TRRespass: Exploiting the Many Sides of Target Row Refresh

Pietro Frigo*† Emanuele Vannacci*† Hasan Hassan§ Victor van der Veen¶
Onur Mutlu§ Cristiano Giuffrida* Herbert Bos* Kaveh Razavi*

*Vrije Universiteit Amsterdam §ETH Zürich ¶Qualcomm Technologies Inc.

†Equal contribution joint first authors

***Abstract*—After a plethora of high-profile RowHammer attacks, CPU and DRAM vendors scrambled to deliver what was meant to be the definitive hardware solution against the RowHammer problem: *Target Row Refresh* (*TRR*). A common belief among practitioners is that, for the latest generation of DDR4 systems that are protected by TRR, RowHammer is no longer an issue in practice. However, in reality, very little is known about TRR. How does TRR exactly prevent RowHammer? Which parts of a system are responsible for operating the TRR mechanism? Does TRR completely solve the RowHammer problem or does it have weaknesses?**

In this paper, we demystify the inner workings of TRR and debunk its security guarantees. We show that what is advertised as a single mitigation mechanism is actually a series of different solutions coalesced under the umbrella term Target Row Refresh. We inspect and disclose, via a deep analysis, different existing TRR solutions and demonstrate that modern implementations operate entirely inside DRAM chips. Despite the difficulties of analyzing in-DRAM mitigations, we describe novel techniques for gaining insights into the operation of these mitigation mechanisms. These insights allow us to build *TRRespass*, a scalable black-box RowHammer fuzzer that we evaluate on 42 recent DDR4 modules.

***TRRespass* shows that even the latest generation DDR4 chips with in-DRAM TRR, immune to all known RowHammer attacks, are often *still vulnerable* to new TRR-aware variants of RowHammer that we develop. In particular, *TRRespass* finds that, on present-day DDR4 modules, RowHammer is still possible when *many* aggressor rows are used (as many as 19 in some cases), with a method we generally refer to as *Many-sided RowHammer*. Overall, our analysis shows that 13 out of the 42 modules from all three major DRAM vendors (i.e., Samsung, Micron, and Hynix) are vulnerable to our TRR-aware RowHammer access patterns, and thus one can still mount existing state-of-the-art system-level RowHammer attacks. In addition to DDR4, we also experiment with LPDDR4(X)[1] chips and show that they are susceptible to RowHammer bit flips too. Our results provide concrete evidence that the pursuit of better RowHammer mitigations must continue.**

I. Introduction

Is RowHammer a solved problem? The leading DRAM vendors have already answered this question with a resounding "yes", advertising the latest generation DDR4 systems as RowHammer-free and using *Target Row Refresh* (*TRR*) as the "silver bullet" that eradicates the vulnerability [63], [70]. Unfortunately, very little is known about the actual implementation or security of TRR on modern systems. Even the major consumers of DRAM in the industry have to simply take the DRAM vendors at their word as the vendors do not disclose the details of the TRR schemes they implement. In this paper, we question this *security by obscurity* strategy and analyze the mechanisms behind TRR to bypass this prevalent mitigation. Our results are worrisome, showing that RowHammer is not only still unsolved, but also that the vulnerability is widespread even in latest off-the-shelf DRAM chips. Moreover, once the RowHammer mitigation mechanism is turned off,[2] we observe bit flips with as few as 45K DRAM row activations, showing that DDR4 and LPDDR4(X) chips are more vulnerable to RowHammer than their DDR3 predecessors, which can tolerate much higher row activation counts (e.g., ∼139K [51]).

RowHammer. Within only five years since its discovery, exploits based on the RowHammer vulnerability [51] have spread to almost every type of computing system [71], [72]. Personal computers [14], [27], [28], [81], [88], cloud servers [23], [33], [54], [77], [79], [89], [96], and mobile phones [25], [91], [92] have all fallen victim to attacks with RowHammer bit flips triggered from native code [12], [23], [27], [33], [77]–[79], [81], [96], JavaScript in the browser [14], [25], [28], [81] and even remote clients across the network [66], [89]. From an academic demonstration, the RowHammer vulnerability has evolved into a major security vulnerability for the entire industry. In response, hardware vendors have scrambled to address the RowHammer issue.

Target Row Refresh. Reliable solutions against RowHammer simply do not exist for older hardware and stopgap solutions such as using ECC and doubling (or even quadrupling) the refresh rate have proven ineffective [7], [23], [51]. In the early days of the DDR4 specification, DRAM vendors announced they would deploy the Target Row Refresh (TRR) mitigation mechanism on newer-generation DDRx systems to eradicate the RowHammer vulnerability [63], [70]. While reports of bit flips on DDR4 devices [27], [56], [66] suggest that the deployment of such mitigation mechanisms may not have been prompt, it is commonly assumed that TRR technology on recent DDR4 systems has put an end to RowHammer attacks [2], [3]. Nowadays, the leading DRAM vendors explicitly advertise RowHammer-free modules [63], [70]. Our initial assessment confirms that none of the *known* RowHammer variants produce bit flips on 42 recent DDR4 modules. However, little is known

[1] We refer to both LPDDR4 and LPDDR4X chips as LPDDR4(X).

[¶] Victor contributed to the research on DDR4 modules.

[2] We turn off the in-DRAM RowHammer mitigation mechanism by disabling `REFRESH` commands, as we explain in Section V.

about TRR beyond what its name suggests, namely that it generates extra refreshes for rows targeted by RowHammer.

The many sides of TRR. In this paper, we take a closer look at the TRR implementations on modern systems. In contrast to what the literature suggests [66], we show that TRR is not a single mitigation mechanism but rather a family of solutions, implemented either in the CPU's memory controller or in the DRAM chips themselves. One of the best-known implementations of TRR-like functionality, Intel's *pTRR* [46], appeared in the memory controllers of Intel CPUs as early as 2014 to protect vulnerable DDR3 modules. Interestingly, while memory controller-based TRR implementations still exist in modern DDR4 systems, we show that they are now mostly dormant. This is presumably because such functionality is considered superfluous now that the DRAM vendors advertise RowHammer-free modules with in-DRAM TRR, i.e., TRR implemented entirely inside the DRAM chips [63], [70].

Unfortunately, none of the in-DRAM TRR variants are well documented. As a result, their security guarantees are buried deep inside the DRAM chips that embed them. This poses a major threat to the security of modern systems, if they turn out to be vulnerable after all.

***TRRespass*.** To compensate for the lack of information, we investigate the mechanisms behind TRR and show that new TRR-aware attacks can still exploit the RowHammer vulnerability on modern DDR4 devices. We start our analysis by investigating TRR variants implemented in the memory controller and DRAM chips. We inspect memory controller-based TRR mechanisms using timing side channels to analyze when the memory controller performs a targeted refresh or whether it refreshes the entire DRAM at increased rate. Inspecting more recent in-DRAM TRR mechanisms is more challenging since these mechanisms operate transparently to the memory controller, and thus the rest of the system (e.g., targeted refresh may or may not happen during the fixed `tRFC` refresh latency). To address this challenge, we use SoftMC [31], an FPGA-based memory controller. SoftMC provides us with fine-grained control over the commands sent to DRAM. Using RowHammer bit flips and a careful selection of DRAM commands, we gradually reconstruct the different mitigations deployed on recent DDR4 modules, and uncover how they track the rows being hammered and how they protect the victim rows.

Our analysis shows that, while TRR implementations differ across DRAM vendors, most TRR variations can be bypassed by what we introduce as *Many-sided RowHammer* (i.e., RowHammer with many aggressor rows). Building on this insight, we present *TRRespass* to identify TRR-aware RowHammer access patterns on modern systems. Our fuzzing strategy generates many-sided RowHammer patterns in an entirely black-box fashion, without relying on any implementation details of the memory controller or DRAM chips. We show that relatively simple many-sided RowHammer patterns identified by *TRRespass* can successfully trigger bit flips on DDR4 DRAM chips from all three major DRAM vendors, namely Samsung, Micron, and Hynix (representing over 95% of the DRAM market [1]), as well as on mobile phones employing LPDDR4(X) DRAM chips. Overall, our analysis provides evidence for significant weaknesses in state-of-the-art TRR implementations, showing they can be bypassed to expose the vulnerable DDR4 substrate to state-of-the-art system-level RowHammer attacks.

Contributions. We make the following contributions:

- We present the first overview of different Target Row Refresh (TRR) implementations available on modern systems, which have been publicized as an effective solution to the RowHammer problem.
- We analyze the memory-controller-based and in-DRAM TRR implementations by the leading hardware vendors.
- We present *TRRespass*, a black-box RowHammer fuzzer, which can automatically identify TRR-bypassing RowHammer access patterns on 13 of 42 tested DDR4 modules from all three major DRAM manufacturers as well as 5 of 13 tested mobile phones.
- We use the RowHammer access patterns that *TRRespass* identifies on modern TRR-protected DDR4 and LPDDR4(X) DRAM chips to show how attackers can use TRR-aware RowHammer access patterns to mount state-of-the-art RowHammer attacks on these modules.

II. RowHammer on DDR4: still a problem?

Prior research has characterized [23], [51], [88], [91] and exploited the RowHammer vulnerability of DRAM [23], [25], [27], [28], [79], [89], [96]. While there has been systematic research on the vulnerability on DDR3 systems [51], [88], relatively little is known about the extent of RowHammer on recent DDR4 systems. In this section, we first provide the necessary background on DRAM and RowHammer for understanding the rest of the paper. We refer the reader to prior work [18]–[21], [29], [30], [50], [51], [57]–[61], [67], [68], [82]–[85], [97] for a more detailed description of DRAM organization and operation. Then, we perform a preliminary analysis on recent DDR4 systems using existing "hammering" patterns in the literature [27], [51], [88] to investigate the current status of the RowHammer vulnerability on DDR4.

A. DRAM Organization

Figure 1 depicts the high-level organization of a DRAM-based main memory subsystem. The CPU communicates with DRAM through the *Memory Controller* (from now on also referred to as MC). The MC is responsible for issuing memory requests to the corresponding DRAM *channel*. DRAM channels operate independently from each other and a single channel can host multiple memory modules (or *DIMMs*). DRAM *chips* in a DIMM are organized as a single *rank* or multiple *ranks*. The DRAM chips that form a rank operate in lock-step, simultaneously receiving the same DRAM command but operating on different data portions. Thus, a rank composed of several DRAM chips appears as a single large memory to the system. A DRAM chip contains multiple DRAM *banks* that operate in parallel.

Inside a bank. A DRAM bank can be logically seen as a two-dimensional array of DRAM *cells* (Figure 2). Cells that share a *wordline* are referred to as a DRAM *row*. The *row decoder* selects (i.e., activates) a row to load its data into the *row buffer*, where data can be read and modified. A DRAM cell consists of two components: (i) a *capacitor* and (ii) an *access transistor*. The capacitor stores a single bit of information as electrical

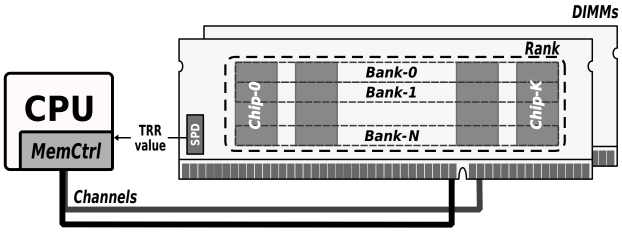


Fig. 1: High-level DRAM organization.

charge. During an access to a cell, the corresponding wordline enables the access transistor of the cell, which connects the cell capacitor to the *bitline*. Thus, to read/write data in a specific DRAM row, the memory controller first issues an `ACTIVATE` command to bring the row's data into the row buffer. The row buffer consists of *sense amplifiers*, each connected to a bitline. Because row activation destroys the data stored in the cell capacitor, a sense amplifier not only successfully determines the bit stored in the cell, but also restores the charge back into the capacitor. After the activated row of cells is fully restored, the memory controller can issue a `PRECHARGE` command to close the row and prepare the bank for activating a different row.

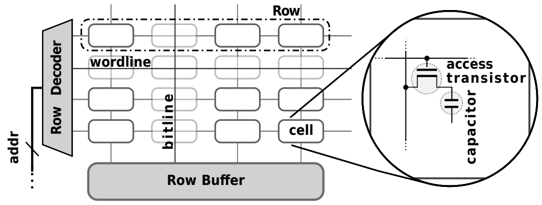


Fig. 2: DRAM bank organization (logical).

DRAM cell capacitors are not ideal and they gradually lose their charge over time. Thus, the memory controller needs to *refresh* the contents of all cells periodically (usually every $64\,ms$ [40], [44], [67]) to prevent data loss.

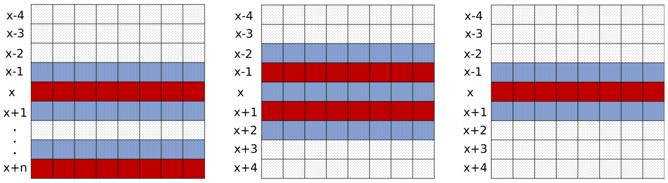
B. RowHammer

RowHammer is a well-known DRAM vulnerability that has been investigated since 2012 [8]–[10], [26], [51]. When a particular DRAM row is repeatedly activated and precharged many times (i.e., hammered), electro-magnetic interference between the hammered row and its neighbor rows can cause the cell capacitors in the neighbor rows to leak much faster than under normal operation. Rows that are hammered are referred to as *aggressor* rows, whereas their neighbor rows are referred to as *victim* rows. Kim et al. [51] are the first to perform a large-scale study of the properties of RowHammer bit flips on DDR3 modules. They report ~85% of the tested modules to be vulnerable to RowHammer. Since one can cause RowHammer bit flips solely by performing memory accesses, RowHammer quickly became a popular vector for developing real-world attacks [5], [11]–[13], [17], [23]–[25], [27], [28], [39], [66], [76]–[79], [81], [88], [89], [91], [92], [96], [98].

Attacks. Seaborn and Dullien [81] initially demonstrated RowHammer attacks for compromising the Linux kernel. Afterwards, other researchers exploited RowHammer to break cloud isolation [23], [33], [54], [77], [79], [89], [96], "root" mobile devices [91], [92], take over browsers [14], [25], [28], and attack server applications over the network [66], [89]. All these attacks demonstrate the severity of the RowHammer threat and the need to build effective defenses.

Defenses. Various software-based RowHammer defenses advocate for the detection of the RowHammer patterns [7], cross-domain [16] (or more general) memory isolation [53], [89], [92], or software-controlled ECC [23]. Unfortunately, these defenses are complex, expensive, and/or incomplete. As a result, they are not deployed in practice. Immediately-deployable hardware-based defenses, such as doubling (or even quadrupling) the refresh rate or using existing DRAM modules with error-correction code (ECC) capability to protect against RowHammer, are used in the field, yet they have been shown to be insecure [7], [23], [51].

DDR4: Towards a RowHammer-less landscape. Most prior RowHammer research focuses on DDR3 systems [5], [11]–[13], [17], [23]–[25], [28], [39], [51], [77]–[79], [81], [88], [89], [91], [92], [96], [98]. While there are reports of bit flips on DDR4 chips in prior work [27], [56], [66], these results are on earlier generations of DDR4. Through communication with industry, we have confirmation that some early-generation DDR4 chips did not have the "TRR" mitigation enabled by default. In order to understand the modern landscape we test a set of 42 recent DDR4 modules against all standard hammering patterns: (i) *single-sided*, which simply activates two arbitrary (*aggressor*) rows in the same bank to induce bit flips in their adjacent *victim* rows (Figure 3a); (ii) *double-sided*, which uses the same access patterns as *single-sided* but the two aggressor rows are chosen to both be adjacent to a single victim row to amplify the effect of hammering (Figure 3b); and (iii) *one-location*, which activates a single row (Figure 3c) and only applies to systems where the MC employs a closed-row [35], [52] or adaptive [47] page policy.



(a) Single-sided **(b)** Double-sided **(c)** One-location

Fig. 3: Standard hammering patterns. The aggressor rows are highlighted in red (🟥) and victim rows are highlighted in blue (🟦).

As we show in Figure 4, our analysis reveals that none of these patterns manifest any bit flip on the modules we test, even when using the exact test suites provided by prior work [27], [51], [88]. Our results suggest that recent DDR4 chips include effective mitigations against the best *known* hammering patterns, matching claims of DRAM vendors [63], [70]. This raises the important question: *Is RowHammer a solved problem?*

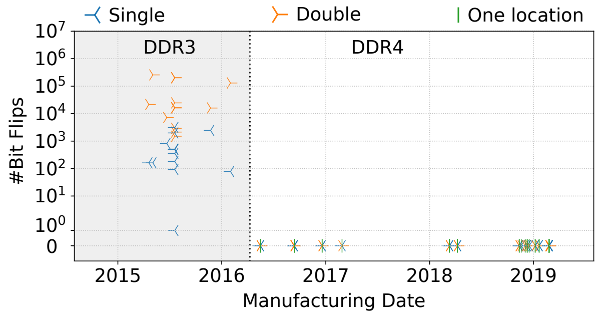


Fig. 4: Bit flips vs. manufacturing date.[3] Analysis of standard hammering patterns [51], [81], [88] on new DDR4 modules. We compare our results with the dataset of Tatar et al. [88] on DDR3 modules (shown in the left part of the chart).

III. Overview

We start our analysis by showing that TRR is not a single RowHammer mitigation mechanism. Specifically, we demonstrate that TRR is an umbrella term for different solutions at different levels of the hardware stack. Next, we analyze what is arguably the best-known TRR implementation in memory controllers, Intel pTRR [46], and show that it is not deployed in any consumer system we tested (Section IV). Since our results indicate that recent systems do not use TRR implemented at the memory controller level, we analyze TRR implementations in the DRAM chip. We examine in detail the effectiveness of the TRR mitigations that different manufacturers employ inside their chips. In particular, we show that once we reach a solid understanding of the behavior of the mitigation mechanism and build targeted access patterns accordingly, existing in-DRAM RowHammer mitigations become ineffective and one can still trigger RowHammer bit flips under standard conditions (Section V).

We observe that 1) different DRAM chips across vendors and generations can employ different TRR implementations and 2) the distribution of DRAM cells that are vulnerable to RowHammer is different for every chip. Since extensive investigation of every possible memory module is not practical, we generalize the insights gained from our investigation to build *TRRespass*: a black-box fuzzer for "TRR-aware" RowHammer analysis and exploitation (Section VI). We show how *TRRespass* can construct a plethora of new and effective RowHammer patterns on multiple TRR-protected DRAM modules. We analyze these patterns, which we collectively refer to as many-sided RowHammer (Section VII), and discuss the implications of TRR-aware hammering exploitation, showing how an attacker armed with *TRRespass* can mount successful state-of-the-art RowHammer attacks on recent DDR4 systems (Section VIII).

IV. Analyzing the memory controller

After the initial discovery of RowHammer [8]–[10], [26], [51], BIOS vendors first responded to the vulnerability by doubling the DRAM refresh rate [6], [7], [64]. However, increasing the refresh rate incurs high overhead as more refresh operations consume more energy and delay actual data transfers [51], [67]. As a consequence, manufacturers of newer CPU generations designed and deployed more efficient and effective hardware-based RowHammer mitigations [4], [8], [9], [15], [26], [46]—solutions that would also prevent attacks on vulnerable DDR3 chips.

As the MC services all incoming memory requests from CPU cores, it can efficiently track the requests and implement countermeasures in case of a RowHammer attack. Specifically, the MC can actively monitor the number of activations to specific DRAM rows and then thwart an attack by sending additional activations to DRAM rows that might be affected by RowHammer. Intel's *pseudo-TRR* [46] (or pTRR) is the most prominent example of a RowHammer defense that is deployed in the memory controller. However, while it is widely cited in the literature [7], [27], [28], [56], [66], [81], [89], [91], very little is actually known about the pTRR mechanism. In this section, we aim to verify the existence of pTRR and analyze different Intel systems to better understand the deployment and effectiveness of pTRR.

A. TRR-compliant Memory

To protect DRAM from RowHammer using additional targeted refresh operations, the MC must know the *maximum number of* `ACTIVATE`s *a row can bear* before any bit in its neighboring rows flips. Since the discovery of RowHammer, manufacturers typically store this information on the *Serial Presence Detect* (SPD) chip [50] of the DRAM module and refer to it as *Maximum Activate Count* (`MAC`). The SPD is a small read-only memory chip containing information about the memory module (Figure 1). The CPU reads the SPD at boot time to gather all the necessary parameters required to initialize the memory controller, including the `MAC` field. DRAM modules disclosing this field have been available approximately since 2014 and we denote them as *TRR-compliant*. We discuss further details in Appendix A.

The JEDEC standard specifies three possible configurations for the `MAC` value: (i) *unlimited*, if the DRAM module claims to be RowHammer-free; (ii) *untested*, if the DRAM module was not inspected after production; or (iii) a discrete value that describes the actual number of activations the DRAM module can bear (e.g., *300K*). We read out the `MAC` of the 42 DDR4 modules we test. We find that, regardless of the DRAM manufacturer, most of these modules claim to be RowHammer-free by reporting an *unlimited* `MAC` value (Table II).

B. Intel pTRR Explained

We now take a closer look at the only publicly advertised MC-based solution for Intel CPUs: *pseudo-TRR* (or pTRR) [46]. Introduced in the Ivy Bridge EP server family [46], pTRR refreshes victim rows when the number of row activations issued to the DRAM exceeds the `MAC` value—according to Intel's public documentation [46]. Unfortunately, this solution is not applicable to non-TRR-compliant modules (i.e., those without a `MAC` value or `MAC` set to *untested*). As a result, when such modules are employed, the system defaults to double refresh mode.

Observing pTRR. We analyze the only system officially reported to support pTRR: Xeon E5-2620 v2, with DDR3

[3] Following prior work [88], we approximate the manufacturing date with the purchase date when the former is unavailable (Table II shows the modules for which we applied such approximation).

memory [46]. We disable write-protection [41], [42] on the SPD of a DDR3 module and we perform the following two experiments.

① We overwrite the MAC value setting to two configurations: *untested*, simulating a non-TRR-compliant DRAM module, and *unlimited*. As mentioned above, when non-TRR-compliant memory is employed, the system should resort to double refresh rate, making it possible to detect the mitigation via frequency analysis of the access latency of uncached memory reads [69]. Indeed, we can observe that with MAC value set to *untested*, the system resorts to double refresh (Figure 5).

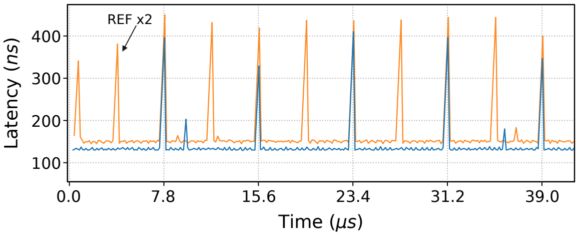


Fig. 5: Intel pTRR - Frequency of `REFRESH` commands. Uncached memory access latency with MAC value set to *Unlimited* and *Untested* on Xeon E5-2620 v2. The peaks reveal the delay introduced by the REFRESH command. We observe twice as many peaks when the MAC value is set to *Untested*.

② We overwrite the MAC value to different discrete values, expecting to observe a difference in the number of bit flips. In the leftmost stack of Figure 6, we show the result of this experiment when hammering the same chunk of memory with MAC value set to *400K* or to *unlimited*. We observe that the number of bit flips drastically decreases when pTRR is enabled (i.e., MAC value set to 400K). Additionally (not shown in Figure 6), we discover that when setting the MAC value to the minimum value defined in the DDR3 specification [41] (i.e., 200K), the system treats the module as a non-TRR-compliant module; that is, it enables double refresh. We do not analyze the effectiveness of pTRR in mitigating RowHammerbit flips in this paper. Lipp et al. [66] report bit flips on a pTRR-enabled system, and operating at increased refresh rate (i.e., double refresh rate) is known to be ineffective [7], [23], [51] at protecting against RowHammer.

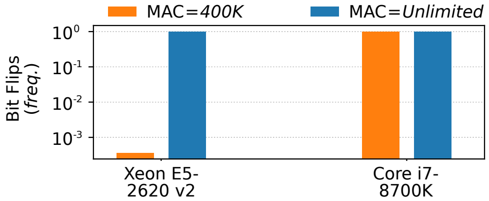


Fig. 6: Intel pTRR - Bit flips observed with different `MAC` values. Frequency of observed bit flips for different MAC configurations. Comparison between a system employing pTRR (Xeon E5-2620 v2) and a system with no MC-based RowHammer mitigation (Core i7-8700K).

pTRR has limited deployment. We run the two experiments for analyzing pTRR on 6 other Intel CPUs from different architecture families that are descendants of Ivy Bridge—both server and consumer lines. Surprisingly, the first experiment ① shows that, when the MAC value is set to *untested*, the memory controller of each of these 6 CPUs still refreshes the DRAM with the default (not double) refresh rate. This observation shows that no RowHammer mitigation is present at the memory controller level in these CPUs. We corroborate this hypothesis by carrying out the second experiment ② where we measure the number of bit flips as we vary the MAC value. We use the new RowHammer patterns we present in Section VI for the CPUs that support DDR4. In contrast to Xeon E5-2620 v2 server-line CPU, which is reported to support pTRR [46], the second experiment on consumer-line CPUs does *not* identify a different number of bit flips for different MAC values. In the rightmost stack of Figure 6, we show the results for the Intel Core i7-8700K consumer-line CPU as an example to illustrate the difference between any of these consumer systems and a pTRR-enabled system. This experiment confirms that pTRR is in fact not present in customer-line CPUs that we test. We list the deployment of MC-based RowHammer mitigations in both server- and consumer-line CPUs in Table I.

TABLE I: Memory controller defenses. Defenses detected in our experiments on Intel CPUs starting from the Ivy Bridge family.

CPU	*Family*	*Year*	*DRAM generation*	*Defense*
Server Line				
Xeon E5-2620 v4	Broadwell	2016	DDR4	REF×2
Xeon E5-2620 v2	Ivy Bridge EP	2013	DDR3	pTRR
Xeon E3-1270 v3	Haswell	2013	DDR3	—
Consumer Line				
Core i9-9900K	Coffee Lake R	2018	DDR4	—
Core i7-8700K	Coffee Lake	2017	DDR4	—
Core i7-7700K	Kaby Lake	2017	DDR4	—
Core i7-5775C	Broadwell	2015	DDR3	—

C. Discussion

Our experiments show that the memory controller-based RowHammer mitigations are deployed only in specific families of Intel processors. While we find that pTRR and other mitigations (e.g., double refresh) are used in high-end Xeon servers, our results show that neither DDR3 nor DDR4 consumer systems appear to enable any MC-based mitigation. In Figure 7, we reconstruct a timeline of RowHammer mitigations on Intel platforms based on the results of our analysis. With both DDR3 and DDR4, only server platforms appear to benefit from mitigations inside the memory controller while consumer platforms do not. Based on earlier reports of bit flips using standard RowHammer patterns on consumer DDR4 memory [27], [56], [66], we can speculate that in-DRAM mitigations are widely-deployed only since 2016 (i.e., the earliest manufacturing date of a DRAM module with MAC set to *unlimited* among all modules that we list in Table II). In other words, DRAM manufacturers' promises of a RowHammer-less future [63], [70] hinge entirely on the security of their undocumented in-DRAM TRR mitigations. Unfortunately, as we show in the next sections, analyzing and understanding such mitigations can reveal significant weaknesses that can be exploited to mount RowHammer attacks on modern DDR4 DRAM chips.

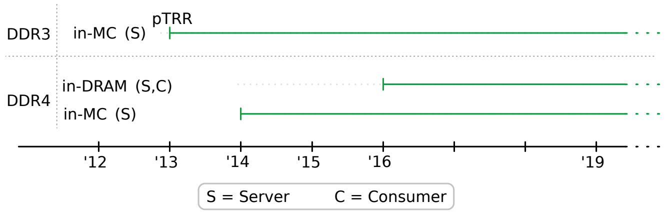


Fig. 7: TRR Timeline. Timeline of deployment of TRR as RowHammer mitigation. MC-based mitigations are deployed in both DDR3 and DDR4 server systems since 2013 [46]. In contrast, in-DRAM mitigations appear with DDR4 for both consumer and server systems [63], [70].

V. Inside the DRAM chips

We dig deeper to understand the RowHammer protection that the DRAM vendors implement inside recent DRAM chips, which are advertised as RowHammer-free [63], [70]. So far, the DRAM vendors have not publicly shared the details of the exact RowHammer protection mechanisms they implement in the form of TRR. Therefore, we experiment with and analyze real DRAM chips to shed light on the inner workings of the TRR mechanisms implemented by different vendors in different DRAM chip generations. Performing such an analysis using a general-purpose CPU is extremely challenging because the memory controller provides a very high-level interface to the CPU (i.e., the programmer can interface with the DRAM using only load/store instructions). However, to perform accurate experiments, we need fine-grained control over the low-level commands sent to the DRAM. Therefore, we leverage an open-source FPGA-based memory controller, SoftMC [31], [80], which enables the programmer to issue arbitrary DRAM commands in a cycle-accurate manner. We extend SoftMC to support experimental studies on DDR4 modules. We first discuss our hypotheses for potential ways of implementing in-DRAM TRR. Then, we present case studies for two DRAM modules from different manufacturers. Our results show that different manufacturers implement vastly different TRR mitigations.

A. Building Blocks and Hypotheses

While literature indicates that each manufacturer may implement its own variant of TRR [34], [38], [45], [49], [65], [74], [75], [95], we abstract the implementation details and unravel the two main requirements for supporting TRR: the sampler and the inhibitor. We define these requirements as building blocks and present a series of hypotheses that we verify in the next sections.

The Sampler. A sampling mechanism is required to track which aggressor rows are being hammered. Solutions vary from basic frequency-based sampling to more complex designs that track activations per row. In frequency-based implementations, sampling occurs at fixed periods in time within a refresh interval [34], [74], [95]. For example, a TRR implementation may determine aggressor rows by monitoring every 3rd and 4th access after a `REFRESH`. The more complex designs that track accesses on a per-row basis, keep activation counters for a number of rows [45], [65] and select aggressors based on their individual activation counts. Despite differences in its implementation, the goal of the sampler remains the same: track which rows are being hammered in order to identify their *target* victim rows.

Our first hypothesis is that the sampler has a limited size s. In other words, there is a maximum number of aggressor rows that the sampler can track. Phrased differently, the TRR mitigation can protect only a limited number of victim rows.

The Inhibitor. Once the sampler is aware of the aggressor rows, the mitigation must thwart the hammering process. As the name *Target Row Refresh* suggests (and different designs confirm [34], [74], [95]), an effective solution consists of generating extra refreshes for the victim rows. Nonetheless, more sophisticated designs incorporate the possibility of row remapping [38], [48], [67].

Our second hypothesis is that the inhibitor acts at refresh time—based on the literature [34], [74], [95]. Refresh operation is the responsibility of the memory controller, which issues one `REFRESH` command every $7.8\,\mu s$ (`tREFI`). Since DDR is a synchronous protocol [40], [44], the memory controller must remain idle for a fixed period of time (`tRFC`) before it can send subsequent commands to the bank. Any, possibly additional, targeted refreshes must still respect these timing constraints for the DRAM module to be compliant with the DDRx standard. That is, only a limited number of target rows can be refreshed.

Goals. Based on the aforementioned assumptions we define the following questions that we want to answer.

- What is the size of the sampler?
- How does the sampler track aggressor rows? For example, does it record row activation commands at a constant frequency or based on a function of time?
- How does the inhibitor work? Can it prevent bit flips?

In the following, we try to answer these questions by analyzing TRR via two different case studies.

B. Case I: Module $\mathcal{C}_{12}$

Our first study examines a module from manufacturer $\mathcal{C}$. We first find the minimum number of activations that are required to trigger bit flips on this module. To do so, we disable refresh, which prevents TRR from performing refresh on victim rows, and perform a double-sided RowHammer sweep of a single DRAM bank. The results, plotted in Figure 8, show that we can trigger bit flips with as few as 50K activations. This indicates that DRAM cells in DDR4 are generally considerably weaker compared to DRAM cells in the predecessor DDR3 standard, which requires at least ~139K activations [51]. Nonetheless, for future experiments reported in this paper we use a higher activation count so that we can observe more bit flips and draw stronger conclusions.

Mastering refresh. Knowing the physical limitations of the DRAM module, we now reintroduce the `REFRESH` command. We decide to batch refresh operations together with the goal of understanding the relationship between them and the effectiveness of the TRR mitigation. We perform a series of hammers (i.e., activations of aggressors) followed by r refreshes for ten rounds—we carry out 8K hammers per round. In Figure 9, we report the results of this experiment

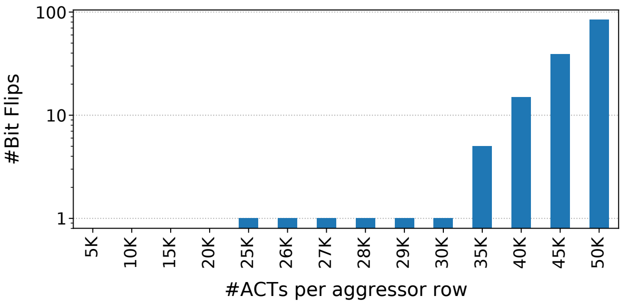


Fig. 8: Bit flips vs. number of activations. Module $\mathcal{C}_{12}$: We can observe bit flips with as few as 25K activations per aggressor row (i.e., 50K activations in total due to double-sided hammering).

for RowHammer configurations with different numbers of aggressors. Let us first consider only the third column of the plot: double-sided RowHammer. We observe that adding a single refresh causes the number of bit flips to drop from 2,866 to only one (for $r = 1$), and then to zero (for any $r \geq 2$). This experiment provides an insightful result: since sending multiple `REFRESH` commands varies the number of bit flips, the TRR mitigation must act on every refresh command.

> ***Observation* 1:** The TRR mitigation acts (i.e., carries out a targeted refresh) on **every** refresh command.

Next, we take a closer look at the sampler size s to find how many rows the mitigation can handle. We increase the number of aggressors n while keeping the number of `ACTs` per aggressor row constant. For every additional aggressor row, we have an additional victim row. For example, with 3 aggressor rows, the hammering configuration looks like `VAVAVAV` where `V` stands for a victim row and `A` stands for an aggressor row. The fourth column in Figure 9 shows the behavior when hammering three aggressors ($n = 3$). Here we observe something different: the number of bit flips decreases significantly when introducing up to two refreshes. However, it plateaus for $r \geq 3$ *without* going down to zero. Notice that when hammering both 2 and 3 rows, the plateaus happen when $r = n$. This suggests that *the TRR mitigation samples more than one aggressor within a refresh interval while it can refresh only one victim per refresh operation*. The DRAM can refresh only one of the victim rows likely as a consequence of the tight timing constraints imposed by the `tRFC` parameter. Moreover, we can deduce from the remaining non-zero bit flips that the sampler is likely to discard the aggressor row from its table once one of its victims has been refreshed. We can recover the size of the sampler by performing the same experiment for different numbers of aggressors n. While increasing n, we search for the scenario where the number of bit flips stabilizes for $r < n$. When this happens, we can conclude that we have overflowed the sampler. We show the results of this experiment for different values of n in Figure 9. As speculated, we see the number of bit flips leveling off (i.e., remaining constant on the y-axis) for $r \geq 4$, revealing the size of the sampler to be $s = 4$: the sampler in this module can track only 4 aggressor rows.

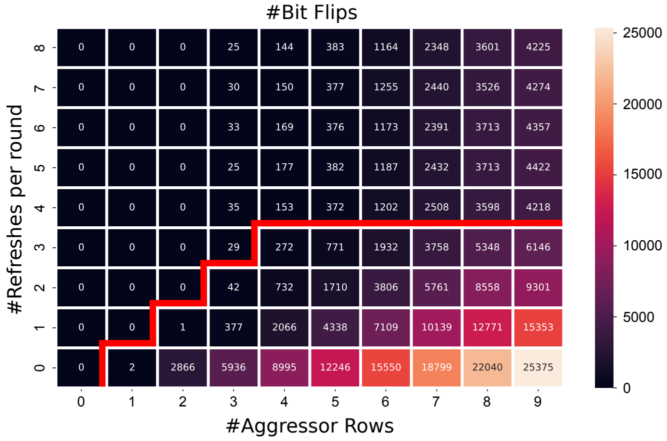


Fig. 9: Refreshes vs. Bit Flips. Module $\mathcal{C}_{12}$: Number of bit flips detected when sending r refresh commands to the module. We report this for different number of aggressor rows (n). For example, when hammering 5 rows, followed by sending 2 refreshes, we find 1,710 bit flips. This figure shows that the number of bit flips stabilizes for $r \geq 4$, implying that the size of the sampler may be 4.

> ***Observation* 2:** The mitigation can sample **more than one** aggressor per refresh interval.
> ***Observation* 3:** The mitigation can refresh only a **single** victim within a refresh operation (i.e., time `tRFC`).
> ***Observation* 4:** Sweeping the number of refresh operations and aggressor rows while hammering reveals the sampler size.

Based on these observations, we conclude that hammering more than 4 rows should circumvent the mitigation. We confirm this by running a test on our SoftMC FPGA infrastructure [31] with standard conditions (i.e., `tREFI` $= 7.8\,\mu s$). Indeed, Figure 10 shows that we overwhelm the mitigation by hammering 5 rows. Figure 10 provides another insight: it shows that for every number of aggressors >5, the number of bit flips decreases drastically compared to 5-sided RowHammer—suggesting that the sampler selects rows in a specific fashion. While we tried to understand this behavior of the sampler, the lack of visibility inside the DRAM chip made it challenging. Regardless, this additional information is not necessary given that hammering 5 aggressors in standard conditions already bypasses the in-DRAM mitigation.

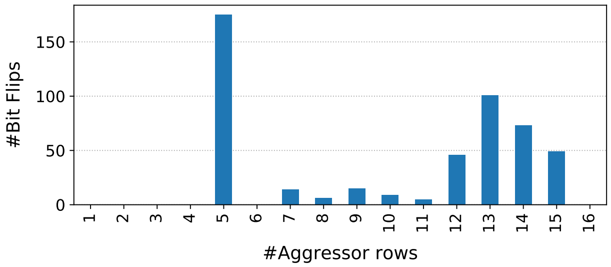


Fig. 10: Bit flips vs. number of aggressor rows. Module $\mathcal{C}_{12}$: Number of bit flips in bank 0 as we vary the number of aggressor rows. Using SoftMC, we refresh DRAM with standard `tREFI` and run the tests until each aggressor rows is hammered 500K times.

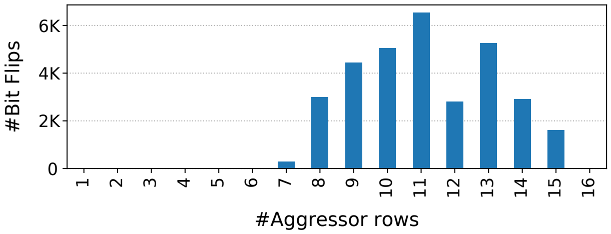


Fig. 11: Bit flips vs. number of aggressor rows. Module $\mathcal{A}_{15}$: Number of bit flips in bank 0 as we vary the number of aggressor rows. Using SoftMC, we refresh DRAM with standard `tREFI` and run the tests until each aggressor rows is hammered 500K times.

C. Case II: Module $\mathcal{A}_{15}$

To provide an understanding of the different flavors of in-DRAM TRR, we further study the behavior of a memory module from a different manufacturer: $\mathcal{A}_{15}$. We quickly test and confirm that the mitigation acts at every refresh command, corroborating the observation made in the previous case study. We then move to analyzing the relationship between the number of bit flips and the number of aggressors n, with the default refresh rate, depicted in Figure 11. We find that we can reliably flip bits for $n \geq 7$, indicating a sampler of size 6.

Double-sided RowHammer resurrected. Although we now already bypass the mitigation, we take this one step further and try to analyze the sampler to see if we can revive the more efficient double-sided RowHammer attack. Our approach consists of finding the minimal set of *dummy* rows that allows us to trick the mitigation mechanism into refreshing all other neighbors of the hammered rows but our victim. For this, we focus on a single row that we know to be susceptible to bit flips and for which we find in advance the threshold of hammers required to observe bit flips. Based on this threshold, we carry out successive experiments while modifying two parameters: (i) the distribution of activations across aggressor and dummy rows and (ii) the number of dummies starting from 6 (i.e., the supposed size of the sampler). To our surprise, regardless of the configuration, we could not detect any bit flip.

Investigating further, we discover two more parameters that were previously unaccounted for:

DRAM command order dependency. The sampler may act at specific DRAM commands issued within a refresh interval and thus it may not necessarily sample based only on frequency. In the case of module $\mathcal{A}_{15}$, the sampler seems to record the first α activations after a refresh command—where $\alpha \leq 6$.

Address dependency. In module $\mathcal{A}_{15}$, we observe a dependency between the aggressor-row address and the dummy rows' addresses. That is, when hammering two aggressor rows, we detect more bit flips when we pick particular dummy rows compared to picking random dummy rows. This suggests that the design of the sampler involves optimizations to reduce the storage cost of tracking row activations such that multiple aggressor rows' addresses may conflict.

***Observation* 5:** The sampler records row activations at specific commands and likely at specific ordering of commands (i.e., it performs **command-order-based sampling**).
***Observation* 6:** The sampling mechanism is affected by the addresses of aggressor rows (i.e., sampling is **row-address-dependent**).

D. Running on the CPU: Module $\mathcal{A}_{15}$

While we observe a considerable number of bit flips when we use the (optimal) activation pattern discovered by SoftMC, a custom FPGA memory controller does not represent a widespread threat model. As a consequence, we want to check if we can reproduce the same access pattern when running on commodity hardware, such as a regular desktop computer.

During the analysis process, we find the mitigation of the $\mathcal{A}_{15}$ memory module to be command order and address dependent. This represents a great challenge when trying to reproduce access patterns that cause bit flips from the CPU. In fact, in order to fool the mitigation, we need to carry out a specific series of activations right after a `REFRESH` command to keep the inhibitor busy with another set of rows than the intended victim row. This means we need to synchronize our access pattern with the `REFRESH` command. Even though we can detect refresh operations (Section IV), synchronizing our access pattern with them is much more difficult. We re-implement the access patterns discovered in the analysis process, which we explain in Section V-C, to run on the CPU. However, we observe much fewer bit flips compared to what we obtain with SoftMC, suggesting we may not be able to perfectly synchronize the hammering pattern with the refresh operations using a CPU. This is likely due to the fact that the memory controller applies various optimizations that can reorder memory requests and refresh commands.

E. Observations

Our experiments in Section V-D show the difficulty of reproducing our FPGA results—those obtained in a simplified, controlled environment—on a modern CPU. This advocates for a better solution for finding effective access patterns that trigger bit flips on TRR-protected DDR4 chips. In the next section, we introduce *TRRespass*, a black-box RowHammer test suite that generates effective access patterns to bypass in-DRAM TRR solutions.

TRRespass is inspired by the insights that we obtained using our analysis of TRR-protected DDR4 chips in this section. More specifically, we take advantage of the following insights:

1. The sampler can track a limited number of aggressor rows. Thus, we may need to *overflow* the sampler's aggressor rows *table* in order to bypass the TRR mitigation.
2. The sampler may sample activations at specific commands, at a specific frequency, or both.
3. The sampler design may be row address dependent. Therefore, some rows may be easier to hammer than others and the same set of rows activated in different order may yield completely different results.
4. The cells in DDR4 chips are much more RowHammer-prone than those on DDR3 [51], requiring fewer activations to trigger bit flips.

In the next section, we describe how we use these observations to build a (guided) black-box fuzzer that can cause bit flips on TRR-protected DDR4 modules.

VI. *TRRespass*: A TRR-aware Rowfuzzer

To convert the knowledge that we gathered from our analysis process on the FPGA-based SoftMC platform into practical attacks that we can launch from regular software on a CPU, we developed a guided black-box fuzzer for RowHammer called *TRRespass*. When searching for usable access patterns, a CPU-based fuzzer has two main advantages over an FPGA-based approach: (i) it allows an attacker to completely ignore the memory controller (and the optimizations it implements), and (ii) it provides a scalable approach to testing for RowHammer bit flips. Indeed, since different manufacturers deploy very different TRR solutions as we show in Section V, trying to obtain a detailed understanding of the full behavior of every TRR-protected memory module is not practical. Even so, we will demonstrate that these details in most cases do not get in the way of finding effective patterns: *TRRespass* was able to automatically find access patterns that trigger bit flips on modules we did *not* analyze, and even on mobile platforms using LPDDR4(X) chips—albeit in a simplified way.

A. Design

Based on the observations in Section V, *TRRespass*' fuzzing strategy is based on two parameters: *Cardinality* and *Location*.

Cardinality. Cardinality represents the number of aggressor rows hammered. We show in Section V-B that some modules require a large number of aggressor rows to overflow the sampler and induce bit flips. For instance, Figure 11 indicates that we need at least 7 rows to observe bit flips in module $\mathcal{A}_{15}$. On the other hand, increasing the cardinality too much is counterproductive. In particular, a DRAM module cannot carry out more than a certain number of activations within the $64\,ms$ interval between two refreshes of the same row. The maximum number of row activations that can be performed within $64\,ms$ mainly depends on the *row cycle time* (t_{RC}) that defines the number of clock cycles between two `ACTIVATE` commands to the same bank. In most modules $t_{RC} \approx 45\,ns$. It follows that the maximum number of activations that we can perform within a $64ms$ interval is 1.4×10^6 ($64\,ms \div 45\,ns$). If we tune the fuzzer to hammer each aggressor row at least 50K times (see Section V-B), the upper limit for the cardinality is 28 rows.

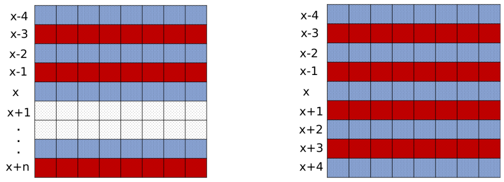
Location. Based on the results of Section V-C, we know that the sampler may have dependence on row addresses. Thus, we want to randomize the location of the aggressors to maximize the probability of bypassing address-dependent TRR mitigations. Moreover, by picking the access pattern randomly, we implicitly bypass any feature of the sampler in the time domain. That is, regardless of the design of the sampler (command-order-based or frequency-based), choosing random values for the distances between the aggressors also randomizes the aggressors' relative positions in the access pattern. Given a set of aggressors, we choose to activate them in a round-robin fashion since our experiments show that other strategies do not bring benefits in terms of the number of bit flips.

Fuzzing strategy. *TRRespass* evaluates randomly-generated access patterns based on the number of unique bit flips. It generates the patterns by randomizing the cardinality and location parameters. If a bank contains n rows, evaluating the combinations of all n rows taking k at a time ($k < n$) would be impractical as n is in the order of tens of thousands in modern DRAMs. To obtain results within a reasonable time frame, the fuzzer therefore allocates a smaller chunk of memory, spanning a subset of rows across different banks, and builds RowHammer access patterns that respect the geometry of the memory configuration [76]. The number of patterns that the fuzzer can test in a given time frame is determined by the number of hammering rounds (i.e., activations ÷ cardinality). We pick this value such that we generate activations that cover more than 3 × *refresh period*. This configuration makes sure that the victim rows are hammered for at least an entire $64\,ms$ interval before their refresh.

B. TRRespass-ing over DDR4

We evaluate our fuzzer and all other experiments on an Intel Core i7-7700K, mounted on an ASUS STRIX Z270G motherboard. We acquire a set of 42 memory modules produced by the three leading DRAM manufacturers (currently holding around 95% of the market [1]). As shown in Table II, the set consists of 16 modules from vendor $\mathcal{A}$, 12 from $\mathcal{B}$, and 14 from $\mathcal{C}$. We tested all the memory modules singularly to draw conclusions about the individual chips. We ran *TRRespass* for more than 6 hours on each module, scanning a memory chunk of 128 adjacent rows from each bank. We now describe the results obtained through *TRRespass*' black-box analysis.

Many-sided RowHammer. In one of our initial tests, *TRRespass* assembled a very simple and elegant access pattern that turned out to be effective on most $\mathcal{B}$ modules: *assisted double-sided*. That is, a double-sided pattern with a "sidekick" row. As shown in Figure 12a, this pattern hammers rows $x-1, x+1$, similarly to double-sided RowHammer, plus an extra one ($x+n$, where $n > 2$).



(a) Assisted double-sided **(b)** 4-sided

Fig. 12: Hammering patterns discovered by *TRRespass*. Aggressor rows are in red (🟥) and victim rows are in blue (🟦).

The analysis on all the 42 DRAM modules then allowed us to generalize the assisted double-sided access pattern to a broader class of access patterns which we call *Many-sided RowHammer*. Our results show that an attacker can benefit from sophisticated hammering patterns that exploit repeated accesses to *many* aggressor rows. We now refer to the discovered patterns using the nomenclature n-sided where n is the *cardinality* of the pattern. For instance, assisted double-sided which is effective on $\mathcal{B}$ modules (Figure 12a), falls under the category of 3-sided RowHammer. Note that while we omit the *location* of the aggressors from this discussion, this

TABLE II: *TRRespass* results. We report the number of patterns found and bit flips detected for the 42 DRAM modules in our set.

Module	*Date*	*Freq.*	*Size*	*Organization*			*MAC*	*Found*	*Best Pattern*	*Corruptions*			*Double*
	(yy-ww)	*(MHz)*	*(GB)*	*Ranks*	*Banks*	*Pins*		*Patterns*		*Total*	$1 \rightarrow 0$	$0 \rightarrow 1$	*Refresh*
$\mathcal{A}_{0,1,2,3}$	16-37	2132	4	1	16	×8	UL	—	—	—	—	—	—
$\mathcal{A}_{4}$	16-51	2132	4	1	16	×8	UL	4	9-sided	7956	4008	3948	—
$\mathcal{A}_{5}$	18-51	2400	4	1	8	×16	UL	—	—	—	—	—	—
$\mathcal{A}_{6,7}$	18-15	2666	4	1	8	×16	UL	—	—	—	—	—	—
$\mathcal{A}_{8}$	17-09	2400	8	1	16	×8	UL	33	19-sided	20808	10289	10519	—
$\mathcal{A}_{9}$	17-31	2400	8	1	16	×8	UL	33	19-sided	24854	12580	12274	—
$\mathcal{A}_{10}$	19-02	2400	16	2	16	×8	UL	488	10-sided	11342	1809	11533	✓
$\mathcal{A}_{11}$	19-02	2400	16	2	16	×8	UL	523	10-sided	12830	1682	11148	✓
$\mathcal{A}_{12,13}$	18-50	2666	8	1	16	×8	UL	—	—	—	—	—	—
$\mathcal{A}_{14}$	19-08†	3200	16	2	16	×8	UL	120	14-sided	32723	16490	16233	—
$\mathcal{A}_{15}$‡	17-08	2132	4	1	16	×8	UL	2	9-sided	22397	12351	10046	—
$\mathcal{B}_{0}$	18-11	2666	16	2	16	×8	UL	2	3-sided	17	10	7	—
$\mathcal{B}_{1}$	18-11	2666	16	2	16	×8	UL	2	3-sided	22	16	6	—
$\mathcal{B}_{2}$	18-49	3000	16	2	16	×8	UL	2	3-sided	5	2	3	—
$\mathcal{B}_{3}$	19-08†	3000	8	1	16	×8	UL	—	—	—	—	—	—
$\mathcal{B}_{4,5}$	19-08†	2666	8	2	16	×8	UL	—	—	—	—	—	—
$\mathcal{B}_{6,7}$	19-08†	2400	4	1	16	×8	UL	—	—	—	—	—	—
$\mathcal{B}_{8}$◇	19-08†	2400	8	1	16	×8	UL	—	—	—	—	—	—
$\mathcal{B}_{9}$◇	19-08†	2400	8	1	16	×8	UL	2	3-sided	12	—	12	✓
$\mathcal{B}_{10,11}$	16-13†	2132	8	2	16	×8	UL	—	—	—	—	—	—
$\mathcal{C}_{0,1}$	18-46	2666	16	2	16	×8	UL	—	—	—	—	—	—
$\mathcal{C}_{2,3}$	19-08†	2800	4	1	16	×8	UL	—	—	—	—	—	—
$\mathcal{C}_{4,5}$	19-08†	3000	8	1	16	×8	UL	—	—	—	—	—	—
$\mathcal{C}_{6,7}$	19-08†	3000	16	2	16	×8	UL	—	—	—	—	—	—
$\mathcal{C}_{8}$	19-08†	3200	16	2	16	×8	UL	—	—	—	—	—	—
$\mathcal{C}_{9}$	18-47	2666	16	2	16	×8	UL	—	—	—	—	—	—
$\mathcal{C}_{10,11}$	19-04	2933	8	1	16	×8	UL	—	—	—	—	—	—
$\mathcal{C}_{12}$‡	15-01†	2132	4	1	16	×8	UT	25	10-sided	190037	63904	126133	✓
$\mathcal{C}_{13}$‡	18-49	2132	4	1	16	×8	UT	3	9-sided	694	239	455	—

† The module does not report manufacturing date. Therefore, we report purchase date as an approximation. UL = Unlimited
‡ Analyzed using the FPGA-based SoftMC. UT = Untested
◇ The system runs with double refresh frequency in standard conditions. We configured the refresh interval to be $64\,ms$ in the BIOS settings.

parameter in some cases does play a role in the effectiveness of the pattern and we further discuss it in Appendix B.

Results. *TRRespass* discovered effective access patterns for 13 of the 42 TRR-protected memory modules in our set. Table II reports the results for the number of access patterns identified and the structure of the most effective pattern. One interesting insight we gain from our analysis is that *there is not a single effective access pattern per module*. In fact, we can see that all the modules where *TRRespass* induces bit flips are vulnerable to at least two different access patterns. On $\mathcal{B}$ modules, we could identify access patterns on 4 out of the 12 modules we analyzed, and always with simple 4-sided and 3-sided patterns as presented in Figure 12. On the other hand, none of these patterns appear to work on the other vendors' chips. For example, in Figure 13, we show the number of aggressor rows required to trigger bit flips on module $\mathcal{A}_{10}$. We can see that no bit flip can be triggered with fewer than 8 aggressor rows. *TRRespass* successfully triggers bit flips on 7 of 16 $\mathcal{A}$ modules, with several very different patterns. $\mathcal{A}_4$ and $\mathcal{A}_{15}$ are mainly vulnerable to the 9-sided variant, $\mathcal{A}_{10}$ and $\mathcal{A}_{11}$ to different variants of the 10-sided pattern, and $\mathcal{A}_8$ and $\mathcal{A}_9$ to a 19-sided pattern. On $\mathcal{C}$ modules, *TRRespass* discovers effective RowHammer patterns on only 2 of 14 modules. We observe that $\mathcal{C}_{12}$ and $\mathcal{C}_{13}$ are vulnerable to 9-sided and 10-sided hammering patterns.[4]

A scalable framework. The results of *TRRespass* on module $\mathcal{A}_{15}$ demonstrate how a black-box approach can be extremely beneficial. In Section V, we describe how complex it can be to reproduce the optimal access pattern discovered using SoftMC on a CPU system. In contrast, *TRRespass* discovers two very successful access patterns that generate a significant number of bit flips automatically.

[4] While *TRRespass* identifies effective RowHammer access patterns only on 13 out of 42 modules, this does not mean that the other modules are immune to RowHammer. Similarly, these results do not necessarily show that that memory modules from a specific vendor are more or less vulnerable than modules from other vendors. Similar to regular software fuzzers, it may simply be a matter of time and better strategies to find access patterns that lead to bit flips. Our testing is also *not* exhaustive due to limited testing time and resources.

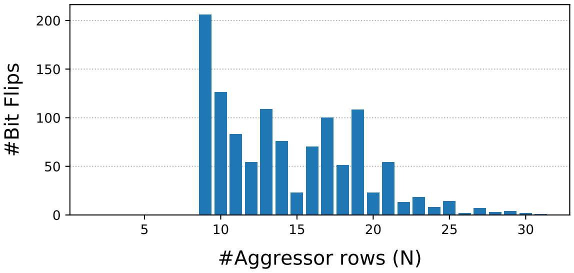


Fig. 13: Bit flips vs. number of aggressor rows. Module $\mathcal{A}_{10}$: Number of bit flips triggered with *N-sided* RowHammer for varying number of *N* on Intel Core i7-7700K. Each aggressor row is one row away from the closest aggressor row (i.e., VAVAVA... configuration) and aggressor rows are hammered in a round-robin fashion.

C. TRRespass on LPDDR4(X)

In order to understand how widespread the issue is, we implement a simplified version of *TRRespass* for ARMv8 to test LPDDR4 [40] and LPDDR4X [43] chips on mobile phones. Due to the fragmented nature of the Android ecosystem and the limited privileges (and resources) available on some of these devices, we drop one of the two fundamental parameters used in our previous design: *location*. In other words, we simply map a big chunk of memory and find a pool of addresses that belong to the same bank [76]. Van der Veen et al. [91], [92] rely on uncached memory due to the lack of cache flushing instructions on ARMv7. This restriction does not apply any longer on ARMv8, and thus we do not use uncached memory. In our experiments, *TRRespass* discovers effective hammering patterns on 5 of the 13 devices, proving that TRR-protected mobile platforms are also still vulnerable to RowHammer (data is shown in Table III). Not all mobile platforms report information about the memory manufacturer and we do not have fine grained control over the memory allocations. As a result, we cannot draw any conclusion with regard to the extent of the vulnerability on LPDDR4(X). Furthermore, phones from the same model can use DRAM chips from different manufacturers. This means that even if *TRRespass* finds RowHammer bit flips on a certain phone from a specific model, another phone from the same model may not exhibit these bit flips. Similarly, the opposite can also be true.

VII. Evaluation

In this section, we systematically evaluate our 42 DDR4 DRAM modules against the optimal RowHammer access pattern (i.e., the one that yields the most bit flips) identified by *TRRespass* for each module.

A. Results

We test each of the 42 modules using the most-bit-flip-incurring RowHammer pattern that we discover for each module in Section VI-B. For every module, we perform a sweep over 256 MB of contiguous physical memory.[5] We then examine the memory for RowHammer bit flips in both *true* cells and *anti* cells [51], [68]. In other words, we look for both 1 → 0 and 0 → 1 bit flips. We show the results for all the 42 modules in Table II. We now provide a detailed explanation of these results by discussing them separately for each DRAM vendor.

TABLE III: LPDDR4(X) results. Mobile phones tested against *TRRespass* on ARMv8 sorted by production date. We found bit flip inducing RowHammer patterns on 5 out of 13mobile phones.

Mobile Phone	*Year*	*SoC*	*Memory (GB)*	*Found Patterns*
Google Pixel	2016	MSM8996	4†	✓
Google Pixel 2	2017	MSM8998	4	—
Samsung G960F/DS	2018	Exynos 9810	4	—
Huawei P20 DS	2018	Kirin 970	4	—
Sony XZ3	2018	SDM845	4	—
HTC U12+	2018	SDM845	6	—
LG G7 ThinQ	2018	SDM845	4†	✓
Google Pixel 3	2018	SDM845	4	✓
Google Pixel 4	2019	SM8150	6	—
OnePlus 7	2019	SM8150	8	✓
Samsung G970F/DS	2019	Exynos 9820	6	✓
Huawei P30 DS	2019	Kirin 980	6	—
Xiaomi Redmi Note 8 Pro	2019	Helio G90T	6	—

† LPDDR4 (not LPDDR4X)

Vendor $\mathcal{A}$. In Section VI, we show *TRRespass* can bypass how mitigations from manufacturer $\mathcal{A}$. We can recover multiple effective access patterns for 7 of the 16 modules in our experiments. In Table II, we provide the number of bit flips that we observe on the vulnerable $\mathcal{A}$ modules. The results are worrisome: we find more than 16K bit flips on average across the 7 vulnerable modules. In addition to the large number of bit flips, we also observe that the bit flips occur with significantly fewer row activations on vendor $\mathcal{A}$'s DDR4 modules compared to previous generation DDR3 DRAM devices. For example, on $\mathcal{A}_8$ and $\mathcal{A}_9$, we can effectively perform 19-sided RowHammer with as few as ~45K row activations to each of the effective aggressor rows (i.e., the aggressor rows adjacent to the target victim row(s)) within the 64ms refresh period. In contrast, Kim et al. [51] show that bit flips occur with ~139K or more DRAM row activations on older DDR3 modules.

Vendor $\mathcal{B}$. In Section VI, we describe assisted double-sided (i.e., 3-sided) and 4-sided hammering as two effective patterns against a subset of our memory modules from vendor $\mathcal{B}$. However, the low bit flip counts in Table II show that bypassing the TRR mitigation on these modules is non-trivial. We run further experiments on these modules to understand the limited number of bit flips we observe. We make two observations. First, when we repeat for multiple iterations the

[5] We avoid testing the entire capacity of the DRAM modules and instead test 256 MB of each module to reduce testing time. We note that this could potentially cause *TRRespass* to miss the most RowHammer-prone portions of a module. Thus, we believe *TRRespass* is likely to be more effective than what we report in this paper.

same RowHammer experiment using the aggressor rows that we know can cause bit flips, we observe a varying number of bit flips in the victim row(s) across different iterations. Figure 14 shows the number of bit flips that we can trigger on a specific row, using 3 aggressors in module $\mathcal{B}_0$. We observe from the figure that different iterations (i.e., samples) of the same test reveal a different number of bit flips in the same victim row. Second, when hammering the same module in a multi-DIMM configuration (i.e., two identical modules on the same system), we often observe more bit flips. These results hint at the presence of a parameter *TRRespass* cannot (yet) bypass. The fact that we occasionally observe a large number of bit flips suggests that these modules are quite susceptible to RowHammer, and causing more bit flips may be only a matter of improving our fuzzing strategy.

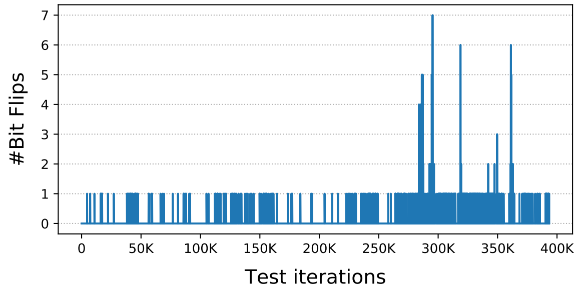


Fig. 14: Bit flips vs. Test iterations. Module $\mathcal{B}_0$: Number of bit flips over test iterations. In each iteration, the aggressor rows are hammered for three refresh intervals.

Vendor $\mathcal{C}$. *TRRespass* identifies effective patterns on 2 of 14 $\mathcal{C}$ modules. However, we see a steep drop in the number of bit flips on modules from newer generations. Module $\mathcal{C}_{12}$, produced before 2015, is the oldest and most vulnerable module in our test set (Table II). Modules of newer generations are less vulnerable (if at all) to the patterns identified by *TRRespass*. This suggests that the in-DRAM TRR implementation has evolved over time. We perform further experiments on $\mathcal{C}_{13}$ to confirm this hypothesis. We discover that, instead of performing a single targeted refresh during each regular refresh operation, the TRR mitigation employed by $\mathcal{C}_{13}$ performs *multiple* targeted refreshes during each regular refresh operation. While we can confirm that recent DRAM chips are still vulnerable to RowHammer, further research is required to better understand newer TRR mitigations and to find more effective fuzzing strategies against them.

B. Increasing the Refresh Rate

As mentioned in Section V-A, the memory controller issues a `REFRESH` command to the memory device every $7.8\,\mu s$, to ensure that all cells are refreshed within a $64\,ms$ interval. Doubling (or even quadrupling) the refresh rate (i.e., double-refresh) was proposed in the past [6], [7], [46], [51], [64] as an immediate countermeasure against RowHammer attacks, since doing so reduces the amount of time required for hammering. As discussed in Section IV, some server platforms employ double-refresh as default behavior or enable it when a non-TRR-compliant DRAM module is in use. This is usually not the case on consumer platforms.[6] However, `tREFI` can (sometimes) be set in the BIOS. Although double-refresh was demonstrated in the past to not fully eliminate the RowHammer vulnerability [7], [51], the introduction of in-DRAM TRR may have changed the situation. In fact, since TRR acts mainly at refresh time, doubling the refresh operations could improve TRR's security guarantees, enforcing the inhibitor operations more frequently. To test this hypothesis, we evaluate our modules against *TRRespass* when running them with double refresh.

Our experiment reveals the presence of new hammering patterns that are still able to trigger bit flips in three modules (as indicated in the rightmost column in Table II). This result further undermines the efficacy of double refresh as a stopgap solution against RowHammer even when in-DRAM TRR is deployed.

C. Repeatability of the Bit Flips

Repeatability is a fundamental factor in RowHammer exploitation. The ability to reliably trigger a bit flip repeatedly [51] is what made RowHammer so popular in adversarial scenarios [14], [25], [28], [71], [72], [79], [81], [89], [91], [96].

We study the repeatability of these many-sided RowHammer bit flips to better understand their properties. We pick one DRAM module per DRAM vendor ($\mathcal{A}_{14}$, $\mathcal{B}_1$, $\mathcal{C}_{13}$) and we run the best pattern for each module. When a bit flip occurs, we try to repeat it. Our experiment confirms that bit flips are repeatable in a reliable way for all the modules. However, it may require multiple attempts before obtaining the same bit flip again and sometimes we may observe many other spurious bit flips generated by the same pattern (see our analysis of Vendor $\mathcal{B}$ modules in Section VII-A). We discuss the implications of this phenomenon for the exploitation of these bit flips in Section VIII.

VIII. Exploitation with *TRRespass*

TRRespass generates many-sided RowHammer patterns to bypass TRR on modern DDR4 modules. While such access patterns are more sophisticated than standard RowHammer access patterns [27], [51], [88], we now show that their practical exploitability is not only possible, but also similar, in spirit, to existing state-of-the-art RowHammer attacks. For this purpose, we show how we craft many-sided RowHammer exploits using the general RowHammer exploitation framework used by prior work in the area [79]. The exploitability investigated by the framework revolves around three fundamental steps: (i) *Memory templating*, (ii) *Memory massaging*, and (iii) *Exploitation*.

Memory Templating. In this step, the attacker scans memory with RowHammer access patterns, looking for vulnerable memory pages (or *templates*) where one or more bits can be flipped at a specific offset. For templating to be successful, an attacker needs to use the desired patterns when accessing DRAM. Prior work has already demonstrated the feasibility of using double-sided RowHammer patterns using either huge (2MB) pages [79] or a variety of side channels to identify

[6] As we report in Table II, we occasionally detect double-refresh behavior on particular DRAM modules. This suggests that the memory controller may employ module-dependent mechanisms for RowHammer mitigation.

physically contiguous memory ranges [25], [37], [54], [91]. For many-sided RowHammer, we can use the former mechanism as long as we can fit all the aggressor rows in a single huge page (similar to double-sided RowHammer). This is possible for simple variants such as 3-sided RowHammer, but not for complex variants such as 19-sided (which may require two or more consecutive huge pages). However, many of these modules vulnerable to lengthy patterns are also vulnerable to a series of different other patterns (often shorter). Moreover, the results on LPDDR4(X), where we simply hammer random addresses belonging to the same bank, demonstrate that the location of the aggressors is not always a fundamental parameter—relaxing the assumptions for the attacker. In the case where only extended patterns (e.g., 19-sided) are effective or in the absence of huge pages in the system, we can still use a variety of page allocator side channels [25], [54], [91] or speculative side channels [37], [93] to locate a sufficiently large contiguous memory chunk to fit our many-sided RowHammer patterns and template memory.

Memory Massaging. Once vulnerable templates are available, the attacker needs to implement some form of memory massaging to lure the victim into mapping the target data onto one of the available templates. Any of the memory massaging techniques described in prior work still apply with no modifications to many-sided RowHammer, given that memory massaging is pattern-agnostic [14], [25], [27], [28], [79], [91], [92].

Exploitation. Once the target data is mapped onto the target template, the attacker needs to trigger the same RowHammer bit flips using the previously templated access patterns to complete the final exploitation step. For this step to be successful, the attacker needs to ensure that, with high probability, (i) the templated bit flips are repeatable, and (ii) there are no spurious (non-templated) bit flips in the victim page. Prior work has shown that these assumptions hold in practice for state-of-the-art attacks based on standard access patterns. Compared to such patterns, many-sided patterns incur similar (albeit lower) repeatability, as discussed in Section VII-C. In practice, this means the attacker may have to perform the access patterns multiple times for reliable exploitation. Moreover, to ensure there are no spurious bit flips across runs, the attacker can trivially mask irrelevant columns in the aggressor rows as shown in previous work [23], [33], [54] or otherwise use these bit flips as part of a compatible attack vector (e.g., corrupting multiple bits of a cryptographic key [79]).

Overall, *TRRespass*-based exploitation is very similar to existing RowHammer attacks. As shown in the next section, once effective many-sided access patterns are available, an attacker can reliably mount real-world RowHammer attacks on modern DDR4 systems in a matter of minutes.

A. Exploitation on DDR4

Armed with (repeatable) templates, we now study the effectiveness of different RowHammer exploits on modern DDR4 systems. To this end, we implement three example attacks: (i) the original RowHammer exploit targeting PTEs (*Page Table Entries*) to obtain kernel privileges from Seaborn and Dullien [81], (ii) the RSA exploit from Razavi et al. [79] that corrupts public keys to gain access to a co-hosted VM, and (iii) the *opcode flipping* exploit on the `sudo` binary from Gruss et al. [27]. The PTE exploit [81] takes advantage of bit flips on the *Page Frame Number* (PFN) to probabilistically redirect the virtual to physical mapping of an attacker-controlled page to another page table page. This relies on page table spraying to increase the probability of referencing another page table page with the corrupted PFN. The exploit from Gruss et al. [27] shows that it is possible to target code pages in the page cache to compromise opcodes and bypass permission checks on the `sudo` binary. Gruss et al. [27] report 29 vulnerable opcodes to use for this purpose. Razavi et al. [79] propose an attack to compromise an RSA public key stored in the page cache. They prove that causing a bit flip in the *modulus* of a 1024-bit or 2048-bit RSA public key makes the modulus factorizable with a probability of 12-22%. For our analysis, we target a 2048-bit RSA public key.

We assume an attacker capable of performing memory massaging—placing an exploitable target on one of the vulnerable memory pages—using any well-known technique [14], [25], [27], [28], [79], [91], [92]. Table IV presents our results for two sample modules for each vendor—the most and least vulnerable from the same manufacturer. As part of our analysis, we also record τ (i.e., time to template a single row), since many-sided RowHammer requires more time to carry out templating compared to previous RowHammer variants. As expected, we see a large discrepancy across the different modules, which matches the largely different number of bit flips reported in Table II. In the case of $\mathcal{B}$ modules, where *TRRespass* is able to generate very few bit flips, we are unable to reproduce any attack. On the other hand, on the other 4 modules from vendors $\mathcal{A}$ and $\mathcal{C}$ we can (overall) find templates to reproduce all the attacks. On $\mathcal{C}_{12}$, we can reproduce the PTE attack [81] in as little as $2.3\,s$, while the RSA-2048 exploit [79], when successful, can take up to $39\,m\,48\,s$ ($\mathcal{A}_4$). Bypassing `sudo` permission checks [27] turned out to be possible only on $\mathcal{C}_{12}$ in $54\,m\,16\,s$. Note that we assume existing templating strategies as is: we did not attempt to craft more sophisticated attacks, since our goal is solely to test existing RowHammer variants. Overall, our results show that RowHammer still presents a significant threat to the security of modern DDR4 systems, even in the presence of in-DRAM TRR mitigations.

TABLE IV: Time to exploit. Time to find the first exploitable template on two sample modules from each DRAM vendor.

Module	τ *(ms)*	*PTE* [81]	*RSA-2048* [79]	*sudo* [27]
$\mathcal{A}_{14}$	188.7	4.9s	6m 27s	—
$\mathcal{A}_4$	180.8	38.8s	39m 28s	—
$\mathcal{B}_1$	360.7	—	—	—
$\mathcal{B}_2$	331.2	—	—	—
$\mathcal{C}_{12}$	300.0	2.3s	74.6s	54m16s
$\mathcal{C}_{13}$	180.9	3h 15m	—	—

τ: Time to template a single row: time to fill the victim and aggressor rows + hammer time + time to scan the row.

IX. Related Work

RowHammer. In their seminal work, Kim et al. [51] are the first to rigorously introduce and characterize the RowHammer vulnerability. Following this work, a large number of attacks compromising a variety of different systems [5], [12], [13], [17], [24], [25], [28], [39], [77], [79], [81], [89], [91], [92], [96], [98] and characterization studies [22], [23], [27], [78], [88] emerged, as described in a recent retrospective article [72]. Prior works rely on three main classes of RowHammer patterns to induce bit flips: (i) single-sided, (ii) double-sided, and (iii) one-location RowHammer. None of these techniques are effective against modern DDR4 modules with in-DRAM RowHammer mitigations. Lanteigne [55], [56] proposes a technique (called regional RowHammer), where a small 2 MB region (e.g., a Linux hugepage) is hammered using multiple software threads to increase the DRAM row activation rate. In fact, we are not the first to use the term n-sided RowHammer for $n = 4$, as Lanteigne refers to his technique as *quad-sided* [55]. However, his technique does not provide a clear or methodical way of picking aggressor rows that are close to each other in a bank, and instead aims to maximize the number of row activations. We show that merely maximizing the number of activations is not sufficient to bypass in-DRAM RowHammer mitigations.

Software-based mitigations. Herath and Fogh [32] and Aweke et al. [7] suggested "hybrid" mitigations based on hardware performance counters to detect suspicious hammering-like activity. Other mitigations, such as CATT [16] and GuardION [92], try to enforce DRAM-based data isolation to prevent RowHammer attacks from corrupting sensitive data. Nevertheless, recent work has shown how these mitigations cannot stop more sophisticated attacks [25], [27]. With the correct DRAM mapping functions, ZebRAM [53] can protect the entire system by extending isolation to the entire DRAM. Unfortunately, ZebRAM becomes expensive when the active working set of an application is larger than half of DRAM capacity.

Hardware-based mitigations. Although doubling the refresh rate or using ECC memory are immediately-deployable solutions, they have proven insufficient to stop RowHammer [7], [23], [51]. Other hardware-based mitigation techniques have been proposed [62], [86], [87] but, to our knowledge, these have not been deployed in real systems. Kim et al. propose Probabilistic Adjacent Row Activation (PARA) [51], which is a low overhead mechanism to prevent RowHammer bit flips. When a row is activated, with a very small probability, PARA refreshes rows adjacent to the activated row. A variant of PARA, Hardware RHP, appears to be employed by some Intel memory controllers [36], [73], [90], [94]. This is a new RowHammer measure in the memory controller and its robustness is yet to be independently validated. In recent years, TRR has become the hardware-based RowHammer mitigation of choice, first deployed in the MC on DDR3 systems and then in-DRAM on DDR4. While DDR3 systems have been widely studied, only a few studies have reported RowHammer bit flips on DDR4 [27], [56], [66]. Compared to our analysis, such studies have induced bit flips on selected earlier-generation DDR4 modules. In contrast, we study several generations of DDR4 modules (including the most-recent off-the-shelf devices) and find that, while standard access patterns are no longer effective, new many-sided RowHammer patterns can still induce bit flips on many TRR-protected DDR4 modules in the market today.

X. Conclusion

This paper shows that, despite significant mitigation efforts, modern DDR4 DRAM systems are still vulnerable to RowHammer bit flips—and even more vulnerable than DDR3 DRAM systems, once the mitigations are bypassed. In particular, we demonstrate that *Target Row Refresh* (TRR), publicized by CPU and DRAM vendors as the definitive solution to RowHammer, can be bypassed to cause RowHammer bit flips. First, we show that TRR is an umbrella term for a variety of mitigations deployed at the memory controller or in DRAM chips. Second, we analyze common TRR implementations in the memory controller (using timing side channels) and in DRAM chips (using an FPGA-based memory controller, SoftMC). Our analysis shows that the consumer CPUs we test rely on in-DRAM TRR to mitigate the RowHammer vulnerability and do not employ TRR at the memory controller level. We discover that modern (in-DRAM) TRR implementations are generally vulnerable to *many-sided RowHammer*, a new hammering strategy that hammers *many* (i.e., at least 3) aggressor rows concurrently. Finally, we present *TRRespass*, a black-box many-sided RowHammer fuzzer that, unaware of the implementation of the memory controller or the DRAM chip, can still find sophisticated hammering patterns to mount real-world attacks for many of the DDR4 DRAM modules in the market. Our results provide evidence that the pursuit of effective RowHammer mitigations must continue and that the *security by obscurity* strategy of DRAM vendors puts computing systems at risk for extended periods of time.

Disclosure

We disclosed our new RowHammer attacks to all affected parties in November of 2019. This triggered an industry-wide effort in addressing the issues raised in this paper. Unfortunately, due to the nature of these vulnerabilities, it will take time before effective mitigations will be in place. Further developments on these vulnerabilities are tracked under CVE-2020-10255. The paper remained confidential until the public disclosure date of March 10, 2020.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback and Robin Webbers for helping us in our analysis of LPDDR4(X) systems. This work was supported by the European Union's Horizon 2020 research and innovation programme under grant agreements No. 786669 (ReAct) and No. 825377 (UNICORE), by Intel Corporation through the Side Channel Vulnerability ISRA, and by the Netherlands Organisation for Scientific Research through grants NWO 639.023.309 VICI "Dowsing", NWO 639.021.753 VENI "PantaRhei", and NWO 016.Veni.192.262. This paper reflects only the authors' view. The funding agencies are not responsible for any use that may be made of the information it contains.

References

[1] "DRAM Chip Market Share by Manufacturer Worldwide from 2011 to 2019," https://www.statista.com/statistics/271726/global-market-share-held-by-dram-chip-vendors-since-2010, 2019.

[2] "RAMBleed DRAM Vulnerabilities," https://blogs.oracle.com/security/rambleed, 2019.

[3] "Researchers Use RowHammer Bit Flips to Steal 2048-bit Crypto Key," https://arstechnica.com/information-technology/2019/06/researchers-use-rowhammer-bitflips-to-steal-2048-bit-crypto-key/, 2019.

[4] Advanced Micro Devices, "AMD Generic Encapsulated Software Architecture (AGESA$^{\text{TM}}$) Interface Specification for Arch2008," 2017.

[5] M. T. Aga *et al.*, "When Good Protections Go Bad: Exploiting Anti-DoS Measures to Accelerate Rowhammer Attacks," in *HOST*, 2017.

[6] Apple Inc., “About the Security Content of Mac EFI Security Update 2015-001,” https://support.apple.com/en-us/HT204934, june 2015.

[7] Z. B. Aweke *et al.*, “ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks,” in *ASPLOS*, 2016.

[8] K. S. Bains and J. B. Halbert, “Distributed row hammer tracking,” US Patent 9 299 400B2, 2016.

[9] K. S. Bains *et al.*, “Row hammer refresh command,” US Patent 9 236 110B2, 2016.

[10] K. S. Bains *et al.*, “Method, apparatus and system for providing a memory refresh,” US Patent 9 030 903B2, 2015.

[11] A. Barenghi *et al.*, “Software-Only Reverse Engineering of Physical DRAM Mappings for RowHammer Attacks,” in *IVSW*, 2018.

[12] S. Bhattacharya and D. Mukhopadhyay, “Curious Case of Rowhammer: Flipping Secret Exponent Bits using Timing Analysis,” in *CHES*, 2016.

[13] S. Bhattacharya and D. Mukhopadhyay, “Advanced Fault Attacks in Software: Exploiting the RowHammer Bug,” in *Fault Tolerant Architectures for Cryptography and Hardware Security*, 2018.

[14] E. Bosman *et al.*, “Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector,” in *S&P*, 2016.

[15] K. M. Brandl, “Data processor with memory controller for high reliability operation and method,” US Patent 9 281 046B2, 2016.

[16] F. Brasser *et al.*, “CAn’t Touch This: Software-only Mitigation against Rowhammer Attacks targeting Kernel Memory,” in *USENIX Sec.*, 2017.

[17] S. Carre *et al.*, “OpenSSL Bellcore’s Protection Helps Fault Attack,” in *DSD*, 2018.

[18] K. K. Chang *et al.*, “Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization,” in *SIGMETRICS*, 2016.

[19] K. K. Chang *et al.*, “Improving DRAM Performance by Parallelizing Refreshes with Accesses,” in *HPCA*, 2014.

[20] K. K. Chang *et al.*, “Low-cost Inter-linked Subarrays (LISA): Enabling Fast Inter-subarray Data Movement in DRAM,” in *HPCA*, 2016.

[21] K. K. Chang *et al.*, “Understanding Reduced-Voltage Operation in Modern DRAM Devices: Experimental Characterization, Analysis, and Mechanisms,” in *SIGMETRICS*, 2017.

[22] L. Cojocar *et al.*, “Are We Susceptible to Rowhammer? An End-to-End Methodology for Cloud Providers,” in *S&P*, 2020.

[23] L. Cojocar *et al.*, “Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks,” in *S&P*, 2019.

[24] A. P. Fournaris *et al.*, “Exploiting Hardware Vulnerabilities to Attack Embedded System Devices: A Survey of Potent Microarchitectural Attacks,” *Electronics*, 2017.

[25] P. Frigo *et al.*, “Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU,” in *S&P*, 2018.

[26] Z. Greenfield *et al.*, “Method, apparatus and system for determining a count of accesses to a row of memory,” US Patent 20 140 085 995A1, 2014.

[27] D. Gruss *et al.*, “Another Flip in the Wall of Rowhammer Defenses,” in *S&P*, 2018.

[28] D. Gruss *et al.*, “Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript,” in *DIMVA*, 2016.

[29] H. Hassan *et al.*, “CROW: A Low-Cost Substrate for Improving DRAM Performance, Energy Efficiency, and Reliability,” in *ISCA*, 2019.

[30] H. Hassan *et al.*, “ChargeCache: Reducing DRAM Latency by Exploiting Row Access Locality,” in *HPCA*, 2016.

[31] H. Hassan *et al.*, “SoftMC: A Flexible and Practical Open-Source Infrastructure for Enabling Experimental DRAM Studies,” in *HPCA*, 2017.

[32] N. Herath and Anders Fogh, “These are Not Your Grand Daddy’s CPU Performance Counters,” in *Black Hat Briefings*, 2015.

[33] S. Hong *et al.*, “Terminal Brain Damage: Exposing the Graceless Degradation in Deep Neural Networks Under Hardware Fault Attacks,” in *USENIX Sec.*, 2019.

[34] D. Hwa Hong, “Smart Refresh Device,” US Patent 9 311 984B1, 2016.

[35] Intel Corp., “Intel® Xeon® Processor E5 v4 Product Family,” 2016.

[36] Intel Corp., “CannonLake Intel Firmware Support Package (FSP) Integration Guide,” https://usermanual.wiki/Pdf/CannonLakeFSPIntegrationGuide.58784693.pdf, 2017.

[37] S. Islam *et al.*, “SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks,” *arXiv preprint 1903.00446*, 2019.

[38] Y. Ito and Y. He, “Semiconductor Device,” US Patent 9 805 783B2, 2017.

[39] Y. Jang *et al.*, “SGX-Bomb: Locking Down the Processor via RowHammer Attack,” in *SysTEX*, 2017.

[40] JEDEC, “JESD209-4, LPDDR4 Specification,” 2014.

[41] JEDEC, “SPD Annex K - Serial Presence Detect (SPD) for DDR3 SDRAM Modules, v6,” 2014.

[42] JEDEC, “SPD Annex L - Serial Presence Detect (SPD) for DDR4 SDRAM Modules, v3,” 2015.

[43] JEDEC, “JESD209-4, LPDDR4X Specification,” 2017.

[44] JEDEC, “JESD79-4B, DDR4 Specification,” 2017.

[45] B. I. Jung *et al.*, “Memory Device, Memory System, and Operating Methods thereof,” US Patent 9 257 169B2, 2016.

[46] M. Kaczmarski, “Thoughts on Intel® Xeon® E5-2600 v2 Product Family Performance Optimisation – component selection guidelines,” 2014.

[47] O. D. Kahn and J. R. Wilcox, “Method for Dynamically Adjusting a Memory Page Closing Policy,” US Patent 6 799 241, 2004.

[48] S. Khan *et al.*, “PARBOR: An Efficient System-Level Technique to Detect Data-Dependent Failures in DRAM,” in *DSN*, 2016.

[49] D. S. Kim and J. I. Kim, “Refresh control device and semiconductor device including the same,” US Patent 9 818 469B1, 2017.

[50] Y. Kim *et al.*, “A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM,” in *ISCA*, 2012.

[51] Y. Kim *et al.*, “Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors,” in *ISCA*, 2014.

[52] Y. Kim *et al.*, “ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers,” in *HPCA*, 2010.

[53] R. K. Konoth *et al.*, “ZebRAM: Comprehensive and Compatible Software Protection Against Rowhammer Attacks,” in *OSDI*, 2018.

[54] A. Kwong *et al.*, “RAMBleed: Reading Bits in Memory Without Accessing Them,” in *S&P*, 2020.

[55] M. Lanteigne, “A Tale of Two Hammers: A Brief Rowhammer Rowhammer Analysis of AMD vs. Intel.” ThirdIO Inc., 2016.

[56] M. Lanteigne, “How Rowhammer Could Be Used to Exploit Weaknesses in Computer Hardware,” in *SEMICON*, 2016.

[57] D. Lee *et al.*, “Adaptive-Latency DRAM: Optimizing DRAM Timing for the Common-Case,” in *HPCA*, 2015.

[58] D. Lee *et al.*, “Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture,” in *HPCA*, 2013.

[59] D. Lee *et al.*, “Simultaneous Multi-Layer Access: Improving 3D-Stacked Memory Bandwidth at Low Cost,” *TACO*, 2016.

[60] D. Lee *et al.*, “Design-Induced Latency Variation in Modern DRAM Chips: Characterization, Analysis, and Latency Reduction Mechanisms,” in *SIGMETRICS*, 2017.

[61] D. Lee *et al.*, “Decoupled Direct Memory Access: Isolating CPU and IO Traffic by Leveraging a Dual-Data-Port DRAM,” in *PACT*, 2015.

[62] E. Lee *et al.*, “TWiCe: Preventing Row-Hammering by Exploiting Time Window Counters,” in *ISCA*, 2019.

[63] J.-B. Lee, “Green Memory Solution,” in *Samsung Electronics, Investor’s Forum*, 2014.

[64] Lenovo, “Row Hammer Privilege Escalation,” https://support.lenovo.com/us/en/product_security/row_hammer, March 2015.

[65] J. Lin, “Handling Maximum Activation Count limit and Target Row Refresh in DDR4 SDRAM,” US Patent 9 589 606B2, 2017.

[66] M. Lipp *et al.*, “Nethammer: Inducing Rowhammer Faults Through Network Requests,” *arXiv preprint 1805.04956*, 2018.

[67] J. Liu *et al.*, “RAIDR: Retention-Aware Intelligent DRAM Refresh,” in *ISCA*, 2012.

[68] J. Liu *et al.*, “An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms,” in *ISCA*, 2013.

[69] M. Majkowski, “Every 7.8μs your computer’s memory has a hiccup,” https://blog.cloudflare.com/every-7-8us-your-computers-memory-has-a-hiccup/, 2018.

[70] Micron, “DDR4 SDRAM Datasheet,” p. 380, 2016.

[71] O. Mutlu, “The RowHammer Problem and Other Issues We May Face as Memory Becomes Denser,” in *DATE*, 2017.

[72] O. Mutlu and J. S. Kim, “RowHammer: A Retrospective,” *TCAD*, 2019.

[73] Omron, “NY-series Industrial Box PC - Hardware User’s Manual,” https://assets.omron.eu/downloads/manual/en/v6/w553_ny-series_industrial_box_pc_users_manual_en.pdf, 2019.

[74] J.-B. Park, “Memory and Memory System including the same,” US Patent 9 396 786B2, 2016.

[75] M. S. Park, “Memory Device to Alleviate the Effects of Row Hammer Condition and Memory System Including the Same,” US Patent 9 685 240B1, 2017.

[76] P. Pessl *et al.*, “DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks.” in *USENIX Sec.*, 2016.

[77] D. Poddebniak *et al.*, “Attacking Deterministic Signature Schemes Using Fault Attacks,” in *EuroS&P*, 2018.

[78] R. Qiao and M. Seaborn, “A New Approach for Rowhammer Attacks,” in *HOST*, 2016.

[79] K. Razavi *et al.*, “Flip Feng Shui: Hammering a Needle in the Software Stack,” in *USENIX Sec.*, 2016.

[80] SAFARI Research Group, “SoftMC — GitHub Repository,” https://github.com/CMU-SAFARI/SoftMC.

[81] M. Seaborn and T. Dullien, “Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges,” in *Black Hat USA*, 2015.

[82] V. Seshadri *et al.*, “RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization,” in *MICRO*, 2013.

[83] V. Seshadri *et al.*, “Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology,” in *MICRO*, 2017.

[84] V. Seshadri *et al.*, “Gather-Scatter DRAM: In-DRAM Address Translation to Improve the Spatial Locality of Non-Unit Strided Accesses,” in *MICRO*, 2015.

[85] V. Seshadri and O. Mutlu, “In-DRAM Bulk Bitwise Execution Engine,” *arXiv:1905.09822*, 2019.

[86] S. M. Seyedzadeh *et al.*, “Counter-Based Tree Structure for Row Hammering Mitigation in DRAM,” *IEEE CAL*, 2017.

[87] M. Son *et al.*, “Making DRAM Stronger Against Row Hammering,” in *DAC*, 2017.

[88] A. Tatar *et al.*, “Defeating Software Mitigations against Rowhammer: A Surgical Precision Hammer,” in *RAID*, 2018.

[89] A. Tatar *et al.*, “Throwhammer: Rowhammer Attacks over the Network and Defenses,” in *USENIX ATC*, 2018.

[90] TQ-Systems, “TQMx80UC User’s Manual,” https://www.tq-group.com/filedownloads/files/products/embedded/manuals/x86/embedded-modul/COM-Express-Compact/TQMx80UC/TQMx80UC.UM.0102.pdf, 2020.

[91] V. van der Veen *et al.*, “Drammer: Deterministic Rowhammer Attacks on Mobile Platforms,” in *CCS*, 2016.

[92] V. van der Veen *et al.*, “GuardION: Practical mitigation of DMA-based rowhammer attacks on ARM,” in *DIMVA*, 2018.

[93] S. van Schaik *et al.*, “RIDL: Rogue in-flight data load,” in *S&P*, 2019.

[94] VersaLogic Corp., “Blackbird BIOS Reference Manual,” https://www.versalogic.com/wp-content/themes/vsl-new/assets/pdf/manuals/MEPU_4462_4562_BRM.pdf, 2019.

[95] G. D. Wolff, “Apparatuses and methods for distributing row hammer refresh events across a memory device,” US Patent 20 180 218 767A1, 2018.

[96] Y. Xiao *et al.*, “One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation.” in *USENIX Sec.*, 2016.

[97] T. Zhang *et al.*, “Half-DRAM: A High-bandwidth and Low-power DRAM Architecture from the Rethinking of Fine-grained Activation,” in *ISCA*, 2014.

[98] Z. Zhang *et al.*, “Triggering Rowhammer Hardware Faults on ARM: A Revisit,” in *ASHES*, 2018.

APPENDIX A
TRR-COMPLIANT MEMORY

In Section IV, we define TRR-compliant memory. Here we expand on this concept, also explaining the difference between TRR-compliant and pTRR-compliant memory.

The `MAC` field is a field of one byte located at byte 41 on the SPD of a DDR3 module [41] and byte 7 on the SPD of a DDR4 module [42]. This field reports information about the module’s resiliency to RowHammer. In the single byte allocated to the `MAC` value inside the SPD [41], [42], only the 6 least significant bits are used to store information about the module’s limits in the form of `MAC` and `tMAW` (Figure 15), where `MAC` is the *Maximum Activate Count* and `tMAW` is the *Maximum Activate Window*, which simply acts as a multiplier for `MAC` (Figure 15). The remaining two most significant bits are flagged as reserved. As we mention in Section IV the `MAC` value can take three configurations:

- *unlimited*, as value `0b1000`;
- *untested*, as value `0b0000`; or
- discrete values from *200K* to *700K* with steppings of $+100K$—values `0b0001` to `0b0110`.

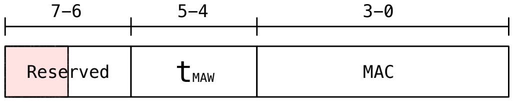


Fig. 15: SPD’s `MAC` field. Bit 7 needs to be set in order to enable pTRR [46].

In one of our early experiments, we discovered that our definition of TRR-compliant modules slightly diverges from Intel’s definition of pTRR-compliant modules [46]. In fact, we discovered that in order to enable pTRR, bit 7 (one of the reserved bits) needs to be set. If not, regardless of the `MAC` and `tMAW` values, the system treats the module as non-compliant. This is likely a legacy feature which stems from the fact that pTRR [46] was introduced before TRR became part of the JEDEC standard [41].

APPENDIX B
TRRespass-ING PATTERNS

In Section VI-B, we explain the new n-sided hammering patterns we use in our experiments. We now provide a more general definition of these hammering patterns.

TRRespass randomizes two parameters: *cardinality* and *distance*. Cardinality and distance together define a novel hammering pattern that we refer to as $\langle n$-sided $\mid$ dist=$d\rangle$ RowHammer. The pattern consists of $\frac{n}{2}$ *pairs* of aggressor rows, where the two aggressor rows in each pair are placed one victim row apart (similar to double-sided RowHammer). The distance d defines the number of rows between the aggressor row pairs. For example, the ⟨4-sided | dist=3⟩ pattern contains two aggressor row pairs (four aggressor rows in total), and the two aggressor row pairs are three rows apart from each other. The n-sided pattern, which we refer to throughout the paper, is another example, where the distance between the aggressor row pairs is one row.

Figure 16 shows the number of bit flips that occur in module $\mathcal{A}_{10}$ when we use the ⟨10-sided | dist=D⟩ hammering pattern while sweeping the parameter D. We note that the number of bit flips increases and decreases as we vary D, reaching its maximum at $D = 12$. This observation confirms that the distance between aggressor row pairs has a primary role in assembling an effective hammering pattern.

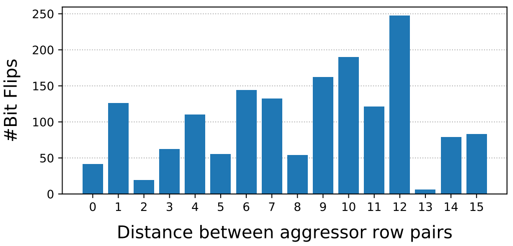


Fig. 16: Bit flips induced by ⟨10-sided | dist=D⟩ RowHammer-pattern as a function of D. X-axis plots the distance between each aggressor row pair. Y-axis reports the number of unique bit flips.