Joshua Diaz

ABSTRACT

Bit flip errors are ubiquitous in modern DRAM systems. Recent works show that bit flip errors not only impair the data integrity, but also threaten its availability and confidentiality. In this paper, we discuss two major sources of bit flips, disturbance and variable retention time (VRT). Two types of attacks are then designed and implemented based on these – a row hammer attack and a retention time side channel attack respectively. We then implemented two countermeasures to the row hammer attack and discussed results, showing that row hammering can be prevented using a purely hardware strategy.

1. INTRODUCTION

In modern computer systems, main memory is typically implemented as dynamic random-access memory (DRAM). The correct functioning of the DRAM can largely decide the availability and integrity of a system. The most prevailing error in this type of memory is the bit flip error, which is caused by the physical characteristics of DRAM cells. These cells each store one bit of information in the form of charge on a capacitor. Ideally, the capacitor in this cell should be either fully charged or uncharged, however, the charge on high-set capacitors leaks over time. Should that charge be left to leak without refilling, it will eventually cause the bit to flip. To mitigate this effect, DRAM is periodically "refreshed", bringing the charge values back to the proper amount. The typical *refresh* rate today is once every 64 ms [1].

For a long period of time, the bit flip error has been able to be overcome by refreshing and error correction code (ECC). Nevertheless, recent research has shown that new challenges emerge as the density of DRAM cells increases. As the bitlines and wordlines get closer and closer, the electrical interference causes issues that never existed in older generations of DRAMs. For example, the variable retention time [1] and the row hammer [2]. These phenomenons can induce bit flip errors in a conventional memory system using a fixed refresh rate. Bit flips not only impair the data integrity, but also expose the vulnerability for the attackers.

In row hammer attacks, one continuously acceses one row, making some bits in the neighboring rows flip [2]. Physically, the row hammer vulnerability is caused by the disturbance brought by toggling a wordline, which stresses inter-cell coupling effects that accelerate charge leakage from nearby rows. It is ubiquitous among current DRAM chips because the size of the capacitors and transistors becomes smaller, and the wordlines and bitlines are getting closer and closer. In [2] the authors found row hammer vulnerability in 110 out of 129 models of DRAM. And an error in every 1.7K cells is injected by merely 139K accesses.

In [1], the variability of DRAM retention time is discussed in detail. Especially, the authors point out that the retention time of the DRAM cells is dependent to the data pattern that the cells hold. For example, the retention time profiling is usually performed with simple data patterns, e.g., “all 1s”, which overestimates the longevity of the charged cells. The experiments in their work show that testing with “all 1s” or “all 0s” identifies less than 15% of all weak cells. This is due to as the density of the DRAM cells grows, the cells are close to each other. The circuit-level coupling effects make the neighboring cells influence each other. Thus, the data pattern stored in the DRAM decides the retention time. Typically, when a cell is charged but its neighbor cells are not, the leakage is faster than usual. In this case, the cell has a shorter retention time. This phenomenon reveals the vulnerability of a retention time based side channel attack. To reveal the value in a victim row, the attacker can set the refresh rate of its neighbor row as a intermediate value between the normal and the shorter retention time of the cells. Thus, the information in the victim row can be deduced based on the bit flips in the neighbor row.

In this paper, we focus on the two phenomenons introduced above. We launch a hardware based row hammer attack and a retention time side channel attack on an FPGA-based platform. Moreover, we implement two hardware-based countermeasure for the row hammer attack, and evaluate their effect. The evaluation and countermeasure for the side channel attack are left for future work due to the limitation of the experiment platform and time. The rest part of the paper is organized as follow. In section

2 we introduce the related work. Afterwards we present the architecture for the experiment in section 3. Next we show our row hammer attack and retention time side channel attack in section 4 and 5 respectively. Then the evaluation on the row hammer attack is analyzed in section 6. At last we discuss the conclusion and the future work in section 7.

2. RELATED WORK

2.1 Row Hammer

Recent work shows that attackers can exploit row hammer to implement powerful hardware attacks. van der Veen et al. show that deterministic row hammer attacks are feasible on commodity mobile platforms and that they cannot be mitigated by current defenses [3]. They implemented the first Android root exploit relying on no software vulnerability or user permissions, but merely on row hammer. Jung et al. present a technique to reconstruct the physical address of DRAM cells without opening the package. They apply the temperature gradient to the DRAM devices and analysis the retention errors along with it. This reverse engineering makes row hammer attack extremely easy since the exact positions of neighbor rows can be exposed. Seaborn et al. present the attack to gain the kernel privilege of x86 machine based on row hammer [4]. They show a probability of 2% to rewrite the writable bit and 31% to flip a physical page table. All these works show that row hammer is a hardware bug that is easy to exploit.

The existing methods to mitigate row hammer are either software-based or hardware-based. Qiao et al. propose to blacklist the non-temporal instructions [5]. Nonetheless, the blacklisting is only effective for the row hammer attacks based on special instructions, such as *CLFLUSH*. It cannot prevent the attack that rely on no software vulnerability, such as the mechanism in [3]. Aweke et al. design a method to detect row hammer attacks by tracking the locality of DRAM accesses and last level cache miss rate using existing hardware performance counters [6]. Using this method, the rows being frequently accessed can be found. The system selectively refresh their neighbor rows to prevent row hammer attack. This method requires only 1% performance overhead. However, it cannot prevent the attack that bypasses the cache.

Alternatively, in [2] the authors mentioned several hardware-based methods that can mitigate the row hammer. One of the most conventional and general method is using ECC. ECC is easy to implement and can help with all kinds of bit flip errors. However, ECC consumes a large amount of the memory, which reduces the utilization of the chip and increases the cost. The typical capacity overhead of ECC is as large as 12.5%. Another conventional hardware countermeasure is victim cells [7]. The erroneous cells can be identified and mapped to spare cells. But usually discovering all the weak cells takes several days or longer [2]. Besides, it is not practical to find all the vulnerable cells. The authors of [2] propose a refresh-based method called PARA to mitigate the row hammer. The idea is whenever a row is opened, one of its neighbor row will also be opened with a low probability. Statistically, the row hammer can be well covered by PARA. So far, most of the countermeasures require either system level or software level support. The effect of the pure hardware-based countermeasures such as PARA is not attested, although hardware methods are generally easier to implement and can solve the row hammer threat at root.

2.2 Retention Time

The variability of the DRAM retention time makes the refresh operation inefficient to mitigate all the bit flip errors. To solve the bit flips introduced by data pattern dependency (DPD), Khan et al. propose an efficient system-level technique called PARBOR. It determines the locations of the physically neighboring DRAM cells in the system address space, and uses this information to detect data-dependent failures [8]. PARBOR can uncovers 21.9% more data dependent failures compared to the random pattern test. Nonetheless, although it is easier to test DPD errors with the knowledge of neighboring bitline addresses, the user still needs to design specific data patterns to comprehensively reveal all the possible vulnerabilities.

Qureshi et al. propose a multirate refresh scheme called *AVATAR* [9]. The basic idea of AVATAR is adopting error correction code (ECC) DIMMs in a system. At first, AVATAR performs an initial retention time test to populate a row refresh table in which the refresh rate of each row is recorded. When accessing a row, the error detection and correction happens. If any word in a row encounters an ECC error, the system upgrades the refresh rate of that row. ECC scrubbing reduces the failure rate of the system to as low as every tens of years. Meanwhile it reduces the refresh operations by 62% to 72% and increases the performance by more than 35%. However, AVATAR does not increase the refresh rate until an error is found. So it relies on the resiliency of the system. Besides adopting ECC harms the utilization of the memory.

Alternatively, Patel el al. came up with *Reach Profiler* (REAPER) [10]. They profile the failing cells at a longer refresh interval and higher temperature relative to the target conditions, in order to maximize failure coverage while minimizing the false positive rate and profiling runtime. This methodology not only reduces the longevity of retention time test so as to enable the online profiling, but also gives a reasonably conservative retention time. The refresh rate based on the result of REAPER keeps a large margin to endure the influence of data pattern. The experiments show that REAPER can attain a coverage of failing DRAM cells larger than 99% while running 2.5x faster than conventional profiling techniques, meanwhile the false positive rate is less than 50%. This method provides a good insight for the design of the retention time test. However, there is no work utilizing REAPER to build an efficient online profiling system. Though different countermeasures have been discussed for the retention time related bit flip errors, no current work considers the potential attack based on the variable retention time.

3. FPGA-BASED EXPERIMENT INFRASTRUCTURE

A memory controller is implemented on an FPGA, which serves as the experiment infrastructure in our project. In this section we introduce the design of the memory controller. A DE0-Nano development board is used as the basic platform. It contains a Cyclone IV FPGA as well as 32MB on-board DRAM. Besides, in the retention time experiment we connect our development board to the PC using an ATmega2560 microcontroller, so as to program and control the experiment on the host machine.

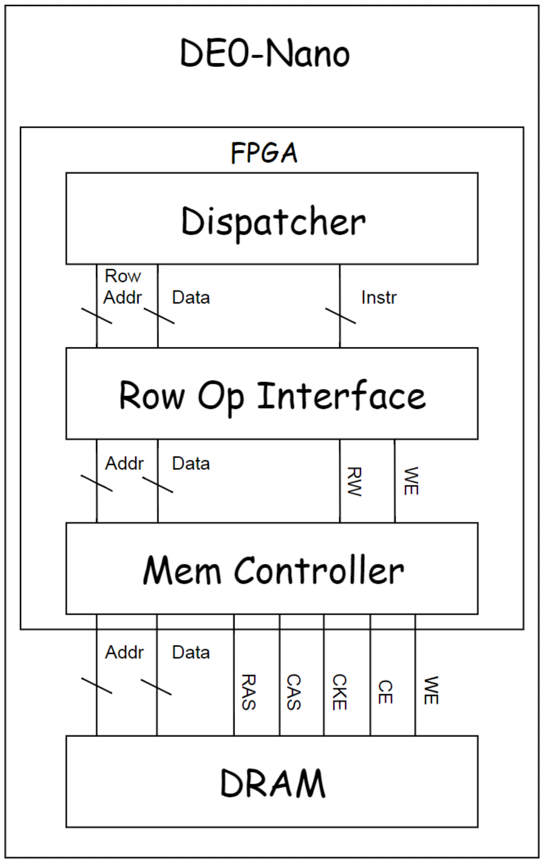

Figure 1: The overall architecture of the FPGA-based experiment infrastructure

Figure 3 shows the overall design of our experiment infrastructure. The goal of our design is to flexibly perform any user specified experiment on the real DRAM device. Hence, a memory controller is first implemented which controls the DRAM using the DDR commands, which are encoded by five signals. On top of the memory controller, a row operation interface is designed which decomposes the row level operations into separate commands, and issues them to the memory controller. Last, the dispatcher performs the experiment by dispatching the row level operations to the row operation interface. The user can design their own experiment by programming the dispatcher directly. Alternatively, inspired by [11], we implement a glue dispatcher which receives the command from a host machine and decodes the command into row level operations. The glue dispatcher communicates with the host via a microcontroller. In this way, the user can program their experiment using a high-level programming language such as C, while still performing the experiment on the real DRAM device.

3.1 Memory Controller

In order to fully control the behavior of the DRAM, we implement a memory controller that directly signals the DRAM using DDR commands. The memory controller takes a `RW` (read/write) signal and a `WE` (write enable) signal as the input. The input address is 24-bit long. The memory controller split the input address from high to low into 2-bit bank address, 13-bit row address, and 9-bit column address. The bank address is directly sent to the DRAM as bank selection signal. The row and column addresses, however, have to go through the same address bus in different cycles, as the DRAM interface only provides a 13-bit address bus. When issuing a row-level DDR command, i.e., `ACTIVATE` or `PRECHARGE`, the row address is sent to the address bus. For the column-level commands (`READ` and `WRITE`), the lower 9 bits of the address bus have to be set as the column address.

In order to read or write a word, the corresponding DDR commands have to be issued to the DRAM in the correct order. Typically, a `PRECHARGE` should be first issued to close the opened rows, followed by an `ACTIVATE` opening the target row. Afterwards the `READ` or `WRITE` commands can be issued. Each `READ` reads only one 16-bit word. Thus, if the user hopes to read or write a whole row, separate commands have to be issued. After the last read or write operation to a row, a `PRECHARGE` is followed to close the row, and get prepared for the next command. Note that the time constraint between each pair of commands has to be observed. Hence after each command there are always several empty cycles. Besides, automatic refresh is also supported by the DDR interface, by a `REFRESH` command. However, the rows to be refreshed are chosen by an inner counter. Hence, although the memory controller we implemented does provide a signal for auto refresh (by setting `RW` as low whereas `WE` as high), the automatic refresh is barely used in practice. Instead, the user can issue a manual refresh by a single read operation, which first opens a row, and then close it. In this case the user can specify the row address that they hope to refresh.

The DDR commands are encoded into five signals, i.e., `CE` (chip enable), `CKE` (clock enable), `RAS` (row address strobe), `CAS` (column address strobe), and `WE` (write enable). By setting correct value to these signals, the corresponding commands are issued. For example, the controller sets `CKE` = 1, `CE` = 0, `RAS` = 0, `CAS` = 1, and `WE` = 1 to create and `ACTIVATE` command. Using the memory controller, the FPGA can get the total control of the behavior of the DRAM. So any intended behavior can be performed.

3.2 Row Operation Interface

In our experiments, row-level operations are frequent. As a result, we wrap the memory controller using a row operation interface, to allow the users efficiently sending the row level operations. The row operation interface takes a 2-bit instruction as the input. The instruction can be either *read*, *write*, *wait*, or *access*. The first two operation will read or write a whole row specified by the user. A *word_ready* bit is given as one of the output signals of the interface. The user snoops that bit once a *read* operation is issued. Each time the *word_ready* jumps from 0 to 1, a word is read out and registered in the data bus. It is the user's responsibility to get the data from the bus before the next word is read out. The *wait* instruction is mostly used in retention time test. The user can specify the cycles to wait. When the cycles is 0, the *wait* operation is equivalent to a `NOP` command. The *access* instruction opens the row specified by the user, and then close it. Two occasions need the *access* operation. First, in the row hammer attack, the neighbor of the victim row needs to be continuously opened and closed. The user can achieve this operation by issuing a bunch of *access* instructions. Second, the user can manually refresh any row by issuing an *access* command. By allowing manual refreshing, the flexibility of the design of the experiments is largely ensured.

3.3 Dispatcher

Given the row operation interface, the user can implement the desired experiment by programming their own dispatcher. Our row hammer attack is implemented in this way. Alternatively, inspired by [11], we also implemented a decoder dispatcher that communicates with a microcontroller (ATmega 2560 MCU) that connects the FPGA to a host PC. With this extension, the user is allowed to program their experiment on the PC side, without involving any hardware programming effort. For example, the user can check if the retention time of a memory cell is larger than a different one in the same row. The full functioning of a retention time experiment utilizing the decoder dispatcher is outlined in section 5. The decoder dispatcher is directly linked to the MCU via a 32-bit bus. The decoder is triggered by the setting of the valid-line that goes from the MCU to the FPGA. Upon this being set, it samples the signals being transmitted from the dispatcher's instruction. These signals, once decoded by the decoder, determine what values are communicated to the memory controller or if the hardware should stall for an amount of time (such as in the *wait_and_read* instruction discussed in section 5).

4. ROW HAMMER AND ITS COUNTERMEASURES

The first experiment we implemented is a hardware-based row hammer attack, and its countermeasures. This part is totally programmed into the FPGA and requires no software support. The dispatcher of the row hammer attack works as follows. First, the user specifies the bank and the row address as the victim, as well as the data pattern. A *write* instruction is then issued to fill the victim row with the desired data pattern. Afterwards, the dispatcher transmits another *write* instruction writing the neighbor row. Then the neighbor row needs to be opened and closed continuously for at least thousands of times, by issuing the *access* instruction over and over again. The dispatcher keeps a counter recording the number of the accesses. By modifying the size of the counter, the user can specify the degree of the interference. Meanwhile, the victim row needs to be refreshed at the normal rate, i.e., every 64 ms. Another counter records the refresh cycle, which reduces by one at each clock cycle. Once the counter is reduced to 0, an *access* is issued to the victim row to refresh it. The neighbor row does not need refresh operations since it is continuously accessed. After the row hammer phase, the data in the victim row is read out word by word. A 8-bit register records the number of bit flips. Once a word is ready, the dispatcher reads it out and compare with the initial data pattern. If there are some mismatching bits, the register increases accordingly. After all the words are read and compared, the register is displayed on the LEDs on the development board.

Two countermeasures for the row hammer are also implemented. The first is called deterministic adjacent row activation (DARA). DARA keeps a counter for each row. The counters are set as 0 at the beginning. Each time an access for a row happens, the according counter is increased by 1. Once the counter is filled, an activation is dispatched to the adjacent row, and the counter is reset to 0. Theoretically, this method can largely overcome the bit flips introduced by row hammer. However, the required overhead is too large. Assigning a counter to every single row largely increases the area overhead of the memory controller. Besides, if the size of the counter is too small, it will overflow easily. Statistically, it will increase the refresh rate and impair the overall performance. Meanwhile, the energy consumption is also increased. Hence, DARA has to find a balanced trade-off between the area and energy overhead.

Alternatively, a probabilistic adjacent row activation (PARA) method is introduced in [2]. The insight of this method is that each time when a row is opened and closed, its neighbor is also opened with a low probability. If one row is continuously accessed, statistically its neighbor will be refreshed eventually. In order to implement PARA in hardware, only a linear-feedback shift register (LFSR) is adopted to generate pseudo random number. When the value in the register happens to be 0, the neighbor row is accessed. The area overhead of a LFSR is negligible compared to the DARA. As a result, the system designer can keep the probability low as long as it can mask the influence of row hammer. However, as PARA is a probabilistic method, there is always a small probability that the bit flip still happens. Besides, the design of LFSR is also critical. The feedback function should be complex enough to ensure the number generated by the LFSR is random enough.

5. RETENTION TIME SIDE CHANNEL

We implemented our second experiment with the same FPGA based memory controller as the row hammer attack, as it allows us to control refresh rates which is necessary to the success of the attack. The retention time side channel varies in effectiveness between rows of a DRAM unit due to its nature of being an exploit on the memory's physical properties. To customize experimentation in lieu of this, commands to the FPGA's memory controller were sent from an ATmega 2560 microcontroller from which different rows can be accessed and analyzed throughout the entire experiment. This host PC communicates an instruction(write/wait and read - more on this to follow), a row address, relevant data for the instruction, and a valid bit, all over a 32 bit bus connected to the FPGA's development board.

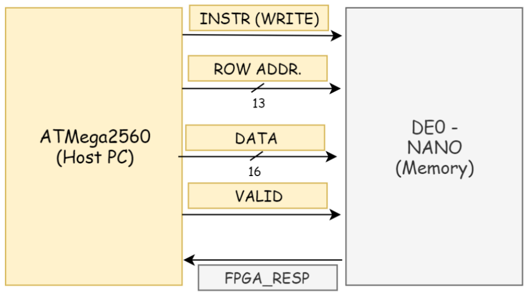

Figure 2: Bus communication between MCU and FPGA for the write instruction

A single test for DRAM retention time vulnerability goes through two stages - *writing*, and *wait_and_reading*. First, the information that we seek to uncover is written to a victim row R_v. The address of this row , along with the 16-bit word of data to be written to it is chosen by the user from the microcontroller interface. Once these lines are set on the bus, the microcontroller sets a valid line that signals the FPGA to sample its lines. The microcontroller will block any proceeding functions until the FPGA sends back an acknowledgment to the ATmega2560 over a line specified for this purpose. This same write command is then used to write data to a row neighboring the victim row (R_{v+1} or R_{v-1}), denoted the *attacking row*. This row is typically written to consist of all-1s, as the retention time bit flip works by flipping the cell from logic value 1 to logic value 0 at different times dependent on its neighbor. Both lines still have refreshing enabled.

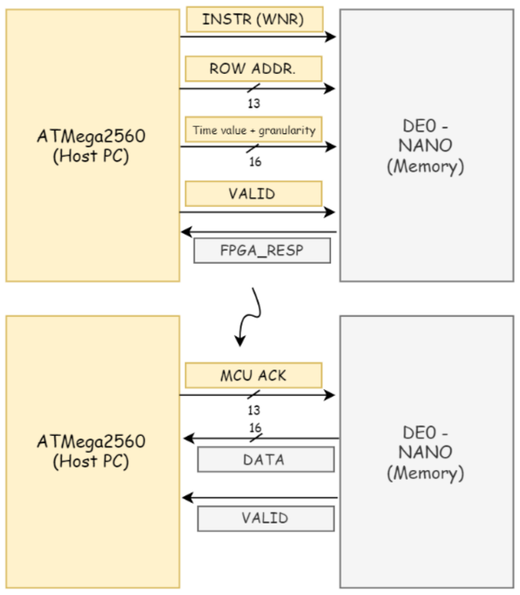

Figure 3: Stage bus communication between MCU and FPGA for the wait and read instruction

With both lines written, we may begin running experiments to determine the retention time of the memory cells in the attacking row. The *wait_and_read* command is specifically designed for this purpose. When used, it sends the row address that we expect to read out along with a time value and time granularity (seconds, milliseconds). It proceeds to block until the FPGA returns with an acknowledgment. The FPGA receives the input, and decodes it to send to the dispatcher. This unit sends the signal to the memory controller to turn off refreshing in the attack row, then waits for the appropriate amount of cycles corresponding to the users time requests (e.g. 50,000,000 cycles to wait one second). This lets the memory cells in the attack row discharge, the rate at which they do being affected by the charge (i.e. logic value) of the neighboring cell (in the victim row). After this, the data may read and the FPGA sends it back to the MCU for analysis By iteratively performing this *wait_and_read* with different amounts of time and checking the value of the data at each iteration - it is possible to measure which bits in our attack row have above-average retention times. Using this, it is possible to deduce which cells in the neighboring row are set to logical value 1. The usage of the external CPU here is useful as it allows tests to be run autonomously, making note of when these outliers are detected.

6. EVALUATION

In this section, we show the evaluation of the row hammer experiment. For each test, we run the row hammer experiment for 50 to 100 times and compute the average number of bit flips. The random number generator we adopted in the PARA is a 16-bit Galois LFSR. Due to the limitation of the infrastructure and time, we leave the evaluation for the retention time side channel attack for future work.

6.1 Number of Accesses

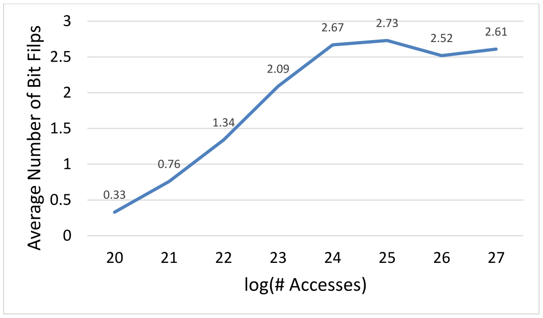

Figure 4: The overall architecture of the FPGA-based experiment infrastructure

Figure 6.1 shows the relationship between the average error rate and the number of accesses in the row hammer experiment. The bit flip starts to happen until the number of accesses is as large as 2^{20}, which is much larger than the threshold mentioned in [2], which is 137K. This is expected due to two reasons. First, the density of our on-board DRAM is lower than the DDR3 DRAM, which is evaluated in [2]. Second, as our development board only has a 50 MHz clock, the frequency is lower than their experiment infrastructure (200 MHz). So the interference is not that severe. As the figure shows, the error rate grows as the number of accesses grows. However, the error rate reaches the upper bound at around 2^{24}. This is expected as the authors of [2] point out that around one out of 1.7K cells is vulnerable to row hammer. In our DRAM bank each row has 8196 cells. Hence, each row has 4 to 5 potential cells vulnerable to row hammer. Considering that the new generation of DRAM is more vulnerable to row hammer, whereas our DRAM is older than DDR3, it is reasonable that for each row there is around 3 vulnerable cells. As a result, we choose 2^{24} as the number of accesses in the following experiments. Table 1 shows the result.

6.2 Data Pattern

In this experiment, we show the impact of the data pattern on the error rate. We tried four data patterns mentioned in [2]: *Solid*, *RowStripe* (all-1s for victim and all-0s for neighbor), *ColStripe* (repeated '01'-pair for both rows), and *Checkered* (repeated '01' for victim and '10' for neighbor).

Table 1: Error rate for different data patterns

Data Pattern	Avg. Bit Flips
Solid	1.92
RowStripe	2.67
ColStripe	2.33
Checkered	2.21

The result shows that row hammer effect is most severe with *RowStripe* data pattern, whereas *Solid* is the least. This phenomenon is consistent with [2]. However, in our experiment, *ColStripe* brings more errors than *Checkered*. We attribute this inconsistency to two reasons. First, as our experiment platform supports checking only one row at a time, the coverage of our experiment is not large enough. With only a small amount of rows tested, it is possible that those rows behave differently than the overall distribution. The second possible reason is that row hammer is non-deterministic. The result of the experiment can naturally have some noise, especially considering the difference between those two results are very small. Generally, our experiment shows that the data pattern has influence on the error rate. When the neighbor row stores different row than the victim row, the error is more likely to happen.

6.3 Countermeasures

We test both of our countermeasures with the most severe RowStripe data pattern. Since the threshold to introduce an error is 173K, which is around 2^{17}, we adopt a 16-bit counter in the DARA. Theoretically, DARA with 16-bit counter can overcome the row hammer vulnerability even for the DDR3 DRAM tested in [2]. The result is as expected, no bit flip error reported in the experiment, since the actual threshold for the bit flip in our DRAM is 2^{20}, far larger than 2^{16}.

In order to make an apple-to-apple comparison, in the PARA we set the LFSR as 16-bit as well. However, this time a 0.02 bit flip is reported. Considering that our LFSR is relatively simple, the cycle for the random number is not long enough, which impairs the effect of our countermeasure. Moreover, we use the switches on the board to control the value of the seed. In each test, we randomly adjust the value of the switches. Nonetheless, there are only 4 switches which can generate only 16 seeds. This limitation further degrades the performance of our method. Nevertheless, our implementation of PARA is already effective enough considering the PARA is a non-deterministic method. Even with a perfect pseudo random number generator, the bit flip is still likely to happen. In general, our experiment shows that the hardware-based method can effectively overcome the row hammer vulnerability, no matter using DARA or PARA.

7. CONCLUSION

We set out to explore various bit-flip attacks that are possible through physical caveats in DRAM. The two techniques that we looked into were the row hammer attack, and the retention time side channel attack, which when used maliciously may threaten the integrity, availability, and even confidentiality of the systems they target. By making use of the Cyclone IV FPGA, we implemented the row hammer attack, revealing its capabilities as a hardware vulnerability. We then experimented with two previously proposed counter attacks, and implemented them in hardware. By evaluating their performances and capability of preventing errors, we showed how improving the hardware itself could prevent malicious activity in memory. Making use of the same FPGA with an ATMega2560 MCU, we implemented the retention time side channel attack. Although effective, this attack is more difficult to produce experimental data for with such a low-end CPU, so we decided to leave this as a proof-of-concept, and the experiment of determining effectiveness and implementing counterattacks for this attack as future work. As DRAM memory only grows more dense and complex, it is important to take into consideration the vulnerabilities left in as flaws in design. By widening the threat model of security design, memory developers can prevent the potential for exploiting these attacks before vulnerabilities in software allow high-level users to exploit them.

8. GROUP DYNAMICS

During the development process of the project, Kailin takes charge of the row hammer attack part whereas Joshua accounts for the retention time side channel part. Kailin's work includes implementing the FPGA based memory controller, row operation interface, and the row hammer attack and countermeasures. Joshua's work contains designing the communication protocol between the FPGA and the microcontroller, implementing the decoder, and the retention time side channel attack. The report is written by both people, each one in charge of the sections corresponding to his work. The outline of the project, and the experiment platform are discussed and designed by two people together.

References

[1] Jamie Liu, Ben Jaiyen, Yoongu Kim, Chris Wilkerson, and Onur Mutlu. An experimental study of data retention behavior in modern dram devices: Implications for retention time profiling mechanisms. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 60–71, New York, NY, USA, 2013. ACM.

[2] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 361–372, Piscataway, NJ, USA, 2014. IEEE Press.

[3] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic rowhammer attacks on mobile platforms. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1675–1689, New York, NY, USA, 2016. ACM.

[4] Mark Seaborn and Thomas Dullien. Exploiting the dram rowhammer bug to gain kernel privileges.

[5] Rui Qiao and Mark Seaborn. A new approach for rowhammer attacks. *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 161–166, 2016.

[6] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin. Anvil: Software-based protection against next-generation rowhammer attacks. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 743–755, New York, NY, USA, 2016. ACM.

[7] Masashi Horiguchi and Kiyoo Itoh. *Nanoscale Memory Repair*, pages 19–67. Springer New York, New York, NY, 2011.

[8] S. Khan, D. Lee, and O. Mutlu. Parbor: An efficient system-level technique to detect data-dependent failures in dram. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 239–250, June 2016.

[9] M. K. Qureshi, D. Kim, S. Khan, P. J. Nair, and O. Mutlu. Avatar: A variable-retention-time (vrt) aware refresh for dram systems. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 427–437, June 2015.

[10] Minesh Patel, Jeremie S. Kim, and Onur Mutlu. The reach profiler (reaper): Enabling the mitigation of dram retention failures via profiling at aggressive conditions. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 255–268, New York, NY, USA, 2017. ACM.

[11] H. Hassan, N. Vijaykumar, S. Khan, S. Ghose, K. Chang, G. Pekhimenko, D. Lee, O. Ergin, and O. Mutlu. Softmc: A flexible and practical open-source infrastructure for enabling experimental dram studies. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 241–252, Feb 2017.