Embedded systems security

Hardware architecture/microarchitecture attacks

Prof. Stefano Di Carlo

RowHammer

The Silent Threat to Computer Security

Lecture credits:

Nicolas Abril

Massimiliano Di Todaro

Matteo Isoldi

Diego Porto

Co-funded by the European Union

La Région Auvergne-Rhône-Alpes

Agence Erasmus+ France · Education Formation

Erasmus+

Acknoledgments

This material was initially developed as part of an assignment for the Operating Systems for embedded systems course delivered at Politecnico di Torino by Prof. Stefano Di Carlo during the academic year 2022/20023.

Credits for the preparation of this material go to:

- Nicolas Abril (`https://github.com/developedby`)
- Massimiliano Di Todaro (`https://github.com/b0hYee`)
- Matteo Isoldi (`https://github.com/mditodaro`)
- Diego Porto (`https://github.com/akhre`)

Contents

1 Overview of the problem

This brief presentation discusses the RowHammer attack, one of the most significant hardware failure that can cause a widespread system security vulnerability. RowHammer is a phenomenon observed when repeatedly accessing a row in a modern DRAM chip at a sufficient rate 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 the effect of circuit-level cell-to-cell interference in a scaled memory technology. There are several key moments in the history of RowHammer in terms of research: Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors (Kim et al. 2014) Exploiting the DRAM rowhammer bug to gain kernel privileges (Seaborn and Dullien from Project Zero, 2015) Drammer: Deterministic Rowhammer Attacks on Mobile Platforms (van der Veen et al. 2016)

2 DRAM

To understand how RowHammer works, a bit of background regarding DRAMs is needed.

Our systems usually present printed circuit boards that act as the main memory for the CPU. These circuit boards are called DIMM (Dual Inline Memory Module) or simply RAM sticks, and they function as volatile memory for instructions and data. A DIMM contains a certain number (usually eight) of microchips, the memory modules. Each chip is then organized into banks: typically, a single chip will have four, eight, or sixteen banks. The memory controller uses log2(number of banks) bits of the address specified for a read/write operation on the memory to access a specific bank. For example, if a chip has eight banks, the memory controller needs to use three bits of the address to select which bank to use for a read or write operation. A bank is further subdivided into arrays of bit cells; each matrix has the following:

- Vertical bit lines with a parasitic capacitance Cl;
- Horizontal word lines to select the correct word;
- A pre-charging circuit: connects the bit lines to a voltage reference of Vdd/2;

- A sense amplifier: a differential amplifier used to detect small voltage differences to read the content of the bit cells;
- Input and output circuits: are switches driven by the column decoder and mux.

The decoder of each array is fed with the same row address so that when a row address is chosen as part of a read or write cycle, the same word line is selected in each array. Furthermore, the multiplexer and demultiplexer of each array are fed with the same column address so that when a column address is chosen, the same column is selected in each array. In other words, when a memory address is applied to a group of arrays, only one cell is selected in each array but at the same position. This means a different binary digit can be read from or written to the same position in each array. In this way, we have a memory device that can read or write a specific number of bits in one go with one address. We need precisely eight arrays to read or write a byte of data.

Each cell of an array stores one single bit, and it's implemented using a transistor and a capacitor (1T1C structure). This is the currently most used structure for memory cells because it is more suitable for densely packed memory chips as it occupies a tiny area. The transistor enables (or disables) the connection to the capacitor CS. In contrast, the capacitor itself is obtained by exploiting the parasitic capacitance of a trench transistor (which is a special transistor with a very large gate area), and it is used to keep a single bit of data in the bit-cell of the DRAM. The transistor is connected to the word line through the gate and the bit line through the source.

In the case of a reading, when the correct word line is selected and activated, there is a charge sharing between the parasitic capacitance Cl of the line and the capacitance Cs of the bit cell. The sense amplifier can detect the small change (both positive and negative) in the line's voltage and identify the content stored inside the cell. It is important to note that the information is lost after the reading, so it is necessary to write the data back both in the bit cells of the selected word and also in the other bit cells in the same row. Unfortunately, standard read events can't be relied on to hit every used DRAM row, so a specific refresh operation is needed.

To write data, the desired values must be applied to the bit lines to charge (or discharge) the storage capacitor in the selected bit cell.

There is another problem because storage capacitors leak charge over time due to disturbances, so each memory cell must be regularly refreshed to allow DRAM to store data for longer periods. The refresh process reads the data into the sense amplifiers and writes them straight back into the cells. All this happens automatically, so the memory controller times refresh cycles to occur without the CPU being aware of it and while the RAM module does other things that prevent it from reading or writing memory (such as transmitting read data).

To consider even the worst-case scenarios, DRAM refresh times are low. This choice ensures that no data is ever lost, but it also affects power usage and performance because the memory can't be accessed during the refresh operation.

It is important to know that:

- Performance can be limited by how often the DRAM cells need refreshing;
- Each memory cell is refreshed, generally, every 64 milliseconds. This prevents them from refreshing all at once, causing a bottleneck. So each row of cells is refreshed independently. The time needed to refresh a single cell is small, about 75-120ns;

3 Introducing RowHammer: flipping bit problem description

In the past few years, technological development has been directed toward minimizing the size occupied by the bit cells inside DRAM so that more could be placed on the same silicon surface to reduce the cost-per-bit of memory. RowHammer is an important interference phenomenon that happens at the circuit level and that is related to technology scaling. This is because increasing the cell density also harms memory reliability:

- A small cell can hold only a limited amount of charge, which makes it more vulnerable to data loss because it reduces the amount of noise that the bit cell can withstand without compromising the information stored in the capacitor;
- The proximity of cells introduces electromagnetic coupling effects between them, causing them to interact with each other in undesirable ways;

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 that can lead to data loss.

In particular, many activations of the same row force the word line to toggle repeatedly, causing voltage fluctuations that disturb nearby rows, inducing some of their cells to leak charge at an accelerated rate. In contrast, others recover this available charge, making them charged. If a cell loses or acquires too much charge before it is restored to its original value (through a refresh operation), it experiences a disturbance error. This interference can result in bit flips in DRAM regions that are physically nearby to DRAM rows that are rapidly accessed (these rows are called hammered rows). So, with the rowhammer vulnerabilities, the attacker, who has access to a DRAM address X, could modify data in a different location Y that is physically near to X in the DRAM.

4 Difficulties in performing RowHammer

To make a RowHammer attack work properly, the system must access the DRAM with high frequency. For this reason, a cache memory could interfere significantly with the flow of the attack by reducing DRAM accesses.

Two different strategies can be adopted to do uncached accesses and reach the DRAM effectively:

1. Remove data from the cache;
2. Don't put data in the cache;

By doing this, subsequent accesses will be performed on DRAM. To avoid using cache and addressing the DRAM directly, different techniques are used:

1. Flushing the cache: based on native x86 clflush instruction (Kim et al.). Since this is based on a Flush and Reload loop, the attack is performed on the memory and cache. It is an instruction-dependent technique.

   The clflush instruction can be used at all privilege levels and is subject to all permission checking and faults associated with a byte load.

We followed this approach by using these three different Arm-V8 instructions:

(a) DC CVAC: Clean by Virtual Address to Point of Coherency;

(b) DC CIVAC: Clean and Invalidate by Virtual Address to Point of Coherency;

(c) DC ZVA: Cache zero by Virtual Address.

These three instructions are unprivileged by default. Despite that, the presence of a system control register can limit their use by setting it appropriately in kernel space. For this reason, these instructions are not considered good primitives for RowHammer exploitation.

However, it's still reasonable to use them for two aspects:

- Disabling these instructions can limit the use of some applications that needs them, making them not runnable
- On specific platforms, it could be challenging to disable these instructions. Since the introduction of Linux kernel 4.8, an exception handler for cache maintenance has been present. Now, even if the kernel disabled these instructions, the exceptions raised by executing them would trap into the kernel, and the handler would use the corresponding instructions to take care of the desired cache operation.

2. Non-temporal access: using operations that don't follow standard cache-coherency rules. If there is memory access, there is no need to cache the data because the memory controller assumes that that data won't be used anymore.

   Several NT instructions bypass the cache. The problem with this approach is that if a store operation needs to be performed while bypassing the cache, the operation is effective only if the write-combined buffer (WC) was previously filled up, allowing one store operation on the DRAM with a lower rate (this isn't good for RowHammer since it requires a higher rate).

3. Cache eviction: evict data from the cache by doing multiple loads that go to the same set of the desired evicted data.

After bypassing the cache using the previously mentioned technique, there's still a problem: a fixed map between physical addresses and the DRAM layout. Since such information is not available to the public, the only possible strategy is to use reverse engineering to get to know it.

Given how the DRAM is organized and how the row buffer, which is in charge of storing the most recently used row in a bank, works, it is possible to understand how to find two rows belonging to the same bank. Specific algorithms can be used to identify these rows: they use the fact that to access the attacker rows, higher latency is expected due to a row buffer conflict, which can't be experienced when accessing rows of different banks.

The mapping is relatively more straightforward in ARM-based processors (like the one in the Raspberry Pi 3) than on x86. At the end of this reverse engineering process, we aim to determine the lowest row bit in the address to correctly identify different rows of the same bank (neighboring rows that can be used as aggressors).

Modern processors use a Virtual Memory system where the addresses used by programs (virtual addresses) are translated into physical addresses used by the memory system. This translation is performed by a part of the processor called a Memory Management Unit (MMU). In Arm-V8 architecture, the MMU uses translation tables stored in memory to translate virtual addresses to physical addresses. The MMU will automatically read the translation tables when necessary. Privileged software, such as an operating system, programs the MMU to translate between these two address spaces. In UNIX-like operating systems, it is possible to manage the memory through paging. From the OS point of view, a virtual page is mapped to a physical page in memory, which is usually less than or at least equal in size to a row. The kernel determines the mappings, so the ultimate authority on physical addresses is the kernel, even though it always operates with the help of the MMU. User-space processes are usually unaware of these mapping procedures since they deal only with logical addresses. However, a program with sufficient privileges can retrieve its physical map by reading a specific file called pagemap, stored in the /proc directory. In the pagemap of a particular process, we can find which physical frame each virtual page is mapped to. It contains one 64-bit value for each virtual page, which includes the Page Frame Number (PFN).

The PFN can be easily computed from the physical address by dividing it by the page size (or by shifting the physical address with PAGESHIFT bits to the right). Therefore, the address can be obtained by making the opposite operation on the PFN.

The interesting aspect is that, due to the RowHammer vulnerability, since Linux 4.0, only users with admin privileges can get PFNs. With 4.0 and 4.1, unprivileged accesses fail with a permission error. Starting from Linux 4.2, the PFN is zeroed if the user has no admin capabilities.

5 Techniques exploiting RowHammer

Rowhammer can be triggered by many different techniques, depending on the attack vector, the targeted architecture, and the desired outcome. Lou et al. propose a framework for how rowhammer attacks work in general and how to classify them according to the techniques used and the result of the attack.

In general, all Rowhammer-based attacks work by following these 3 or 4 steps:

1. Location Preparation: find where there are vulnerable bits and which rows can be used to attack them. Then, it needs to place the attacked data about the identified vulnerable rows in specific places.

2. Rapid Hammering: with all the data structures being put in the required places, the attack hammers the attacker rows to corrupt the desired memory bits. It must bypass the cache to trigger a memory read/write and use a specific access pattern to cause bit flips while evading the system's protections efficiently. The most common pattern is double-sided rowhammer, where the two rows adjacent to the victim are accessed alternately. Still, there are other patterns like single-sided, multi-sided, non-uniform access, half-double, and many others.

3. Exploit verification: after hammering the memory, the attack needs to judge if the outcome was achieved. This can be done by reading the changed memory or indirectly by seeing if some system behavior changed, like a different software behavior or differences in timing.

4. Store Injected Errors: step only for attacks that need multiple-bit change. They need to flip one of the bits, propagate the maliciously modified structure back, place it in a new vulnerable memory position, and repeat the process.

The framework also classifies Rowhammer attacks based on these criteria:

- Attack Origin: if the source of the attack is remote, like malicious JS script code sent by a server or network packets that trigger rowhammer, or local, as any other regular program. Local programs can be either privileged or unprivileged. Although privileged processes can already access anything in their system, it's possible to use rowhammer to access other systems in the same machine, like in a multi-VM hypervisor setting. The origin could also include other non-conventional sources, like peripherals that can use DMA or FPGAs that share the memory bus with a CPU.

- Intended Application: what kind of object does the attack try to corrupt? Whether the target is privileged or not, and whether the targeted memory is readable. Different techniques may be required depending on the permissions we have in relation to the target. Also, attacks on bits of higher priority usually lead to more disastrous failures.

This doesn't include the application of the attack, which could be many:

- Random Data Corruption: flipping non-specific but important bits until it causes the system to misbehave or malfunction.

- Privilege escalation: an attack could achieve root permission to the desired process, tampering with the memory of another process and even kernel level, unrestricted access to memory.

- Sandbox Escape: a process can indirectly escape the sandbox by changing the memory of a different process or directly giving itself access to the underlying system. This can be applied to browsers, containers, the Android sandbox, and virtual machines.

- Denial of Service: flipping some precise bits can cause a system to crash or lag and cause some programs to get stuck in a loop. Some of the RowHammer mitigations also can slow down memory access to prevent bit flips, essentially changing the hammering from an integrity problem to a denial of service one.

- Secret Key Cracking: by reading bits of restricted memory, private encryption keys can be partially or entirely obtained, enabling further attacks.

- Modifying behavior of other programs: by changing the executed instructions or some data the target program uses, an attacker can cause incorrect and undesired results.

These are some attacks, but many others are restricted mainly by the attacker's creativity. There are dozens of different exploits publicly published. With this classification in mind, we can have a look at some famous existing attacks:

- 2015, Seaborn et al.: Use rowhammer to manipulate Page Tables on Linux, which can lead to the process having unrestricted memory access, introducing double-sided hammering.
- Rowhammer.js: By leveraging large Javascript arrays and a very efficient cache eviction mechanism, this attack manages to escape the browser sandbox in a variety of different host systems, being able to interfere with other browser processes and potentially even compromise the whole system.
- RAMBleed: Is able to read somewhat arbitrary memory by sandwiching a vulnerable page containing a secret that must be disclosed on both sides of the row. It places controlled pages on the same rows of the secret and hammers them, making the secret leak to the vulnerable row.
- Blacksmith: Applies hammering to many rows, not only 1 or 2 adjacent ones, and accesses them with a non-uniform pattern. It includes a fuzzer to find the optimum access pattern across different parameters. This escapes most simple rowhammer mitigations while still being able to trigger bit flips efficiently.
- RAMPage: Exploits Android-specific DMA management APIs that give programs access to uncached memory, managing to provide an app root access to the system.

The same principle used for Rowhammer can also be applied to other types of memory that store data as charges in a capacitor-like circuit. There are existing exploits for NAND Flash, but it could be used for other forms of storage that work with this same underlying mechanism.

6 Mitigation techniques for RowHammer

There are a lot of possible attacks, but there are also many diverse solutions that engineers and researchers proposed through papers, new standards, and new DRAM internal designs.

The academic world focused on devising new ways to tackle the problem based on the different kinds of attacks implemented. They proposed numerous solutions that can be generally classified into four categories:

- Tracking approaches

  These approaches track microarchitectural events such as DRAM row activations and cache misses. Corrective action is performed when these events occur at a high rate (the victim row is refreshed, or the aggressor rows are blocked until the next background refresh). To track these events, CAM (content addressable memory) or SRAM memory must store metadata about row accesses in all DRAM chips in the system. This increases the cost of the memory as a whole. Still, it also decreases the performance since their implementation adds an overhead for accessing, reading, and updating these memories once a row activation is done, increasing memory access latency. The Graphene mitigation represents one example of a tracking approach. Graphene is based on an algorithm generally used for counting frequent item accesses (the Misra-Gries algorithm) to identify aggressor rows. Once an aggressor row has been detected, it sends a command to the DRAM to order the refreshing of its neighboring rows. A problem arises if this additional memory is implemented outside DRAM since their internal structure isn't publicly available: manufacturers need to create a way for memory controllers to request single victim row refreshes with the information gathered. Because, by tracking aggressor rows, memory controllers cannot identify victim rows. BlockHammer, mitigation similar to Graphene, is designed to be implemented without the need for proprietary information about DRAM chips because it freezes access to bank rows directly from the MMU.

- Sampling approaches

  The MMU monitors row accesses, and after every row activation, it can be considered an aggressor row based on a certain probability. In this case, memory controllers request the DRAM to perform a refresh operation on the target victim rows of the aggressor one. These approaches are simpler than tracking methods because they do not require additional memory to store row access information. However, since they are implemented inside memory controllers, the same communication problem with the DRAM remains. They also require the memory controller to use a random number generator module to make the sampling operation more secure. One example of this approach is PRA (Probabilistic Row Activation).

- Partitioning approaches

  With these approaches, the memory locations contained in a bank are divided into small sections (data units) delimited by a perimeter. When an attacker tries to hammer a row, its influence is confined by this memory perimeter (usually, a data unit consists only of a single row). This is generally done by adding additional rows between each perimeter, called guard rows (rows that don't store any data), to ensure that the rows inside a perimeter are sufficiently far from another perimeter and cannot influence each other electrically.

  More complex to implement because of the mismatch in data addressing that can be generated between the CPU and the DRAM.

  The most notable example of this mitigation is the one who defined this approach: GuardION.

- Clean-slate approaches

  These approaches try to reduce electro-magnetical effects between nearby rows. Some act on the row and cell placement inside the memory, others on using electron energy barriers between cells to avoid interferences, etc. They primarily propose different implementations of the internal memory structure to remove possible RowHammer causes.

While the academic world has published these many proposals, DRAM manufacturers, without a standardized way for dealing with the vulnerability, have tried in their way to deal with the problem by deploying their products TRR or Target Row Refresh.

Since TRR is not part of a standard, each manufacturer used different techniques to implement TRR, but at the core, the building blocks are the same and generally:

- a sampler: it tracks which rows are being hammered to identify their target victim rows;

- an inhibitor: it applies the mitigation mechanism to prevent bit flips in the identified victim row.

Although DRAM vendors advertise their products as RowHammer-free, researchers have demonstrated that TRR is insufficient to prevent bit flips and that new attacks can be exploited to corrupt data contained in memory (TRRespass). This is due to the sampler's nature: it can track only a limited number of aggressor rows. So, when the sampler's internal table used for tracking is overflowed, the mitigation can be bypassed effortlessly, even when, due to its black box nature, nobody knows how the table is managed.

JEDEC made another attempt to solve the problem last year. This semiconductor standardization body is responsible for the DDR protocol, with the publication of a revised version of the DDR5 specification (JESD79-5B). With this updated standard version, a new command was introduced for memory controllers to communicate with the memory: the DRFM or Directed Refresh Management. Its working principle is straightforward: it is used to report to the DRAM the identity of an aggressor row (this has been proposed by numerous academic papers before and is at the base of various tracking and sampling solutions). The DRAM, upon receiving this command, will then refresh the victim rows contained inside the blast radius (the physical distance between aggressor and victim rows) of the aggressor row reported.

The specification doesn't cover every aspect of implementing the mechanism behind the DRFM in the DRAM and its usage by memory controllers, so while it allows for flexibility, there can still be problems with using this approach. Even so, this likely represents the first step to solving RowHammer although its effects will be seen in future years.

7 Conclusions

Although a new step was taken with the current standard, its effect will be seen in future years. Even so, RowHammer remains a threat, and its influence rapidly expands toward other types of memories (i.e., NAND Flash). Manufacturers and researchers must continue to explore new mitigation strategies and work to develop more secure and resilient memory architectures to prevent further complications in the future when scaling technology advances will increase cell interferences.

8 Bibliography

- Y. Kim et al. ”Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors.” In ISCA, 2014.
- X. Lou, F. Zhang, Z. L. Chua, Z. Liang, Y. Cheng, and Y. Zhou, ”Understanding Rowhammer Attacks through the Lens of a Unified Reference Framework.” arXiv, 2019. doi: 10.48550/ARXIV.1901.03538.
- A. J. Walker, S. Lee and D. Beery, ”On DRAM Rowhammer and the Physics of Insecurity”, IEEE Transactions on Electron Devices, vol. 68, no. 4, pp. 1400-1410, April 2021, doi: 10.1109/TED.2021.3060362
- D. Gruss, C. Maurice, and S. Mangard, ”Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript.” arXiv, 2015. doi: 10.48550/ARXIV.1507.06955.
- V. van der Veen et al., ”Drammer: Deterministic Rowhammer Attacks on Mobile Platforms”, 2016.
- Seaborn, Mark; Dullien, Thomas (March 9, 2015). ”Exploiting the DRAM rowhammer bug to gain kernel privileges”. googleprojectzero.blogspot.com. Retrieved March 5, 2023.
- A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, “RAMBleed: Reading Bits in Memory Without Accessing Them”, 2020.
- V. van der Veen et al., “GuardION: Practical Mitigation of DMA-Based Rowhammer Attacks on ARM”, in Detection of Intrusions and Malware, and Vulnerability Assessment, 2018, pp. 92–113.
- P. Jattke, V. Van Der Veen, P. Frigo, S. Gunter and K. Razavi, ”BLACKSMITH: Scalable Rowhammering in the Frequency Domain”, 2022 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 2022, pp. 716-734, doi: 10.1109/SP46214.2022.9833772.
- Y. Cai, S. Ghose, Y. Luo, K. Mai, O. Mutlu and E. F. Haratsch, ”Vulnerabilities in MLC NAND Flash Memory Programming: Experimental Analysis, Exploits, and Mitigation Techniques”, 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), Austin, TX, USA, 2017, pp. 49-60, doi: 10.1109/HPCA.2017.61.

- K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, “Flip Feng Shui: Hammering a Needle in the Software Stack”, in Proceedings of the 25th USENIX Conference on Security Symposium, 2016, pp. 1–18.
- X. Chen, M. Merugu, J. Zhang, and S. Ray, “AroMa: Evaluating Deep Learning Systems for Stealthy Integrity Attacks on Multi-Tenant Accelerators”, J. Emerg. Technol. Comput. Syst., Jan. 2023, doi: 10.1145/3579033.
- A. Tatar, R. K. Konoth, E. Athanasopoulos, C. Giuffrida, H. Bos, and K. Razavi, “Throwhammer: Rowhammer Attacks over the Network and Defenses”, in Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference, 2018, pp. 213–225.
- Z. Weissman, T. Tiemann, D. Moghimi, E. Custodio, T. Eisenbarth, and B. Sunar, “JackHammer: Efficient Rowhammer on Heterogeneous FPGA-CPU Platforms”, IACR Transactions on Cryptographic Hardware and Embedded Systems, pp. 169–195, Jun. 2020, doi: 10.46586/tches.v2020.i3.169-195.
- J. S. Kim et al., ”Revisiting RowHammer: An Experimental Analysis of Modern DRAM Devices and Mitigation Techniques”, 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), Valencia, Spain, 2020, pp. 638-651, doi: 10.1109/ISCA45697.2020.00059.
- Y. Xiao et al., ”One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation”, In Proceedings of the 25th USENIX Security Symposium, 2016.
- Z. Zhang et al., ”Triggering Rowhammer Hardware Faults on ARM: A Revisit”, In ASHES ’18: Proceedings of the 2018 Workshop on Attacks and Solutions in Hardware Security, 2018
- Tanj Bennett, Stefan Saroiu, Alec Wolman, and Lucian Cojocar, ”Panopticon: A Complete In-DRAM Rowhammer Mitigation”, In Proceedings of the 1st Workshop on DRAM Security (DRAMSec), 2021
- S. Saroiu, A. Wolman and L. Cojocar, ”The Price of Secrecy: How Hiding Internal DRAM Topologies Hurts Rowhammer Defenses,” 2022 IEEE International Reliability

Physics Symposium (IRPS), Dallas, TX, USA, 2022

- P. Frigo et al., "TRRespass: Exploiting the Many Sides of Target Row Refresh," 2020 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 2020
- D. -H. Kim, P. J. Nair and M. K. Qureshi, "Architectural Support for Mitigating Row Hammering in DRAM Memories," in IEEE Computer Architecture Letters, vol. 14, no. 1, pp. 9-12, 1 Jan.-June 2015
- Onur Mutlu, Ataberk Olgun, and A. Giray Yağlıkcı. 2023. Fundamentally Understanding and Solving RowHammer. In Proceedings of the 28th Asia and South Pacific Design Automation Conference (ASPDAC '23). Association for Computing Machinery, New York, NY, USA, 461–468.

9 Sitography

- *https : //youtu.be/Mhqi70OPW0o*
- *https : //www.technipages.com/what − is − a − refresh − cycle*
- *https : //users.ece.cmu.edu/ yoonguk/papers/kim − isca14.pdf*
- *https : //www.felixcloutier.com/x86/clflush*
- *https : //cmaurice.fr/pdf/ccs16*$_v$*vdveen.pdf*
- *https : //developer.arm.com/documentation/den0024/a/Caches*
- *https : //developer.arm.com/documentation/den0024/a/Caches/Cache − maintenance*
- *https : //www.felixcloutier.com/x86/movnti*
- *https : //www.usenix.org/system/files/conference/usenixsecurity16/sec16*$_p$*aper*$_x$*iao.pdf*
- *https : //developer.arm.com/documentation/101811/0102/Virtual − and − physical − addresses*
- *https : //linux−kernel−labs.github.io/refs/heads/master/lectures/memory−management.html*

- $https://linux-kernel-labs.github.io/refs/heads/master/labs/memory_mapping.html$
- $https://www.kernel.org/doc/Documentation/vm/pagemap.txt$
- $https://stefan.t8k2.com/rh/DRFM/$