RowHammer

Onur Mutlu, *ETH Zürich and CMU*

Summary— As memory scales down to smaller technology nodes, new failure mechanisms emerge that threaten its correct operation. If such failures are not anticipated and corrected, they can not only degrade system reliability and availability but also, even more importantly, open up new security vulnerabilities: a malicious attacker can exploit the exposed failure mechanism to take over an entire system. As such, new failure mechanisms in memory can become practical and significant threats to system security.

In our ISCA 2014 paper [28], we introduce the RowHammer problem in DRAM, which is a prime (and likely the first) example of a real circuit-level failure mechanism that causes a practical and widespread system security vulnerability. RowHammer, as it is now popularly referred to, is the phenomenon that repeatedly accessing a row in a modern DRAM chip causes bit flips in physically-adjacent rows at consistently predictable bit locations. It is caused by a hardware failure mechanism called *DRAM disturbance errors*, which is a manifestation of circuit-level cell-to-cell interference in a scaled memory technology. Specifically, when a DRAM row is opened (i.e., activated) and closed (i.e., precharged) repeatedly (i.e., *hammered*), enough times within a DRAM refresh interval, one or more bits in physically-adjacent DRAM rows can be flipped to the wrong value. Using an FPGA-based DRAM testing infrastructure [21, 34], we tested 129 DRAM modules manufactured by three major manufacturers in seven recent years (2008–2014) and found that 110 of them exhibited RowHammer errors, the earliest of which dates back to 2010. Our ISCA 2014 paper [28] provides a detailed and rigorous analysis of various characteristics of RowHammer, including its data pattern dependence, repeatability of errors, relationship with leaky cells, and various circuit-level causes of the phenomenon.

Our ISCA 2014 paper demonstrates that a very simple user-level program [3, 28] can reliably and consistently induce RowHammer errors in commodity AMD and Intel systems using vulnerable DRAM modules. We released the source code of this program [3], which Google Project Zero later enhanced [4]. Using our user-level RowHammer program, we showed that both read and write accesses to memory can induce bit flips, all of which occur in rows other than the one that is being accessed. Since different DRAM rows are mapped to different software pages, our user-level program could reliably corrupt specific bits in pages belonging to other programs. As a result, RowHammer errors can be exploited by a malicious program to breach memory protection and compromise the system. In fact, we hypothesized, in our ISCA 2014 paper, that our user-level program, with some engineering effort, could be developed into a *disturbance attack* that injects errors into other programs, crashes the system, or hijacks control of the system.

RowHammer exposes a *security threat* since it leads to a serious breach of memory isolation: an access to one memory row (e.g., an OS page) predictably modifies the data stored in another row (e.g., another OS page). Malicious software, which we call *disturbance attacks* [28], or *RowHammer attacks*, can be written to take advantage of these disturbance errors to take over an entire system. Inspired by our ISCA 2014 paper's fundamental findings, researchers from Google Project Zero demonstrated in 2015 that RowHammer can be effectively exploited by user-level programs to gain kernel privileges on real systems [42, 43]. Tens of other works since then demonstrated other attacks exploiting RowHammer. These include remote takeover of a server vulnerable to RowHammer via JavaScript code execution [19], takeover of a victim virtual machine by another virtual machine running on the same system [41], takeover of a mobile device by a malicious user-level application that requires no permissions [48], takeover of a mobile system by triggering RowHammer using the WebGL interface on a mobile GPU [14], takeover of a remote system by triggering RowHammer through the Remote Direct Memory Access (RDMA) protocol [32, 46], and various other attacks [5, 9, 10, 18, 24, 37–39, 47, 51]. Thus, RowHammer has widespread and profound real implications on system security, as it destroys memory isolation on top of which modern system security principles are built.

Our ISCA 2014 paper provides a wide variety of solutions, both *immediate* and *longer-term*, to RowHammer. A popular *immediate* solution we describe and analyze, is to increase the refresh rate of memory such that the probability of inducing a RowHammer error before DRAM cells get refreshed is reduced. Several major system manufacturers have adopted this solution and released security patches that increased DRAM refresh rates (e.g., [7, 13, 22, 31]) in memory controllers deployed in the field. While this solution is practical and effective in reducing the vulnerability, it has the significant drawbacks of increasing energy/power consumption, reducing system performance, and degrading quality of service experienced by user programs. Our paper shows that the refresh rate needs to be increased by 7X if we want to eliminate *every single* RowHammer-induced error we saw in our tests of 129 DRAM modules. Since DRAM refresh is already a significant burden [12, 25, 26, 33, 40] on energy, performance, and QoS, increasing it by any significant amount would only exacerbate the problem. Yet, increased refresh rate is likely the most practical *immediate* solution to RowHammer.

After describing and analyzing six solutions to RowHammer, our ISCA 2014 paper shows that the long-term solution to RowHammer can actually be simple and low cost. We introduce a new idea, called *PARA (Probabilistic Adjacent Row Activation)*: when the memory controller closes a row (after it was activated), with a very low probability, it refreshes the adjacent rows. The probability value is a parameter determined by the system designer or provided programmatically, if needed, to trade off between performance overhead and vulnerability protection guarantees. We show that this solution is very effective: it eliminates the RowHammer vulnerability, providing much higher reliability guarantees than modern hard disks provide today, while requiring no storage cost and having negligible performance and energy overheads [28].

Our ISCA 2014 paper leads to a new mindset that has enabled a renewed interest in hardware security research: real memory chips are vulnerable, in a simple and widespread manner, and this causes real security problems. We believe the RowHammer problem will worsen over time since DRAM cells are getting closer to each other with technology scaling. Other similar vulnerabilities may also be lurking in DRAM and other types of memories, e.g., NAND flash memory or Phase Change Memory, that can potentially threaten the foundations of secure systems [35]. Our work advocates a principled system-memory co-design approach to memory reliability and security research that can enable us to better anticipate and prevent such vulnerabilities.

Significance and Impact

Our ISCA 2014 paper is the first to demonstrate the RowHammer vulnerability. RowHammer is a prime (and likely the first) example of a hardware failure mechanism that causes a practical and widespread system security vulnerability. Thus, the implications of our ISCA 2014 paper on systems security is tremendous, both in the short term and the long term: it is the first work we know of that shows that a real reliability problem in one of the ubiquitous general-purpose hardware components (DRAM memory) can cause practical and widespread system security vulnerabilities.

Our ISCA 2014 paper has already had significant real-world impact on both industry and academia in at least four directions. These directions will continue to exert long-term impact for RowHammer.

First, our work has inspired many researchers to exploit RowHammer to devise new attacks. As mentioned earlier, tens of papers were written in top security venues that demonstrate various practical attacks exploiting RowHammer (e.g., [5,9,10,14,18,19,24,37,39,41,48,51]). These attacks started with Google Project Zero's first work in 2015 [42, 43] and they continue to this date, with the latest ones that we know of being published in Summer 2018 [32, 38, 46, 47]. We believe there is a lot more to come in this direction: as systems security researchers understand more about RowHammer, and as the RowHammer phenomenon continues to fundamentally affect memory chips due to technology scaling problems [35], researchers and practitioners will develop different types of attacks to exploit RowHammer in various contexts and in many more creative ways. Some recent reports suggest that new-generation DDR4 DRAM chips are vulnerable to RowHammer [29, 37], so the fundamental security research on RowHammer is likely to continue into the future.

Second, our work turned RowHammer into a popular phenomenon [1, 2, 6, 16, 20, 29, 36, 43, 50], which, in turn, helped make hardware security "mainstream" in media and the broad security community. It showed that hardware reliability problems can be very serious security threats that have to be defended against. A well-read article from the Wired magazine, all about RowHammer, is entitled "Forget

Software – Now Hackers are Exploiting Physics!" [17], indicating the shift of mindset towards very low-level hardware security vulnerabilities in the popular mainstream security community. Many other popular articles in press have been written about RowHammer, many of which pointing to the our ISCA 2014 work [28] as the first demonstration and scientific analysis of the RowHammer problem. Showing that hardware reliability problems can be serious security threats and pulling them to the popular discussion space, and thus influencing the mainstream discourse, creates a very long term impact for the RowHammer problem.

Third, our work inspired many solution and mitigation techniques for RowHammer from both researchers and industry practitioners. *Apple* publicly mentioned, in their critical security release for RowHammer, that they increased the memory refresh rates due to the "original research by Yoongu Kim et al. (2014)" [7]. Memtest86 program was updated, including a RowHammer test, acknowledging our ISCA 2014 paper [36]. Many academic works developed solutions to RowHammer, working from our original research (e.g., [8,11,15,18,23,27,30,44,45,49]). We believe such solutions will continue to be generated in both academia and industry, extending our paper's impact into the very long term.

Fourth, and perhaps most importantly, RowHammer enabled a shift of mindset among mainstream security researchers: general-purpose hardware is fallible (in a very widespread manner) and its problems are actually exploitable. This shift of mindset enabled many systems security researchers to examine hardware in more depth and understand its inner workings and vulnerabilities better. We believe it is no coincidence that two of the groups that concurrently discovered the Meltdown and Spectre vulnerabilities (Google Project Zero and TU Graz InfoSec) have heavily worked on RowHammer attacks before.

REFERENCES

[1] "RowHammer Discussion Group," https://groups.google.com/forum/#!forum/rowhammer-discuss.
[2] "RowHammer on Twitter," https://twitter.com/search?q=rowhammer&src=typd.
[3] "Rowhammer: Source code for testing the Row Hammer error mechanism in DRAM devices." https://github.com/CMU-SAFARI/rowhammer.
[4] "Test DRAM for bit flips caused by the rowhammer problem," https://github.com/google/rowhammer-test.
[5] M. T. Aga *et al.*, "When Good Protections go Bad: Exploiting anti-DoS Measures to Accelerate Rowhammer Attacks," in *HOST*, 2017.
[6] B. Aichinger, "The Known Failure Mechanism in DDR3 Memory referred to as Row Hammer," http://ddrdetective.com/files/6414/1036/5710/The_Known_Failure_Mechanism_in_DDR3_memory_referred_to_as_Row_Hammer.pdf, September 2014.
[7] Apple Inc., "About the security content of Mac EFI Security Update 2015-001," https://support.apple.com/en-us/HT204934, June 2015.
[8] Z. B. Aweke *et al.*, "Anvil: Software-based protection against next-generation rowhammer attacks," in *ASPLOS*, 2016.
[9] S. Bhattacharya and D. Mukhopadhyay, "Curious Case of RowHammer: Flipping Secret Exponent Bits using Timing Analysis," in *CHES*, 2016.
[10] E. Bosman *et al.*, "Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector," *S&P*, 2016.
[11] F. Brasser *et al.*, "Can't Touch This: Practical and Generic Software-only Defenses Against RowHammer Attacks," *USENIX Sec.*, 2017.
[12] K. Chang *et al.*, "Improving DRAM performance by parallelizing refreshes with accesses," in *HPCA*, 2014.
[13] T. Fridley and O. Santos, "Mitigations Available for the DRAM Row Hammer Vulnerability," http://blogs.cisco.com/security/mitigations-available-for-the-dram-row-hammer-vulnerability, March 2015.
[14] P. Frigo *et al.*, "Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU," *IEEE S&P*, 2018.
[15] H. Gomez *et al.*, "DRAM Row-hammer Attack Reduction using Dummy Cells," in *NORCAS*, 2016.
[16] D. Goodin, "Once thought safe, DDR4 memory shown to be vulnerable to Rowhammer," https://arstechnica.com/information-technology/2016/03/once-thought-safe-ddr4-memory-shown-to-be-vulnerable-to-rowhammer/, 2016.
[17] A. Greenberg, "Forget Software – Now Hackers are Exploiting Physics," https://www.wired.com/2016/08/new-form-hacking-breaks-ideas-computers-work/, 2016.
[18] D. Gruss *et al.*, "Another Flip in the Wall of Rowhammer Defenses," *IEEE S&P*, 2018.
[19] D. Gruss *et al.*, "Rowhammer.js: A remote software-induced fault attack in javascript," *CoRR*, vol. abs/1507.06955, 2015.
[20] R. Harris, "Flipping DRAM bits - maliciously," http://www.zdnet.com/article/flipping-dram-bits-maliciously/, December 2014.
[21] H. Hassan *et al.*, "SoftMC: A Flexible and Practical Open-Source Infrastructure for Enabling Experimental DRAM Studies," in *HPCA*, 2017.
[22] Hewlett-Packard Enterprise, "HP Moonshot Component Pack Version 2015.05.0," http://h17007.www1.hp.com/us/en/enterprise/servers/products/moonshot/component-pack/index.aspx, 2015.
[23] G. Irazoqui *et al.*, "MASCAT: Stopping Microarchitectural Attacks Before Execution," *IACR Cryptology ePrint Archive*, 2016.
[24] Y. Jang *et al.*, "SGX-Bomb: Locking Down the Processor via Rowhammer Attack," in *SysTEX*, 2017.
[25] U. Kang *et al.*, "Co-architecting controllers and DRAM to enhance DRAM process scaling," in *The Memory Forum*, 2014.
[26] S. Khan *et al.*, "The efficacy of error mitigation techniques for DRAM retention failures: A comparative experimental study," *SIGMETRICS*, 2014.
[27] D.-H. Kim *et al.*, "Architectural Support for Mitigating Row Hammering in DRAM Memories," *IEEE CAL*, 2015.
[28] Y. Kim *et al.*, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," in *ISCA*, 2014.
[29] M. Lanteigne, "How Rowhammer Could Be Used to Exploit Weaknesses in Computer Hardware," http://www.thirdio.com/rowhammer.pdf, March 2016.
[30] E. Lee *et al.*, "TWiCe: Time Window Counter Based Row Refresh to Prevent Row-Hammering," *CAL*, 2018.
[31] Lenovo, "Row Hammer Privilege Escalation," https://support.lenovo.com/us/en/product_security/row_hammer, March 2015.
[32] M. Lipp *et al.*, "Nethammer: Inducing Rowhammer Faults through Network Requests," *arxiv.org*, 2018.
[33] J. Liu *et al.*, "RAIDR: Retention-aware intelligent DRAM refresh," *ISCA*, 2012.
[34] J. Liu *et al.*, "An experimental study of data retention behavior in modern DRAM devices: Implications for retention time profiling mechanisms," *ISCA*, 2013.
[35] O. Mutlu, "The RowHammer problem and other issues we may face as memory becomes denser," *DATE*, 2017.
[36] PassMark Software, "MemTest86: The original industry standard memory diagnostic utility," http://www.memtest86.com/troubleshooting.htm, 2015.
[37] P. Pessl *et al.*, "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks," in *USENIX Security*, 2016.
[38] D. Poddebniak *et al.*, "Attacking Deterministic Signature Schemes using Fault Attacks," in *EuroS&P*, 2018.
[39] R. Qiao and M. Seaborn, "A New Approach for Rowhammer Attacks," in *HOST*, 2016.
[40] M. K. Qureshi *et al.*, "AVATAR: A Variable-Retention-Time (VRT) Aware Refresh for DRAM Systems," in *DSN*, 2015.
[41] K. Razavi *et al.*, "Flip Feng Shui: Hammering a Needle in the Software Stack," *USENIX Security*, 2016.
[42] M. Seaborn and T. Dullien, "Exploiting the DRAM rowhammer bug to gain kernel privileges," http://googleprojectzero.blogspot.com.tr/2015/03/exploiting-dram-rowhammer-bug-to-gain.html.
[43] M. Seaborn and T. Dullien, "Exploiting the DRAM rowhammer bug to gain kernel privileges," *BlackHat*, 2016.
[44] S. M. Seyedzadeh *et al.*, "Counter-based Tree Structure for Row Hammering Mitigation in DRAM," *CAL*, 2017.
[45] M. Son *et al.*, "Making DRAM Stronger Against Row Hammering," in *DAC*, 2017.
[46] A. Tatar *et al.*, "Throwhammer: Rowhammer Attacks over the Network and Defenses," *USENIX ATC*, 2018.
[47] A. Tatar *et al.*, "Defeating Software Mitigations Against Rowhammer: A Surgical Precision Hammer," in *RAID*, 2018.
[48] V. van der Veen *et al.*, "Drammer: Deterministic Rowhammer Attacks on Mobile Platforms," *CCS*, 2016.
[49] V. van der Veen *et al.*, "GuardION: Practical Mitigation of DMA-Based Rowhammer Attacks on ARM," in *DIMVA*, 2018.
[50] Wikipedia, "Row hammer," https://en.wikipedia.org/wiki/Row_hammer.
[51] Y. Xiao *et al.*, "One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation," *USENIX Sec.*, 2016.

Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors

Yoongu Kim[1] Ross Daly* Jeremie Kim[1] Chris Fallin* Ji Hye Lee[1]
Donghyuk Lee[1] Chris Wilkerson[2] Konrad Lai Onur Mutlu[1]
[1]Carnegie Mellon University [2]Intel Labs

Abstract. *Memory isolation is a key property of a reliable and secure computing system — an access to one memory address should not have unintended side effects on data stored in other addresses. However, as DRAM process technology scales down to smaller dimensions, it becomes more difficult to prevent DRAM cells from electrically interacting with each other. In this paper, we expose the vulnerability of commodity DRAM chips to disturbance errors. By reading from the same address in DRAM, we show that it is possible to corrupt data in nearby addresses. More specifically, activating the same row in DRAM corrupts data in nearby rows. We demonstrate this phenomenon on Intel and AMD systems using a malicious program that generates many DRAM accesses. We induce errors in most DRAM modules (110 out of 129) from three major DRAM manufacturers. From this we conclude that many deployed systems are likely to be at risk. We identify the root cause of disturbance errors as the repeated toggling of a DRAM row's wordline, which stresses inter-cell coupling effects that accelerate charge leakage from nearby rows. We provide an extensive characterization study of disturbance errors and their behavior using an FPGA-based testing platform. Among our key findings, we show that (i) it takes as few as 139K accesses to induce an error and (ii) up to one in every 1.7K cells is susceptible to errors. After examining various potential ways of addressing the problem, we propose a low-overhead solution to prevent the errors.*

1. Introduction

The continued scaling of DRAM process technology has enabled smaller cells to be placed closer to each other. Cramming more DRAM cells into the same area has the well-known advantage of reducing the cost-per-bit of memory. Increasing the cell density, however, also has a negative impact on memory reliability due to three reasons. First, a small cell can hold only a limited amount of charge, which reduces its noise margin and renders it more vulnerable to data loss [14, 47, 72]. Second, the close proximity of cells introduces electromagnetic coupling effects between them, causing them to interact with each other in undesirable ways [14, 42, 47, 55]. Third, higher variation in process technology increases the number of outlier cells that are exceptionally susceptible to inter-cell crosstalk, exacerbating the two effects described above.

As a result, high-density DRAM is more likely to suffer from *disturbance*, a phenomenon in which different cells interfere with each other's operation. If a cell is disturbed beyond its noise margin, it malfunctions and experiences a *disturbance error*. Historically, DRAM manufacturers have been aware of disturbance errors since as early as the Intel 1103, the first commercialized DRAM chip [58]. To mitigate disturbance errors, DRAM manufacturers have been employing a two-pronged approach: *(i)* improving inter-cell isolation through circuit-level techniques [22, 32, 49, 61, 73] and *(ii)* screening for disturbance errors during post-production testing [3, 4, 64]. We demonstrate that their efforts to contain disturbance errors have not always been successful, and that erroneous DRAM chips have been slipping into the field.[1]

In this paper, we expose the existence and the widespread nature of disturbance errors in *commodity* DRAM chips sold and used today. Among 129 DRAM modules we analyzed (comprising 972 DRAM chips), we discovered disturbance errors in 110 modules (836 chips). In particular, *all* modules manufactured in the past two years (2012 and 2013) were vulnerable, which implies that the appearance of disturbance errors in the field is a relatively recent phenomenon affecting more advanced generations of process technology. We show that it takes as few as 139K reads to a DRAM address (more generally, to a DRAM row) to induce a disturbance error. As a proof of concept, we construct a user-level program that continuously accesses DRAM by issuing many loads to the same address while flushing the cache-line in between. We demonstrate that such a program induces many disturbance errors when executed on Intel or AMD machines.

We identify the root cause of DRAM disturbance errors as voltage fluctuations on an internal wire called the *wordline*. DRAM comprises a two-dimensional array of cells, where each *row* of cells has its own wordline. To access a cell within a particular row, the row's wordline must be enabled by raising its voltage — i.e., the row must be *activated*. When there are many activations to the same row, they force the wordline to toggle on and off repeatedly. According to our observations, such voltage fluctuations on a row's wordline have a disturbance effect on nearby rows, inducing some of their cells to leak charge at an accelerated rate. If such a cell loses too much charge before it is restored to its original value (i.e., *refreshed*), it experiences a disturbance error.

We comprehensively characterize DRAM disturbance errors on an FPGA-based testing platform to understand their behavior and symptoms. Based on our findings, we examine a number of potential solutions (e.g., error-correction and frequent refreshes), which all have some limitations. We propose an effective and low-overhead solution, called *PARA*, that prevents disturbance errors by probabilistically refreshing only those rows that are likely to be at risk. In contrast to other solutions, PARA does not require expensive hardware structures or incur large performance penalties. This paper makes the following contributions.

*Work done while at Carnegie Mellon University.

[1]The industry has been aware of this problem since at least 2012, which is when a number of patent applications were filed by Intel regarding the problem of "row hammer" [6, 7, 8, 9, 23, 24]. Our paper was under review when the earliest of these patents was released to the public.

978-1-4799-4394-4/14/$31.00

- To our knowledge, this is the first paper to expose the widespread existence of disturbance errors in commodity DRAM chips from recent years.
- We construct a user-level program that induces disturbance errors on real systems (Intel/AMD). Simply by reading from DRAM, we show that such a program could potentially breach memory protection and corrupt data stored in pages that it should not be allowed to access.
- We provide an extensive characterization of DRAM disturbance errors using an FPGA-based testing platform and 129 DRAM modules. We identify the root cause of disturbance errors as the repeated toggling of a row's wordline. We observe that the resulting voltage fluctuation could disturb cells in nearby rows, inducing them to lose charge at an accelerated rate. Among our key findings, we show that *(i)* disturbable cells exist in 110 out of 129 modules, *(ii)* up to one in 1.7K cells is disturbable, and *(iii)* toggling the wordline as few as 139K times causes a disturbance error.
- After examining a number of possible solutions, we propose PARA (*probabilistic adjacent row activation*), a low-overhead way of preventing disturbance errors. Every time a wordline is toggled, PARA refreshes the nearby rows with a very small probability ($p \ll 1$). As a wordline is toggled many times, the increasing disturbance effects are offset by the higher likelihood of refreshing the nearby rows.

2. DRAM Background

In this section, we provide the necessary background on DRAM organization and operation to understand the cause and symptoms of disturbance errors.

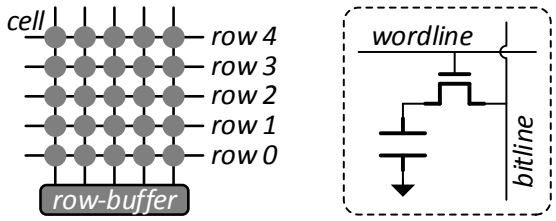
2.1. High-Level Organization

DRAM chips are manufactured in a variety of configurations [34], currently ranging in capacities of 1–8 Gbit and in data-bus widths of 4–16 pins. (A particular capacity does not imply a particular data-bus width.) By itself, an individual DRAM chip has only a small capacity and a narrow data-bus. That is why multiple DRAM chips are commonly ganged together to provide a large capacity and a wide data-bus (typically 64-bit). Such a "gang" of DRAM chips is referred to as a DRAM *rank*. One or more ranks are soldered onto a circuit board to form a DRAM *module*.

2.2. Low-Level Organization

As Figure 1a shows, DRAM comprises a two-dimensional array of DRAM *cells*, each of which consists of a *capacitor* and an *access-transistor*. Depending on whether its capacitor is fully charged or fully discharged, a cell is in either the *charged state* or the *discharged state*, respectively. These two states are used to represent a binary data value.

As Figure 1b shows, every cell lies at the intersection of two perpendicular wires: a horizontal *wordline* and a vertical *bitline*. A wordline connects to all cells in the horizontal direction (*row*) and a bitline connects to all cells in the vertical direction (*column*). When a row's wordline is *raised* to a high voltage, it enables all of the access-transistors within the row, which in turn connects all of the capacitors to their respective bitlines. This allows the row's data (in the form of charge) to be transferred into the *row-buffer* shown in Figure 1a. Better known as *sense-amplifiers*, the row-buffer reads out the charge from the cells — a process that destroys the data in the cells — and immediately writes the charge back into the cells [38, 41, 43]. Subsequently, all accesses to the row are served by the row-buffer on behalf of the row. When there are no more accesses to the row, the wordline is *lowered* to a low voltage, disconnecting the capacitors from the bitlines. A group of rows is called a *bank*, each of which has its own dedicated row-buffer. (The organization of a bank is similar to what was shown in Figure 1a.) Finally, multiple banks come together to form a rank. For example, Figure 2 shows a 2GB rank whose 256K rows are vertically partitioned into eight banks of 32K rows, where each row is 8KB (=64Kb) in size [34]. Having multiple banks increases parallelism because accesses to different banks can be served concurrently.



a. Rows of cells **b.** A single cell

Figure 1. DRAM consists of cells

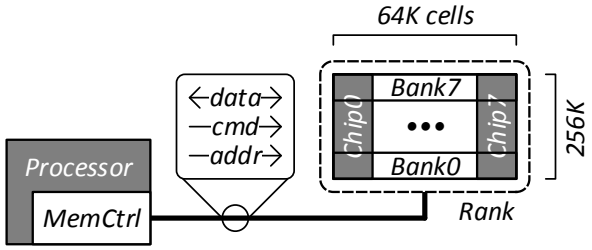


Figure 2. Memory controller, buses, rank, and banks

2.3. Accessing DRAM

An access to a rank occurs in three steps: *(i)* "opening" the desired row within a desired bank, *(ii)* accessing the desired columns from the row-buffer, and *(iii)* "closing" the row.

1. Open Row. A row is opened by raising its wordline. This connects the row to the bitlines, transferring all of its data into the bank's row-buffer.
2. Read/Write Columns. The row-buffer's data is accessed by reading or writing any of its columns as needed.
3. Close Row. Before a different row in the same bank can be opened, the original row must be closed by lowering its wordline. In addition, the row-buffer is cleared.

The *memory controller*, which typically resides in the processor (Figure 2), guides the rank through the three steps by issuing *commands* and *addresses* as summarized in Table 1. After a rank accepts a command, some amount of delay is required before it becomes ready to accept another command. This delay is referred to as a DRAM *timing constraint* [34]. For example, the timing constraint defined between a pair of `ACTIVATE`s to the same row (in the same bank) is referred to as t_{RC} (row cycle time), whose typical value is $\sim$50 *nanoseconds* [34]. When trying to open and close the same row as quickly as possible, t_{RC} becomes the bottleneck — limiting the maximum rate to once every t_{RC}.

Operation	*Command*	*Address(es)*
1. Open Row	`ACTIVATE (ACT)`	Bank, Row
2. Read/Write Column	`READ/WRITE`	Bank, Column
3. Close Row	`PRECHARGE (PRE)`	Bank
Refresh (Section 2.4)	`REFRESH (REF)`	—

Table 1. DRAM commands and addresses [34]

2.4. Refreshing DRAM

The charge stored in a DRAM cell is not persistent. This is due to various leakage mechanisms by which charge can disperse: e.g., subthreshold leakage [56] and gate-induced drain leakage [57]. Eventually, the cell's charge-level would deviate beyond the noise margin, causing it to lose data — in other words, a cell has only a limited *retention time*. Before this time expires, the cell's charge must be restored (i.e., *refreshed*) to its original value: fully charged or fully discharged. The DDR3 DRAM specifications [34] guarantee a retention time of at least 64 *milliseconds*, meaning that all cells within a rank need to be refreshed at least once during this time window. Refreshing a cell can be accomplished by opening the row to which the cell belongs. Not only does the row-buffer read the cell's altered charge value but, at the same time, it restores the charge to full value (Section 2.2). In fact, *refreshing a row and opening a row are identical operations* from a circuits perspective. Therefore, one possible way for the memory controller to refresh a rank is to issue an `ACT` command to every row in succession. In practice, there exists a separate `REF` command which refreshes many rows at a time (Table 1). When a rank receives a `REF`, it automatically refreshes several of its least-recently-refreshed rows by internally generating `ACT` and `PRE` pairs to them. Within any given 64*ms* time window, the memory controller issues a sufficient number of `REF` commands to ensure that every row is refreshed exactly once. For a DDR3 DRAM rank, the memory controller issues 8192 `REF` commands during 64*ms*, once every 7.8*us* (=64*ms*/8192) [34].

3. Mechanics of Disturbance Errors

In general, disturbance errors occur whenever there is a strong enough interaction between two circuit components (e.g., capacitors, transistors, wires) that should be isolated from each other. Depending on which component interacts with which other component and also how they interact, many different modes of disturbance are possible.

Among them, we identify one particular disturbance mode that afflicts commodity DRAM chips from all three major manufacturers. *When a wordline's voltage is toggled repeatedly, some cells in nearby rows leak charge at a much faster rate.* Such cells cannot retain charge for even 64*ms*, the time interval at which they are refreshed. Ultimately, this leads to the cells losing data and experiencing disturbance errors.

Without analyzing DRAM chips at the device-level, we cannot make definitive claims about how a wordline interacts with nearby cells to increase their leakiness. We hypothesize, based on past studies and findings, that there may be three ways of interaction.[2] First, changing the voltage of a wordline could inject noise into an adjacent wordline through *electromagnetic coupling* [15, 49, 55]. This partially enables the adjacent row of access-transistors for a short amount of time and facilitates the leakage of charge. Second, *bridges* are a well-known class of DRAM faults in which conductive channels are formed between unrelated wires and/or capacitors [3, 4]. One study on embedded DRAM (eDRAM) found that toggling a wordline could accelerate the flow of charge between two bridged cells [29]. Third, it has been reported that toggling a wordline for hundreds of hours can permanently damage it by *hot-carrier injection* [17]. If some of the hot-carriers are injected into the neighboring rows, this could modify the amount of charge in their cells or alter the characteristic of their access-transistors to increase their leakiness.

Disturbance errors occur only when the cumulative interference effects of a wordline become strong enough to disrupt the state of nearby cells. In the next section, we demonstrate a small piece of software that achieves this by continuously reading from the same row in DRAM.

4. Real System Demonstration

We induce DRAM disturbance errors on Intel (Sandy Bridge, Ivy Bridge, and Haswell) and AMD (Piledriver) systems using a 2GB DDR3 module. We do so by running Code 1a, which is a program that generates a read to DRAM on every data access. First, the two `mov` instructions read from DRAM at address `X` and `Y` and install the data into a register and also the cache. Second, the two `clflush` instructions evict the data that was just installed into the cache. Third, the `mfence` instruction ensures that the data is fully flushed before any subsequent memory instruction is executed.[3] Finally, the code jumps back to the first instruction for another iteration of reading from DRAM. (Note that Code 1a does not require elevated privileges to execute any of its instructions.)

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

a. Induces errors

```
1 code1b:
2   mov (X), %eax
3   clflush (X)
4
5
6   mfence
7   jmp code1b
```

b. Does not induce errors

Code 1. Assembly code executed on Intel/AMD machines

On out-of-order processors, Code 1a generates multiple DRAM read requests, all of which queue up in the memory controller before they are sent out to DRAM: (req_{X}, req_{Y}, req_{X}, req_{Y}, $\cdots$). Importantly, we chose the values of `X` and `Y` so that they map to the *same* bank, but to *different* rows within the bank.[4] As we explained in Section 2.3, this forces the memory controller to open and close the two rows repeatedly: (ACT_{X}, $READ_{X}$, PRE_{X}, ACT_{Y}, $READ_{Y}$, PRE_{Y}, $\cdots$). Using the address-pair (`X`, `Y`), we then executed Code 1a for millions of iterations. Subsequently, we repeated this procedure

[2] At least one major DRAM manufacturer has confirmed these hypotheses as potential causes of disturbance errors.

[3] Without the `mfence` instruction, there was a large number of hits in the processor's *fill-buffer* [30] as shown by hardware performance counters [31].

[4] Whereas AMD discloses *which* bits of the physical address are used and *how* they are used to compute the DRAM bank address [5], Intel does not. We partially reverse-engineered the addressing scheme for Intel processors using a technique similar to prior work [46, 60] and determined that setting `Y` to `X`+8M achieves our goal for all four processors. We ran Code 1a within a customized Memtest86+ environment [1] to bypass address translation.

using many different address-pairs until every row in the 2GB module was opened/closed millions of times. In the end, we observed that Code 1a caused many bits to flip. For each processor, Table 2 reports the total number of bit-flips induced by Code 1a for two different initial states of the module: all '0's or all '1's.[5,6] Since Code 1a does not write any data into DRAM, we conclude that the bit-flips are the manifestation of disturbance errors. We will show later in Section 6.1 that this particular module — which we named A_{19} (Section 5) — yields *millions* of errors under certain testing conditions.

Bit-Flip	Sandy Bridge	Ivy Bridge	Haswell	Piledriver
'0' → '1'	7,992	10,273	11,404	47
'1' → '0'	8,125	10,449	11,467	12

Table 2. Bit-flips induced by disturbance on a 2GB module

As a control experiment, we also ran Code 1b which reads from only a single address. Code 1b did not induce any disturbance errors as we expected. For Code 1b, all of its reads are to the same row in DRAM: ($req_{\mathtt{X}}$, $req_{\mathtt{X}}$, $req_{\mathtt{X}}$, $\cdots$). In this case, the memory controller minimizes the number of DRAM commands by opening and closing the row just *once*, while issuing many column reads in between: ($\mathtt{ACT}_{\mathtt{X}}$, $\mathtt{READ}_{\mathtt{X}}$, $\mathtt{READ}_{\mathtt{X}}$, $\mathtt{READ}_{\mathtt{X}}$, $\cdots$, $\mathtt{PRE}_{\mathtt{X}}$). As we explained in Section 3, DRAM disturbance errors are caused by the repeated opening/closing of a row, *not* by column reads — which is precisely why Code 1b does *not* induce any errors.

Disturbance errors violate two invariants that memory should provide: *(i)* a read access should *not* modify data at any address and *(ii)* a write access should modify data *only* at the address being written to. As long as a row is repeatedly opened, both read and write accesses can induce disturbance errors (Section 6.2), all of which occur in rows other than the one being accessed (Section 6.3). Since different DRAM rows are mapped (by the memory controller) to different software pages [35], Code 1a — just by accessing its own page — could corrupt pages belonging to other programs. Left unchecked, disturbance errors can be exploited by a malicious program to breach memory protection and compromise the system. With some engineering effort, we believe we can develop Code 1a into a *disturbance attack* that injects errors into other programs, crashes the system, or perhaps even hijacks control of the system. We leave such research for the future since the primary objective in this work is to understand and prevent DRAM disturbance errors.

5. Experimental Methodology

To develop an understanding of disturbance errors, we characterize 129 DRAM modules on an FPGA-based testing platform. Our testing platform grants us precise control over how and when DRAM is accessed on a cycle-by-cycle basis. Also, it does *not* scramble the data it writes to DRAM.[6]

[5]The faster a processor accesses DRAM, the more bit-flips it has. Expressed in the unit of accesses-per-second, the four processors access DRAM at the following rates: 11.6M, 11.7M, 12.3M, and 6.1M. (It is possible that not all accesses open/close a row.)

[6]We initialize the module by making the processor write out all '0's or all '1's to memory. But before this data is actually sent to the module, it is *scrambled* by the memory controller to avoid electrical resonance on the DRAM data-bus [31]. In other words, we do not know the exact "data" that is received by the module. We examine the significance of this in Section 6.4.

Testing Platform. We programmed eight Xilinx FPGA boards [70] with a DDR3-800 DRAM memory controller [71], a PCIe 2.0 core [69], and a customized test engine. After equipping each FPGA board with a DRAM module, we connected them to two host computers using PCIe extender cables. We then enclosed the FPGA boards inside a heat chamber along with a thermocouple and a heater that are connected to an external temperature controller. Unless otherwise specified, all tests were run at 50±2.0°C (ambient).

Tests. We define a *test* as a sequence of DRAM accesses specifically designed to induce disturbance errors in a module. Most of our tests are derived from two snippets of pseudocode listed above (Code 2): TestBulk and TestEach. The goal of TestBulk is to quickly identify the union of all cells that were disturbed after toggling every row many times. On the other hand, TestEach identifies which specific cells are disturbed when each row is toggled many times. Both tests take three input parameters: *AI* (activation interval), *RI* (refresh interval), and *DP* (data pattern). First, AI determines how frequently a row is toggled — i.e., the time it takes to execute one iteration of the inner for-loop. Second, RI determines how frequently the module is refreshed during the test. Third, DP determines the initial data values with which the module is populated before errors are induced. TestBulk (Code 2a) starts by writing DP to the entire module. It then toggles a row at the rate of AI for the full duration of RI — i.e., the row is toggled $N = (2 \times RI)/AI$ times.[7] This procedure is then repeated for every row in the module. Finally, TestBulk reads out the entire module and identifies all of the disturbed cells. TestEach (Code 2b) is similar except that lines 6, 12, and 13 are moved inside the outer for-loop. After toggling just one row, TestEach reads out the module and identifies the cells that were disturbed by the row.

```
1  TestBulk(AI, RI, DP)
2    setAI(AI)
3    setRI(RI)
4    N ← (2 × RI)/AI
5
6    writeAll(DP)
7    for r ← 0···ROW_MAX
8      for i ← 0···N
9        ACT r^th row
10       READ 0^th col.
11       PRE r^th row
12   readAll()
13   findErrors()
```

a. Test all rows at once

```
1  TestEach(AI, RI, DP)
2    setAI(AI)
3    setRI(RI)
4    N ← (2 × RI)/AI
5
6    for r ← 0···ROW_MAX
7      writeAll(DP)
8      for i ← 0···N
9        ACT r^th row
10       READ 0^th col.
11       PRE r^th row
12     readAll()
13     findErrors()
```

b. Test one row at a time

Code 2. Two types of tests synthesized on the FPGA

Test Parameters. In most of our tests, we set AI=55*ns* and RI=64*ms*, for which the corresponding value of N is 2.33×10^6. We chose 55*ns* for AI since it approaches the maximum rate of toggling a row without violating the t_{RC} timing constraint (Section 2.3). In some tests, we also sweep AI up to 500*ns*. We chose 64*ms* for RI since it is the default refresh interval specified by the DDR3 DRAM standard (Section 2.4). In some tests, we also sweep RI down to 10*ms* and up to 128*ms*. For DP, we primarily use two data patterns [65]:

[7]Refresh intervals for different rows are not aligned with each other (Section 2.4). Therefore, we toggle a row for *twice* the duration of *RI* to ensure that we fully overlap with at least one refresh interval for the row.

Manufacturer	Module	Date*	Timing†		Organization		Chip			Victims-per-Module			RI_{th} (ms)
		(yy-ww)	Freq (MT/s)	t_{RC} (ns)	Size (GB)	Chips	Size (Gb)‡	Pins	DieVersion§	Average	Minimum	Maximum	Min
A Total of 43 Modules	A_1	10-08	1066	50.625	0.5	4	1	×16	$\mathcal{B}$	0	0	0	–
	A_2	10-20	1066	50.625	1	8	1	×8	$\mathcal{F}$	0	0	0	–
	$A_{3\text{-}5}$	10-20	1066	50.625	0.5	4	1	×16	$\mathcal{B}$	0	0	0	–
	$A_{6\text{-}7}$	11-24	1066	49.125	1	4	2	×16	$\mathcal{D}$	7.8×10^1	5.2×10^1	1.0×10^2	21.3
	$A_{8\text{-}12}$	11-26	1066	49.125	1	4	2	×16	$\mathcal{D}$	2.4×10^2	5.4×10^1	4.4×10^2	16.4
	$A_{13\text{-}14}$	11-50	1066	49.125	1	4	2	×16	$\mathcal{D}$	8.8×10^1	1.7×10^1	1.6×10^2	26.2
	$A_{15\text{-}16}$	12-22	1600	50.625	1	4	2	×16	$\mathcal{D}$	9.5	9	1.0×10^1	34.4
	$A_{17\text{-}18}$	12-26	1600	49.125	2	8	2	×8	$\mathcal{M}$	1.2×10^2	3.7×10^1	2.0×10^2	21.3
	$A_{19\text{-}30}$	12-40	1600	48.125	2	8	2	×8	$\mathcal{K}$	8.6×10^6	7.0×10^6	$\mathbf{1.0 \times 10^7}$	**8.2**
	$A_{31\text{-}34}$	13-02	1600	48.125	2	8	2	×8	–	1.8×10^6	1.0×10^6	3.5×10^6	11.5
	$A_{35\text{-}36}$	13-14	1600	48.125	2	8	2	×8	–	4.0×10^1	1.9×10^1	6.1×10^1	21.3
	$A_{37\text{-}38}$	13-20	1600	48.125	2	8	2	×8	$\mathcal{K}$	1.7×10^6	1.4×10^6	2.0×10^6	9.8
	$A_{39\text{-}40}$	13-28	1600	48.125	2	8	2	×8	$\mathcal{K}$	5.7×10^4	5.4×10^4	6.0×10^4	16.4
	A_{41}	14-04	1600	49.125	2	8	2	×8	–	2.7×10^5	2.7×10^5	2.7×10^5	18.0
	$A_{42\text{-}43}$	14-04	1600	48.125	2	8	2	×8	$\mathcal{K}$	0.5	0	1	62.3
B Total of 54 Modules	B_1	08-49	1066	50.625	1	8	1	×8	$\mathcal{D}$	0	0	0	–
	B_2	09-49	1066	50.625	1	8	1	×8	$\mathcal{E}$	0	0	0	–
	B_3	10-19	1066	50.625	1	8	1	×8	$\mathcal{F}$	0	0	0	–
	B_4	10-31	1333	49.125	2	8	2	×8	$\mathcal{C}$	0	0	0	–
	B_5	11-13	1333	49.125	2	8	2	×8	$\mathcal{C}$	0	0	0	–
	B_6	11-16	1066	50.625	1	8	1	×8	$\mathcal{F}$	0	0	0	–
	B_7	11-19	1066	50.625	1	8	1	×8	$\mathcal{F}$	0	0	0	–
	B_8	11-25	1333	49.125	2	8	2	×8	$\mathcal{C}$	0	0	0	–
	B_9	11-37	1333	49.125	2	8	2	×8	$\mathcal{D}$	1.9×10^6	1.9×10^6	1.9×10^6	11.5
	$B_{10\text{-}12}$	11-46	1333	49.125	2	8	2	×8	$\mathcal{D}$	2.2×10^6	1.5×10^6	$\mathbf{2.7 \times 10^6}$	11.5
	B_{13}	11-49	1333	49.125	2	8	2	×8	$\mathcal{C}$	0	0	0	–
	B_{14}	12-01	1866	47.125	2	8	2	×8	$\mathcal{D}$	9.1×10^5	9.1×10^5	9.1×10^5	**9.8**
	$B_{15\text{-}31}$	12-10	1866	47.125	2	8	2	×8	$\mathcal{D}$	9.8×10^5	7.8×10^5	1.2×10^6	11.5
	B_{32}	12-25	1600	48.125	2	8	2	×8	$\mathcal{E}$	7.4×10^5	7.4×10^5	7.4×10^5	11.5
	$B_{33\text{-}42}$	12-28	1600	48.125	2	8	2	×8	$\mathcal{E}$	5.2×10^5	1.9×10^5	7.3×10^5	11.5
	$B_{43\text{-}47}$	12-31	1600	48.125	2	8	2	×8	$\mathcal{E}$	4.0×10^5	2.9×10^5	5.5×10^5	13.1
	$B_{48\text{-}51}$	13-19	1600	48.125	2	8	2	×8	$\mathcal{E}$	1.1×10^5	7.4×10^4	1.4×10^5	14.7
	$B_{52\text{-}53}$	13-40	1333	49.125	2	8	2	×8	$\mathcal{D}$	2.6×10^4	2.3×10^4	2.9×10^4	21.3
	B_{54}	14-07	1333	49.125	2	8	2	×8	$\mathcal{D}$	7.5×10^3	7.5×10^3	7.5×10^3	26.2
C Total of 32 Modules	C_1	10-18	1333	49.125	2	8	2	×8	$\mathcal{A}$	0	0	0	–
	C_2	10-20	1066	50.625	2	8	2	×8	$\mathcal{A}$	0	0	0	–
	C_3	10-22	1066	50.625	2	8	2	×8	$\mathcal{A}$	0	0	0	–
	$C_{4\text{-}5}$	10-26	1333	49.125	2	8	2	×8	$\mathcal{B}$	8.9×10^2	6.0×10^2	1.2×10^3	29.5
	C_6	10-43	1333	49.125	1	8	1	×8	$\mathcal{T}$	0	0	0	–
	C_7	10-51	1333	49.125	2	8	2	×8	$\mathcal{B}$	4.0×10^2	4.0×10^2	4.0×10^2	29.5
	C_8	11-12	1333	46.25	2	8	2	×8	$\mathcal{B}$	6.9×10^2	6.9×10^2	6.9×10^2	21.3
	C_9	11-19	1333	46.25	2	8	2	×8	$\mathcal{B}$	9.2×10^2	9.2×10^2	9.2×10^2	27.9
	C_{10}	11-31	1333	49.125	2	8	2	×8	$\mathcal{B}$	3	3	3	39.3
	C_{11}	11-42	1333	49.125	2	8	2	×8	$\mathcal{B}$	1.6×10^2	1.6×10^2	1.6×10^2	39.3
	C_{12}	11-48	1600	48.125	2	8	2	×8	$\mathcal{C}$	7.1×10^4	7.1×10^4	7.1×10^4	19.7
	C_{13}	12-08	1333	49.125	2	8	2	×8	$\mathcal{C}$	3.9×10^4	3.9×10^4	3.9×10^4	21.3
	$C_{14\text{-}15}$	12-12	1333	49.125	2	8	2	×8	$\mathcal{C}$	3.7×10^4	2.1×10^4	5.4×10^4	21.3
	$C_{16\text{-}18}$	12-20	1600	48.125	2	8	2	×8	$\mathcal{C}$	3.5×10^3	1.2×10^3	7.0×10^3	27.9
	C_{19}	12-23	1600	48.125	2	8	2	×8	$\mathcal{E}$	1.4×10^5	1.4×10^5	1.4×10^5	18.0
	C_{20}	12-24	1600	48.125	2	8	2	×8	$\mathcal{C}$	6.5×10^4	6.5×10^4	6.5×10^4	21.3
	C_{21}	12-26	1600	48.125	2	8	2	×8	$\mathcal{C}$	2.3×10^4	2.3×10^4	2.3×10^4	24.6
	C_{22}	12-32	1600	48.125	2	8	2	×8	$\mathcal{C}$	1.7×10^4	1.7×10^4	1.7×10^4	22.9
	$C_{23\text{-}24}$	12-37	1600	48.125	2	8	2	×8	$\mathcal{C}$	2.3×10^4	1.1×10^4	3.4×10^4	18.0
	$C_{25\text{-}30}$	12-41	1600	48.125	2	8	2	×8	$\mathcal{C}$	2.0×10^4	1.1×10^4	3.2×10^4	19.7
	C_{31}	13-11	1600	48.125	2	8	2	×8	$\mathcal{C}$	3.3×10^5	3.3×10^5	$\mathbf{3.3 \times 10^5}$	**14.7**
	C_{32}	13-35	1600	48.125	2	8	2	×8	$\mathcal{C}$	3.7×10^4	3.7×10^4	3.7×10^4	21.3

∗ We report the manufacture date marked on the chip packages, which is more accurate than other dates that can be gleaned from a module.
† We report timing constraints stored in the module's on-board ROM [33], which is read by the system BIOS to calibrate the memory controller.
‡ The maximum DRAM chip size supported by our testing platform is 2Gb.
§ We report DRAM die versions marked on the chip packages, which typically progress in the following manner: $\mathcal{M} \rightarrow \mathcal{A} \rightarrow \mathcal{B} \rightarrow \mathcal{C} \rightarrow \cdots$.

Table 3. Sample population of 129 DDR3 DRAM modules, categorized by manufacturer and sorted by manufacture date

RowStripe (even/odd rows populated with '0's/'1's) and its inverse ∼*RowStripe*. As Section 6.4 will show, these two data patterns induce the most errors. In some tests, we also use *Solid*, *ColStripe*, *Checkered*, as well as their inverses [65].

DRAM Modules. As listed in Table 3, we tested for disturbance errors in a total of 129 DDR3 DRAM modules. They comprise 972 DRAM chips from three manufacturers whose names have been anonymized to A, B, and C.[8] The three manufacturers represent a large share of the global DRAM market [20]. We use the following notation to reference the modules: M_i^{yyww} (M for the manufacturer, i for the numerical identifier, and *yyww* for the manufacture date in year and week).[9] Some of the modules are indistinguishable from each other in terms of the manufacturer, manufacture date, and chip type (e.g., $\mathsf{A}_{3\text{-}5}$). We collectively refer to such a group of modules as a *family*. For multi-rank modules, only the first rank is reflected in Table 3, which is also the only rank that we test. We will use the terms module and rank interchangeably.

6. Characterization Results

We now present the results from our characterization study. Section 6.1 explains how the number of disturbance errors in a module varies greatly depending on its manufacturer and manufacture date. Section 6.2 confirms that repeatedly activating a row is indeed the source of disturbance errors. In addition, we also measure the minimum number of times a row must be activated before errors start to appear. Section 6.3 shows that the errors induced by such a row (i.e., the *aggressor row*) are predominantly localized to two other rows (i.e., the *victim rows*). We then provide arguments for why the victim rows are likely to be the immediate neighbors. Section 6.4 demonstrates that disturbance errors affect only the charged cells, causing them to lose data by becoming discharged.

6.1. Disturbance Errors are Widespread

For every module in Table 3, we tried to induce disturbance errors by subjecting them to two runs of TestBulk:

1. TestBulk(55*ns*, 64*ms*, RowStripe)
2. TestBulk(55*ns*, 64*ms*, ∼RowStripe)

If a cell experienced an error in either of the runs, we refer to it as a *victim cell* for that module. Interestingly, virtually no cell in any module had errors in both runs — meaning that the number of errors summed across the two runs is equal to the number of unique victims for a module.[10] (This is an important observation that will be examined further in Section 6.4.)

For each family of modules, three right columns in Table 3 report the avg/min/max number of victims among the modules belonging to the family. As shown in the table, we were able to induce errors in all but 19 modules, most of which are also the oldest modules from each manufacturer. In fact, there exist date boundaries that separate the modules with errors from those without. For A, B, and C, their respective date boundaries are 2011-24, 2011-37, and 2010-26. Except for A_{42}, B_{13}, and C_6, every module manufactured on or after these dates exhibits errors. These date boundaries are likely to indicate process upgrades since they also coincide with die version upgrades. Using manufacturer B as an example, 2Gb×8 chips before the boundary have a die version of $\mathcal{C}$, whereas the chips after the boundary (except B_{13}) have die versions of either $\mathcal{D}$ or $\mathcal{E}$. Therefore, we conclude that disturbance errors are a relatively recent phenomenon, affecting almost all modules manufactured within the past 3 years.

Using the data from Table 3, Figure 3 plots the normalized number of errors for each family of modules versus their manufacture date. The error bars denote the minimum and maximum for each family. From the figure, we see that modules from 2012 to 2013 are particularly vulnerable. For each manufacturer, the number of victims per 10^9 cells can reach up to 5.9×10^5, 1.5×10^5, and 1.9×10^4. Interestingly, Figure 3 reveals a jigsaw-like trend in which sudden jumps in the number of errors are followed by gradual descents. This may occur when a manufacturer migrates away from an old-but-reliable process to a new-but-unreliable process. By making adjustments over time, the new process may eventually again become reliable — which could explain why the most recent modules from manufacturer A ($\mathsf{A}_{42\text{-}43}$) have little to no errors.

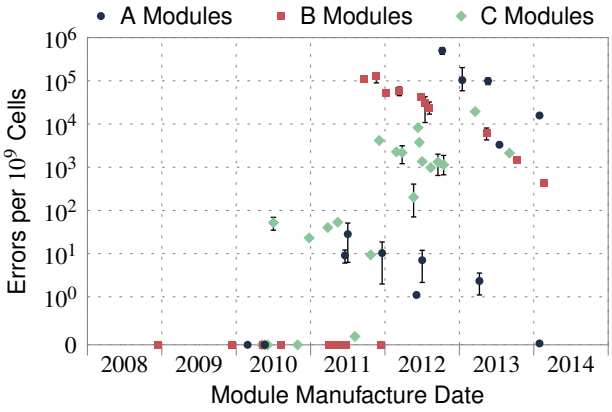


Figure 3. Normalized number of errors vs. manufacture date

6.2. Access Pattern Dependence

So far, we have demonstrated disturbance errors by repeatedly opening, reading, and closing the same row. We express this access pattern using the following notation, where N is a large number: $(\textit{open–read–close})^N$. However, this is not the only access pattern to induce errors. Table 4 lists a total of four different access patterns, among which two induced errors on the modules that we tested: A_{23}, B_{11}, and C_{19}. These three modules were chosen because they had the most errors (A_{23} and B_{11}) or the second most errors (C_{19}) among all modules from the same manufacturer. What is in common between the first two access patterns is that they open and close the same row repeatedly. The other two, in contrast, do so just once and did not induce any errors. From this we conclude that the repeated toggling of the same wordline is indeed the cause of disturbance errors.[11]

[8] We tried to avoid third-party modules since they sometimes obfuscate the modules, making it difficult to determine the actual chip manufacturer or the exact manufacture date. Modules $\mathsf{B}_{14\text{-}31}$ are engineering samples.

[9] Manufacturers do not explicitly provide the technology node of the chips. Instead, we interpret recent manufacture dates and higher die versions as rough indications of more advanced process technology.

[10] In some of the B modules, there were some rare victim cells ($\leq$15) that had errors in both runs. We will revisit these cells in Section 6.3.

[11] For write accesses, a row cannot be opened and closed once every t_{RC} due to an extra timing constraint called t_{WR} (write recovery time) [34]. As a result, the second access pattern in Table 4 induces fewer errors.

Access Pattern	Disturbance Errors?
1. $(open–read–close)^N$	**Yes**
2. $(open–write–close)^N$	**Yes**
3. $open–read^N–close$	No
4. $open–write^N–close$	No

Table 4. Access patterns that induce disturbance errors

Refresh Interval (RI). As explained in Section 5, our tests open a row once every 55*ns*. For each row, we sustain this rate for the full duration of an RI (default: 64*ms*). This is so that the row can maximize its disturbance effect on other cells, causing them to leak the most charge before they are next refreshed. As the RI is varied between 10–128*ms*, Figure 4 plots the numbers of errors in the three modules. Due to time limitations, we tested only the first bank. For shorter RIs, there are fewer errors due to two reasons: *(i)* a victim cell has less time to leak charge between refreshes; *(ii)* a row is opened fewer times between those refreshes, diminishing the disturbance effect it has on the victim cells. At a sufficiently short RI — which we refer to as the *threshold refresh interval* (RI_{th}) — errors are completely eliminated not in just the first bank, but for the entire module. For each family of modules, the rightmost column in Table 3 reports the minimum RI_{th} among the modules belonging to the family. The family with the most victims at RI = 64*ms* is also likely to have the lowest RI_{th}: 8.2*ms*, 9.8*ms*, and 14.7*ms*. This translates into 7.8×, 6.5×, and 4.3× increase in the frequency of refreshes.

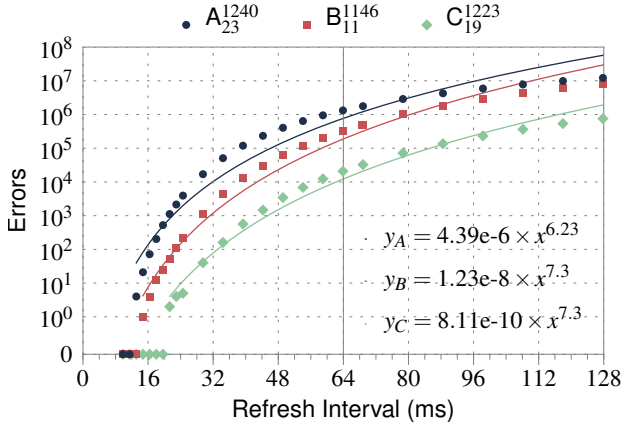


Figure 4. Number of errors as the refresh interval is varied

Activation Interval (AI). As the AI is varied between 55–500*ns*, Figure 5 plots the numbers of errors in the three modules. (Only the first bank is tested, and the RI is kept constant at 64*ms*.) For longer AIs, there are fewer errors because a row is opened less often, thereby diminishing its disturbance effect. When the AI is sufficiently long, the three modules have no errors: ∼500*ns*, ∼450*ns*, and ∼250*ns*. At the shortest AIs, however, there is a notable reversal in the trend: B_{11} and C_{19} have fewer errors at 60*ns* than at 65*ns*. How can there be fewer errors when a row is opened more often? This anomaly can be explained only if the disturbance effect of opening a row is weaker at 60*ns* than at 65*ns*. In general, row-coupling effects are known to be weakened if the wordline voltage is not raised quickly while the row is being opened [55]. The wordline voltage, in turn, is raised by a circuit called the *wordline charge-pump* [38], which becomes sluggish if not given enough time to "recover" after performing its job.[12] When a wordline is raised every 60*ns*, we hypothesize that the charge-pump is unable to regain its full strength by the end of each interval, which leads to a slow voltage transition on the wordline and, ultimately, a weak disturbance effect. In contrast, an AI of 55*ns* appears to be immune to this phenomenon, since there is a large jump in the number of errors. We believe this to be an artifact of how our memory controller schedules refresh commands. At 55*ns*, our memory controller happens to run at 100% utilization, meaning that it always has a DRAM request queued in its buffer. In an attempt to minimize the latency of the request, the memory controller deprioritizes a pending refresh command by ∼64*us*. This technique is fully compliant with the DDR3 DRAM standard [34] and is widely employed in general-purpose processors [31]. As a result, the effective refresh interval is slightly lengthened, which again increases the number of errors.

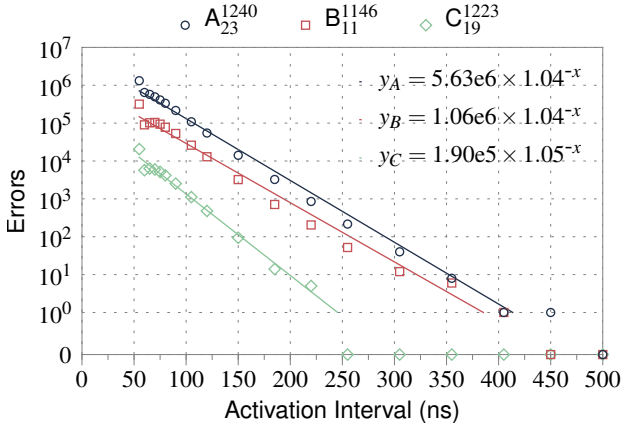


Figure 5. Number of errors as the activation interval is varied

Number of Activations. We have seen that disturbance errors are heavily influenced by the lengths of RI and AI. In Figure 6, we compare their effects by superimposing the two previous figures on top of each other. Both figures have been normalized onto the same x-axis whose values correspond to the number of activations per refresh interval: RI/AI.[13] (Only the left-half is shown for Figure 4, where RI ≤ 64*ms*.) In Figure 6, the number of activations reaches a maximum of 1.14×10^6 (=64*ms*/55*ns*) when RI and AI are set to their default lengths. At this particular point, the numbers of errors between the two studies degenerate to the same value. It is clear from the figure that fewer activations induce fewer errors. For the same number of activations, having a long RI and a long AI is likely to induce more errors than having a short RI and a short AI. We define the *threshold number of activations* (N_{th}) as the minimum number of activations that is required to induce an error when RI=64*ms*. The three modules (for only their first banks) have the following values for N_{th}: 139K, 155K, and 284K.

[12]The charge-pump "up-converts" the DRAM chip's supply voltage into an even higher voltage to ensure that the wordline's access-transistors are completely switched on. A charge-pump is essentially a large reservoir of charge which is slowly refilled after being tapped into.

[13]The actual formula we used is $(RI - 8192 \times t_{RFC})/AI$, where t_{RFC} (refresh cycle time) is the timing constraint between a REF and a subsequent ACT to the same module [34]. Our testing platform sets t_{RFC} to 160*ns*, which is a sufficient amount of time for all of our modules.

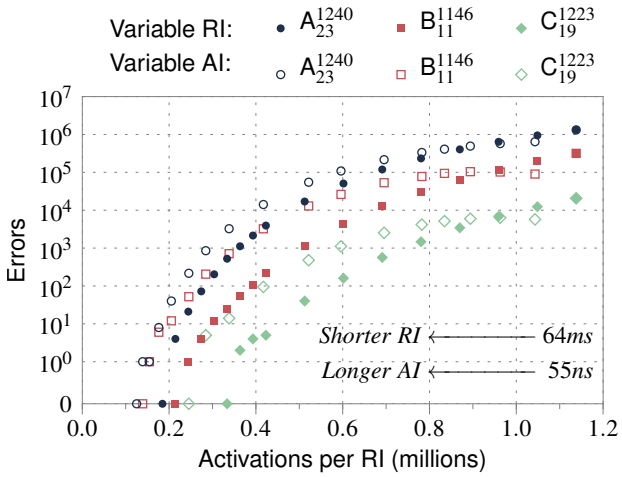


Figure 6. Number of errors vs. number of activations

6.3. Address Correlation: Aggressor & Victim

Most rows in A_{23}, B_{11}, and C_{19} have at least one cell that experienced an error: 100%, 99.96%, and 41.54%. We analyzed the addresses of such victim cells to determine whether they exhibit any spatial locality. We were unable to identify any distinct pattern or skew. By chance, however, some victim cells could still end up being located near each other. For the three modules, Table 5 shows how many 64-bit words in their full address-space (0–2GB) contain 1, 2, 3, or 4 victim cells. While most words have just a single victim, there are also some words with multiple victims. This has an important consequence for error-correction codes (ECC). For example, SECDED (single error-correction, double error-detection) can correct only a single-bit error within a 64-bit word. If a word contains two victims, however, SECDED cannot correct the resulting double-bit error. And for three or more victims, SECDED cannot even detect the multi-bit error, leading to silent data corruption. Therefore, we conclude that SECDED is not failsafe against disturbance errors.

Module	*Number of 64-bit words with X errors*			
	$X=1$	$X=2$	$X=3$	$X=4$
A_{23}	9,709,721	**181,856**	**2,248**	**18**
B_{11}	2,632,280	**13,638**	**47**	0
C_{19}	141,821	**42**	0	0

Table 5. Uncorrectable multi-bit errors (in bold)

Most rows in A_{23}, B_{11}, and C_{19} cause errors when they are repeatedly opened. We refer to such rows as *aggressor rows*. We exposed the aggressor rows in the modules by subjecting them to two runs of TestEach for only the first bank:

1. TestEach(55*ns*, 64*ms*, RowStripe)
2. TestEach(55*ns*, 64*ms*, ~RowStripe)

The three modules had the following numbers of aggressor rows: 32768, 32754, and 15414. Considering that a bank in the modules has 32K rows, we conclude that large fractions of the rows are aggressors: 100%, 99.96%, and 47.04%.

Each aggressor row can be associated with a set of victim cells that were disturbed by the aggressor during either of the two tests. Figure 7 plots the size distribution of this set for the three modules. Aggressor rows in A_{23} are the most potent, disturbing as many as 110 cells at once. (We cannot explain the two peaks in the graph.) On the other hand, aggressors in B_{11} and C_{19} can disturb up to 28 and 5 cells, respectively.

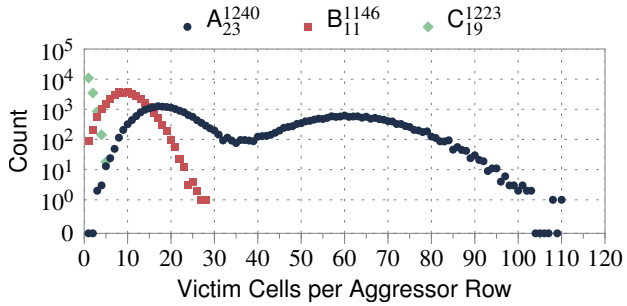


Figure 7. How many cells are affected by an aggressor row?

Similarly, we can associate each aggressor row with a set of *victim rows* to which the victim cells belong. Figure 8 plots the size distribution of this set. We see that the victim cells of an aggressor row are predominantly localized to two rows or less. In fact, only a small fraction of aggressor rows affect three rows or more: 2.53%, 0.0122%, and 0.00649%.

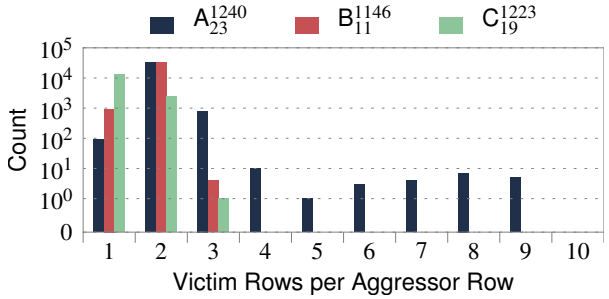


Figure 8. How many rows are affected by an aggressor row?

To see whether any correlation exists between the address of an aggressor row and those of its victim rows, we formed every possible pair between them. For each such pair, we then computed the row-address difference as follows: $VictimRow_{addr} - AggressorRow_{addr}$. The histogram of these differences is shown in Figure 9. It is clear from the figure that an aggressor causes errors in rows only other than itself. This is understandable since every time an aggressor is opened and closed, it also serves to replenish the charge in all of its own cells (Section 2.4). Since the aggressor's cells are continuously being refreshed, it is highly unlikely that they could leak enough charge to lose their data.

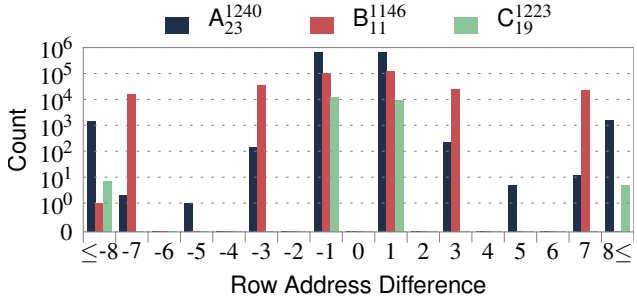


Figure 9. Which rows are affected by an aggressor row?

For all three modules, Figure 9 shows strong peaks at ± 1, suggesting that an aggressor and its victims are likely to have consecutive row-addresses, i.e., they are *logically adjacent*. Being logically adjacent, however, does not always imply that the rows are placed next to each other on the silicon die, i.e., *physically adjacent*. Although every logical row must be mapped to some physical row, it is entirely up to the DRAM manufacturer to decide how they are mapped [65]. In spite of this, we hypothesize that aggressors cause errors in their physically adjacent rows due to three reasons.

- *Reason 1.* Wordline voltage fluctuations are likely to place the greatest electrical stress on the immediately neighboring rows [49, 55].
- *Reason 2.* By definition, a row has only two immediate neighbors, which may explain why disturbance errors are localized mostly to two rows.
- *Reason 3.* Logical adjacency may highly correlate with physical adjacency, which we infer from the strong peaks at ± 1 in Figure 9.

However, we also see discrepancies in Figures 8 and 9, whereby an aggressor row appears to cause errors in non-adjacent rows. We hypothesize that this is due to two reasons.

- *Reason 1.* In Figure 8, some aggressors affect more than just two rows. This may be an irregularity caused by *re-mapped* rows. Referring back to Figure 2 (Section 2.1), the i^{th} "row" of a rank is formed by taking the i^{th} row in each chip and concatenating them. But if the row in one of the chips is faulty, the manufacturer re-maps it to a spare row (e.g., $i \rightarrow j$) [28]. In this case, the i^{th} "row" has *four* immediate neighbors: $i \pm 1^{th}$ rows in seven chips and $j \pm 1^{th}$ rows in the re-mapped chip.
- *Reason 2.* In Figure 9, some aggressors affect rows that are not logically-adjacent: e.g., side peaks at ± 3 and ± 7. This may be an artifact of manufacturer-dependent mapping, where some physically-adjacent rows have logical row-addresses that differ by ± 3 or ± 7 — for example, when the addresses are gray-encoded [65]. Alternatively, it could be that aggressors affect rows farther away than the immediate neighbors — a possibility that we cannot completely rule out. However, if that were the case, then it would be unlikely for the peaks to be separated by gaps at ± 2, ± 4, and ± 6.[14]

Double Aggressor Rows. Most victim cells are disturbed by only a *single* aggressor row. However, there are some victim cells that are disturbed by *two different* aggressor rows. In the first bank of the three modules, the numbers of such victim cells were 83, 2, and 0. In module A_{23}, for example, the victim cell at (row 1464, column 50466) had a '1'→'0' error when *either* row 1463 *or* row 1465 was toggled. In module B_{11}, the victim cell at (row 5907, column 32087) had a '0'→'1' error when row 5906 was toggled, whereas it had a '1'→'0' error when row 5908 was toggled. Within these two modules respectively, the same trend applies to the other victim cells with two aggressor rows. Interestingly, the two victim cells in module B_{11} with two aggressor rows were also the same cells that had errors for both runs of the test pair described in Section 6.1. These cells were the only cases in which we observed both '0'→'1' *and* '1'→'0' errors in the same cell. Except for such rare exceptions found only in B modules, every other victim cell had an error in just a single preferred direction, for reasons we next explain.

6.4. Data Pattern Dependence

Until now, we have treated all errors equally without making any distinction between the two different *directions* of errors: '0'⇆'1'. When we categorized the errors in Table 3 based on their direction, an interesting trend emerged. Whereas A modules did not favor one direction over the other, B and C modules heavily favored '1'→'0' errors. Averaged on a module-by-module basis, the relative fraction of '1'→'0' errors is 49.9%, 92.8%, and 97.1% for A, B, and C.[15]

The seemingly asymmetric nature of disturbance errors is related to an intrinsic property of DRAM cells called *orientation*. Depending on the implementation, some cells represent a logical value of '1' using the charged state, while other cells do so using the discharged state — these cells are referred to as *true-cells* and *anti-cells*, respectively [44]. If a true-cell loses charge, it experiences a '1'→'0' error. When we profiled two modules (B_{11} and C_{19}), we discovered that they consist mostly of true-cells by a ratio of 1000s-to-1.[16] For these two modules, the dominance of true-cells and their '1'→'0' errors imply that victim cells are most likely to *lose* charge when they are disturbed. The same conclusion also applies to A_{23}, whose address-space is divided into large swaths of true- and anti-cells that alternate every 512 rows. For this module, we found that '1'→'0' errors are dominant (>99.8%) in rows where true-cells are dominant: rows 0–511, 1024–1535, 2048–2559, $\cdots$. In contrast, '0'→'1' errors are dominant (>99.7%) in the remainder of the rows where anti-cells are dominant. Regardless of its orientation, a cell can lose charge only if it was initially charged — explaining why a given cell did not have errors in both runs of the test in Section 6.1. Since the two runs populate the module with inverse data patterns, a cell cannot be charged for both runs.

Table 6 reports the numbers of errors that were induced in three modules using four different data patterns and their inverses: Solid, RowStripe, ColStripe, and Checkered. Among them, RowStripe (even/odd rows '0's/'1's) induces the most errors for A_{23} and B_{11}, as well as the second most errors for C_{19}. In contrast, Solid (all '0's) has the fewest errors for all three modules by an order of magnitude or more. Such a large difference *cannot* be explained if the requirements for a disturbance error are only two-fold: *(i)* a victim cell is in the charged state, and *(ii)* its aggressor row is toggled. This is because the same two requirements are satisfied by all four pairs of data patterns. Instead, there must be other factors at play than just the coupling of a victim cell with an aggressor wordline. In fact, we discovered that the behavior of most

[14] Figure 9 presents further indications of re-mapping, where some modules have non-zero values for ± 8 or beyond. Such large differences — which in some cases reach into the thousands — may be caused when a faulty row is re-mapped to a spare row that is far away, which is typically the case [28].

[15] For manufacturer C, we excluded modules with a die version of $\mathcal{B}$. Unlike other modules from the same manufacturer, these modules had errors that were evenly split between the two directions.

[16] At 70°C, we wrote all '0's to the module, disabled refreshes for six hours and read out the module. We then repeated the procedure with all '1's. A cell was deemed to be true (or anti) if its outcome was '0' (or '1') for both experiments. We could not resolve the orientation of every cell.

victim cells is correlated with the data stored in some other cells.[17] A victim cell may have *aggressor cell(s)* — typically residing in the aggressor row — that must be discharged for the victim to have an error. A victim cell may also have *protector cell(s)* — typically residing in either the aggressor row or the victim row — that must be charged or discharged for the victim to have a lower probability of having an error. In its generalized form, disturbance errors appear to be a complicated "N-body" phenomenon involving the interaction of multiple cells, the net result of which would only explain the differences in Table 6.

Module	TestBulk(*DP*) + TestBulk(∼*DP*)			
	Solid	RowStripe	ColStripe	Checkered
A_{23}	112,123	**1,318,603**	763,763	934,536
B_{11}	12,050	**320,095**	9,610	302,306
C_{19}	57	20,770	130	**29,283**

Table 6. Number of errors for different data patterns

7. Sensitivity Results

Errors are Mostly Repeatable. We subjected three modules to ten *iterations* of testing, where each iteration consists of the test pair described in Section 6.1. Across the ten iterations, the average numbers of errors (for only the first bank) were the following: 1.31M, 339K, and 21.0K. There were no iterations that deviated by more than ±0.25% from the average for all three modules. The ten iterations revealed the following numbers of unique victim cells: 1.48M, 392K, and 24.4K. Most victim cells were repeat offenders, meaning that they had an error in every iteration: 78.3%, 74.4%, and 73.2%. However, some victim cells had an error in just a single iteration: 3.14%, 4.86%, and 4.76%. This implies that an exhaustive search for every possible victim cell would require a large number of iterations, necessitating several days (or more) of continuous testing. One possible way to reduce the testing time is to increase the RI beyond the standardized value of 64*ms* as we did in Figure 4 (Section 6.2). However, multiple iterations could still be required since a single iteration at RI=128*ms* does not provide 100% coverage of all the victim cells at RI=64*ms*: 99.77%, 99.87%, and 99.90%.

Victim Cells ≠ Weak Cells. Although the retention time of every DRAM cell is required to be greater than the 64*ms* minimum, different cells have different retention times. In this context, the cells with the shortest retention times are referred to as *weak cells* [45]. Intuitively, it would appear that the weak cells are especially vulnerable to disturbance errors since they are already leakier than others. On the contrary, we did not find any strong correlation between weak cells and victim cells. We searched for a module's weak cells by neither accessing nor refreshing a module for a generous amount of time (10 *seconds*) after having populated it with either all '0's or all '1's. If a cell was corrupted during this procedure, we considered it to be a weak cell [45]. In total, we were able to identify ∼1M weak cells for each module (984K, 993K, and 1.22M), which is on par with the number of victim cells. However, only a few weak cells were also victim cells: 700, 220, and 19. Therefore, we conclude that the coupling pathway responsible for disturbance errors may be independent of the process variation responsible for weak cells.

Not Strongly Affected by Temperature. When temperature increases by 10°C, the retention time for each cell is known to decrease by almost a factor of two [39, 45]. To see whether this would drastically increase the number of errors, we ran a single iteration of the test pair for the three modules at 70±2.0°C, which is 20°C higher than our default ambient temperature. Compared to an iteration at 50°C, the number of errors did not change greatly: +10.2%, −0.553%, and +1.32%. We also ran a single iteration of the test pair for the three modules at 30±2.0°C with similar results: −14.5%, +2.71%, and −5.11%. From this we conclude that disturbance errors are not strongly influenced by temperature.

8. Solutions to Disturbance Errors

We examine seven solutions to tolerate, prevent, or mitigate disturbance errors. Each solution makes a different trade-off between feasibility, cost, performance, power, and reliability. Among them, we believe our seventh and last solution, called *PARA*, to be the most efficient and low-overhead. Section 8.1 discusses each of the first six solutions. Section 8.2 analyzes our seventh solution (PARA) in detail.

8.1. Six Potential Solutions

1. Make better chips. Manufacturers could fix the problem at the chip-level by improving circuit design. However, the problem could resurface when the process technology is upgraded. In addition, this may get worse in the future as cells become smaller and more vulnerable.

2. Correct errors. Server-grade systems employ ECC modules with extra DRAM chips, incurring a 12.5% capacity overhead. However, even such modules cannot correct multi-bit disturbance errors (Section 6.3). Due to their high cost, ECC modules are rarely used in consumer-grade systems.

3. Refresh all rows frequently. Disturbance errors can be eliminated for sufficiently short refresh intervals ($RI \leq RI_{th}$) as we saw in Section 6.2. However, frequent refreshes also degrade performance and energy-efficiency. Today's modules already spend 1.4–4.5% of their time just performing refreshes [34]. This number would increase to 11.0–35.0% if the refresh interval is shortened to 8.2*ms*, which is required by A_{20} (Table 3). Such a high overhead is unlikely to be acceptable for many systems.

4. Retire cells (manufacturer). Before DRAM chips are sold, the manufacturer could identify victim cells and re-map them to spare cells [28]. However, an exhaustive search for all victim cells could take several days or more (Section 7). In addition, if there are many victim cells, there may not be enough spare cells for all of them.

5. Retire cells (end-user). The end-users themselves could test the modules and employ system-level techniques for handling DRAM reliability problems: disable faulty addresses [2, 27, 62, 67], re-map faulty addresses to reserved addresses [52, 53], or refresh faulty addresses more frequently [44, 67]. However, the first/second approaches are ineffective when every row in the module is a victim row (Section 6.3). On the other hand, the third approach is inefficient since it always refreshes the victim rows more frequently —

[17] We comprehensively tested the first 32 rows in module A_{19} using hundreds of different random data patterns. Through statistical analysis on the experimental results, we were able to identify *almost certain* correlations between a victim cell and the data stored in some other cells.

even when the module is not being accessed at all. In all three approaches, the end-user pays for the cost of identifying and storing the addresses of the aggressor/victim rows.

6. Identify "hot" rows and refresh neighbors. Perhaps the most intuitive solution is to identify frequently opened rows and refresh only their neighbors. The challenge lies in minimizing the hardware cost to identify the "hot" rows. For example, having a counter for each row would be too expensive when there are millions of rows in a system.[18] The generalized problem of identifying frequent items (from a stream of items) has been extensively studied in other domains. We applied a well-known method [37] and found that while it reduces the number of counters, it also requires expensive operations to query the counters (e.g., highly-associative search). We also analyzed approximate methods which further reduce the storage requirement: Bloom Filters [11], Morris Counters [50], and variants thereof [18, 21, 66]. These approaches, however, rely heavily on hash functions and, therefore, introduce hash collisions. Whenever *one* counter exceeds the threshold value, *many* rows are falsely flagged as being "hot," leading to a torrent of refreshes to all of their neighbors.

8.2. Seventh Solution: PARA

Our main proposal to prevent DRAM disturbance errors is a low-overhead mechanism called *PARA* (*probabilistic adjacent row activation*). The key idea of PARA is simple: every time a row is opened and closed, one of its adjacent rows is also opened (i.e., refreshed) with some low probability. If one particular row happens to be opened and closed repeatedly, then it is statistically certain that the row's adjacent rows will eventually be opened as well. The main advantage of PARA is that it is *stateless*. PARA does not require expensive hardware data-structures to count the number of times that rows have been opened or to store the addresses of the aggressor/victim rows.

Implementation. PARA is implemented in the memory controller as follows. Whenever a row is closed, the controller flips a biased coin with a probability p of turning up heads, where $p \ll 1$. If the coin turns up heads, the controller opens one of its adjacent rows where either of the two adjacent rows are chosen with equal probability ($p/2$). Due to its probabilistic nature, PARA does *not* guarantee that the adjacent will always be refreshed in time. Hence, PARA *cannot* prevent disturbance errors with absolute certainty. However, its parameter p can be set so that disturbance errors occur at an extremely low probability — many orders of magnitude lower than the failure rates of other system components (e.g., more than 1% of hard-disk drives fail every year [54, 59]).

Error Rate. We analyze PARA's error probability by considering an adversarial access pattern that opens and closes a row just enough times (N_{th}) during a refresh interval but no more. Every time the row is closed, PARA flips a coin and refreshes a given adjacent row with probability $p/2$. Since the coin-flips are independent events, the number of refreshes to one particular adjacent row can be modeled as a random variable X that is binomially-distributed with parameters B(N_{th}, $p/2$). An error occurs in the adjacent row only if it is never refreshed during any of the N_{th} coin-flips (i.e., X=0). Such an event has the following probability of occurring: $(1-p/2)^{N_{th}}$. When p=0.001, we evaluate this probability in Table 7 for different values of N_{th}. The table shows two error probabilities: one in which the adversarial access pattern is sustained for 64*ms* and the other for one *year*. Recall from Section 6.2 that realistic values for N_{th} in our modules are in the range of 139K–284K. For p=0.001 and N_{th}=100K, the probability of experiencing an error in one year is negligible at 9.4×10^{-14}.

Duration	N_{th}=50K	N_{th}=100K	N_{th}=200K
64*ms*	1.4×10^{-11}	1.9×10^{-22}	3.6×10^{-44}
1 *year*	6.8×10^{-3}	9.4×10^{-14}	1.8×10^{-35}

Table 7. Error probabilities for PARA when p=0.001

Adjacency Information. For PARA to work, the memory controller must know which rows are physically adjacent to each other. This is also true for alternative solutions based on "hot" row detection (Section 8.1). Without this information, rows cannot be selectively refreshed, and the only safe resort is to blindly refresh *all* rows in the same bank, incurring a large performance penalty. To enable low-overhead solutions, we argue for the manufacturers to disclose how they map logical rows onto physical rows.[19] Such a *mapping function* could possibly be as simple as specifying the bit-offset within the logical row-address that is used as the least-significant-bit of the physical row-address. Along with other metadata about the module (e.g., capacity, and bus frequency), the mapping function could be stored in a small ROM (called the *SPD*) that exists on every DRAM module [33]. The manufacturers should also disclose how they re-map faulty physical rows (Section 6.3). When a faulty physical row is re-mapped, the logical row that had mapped to it acquires a new set of physical neighbors. The SPD could also store the *re-mapping function*, which specifies how the logical row-addresses of those new physical neighbors can be computed. To account for the possibility of re-mapping, PARA can be configured to *(i)* have a higher value of p and *(ii)* choose a row to refresh from a wider pool of candidates, which includes the re-mapped neighbors in addition to the original neighbors.

Performance Overhead. Using a cycle-accurate DRAM simulator, we evaluate PARA's performance impact on 29 single-threaded workloads from SPEC CPU2006, TPC, and memory-intensive microbenchmarks (We assume a reasonable system setup [41] with a 4GHz out-of-order core and dual-channel DDR3-1600.) Due to re-mapping, we conservatively assume that a row can have up to *ten* different rows as neighbors, not just two. Correspondingly, we increase the value of p by five-fold to 0.005.[20] Averaged across all 29 benchmarks, there was only a 0.197% degradation in instruction throughput during the simulated duration of 100*ms*. In addition, the largest degradation in instruction throughput for any single benchmark was 0.745%. From this, we conclude

[18] Several patent applications propose to maintain an array of counters ("detection logic") in either the memory controller [7, 8, 24] or in the DRAM chips themselves [6, 9, 23]. If the counters are tagged with the addresses of only the most recently activated rows, their number can be significantly reduced [24].

[19] Bains et al. [6] make the same argument. As an alternative, Bains et al. [7, 8] propose a new DRAM command called "targeted refresh". When the memory controller sends this command along with the target row address, the DRAM chip is responsible for refreshing the row and its neighbors.

[20] We do not make any special considerations for victim cells with two aggressor rows (Section 6.3). Although they could be disturbed by either aggressor row, they could also be refreshed by either aggressor row.

that PARA has a small impact on performance, which we believe is justified by the *(i)* strong reliability guarantee and *(ii)* low design complexity resulting from its stateless nature.

9. Other Related Work

Disturbance errors are a general class of reliability problem that afflicts not only DRAM, but also other memory and storage technologies: SRAM [16, 26, 40], flash [10, 12, 13, 19, 25], and hard-disk [36, 63, 68]. Van de Goor and de Neef [64] present a collection of production tests that can be employed by DRAM manufacturers to screen faulty chips. One such test is the "hammer," where each cell is written a thousand times to verify that it does not disturb nearby cells. In 2013, one test equipment company mentioned the "row hammer" phenomenon in the context of DDR4 DRAM [48], the next generation of commodity DRAM. To our knowledge, no previous work demonstrated and characterized the phenomenon of disturbance errors in DRAM chips from the field.

10. Conclusion

We have demonstrated, characterized, and analyzed the phenomenon of disturbance errors in modern commodity DRAM chips. These errors happen when repeated accesses to a DRAM row corrupts data stored in other rows. Based on our experimental characterization, we conclude that disturbance errors are an emerging problem likely to affect current and future computing systems. We propose several solutions, including a new stateless mechanism that provides a strong statistical guarantee against disturbance errors by probabilistically refreshing rows adjacent to an accessed row. As DRAM process technology scales down to smaller feature sizes, we hope that our findings will enable new system-level [51] approaches to enhance DRAM reliability.

Acknowledgments

We thank the reviewers and SAFARI members for their feedback. We acknowledge the support of IBM, Intel, and Qualcomm. This research was partially supported by ISTC-CC, NSF (CCF 0953246, CCF 1212962, and CNS 1065112), and SRC. Yoongu Kim is supported by an Intel fellowship.

References

[1] Memtest86+ v4.20. `http://www.memtest.org`.
[2] The GNU GRUB Manual. `http://www.gnu.org/software/grub`.
[3] Z. Al-Ars. *DRAM Fault Analaysis and Test Generation*. PhD thesis, TU Delft, 2005.
[4] Z. Al-Ars et al. DRAM-Specific Space of Memory Tests. In *ITC*, 2006.
[5] AMD. BKDG for AMD Family 15h Models 10h-1Fh Processors, 2013.
[6] K. Bains et al. Method, Apparatus and System for Providing a Memory Refresh. US Patent App. 13/625,741, Mar. 27 2014.
[7] K. Bains et al. Row Hammer Refresh Command. US Patent App. 13/539,415, Jan. 2 2014.
[8] K. Bains et al. Row Hammer Refresh Command. US Patent App. 14/068,677, Feb. 27 2014.
[9] K. Bains and J. Halbert. Distributed Row Hammer Tracking. US Patent App. 13/631,781, Apr. 3 2014.
[10] R. Bez et al. Introduction to Flash Memory. *Proc. of the IEEE*, 91(4), 2003.
[11] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7), 1970.
[12] Y. Cai et al. Error Patterns in MLC NAND Flash Memory: Measurement, Characterization, and Analysis. In *DATE*, 2012.
[13] Y. Cai et al. Program Interference in MLC NAND Flash Memory: Characterization, Modeling and Mitigation. In *ICCD*, 2013.
[14] S. Y. Cha. DRAM and Future Commodity Memories. In *VLSI Technology Short Course*, 2011.
[15] M.-T. Chao et al. Fault Models for Embedded-DRAM Macros. In *DAC*, 2009.
[16] Q. Chen et al. Modeling and Testing of SRAM for New Failure Mechanisms Due to Process Variations in Nanoscale CMOS. In *VLSI Test Symposium*, 2005.
[17] P.-F. Chia et al. New DRAM HCI Qualification Method Emphasizing on Repeated Memory Access. In *Integrated Reliability Workshop*, 2010.
[18] S. Cohen and Y. Matias. Spectral Bloom Filters. In *SIGMOD*, 2003.
[19] J. Cooke. The Inconvenient Truths of NAND Flash Memory. In *Flash Memory Summit*, 2007.
[20] DRAMeXchange. TrendForce: 3Q13 Global DRAM Revenue Rises by 9%, Samsung Shows Most Noticeable Growth, Nov. 12, 2013.
[21] L. Fan et al. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *Transactions on Networking*, 8(3), 2000.
[22] J. A. Fifield and H. L. Kalter. Crosstalk-Shielded-Bit-Line DRAM. US Patent 5,010,524, Apr. 23, 1991.
[23] Z. Greenfield et al. Method, Apparatus and System for Determining a Count of Accesses to a Row of Memory. US Patent App. 13/626,479, Mar. 27 2014.
[24] Z. Greenfield et al. Row Hammer Condition Monitoring. US Patent App. 13/539,417, Jan. 2, 2014.
[25] L. M. Grupp et al. Characterizing Flash Memory: Anomalies, Observations, and Applications. In *MICRO*, 2009.
[26] Z. Guo et al. Large-Scale SRAM Variability Characterization in 45 nm CMOS. *JSSC*, 44(11), 2009.
[27] D. Henderson and J. Mitchell. *IBM POWER7 System RAS*, Dec. 2012.
[28] M. Horiguchi and K. Itoh. *Nanoscale Memory Repair*. Springer, 2011.
[29] R.-F. Huang et al. Alternate Hammering Test for Application-Specific DRAMs and an Industrial Case Study. In *DAC*, 2012.
[30] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual, 2012.
[31] Intel. 4th Generation Intel Core Processor Family Desktop Datasheet, 2013.
[32] K. Itoh. Semiconductor Memory. US Patent 4,044,340, Apr. 23, 1977.
[33] JEDEC. *Standard No. 21C. Annex K: Serial Presence Detect (SPD) for DDR3 SDRAM Modules*, Aug. 2012.
[34] JEDEC. *Standard No. 79-3F. DDR3 SDRAM Specification*, July 2012.
[35] M. K. Jeong et al. Balancing DRAM Locality and Parallelism in Shared Memory CMP Systems. In *HPCA*, 2012.
[36] W. Jiang et al. Cross-Track Noise Profile Measurement for Adjacent-Track Interference Study and Write-Current Optimization in Perpendicular Recording. *Journal of Applied Physics*, 93(10), 2003.
[37] R. M. Karp et al. A Simple Algorithm for Finding Frequent Elements in Streams and Bags. *Transactions on Database Systems*, 28(1), 2003.
[38] B. Keeth et al. *DRAM Circuit Design. Fundamental and High-Speed Topics*. Wiley-IEEE Press, 2007.
[39] S. Khan et al. The Efficacy of Error Mitigation Techniques for DRAM Retention Failures: A Comparative Experimental Study. In *SIGMETRICS*, 2014.
[40] D. Kim et al. Variation-Aware Static and Dynamic Writability Analysis for Voltage-Scaled Bit-Interleaved 8-T SRAMs. In *ISLPED*, 2011.
[41] Y. Kim et al. A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM. In *ISCA*, 2012.
[42] Y. Konishi et al. Analysis of Coupling Noise between Adjacent Bit Lines in Megabit DRAMs. *JSSC*, 24(1), 1989.
[43] D. Lee et al. Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture. In *HPCA*, 2013.
[44] J. Liu et al. RAIDR: Retention-Aware Intelligent DRAM Refresh. In *ISCA*, 2012.
[45] J. Liu et al. An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms. In *ISCA*, 2013.
[46] L. Liu et al. A Software Memory Partition Approach for Eliminating Bank-level Interference in Multicore Systems. In *PACT*, 2012.
[47] J. A. Mandelman et al. Challenges and Future Directions for the Scaling of Dynamic Random-Access Memory (DRAM). *IBM Journal of R&D*, 46(2.3), 2002.
[48] M. Micheletti. Tuning DDR4 for Power and Performance. In *MemCon*, 2013.
[49] D.-S. Min et al. Wordline Coupling Noise Reduction Techniques for Scaled DRAMs. In *Symposium on VLSI Circuits*, 1990.
[50] R. Morris. Counting Large Numbers of Events in Small Registers. *Communications of the ACM*, 21(10), 1978.
[51] O. Mutlu. Memory Scaling: A Systems Architecture Perspective. In *MemCon*, 2013.
[52] P. J. Nair et al. ArchShield: Architectural Framework for Assisting DRAM Scaling by Tolerating High Error Rates. In *ISCA*, 2013.
[53] C. Nibby et al. Remap Method and Apparatus for a Memory System Which Uses Partially Good Memory Devices. US Patent 4,527,251, July 2 1985.
[54] E. Pinheiro et al. Failure Trends in a Large Disk Drive Population. In *FAST*, 2007.
[55] M. Redeker et al. An Investigation into Crosstalk Noise in DRAM Structures. In *MTDT*, 2002.
[56] K. Roy et al. Leakage Current Mechanisms and Leakage Reduction Techniques in Deep-Submicrometer CMOS Circuits. *Proc. of the IEEE*, 91(2), 2003.
[57] K. Saino et al. Impact of Gate-Induced Drain Leakage Current on the Tail Distribution of DRAM Data Retention Time. In *IEDM*, 2000.
[58] J. H. Saltzer and M. F. Kaashoek. *Principles of Computer Design: An Introduction*. Chapter 8, p. 58. Morgan Kaufmann, 2009.
[59] B. Schroeder and G. A. Gibson. Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You? In *FAST*, 2007.
[60] N. Suzuki et al. Coordinated Bank and Cache Coloring for Temporal Protection of Memory Accesses. In *ICESS*, 2013.
[61] A. Tanabe et al. A 30-ns 64-Mb DRAM with Built-In Self-Test and Self-Repair Function. *JSSC*, 27(11), 1992.
[62] D. Tang et al. Assessment of the Effect of Memory Page Retirement on System RAS Against Hardware Faults. In *DSN*, 2006.
[63] Y. Tang et al. Understanding Adjacent Track Erasure in Discrete Track Media. *Transactions on Magnetics*, 44(12), 2008.
[64] A. J. van de Goor and J. de Neef. Industrial Evaluation of DRAM Tests. In *DATE*, 1999.
[65] A. J. van de Goor and I. Schanstra. Address and Data Scrambling: Causes and Impact on Memory Tests. In *DELTA*, 2002.
[66] B. Van Durme and A. Lall. Probabilistic Counting with Randomized Storage. In *IJCAI*, 2009.
[67] R. Venkatesan et al. Retention-Aware Placement in DRAM (RAPID): Software Methods for Quasi-Non-Volatile DRAM. In *HPCA*, 2006.
[68] R. Wood et al. The Feasibility of Magnetic Recording at 10 Terabits Per Square Inch on Conventional Media. *Transactions on Magnetics*, 45(2), 2009.
[69] Xilinx. *Virtex-6 FPGA Integrated Block for PCI Express*, Mar. 2011.
[70] Xilinx. *ML605 Hardware User Guide*, Oct. 2012.
[71] Xilinx. *Virtex-6 FPGA Memory Interface Solutions*, Mar. 2013.
[72] J. H. Yoon et al. Flash & DRAM Si Scaling Challenges, Emerging Non-Volatile Memory Technology Enablement. In *Flash Memory Summit*, 2013.
[73] T. Yoshihara et al. A Twisted Bit Line Technique for Multi-Mb DRAMs. In *ISSCC*, 1988.