arXiv:2306.17061v5 [cs.CR] 28 Mar 2024

RowPress: Amplifying Read Disturbance in Modern DRAM Chips

Haocong Luo Ataberk Olgun A. Giray Yağlıkçı Yahya Can Tuğrul Steve Rhyner
Meryem Banu Cavlak Joël Lindegger Mohammad Sadrosadati Onur Mutlu

ETH Zürich

Abstract

Memory isolation is critical for system reliability, security, and safety. Unfortunately, read disturbance can break memory isolation in modern DRAM chips. For example, RowHammer is a well-studied read-disturb phenomenon where repeatedly opening and closing (i.e., hammering) a DRAM row *many times* causes bitflips in physically nearby rows.

This paper experimentally demonstrates and analyzes another widespread read-disturb phenomenon, RowPress, in real DDR4 DRAM chips. RowPress breaks memory isolation by keeping a DRAM row open for a long period of time, which disturbs physically nearby rows enough to cause bitflips. We show that RowPress amplifies DRAM's vulnerability to read-disturb attacks by significantly reducing the number of row activations needed to induce a bitflip by one to two orders of magnitude under realistic conditions. In extreme cases, RowPress induces bitflips in a DRAM row when an adjacent row is activated *only once*. Our detailed characterization of 164 real DDR4 DRAM chips shows that RowPress 1) affects chips from all three major DRAM manufacturers, 2) gets worse as DRAM technology scales down to smaller node sizes, and 3) affects a different set of DRAM cells from RowHammer and behaves differently from RowHammer as temperature and access pattern changes. We also show that cells vulnerable to RowPress are very different from cells vulnerable to retention failures.

We demonstrate in a real DDR4-based system with RowHammer protection that 1) a user-level program induces bitflips by leveraging RowPress while conventional RowHammer cannot do so, and 2) a memory controller that adaptively keeps the DRAM row open for a longer period of time based on access pattern can facilitate RowPress-based attacks. To prevent bitflips due to RowPress, we describe and analyze four potential mitigation techniques, including a new methodology that adapts existing RowHammer mitigation techniques to also mitigate RowPress with low *additional* performance overhead. We evaluate this methodology and demonstrate that it is effective on a variety of workloads. We open source all our code and data to facilitate future research on RowPress.

1 Introduction

To ensure system reliability, security, and safety, it is critical to maintain memory isolation: accessing a memory address should not cause unintended side-effects on data stored in other addresses. Unfortunately, with aggressive technology node scaling, dynamic random access memory (DRAM) [24], the prevalent main memory technology, suffers from increased *read disturbance*: accessing (reading) a DRAM cell disturbs the operational characteristics (e.g., stored charge) of other physically close DRAM cells.

RowHammer is an example read-disturb phenomenon where repeatedly opening and closing (i.e., hammering) a DRAM row (called aggressor row) *many times* (e.g., tens of thousands times) can cause bitflips in physically nearby rows (called victim rows) [67, 68].

RowHammer is a critical security vulnerability as attackers can induce and exploit the bitflips to take over a system or leak private or security-critical data [1, 10, 12–20, 22, 28, 30–32, 38–40, 43, 45, 52, 53, 58, 68, 72, 74, 81, 84, 94, 95, 102, 112–114, 117, 118, 124, 131, 132, 145, 146, 148, 149, 153, 154, 156, 160, 170, 173–176]. Prior works [67, 68] experimentally demonstrate that RowHammer significantly worsens as DRAM manufacturing technology scales to smaller nodes. For example, the minimum number of total aggressor row activations to cause at least one bitflip (AC_{min}) has reduced by 14× in less than a decade [67]. To ensure reliable, secure, and safe operation in modern and future DRAM-based systems, it is critical to develop a rigorous understanding of read disturbance effects like RowHammer.

In this paper, we experimentally demonstrate another widespread read-disturb phenomenon, *RowPress*, in real DDR4 DRAM chips. We show that keeping a DRAM row (i.e., aggressor row) open for a long period of time (i.e., a large aggressor row on time, t_{AggON}) disturbs physically nearby DRAM rows.[1] Doing so induces bitflips in the victim row *without* requiring (tens of) thousands of activations to the aggressor row. We characterize RowPress in 164 off-the-shelf DDR4 DRAM chips from all three major manufacturers, and find that RowPress significantly amplifies DRAM's vulnerability to read-disturb attacks (i.e., greatly reduces the minimum number of total aggressor row activations to cause at least one bitflip, AC_{min}).

To illustrate this, Fig. 1 shows the distribution of AC_{min} (y-axis) we measure in 164 DRAM chips across all three major DRAM manufacturers when the aggressor row stays open as much as t_{AggON} (x-axis) between consecutive activations at 80 °C with one (single-sided) and two (double-sided) aggressor row(s) in a box-and-whiskers plot.[2] We study the single- and double-sided RowPress access patterns in detail in §5.2.

The two leftmost boxes in each plot shows the distribution of AC_{min} for the conventional single-sided (orange) and double-sided (blue) RowHammer pattern, where the aggressor row is open for the minimum amount of time ($t_{AggON} = t_{RAS} = 36ns$)[3] allowed by the DRAM specification [56], as done in conventional RowHammer

[1] The industry is aware that keeping a DRAM row open for a long period of time can cause read disturbance: Micron mentions "RAS Clobber" in two earlier patents [50, 158], while Samsung calls this "Passing Gate Effect" in a very recent work placed on arXiv while our paper has been under review [46]. We name this phenomenon "RowPress", which we believe is an intuitive name that immediately shows the difference compared to RowHammer in a figurative way: we "press" (i.e., keep open for a long period of time) instead of "hammer" (i.e., repeatedly open and close) the row.

[2] The box is lower-bounded by the first quartile (i.e., the median of the first half of the ordered set of data points) and upper-bounded by the third quartile (i.e., the median of the second half of the ordered set of data points). The interquartile range (IQR) is the distance between the first and third quartiles (i.e., box size). Whiskers show the minimum and maximum values.

[3] Manufacturer-recommended minimum row open time (t_{RAS}) ranges from 32 ns to 35 ns in DDR4 [56]. We use a 36 ns minimum t_{AggON} 1) to cover the whole range of

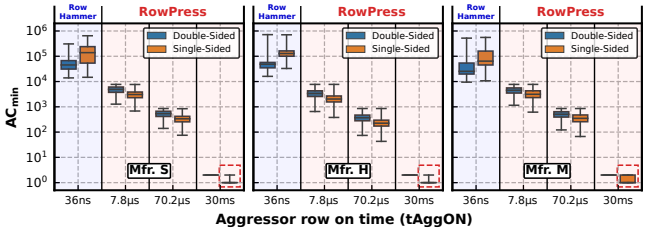

Figure 1: AC_{min} distributions of conventional RowHammer (RH) and three representative cases of RowPress (RP) at $80^{\circ}C$ across 164 DDR4 chips from manufacturers S, H, and M.

attacks [1, 10, 12–20, 22, 28, 30–32, 38–40, 43, 45, 52, 53, 58, 68, 72, 74, 81, 84, 94, 95, 102, 112–114, 117, 118, 124, 131, 132, 145, 146, 148, 149, 153, 154, 156, 160, 170, 173–176]. We observe that as t_{AggON} increases, compared to the most effective RowHammer pattern, the most effective RowPress pattern reduces AC_{min} 1) by 17.6× on average (up to 40.7×) when t_{AggON} is as large as the refresh interval (7.8 µs)[4], 2) by 159.4× on average (up to 363.8×) when t_{AggON} is 70.2 µs, the maximum allowed t_{AggON} [56], and 3) down to *only one* activation for an extreme t_{AggON} of 30 ms (highlighted by dashed red boxes).

Our detailed characterization results and sensitivity studies suggest that RowPress has a different underlying error mechanism compared to the RowHammer phenomenon in DRAM [67, 68, 94, 95, 98, 106, 107, 155, 164, 169]. We experimentally demonstrate that 1) only less than 0.013% of the DRAM cells that exhibit RowPress bitflips also exhibit RowHammer bitflips (§4.3), and 2) RowPress behaves very differently from RowHammer with temperature (§5.1) and access pattern (§5.2) changes. We also show detailed results demonstrating that cells vulnerable to RowPress are very different from cells vulnerable to retention failures (only less than 0.34% overlap).

We demonstrate that a user-level program can induce RowPress bitflips in a real DDR4-based system that already employs RowHammer protection. The program accesses *multiple different* columns of the aggressor DRAM row so that the memory controller keeps the aggressor row open for a longer period of time to serve these accesses. As a result, the program exercises RowPress and induces bitflips, while conventional RowHammer cannot, in the presence of in-DRAM RowHammer mitigation mechanisms (§6). We believe this program can be the basis of a proof-of-concept RowPress attack.

Our characterization results suggest that DRAM-based systems need to take RowPress into account to maintain the fundamental security/safety/reliability property of memory isolation. Based on our findings, we discuss and evaluate the implications of RowPress on existing read-disturb mitigation mechanisms that consider *only* RowHammer. We propose a methodology to adapt RowHammer mitigation techniques to also mitigate RowPress with low *additional* performance overhead by both 1) limiting the *maximum row-open time*, and 2) configuring the RowHammer defense to account for the RowPress-induced reduction in AC_{min}. We experimentally demonstrate that by applying our proposed methodology to two major techniques (PARA [68] and Graphene [109]), we can mitigate both RowHammer and RowPress with an average (maximum) *additional* slowdown of only 3.6% (13.1%) and −0.63% (4.6%), respectively.

We make the following contributions in this paper:

- To our knowledge, this is the first work to experimentally demonstrate the RowPress phenomenon and its widespread existence in real DDR4 DRAM chips from all three major manufacturers.
- We provide an extensive characterization of RowPress on 164 real DRAM chips. Our results show that RowPress 1) significantly amplifies DRAM's vulnerability to read-disturb attacks, 2) gets worse as DRAM technology scales down, and 3) is very different from RowHammer and retention failures in terms of the DRAM cells it affects and in the way it behaves as temperature and access pattern changes.
- We demonstrate that a simple user-level program induces RowPress bitflips on a real DDR4-based system, while a state-of-the-art RowHammer program cannot.
- We describe, analyze, and evaluate four potential ways to mitigate read-disturb attacks exploiting RowPress. We introduce a methodology to adapt existing RowHammer mitigation techniques to also mitigate RowPress with low *additional* performance overhead.
- We open-source [125] all our infrastructure, test programs, and raw data to enable 1) reproduction and replication of our results, and 2) further research on RowPress.

2 Background & Motivation

We provide a high-level introduction to DRAM organization (§2.1), major DRAM operations (§2.2), DRAM timing parameters involved in this work (§2.3), and read-disturb mechanisms in DRAM (§2.4).

2.1 DRAM Organization

Fig. 2 shows the hierarchical organization of modern DRAM-based main memory. The CPU's *memory controller* communicates with a *DRAM module* over a *memory channel*. A module contains one or multiple *DRAM ranks* that share the memory channel. A rank is made up of multiple *DRAM chips* that are operated in a lock-step manner (i.e., all chips receive and process the same command at the same time). Each DRAM chip contains multiple *DRAM banks* ❶ that can be accessed independently.

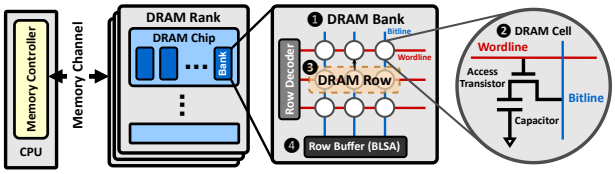

Figure 2: Hierarchical organization of modern DRAM.

Inside a DRAM bank, *DRAM cells* are organized into a two-dimensional array, addressed by rows and columns. A DRAM cell ❷ consists of 1) a capacitor, which stores one bit of information in the

t_{RAS} values and 2) due to the limited DRAM command bus frequency of our testing infrastructure (i.e., we can only send a DRAM command at every 1.5 ns) [101].

[4] *Refresh interval* is the time interval between two consecutive refresh commands that a DRAM row can be kept open [54, 56].

form of electrical charge level, and 2) an access transistor, which connects the capacitor to a bitline, controlled by a wordline. When the row decoder (including wordline drivers) drives a wordline high, the access transistors of all DRAM cells in the row ❸ are enabled, electrically connecting each cell in the row to its corresponding bitline. DRAM cells in the same column share a bitline, which is used to read from and write to the cells via the row buffer ❹ (which contains bitline sense amplifiers, BLSA).

2.2 Major DRAM Operations

DRAM Access. Accessing DRAM consists of three steps. First, the memory controller issues an ACT (activate) command together with a row address to the bank. The row decoder drives the wordline of that row to open the row (i.e., enables the access transistors). Data is then transferred from the DRAM cells in the row to the row buffer through the bitlines. Second, once the data is in the row buffer, the memory controller can send RD/WR commands to read/write data from/to the opened row. Third, the memory controller sends a PRE (precharge) command to close the opened row before accessing another row in the same bank.

DRAM Refresh. DRAM cells lose charge over time, risking *retention failure* induced bitflips if their charge is not restored in time. To avoid this, the memory controller periodically restores each DRAM row's charge levels by sending REF (refresh) commands. Before issuing a REF command, the memory controller must send a PRE command to close any open row to prepare the bank for refresh.

2.3 Key DRAM Timing Parameters

To guarantee correct operation, the memory controller must time DRAM commands according to certain *timing parameters* [54–57]. Fig. 3 shows a timeline of the key DRAM access operations. We describe four key timing parameters involved in this work: 1) t_{RAS}, 2) t_{RP}, 3) t_{REFI}, and 4) t_{REFW}.

t_{RAS} is the minimum time between opening a row with an ACT command and closing the row with a PRE command (❶ in Fig. 3). t_{RP} is the minimum time between sending a PRE command and opening a row with an ACT command (❷ in Fig. 3). t_{REFI} is the default time interval between consecutive REF commands. t_{REFW} is the maximum time window between two refresh operations that target the same row.

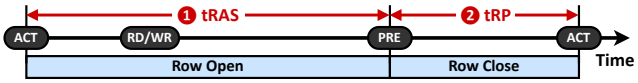

Figure 3: Timeline of key DRAM access operations.

A majority of DRAM timing parameters define lower bounds for the time intervals between pairs of DRAM commands. For example, t_{RAS} is the *minimum* amount of time that the memory controller has to wait before issuing a PRE command to close an open(ed) DRAM row. The memory controller may keep the DRAM row open *longer* than t_{RAS} to serve more RD/WR commands (in anticipation of future requests to the same row [96, 97, 119, 177]), depending on the memory controller's implementation and the workload's access pattern. In general, if the memory controller does *not* postpone REF commands, a DRAM row can be open for a duration of t_{REFI} before it has to be closed to serve a REF command. Otherwise, a DRAM row can be open for up to 9× t_{REFI} because the JEDEC DDR4 standard [56] allows postponing up to eight REF commands. Under normal operating conditions (i.e., within the temperature range of $0°C$ to $85°C$), t_{REFI} is 7.8 µs for commodity DDR4 chips.

2.4 Motivation

There are three major causes of bitflips in DRAM cell arrays: 1) soft errors caused by charged and/or energetic particle strikes [11, 75, 90, 100], 2) data retention failures due to the volatile and leaky nature of DRAM cells [63, 64, 82, 83, 111], and 3) read disturbance (e.g., RowHammer [2, 18, 34, 62, 67, 68, 79, 80, 98, 99, 102, 103, 106, 107, 121, 155, 164, 169, 172]) caused by undesirable interactions between circuit components. Both retention failures and RowHammer get worse as DRAM technology scales down to smaller node sizes.

Read disturbance has significant implications for system reliability, security, and safety because it is a widespread issue and can be exploited to break memory isolation [1, 10, 12–20, 22, 28, 30–32, 38–40, 43, 45, 52, 53, 58, 68, 72, 74, 81, 84, 94, 95, 102, 112–114, 117, 118, 124, 131, 132, 145, 146, 148, 149, 153, 154, 156, 160, 170, 173–176]. Therefore, it is important to identify and understand read disturbance mechanisms in DRAM. **Our goal** is to 1) rigorously and comprehensively characterize and investigate the read disturbance caused by increased aggressor row on time (t_{AggON}), and 2) understand its implications for secure, reliable, and safe operation of DRAM-based systems.

3 Methodology

We describe our DRAM testing infrastructure and the real DDR4 DRAM chips tested. We explain the methodology of our characterization experiments in their respective sections (under §4).

3.1 DRAM Testing Infrastructure

We test commodity DDR4 DRAM chips using an FPGA-based DRAM testing infrastructure that consists of four main components (as Fig. 4 illustrates): 1) a host machine that generates the test program and collects experiment results, 2) an FPGA development board (Xilinx Alveo U200 [161]), programmed with DRAM Bender [101, 122] (based on SoftMC [44, 126]), to execute our test programs, 3) a thermocouple temperature sensor and a pair of heater pads pressed against the DRAM chips to maintain a target temperature level, and 4) a PID temperature controller (MaxWell FT200 [89]) that controls the heaters and keeps the temperature at the desired level.

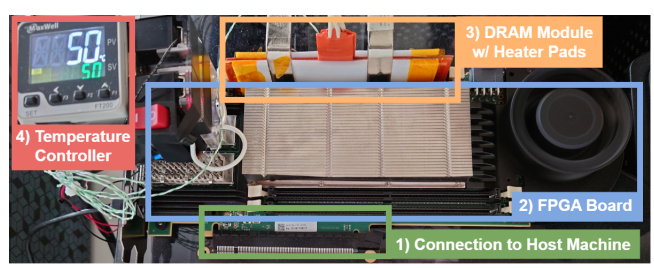

Figure 4: Our DDR4 DRAM testing infrastructure.

Disabling Interference Sources. To observe RowPress' effects at the circuit level, we disable potential sources of interference following a methodology similar to prior works [43, 67, 103, 164]. First, we disable periodic refresh during the execution of our test programs to 1) keep the timings of our test programs precise and 2) disable any existing on-die RowHammer defense mechanisms (e.g., TRR) [32, 43] so as to observe the DRAM chip's fundamental read disturbance behavior at the circuit level. Second, we bound our test programs' execution time strictly within a refresh window (i.e., 64ms t_{REFW}) of the tested DRAM chips to prevent data retention failures from interfering with read-disturb failures. Third, we ensure that the tested DRAM modules and chips have neither rank-level nor on-die ECC. Doing so ensures that we directly observe and analyze all circuit-level bitflips without interference from architecture-level correction and mitigation mechanisms.

3.2 Commodity DDR4 DRAM Chips Tested

Table 1 shows the 164 (21) real DDR4 DRAM chips (modules) that we test from all three major DRAM manufacturers. To demonstrate that RowPress is intrinsic to the DRAM technology and is a widespread phenomenon across manufacturers, we test a variety of DRAM chips spanning different die densities and die revisions from each DRAM chip manufacturer.[5]

Table 1: Tested DDR4 DRAM Chips.

Mfr.	#DIMMs	#Chips	Density	Die Rev.	Org.	Date
Mfr. S (Samsung)	2	8	8Gb	B	x8	20-53
	1	8	8Gb	C	x8	N/A
	3	8	8Gb	D	x8	21-10
	2	8	4Gb	F	x8	N/A
Mfr. H (SK Hynix)	1	8	4Gb	A	x8	19-46
	1	8	4Gb	X	x8	N/A
	2	8	16Gb	A	x8	20-51
	2	8	16Gb	C	x8	21-36
Mfr. M (Micron)	1	16	8Gb	B	x4	N/A
	2	4	16Gb	B	x16	21-26
	1	16	16Gb	E	x4	20-14
	2	4	16Gb	E	x16	20-46
	1	4	16Gb	F	x16	21-50

To account for in-DRAM row address mapping [10, 19, 47, 51, 61, 64, 65, 68, 77, 82, 110, 134, 136, 145], we reverse-engineer the physical row address layout, following the methodology of prior works [43, 67, 103, 164].

4 Major RowPress Characterization

We characterize RowPress by analyzing 1) how DRAM's vulnerability to read disturbance changes as t_{AggON} increases, and 2) properties of RowPress bitflips that distinguish them from RowHammer and retention failure bitflips. We evaluate the sensitivity of RowPress biflips to temperature, access pattern, and aggressor row *off* time (i.e., t_{AggOFF}) in §5. Appendix §C provides further results and plots.

[5]The technology node that a DRAM chip is manufactured with is usually not publicly available. We assume that two DRAM chips from the same manufacturer have the same technology node only if they share both the same die density and die revision code. A die revision code of X indicates that there is no public information available about the die revision (e.g., the original DRAM chip manufacturer's markings have been removed by the DRAM module vendor and the DRAM stepping field in the SPD is `0x00`). More details on the tested chips and a summary of their RowPress and RowHammer characteristics are in Appendix B.

4.1 Experiment Methodology

Metric. To characterize how RowPress amplifies DRAM's vulnerability to read disturbance, we examine how AC_{min} changes as t_{AggON} increases. A lower AC_{min} means more vulnerability to read disturbance.

Access Pattern. Fig. 5 illustrates our RowPress access pattern targeting a single aggressor row (single-sided) to induce bitflips. We 1) activate (ACT) the aggressor row (R0), 2) keep the aggressor row on for a certain amount of time (t_{AggON}), and 3) close the row with a precharge (PRE) command. To respect the timing constraints, we wait until precharge latency t_{RP} is satisfied before repeating the same access pattern. We sweep t_{AggON} from the minimum possible value of 36 ns (i.e., the nominal t_{RAS} value) up to 30 ms. Note that for t_{AggON} = 36 ns, our single-sided RowPress pattern is identical to a single-sided RowHammer access pattern.[6] We test 3072 rows (the first, the middle, and the last 1024 rows) in bank 1 for each DRAM module.

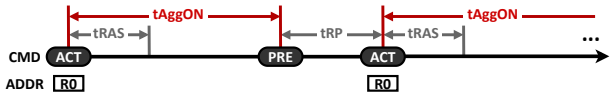

Figure 5: Single-sided RowPress access pattern used to characterize how AC_{min} changes as t_{AggON} increases.

Algorithm. For every t_{AggON} value we evaluate, we find the AC_{min} for each tested row using a modified version of the bisection-method algorithm used by prior works [103, 164]. Instead of a fixed AC_{min} accuracy (e.g., 100 in [164] and 512 in [103]), we enable an accuracy of 1%, rounded up to the next integer (i.e., we terminate the search for AC_{min} when the difference between the current and previous measurements of AC_{min} is no larger than 1% of the previous measurements). We report that we could not induce any bitflip if the test program's execution time exceeds 60ms (which is strictly smaller than the refresh window of 64 ms in DDR4 [56]). For every tested row, we repeat the AC_{min} search five times and report the minimum AC_{min} value we observe.

Data Pattern. We use a checkerboard data pattern [152] where we fill the aggressor row with `0xAA` and victim rows with `0x55`. We consider three adjacent rows on each side of the aggressor row as victim rows. We use this data pattern for all our characterization and sensitivity studies. We study the data pattern sensitivity of RowPress bitflips in detail in §5.3.

Temperature. We maintain the DRAM chip temperature at a normal operating condition of $50°C$. We study the temperature sensitivity of RowPress bitflips in §5.1.

[6]The RowHammer access pattern activates an aggressor row as frequently as possible, and thus closes the row (i.e., precharges the bank) as soon as it can, which is 36 ns (= t_{RAS}) after the row is opened.

4.2 Vulnerability to Read Disturbance

Fig. 6 shows the AC_{min} distribution (y-axis) of different die revisions for all three major DRAM manufacturers as we sweep t_{AggON} (x-axis) from 36 ns to 30 ms in log-log scale. For each manufacturer (i.e., each plot), we group the data based on the die revision (different colors) and aggregate the AC_{min} values from all the rows we test in all chips with the same die revision. Each data point shows the mean AC_{min} value and the error band shows the minimum and maximum of AC_{min} values across all tested rows. We highlight the t_{AggON} values of 7.8 µs (t_{REFI}) and 70.2 µs (9×t_{REFI}) on the x-axis, as they are the two potential upper bounds of t_{AggON}, as dictated by the JEDEC DDR4 standard [56].[7] We mark AC_{min} = 1 on the y-axis. We make three major observations from Fig. 6.

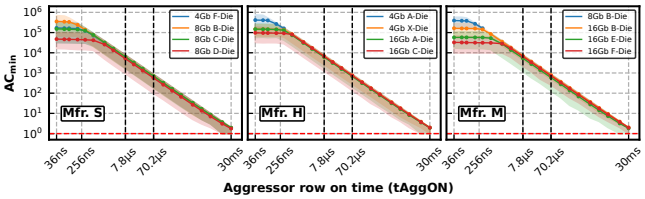

Figure 6: AC_{min} as t_{AggON} increases; single-sided RowPress at 50°C.

Obsv. 1. RowPress significantly reduces AC_{min} as t_{AggON} increases.

For example, for almost all (10 of 12) die revisions from all three DRAM manufacturers,[8] we observe that AC_{min} reduces by 21× on average when t_{AggON} increases from 36 ns to 7.8 µs. For modules with 8Gb B-Dies from Mfr. S, the reduction in mean AC_{min} can reach up to 59×. If t_{AggON} increases from 36 ns to 70.2 µs, the reduction in mean AC_{min} is 190×, and the maximum reduction reaches 537×, as observed in modules with 8Gb B-Dies from Mfr. S.

Obsv. 2. In extreme cases, RowPress causes bitflips with only one aggressor row activation (i.e., AC_{min}= 1).

We observe that for almost all die revisions from all three manufacturers, 1) we can *always* induce bitflips as we continue to increase t_{AggON} until 30 ms, and 2) for 13.1% of the tested rows that experience bitflips, only a single activation of an aggressor row (i.e., AC_{min}= 1), is needed to induce bitflips when t_{AggON} is 30 ms at 50°C. We conclude that, unlike RowHammer, RowPress does not have to rely on repeatedly accessing the aggressor row *many* times to induce bitflips.

Obsv. 3. RowPress is a common DRAM vulnerability across all three major DRAM manufacturers.

We observe that the AC_{min} trends across almost all die revisions from all three major DRAM manufacturers follow a consistent pattern. First, AC_{min} decreases slowly as t_{AggON} starts to increase. For example, when t_{AggON} increases by 5.17× from 36 ns to 186 ns, AC_{min} reduces on average by only 1.17×, 1.04×, and 1.08× for Mfr. S, H, and M, respectively. Second, as t_{AggON} continues to increase (e.g., beyond 7.8 µs), AC_{min} decreases drastically for all three manufacturers, following an approximately straight line in log-log scale. We find that the AC_{min} trend lines when $t_{AggON} \geq$ 7.8 µs for all three manufacturers have very similar slopes: −1.020, −1.013, and −1.013 for Mfr. S, H, and M, respectively. Given the similarity in AC_{min} reduction with increasing t_{AggON} across all tested die revisions from all three major manufacturers spanning 164 chips, we conclude that RowPress is an intrinsic read-disturb phenomenon to the DRAM technology.

Note that a slope close to −1 in log-log scale does *not* mean that AC_{min} reduces linearly as t_{AggON} reduces. Fig. 7 shows a portion of the AC_{min} distribution from Fig. 6 with a smaller range of t_{AggON} values (from 7.8 µs to 70.2 µs) in *linear-linear* scale.

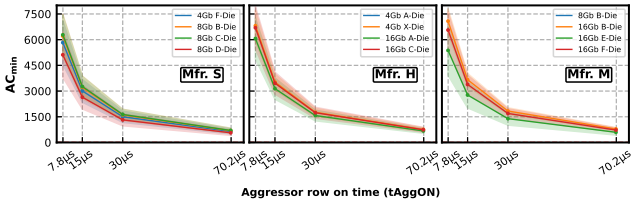

Figure 7: AC_{min} for t_{AggON} between 7.8 µs and 70.2 µs in linear-linear scale; single-sided RowPress at 50°C.

We observe that as t_{AggON} increases, the reduction rate of AC_{min} decreases. The average AC_{min} reduction for Mfr. S, H, and M when t_{AggON} increases from 7.8 µs to 15 µs are $-0.37\,\mu s^{-1}$, $-0.41\,\mu s^{-1}$, and $-0.39\,\mu s^{-1}$, respectively, but only $-0.021\,\mu s^{-1}$, $-0.023\,\mu s^{-1}$, and $-0.021\,\mu s^{-1}$, respectively, when t_{AggON} increases from 30 µs to 70.2 µs. We conclude that AC_{min} does *not* reduce linearly as t_{AggON} increases.

Fig. 8 shows the fraction of the tested rows that have at least one RowPress bitflip (y-axis) as we sweep t_{AggON} (x-axis). Each plot corresponds to a different manufacturer. Each curve represents a different DRAM module and is colored by its die revision.

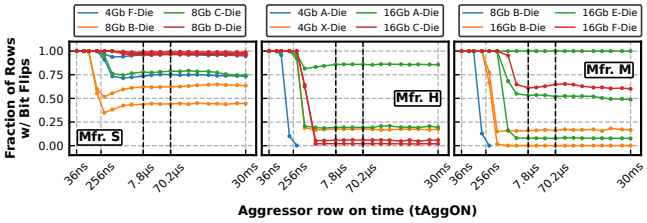

Figure 8: The fraction of rows that experience at least one bitflip; single-sided RowPress at 50°C.

Obsv. 4. RowPress worsens as DRAM technology node scales down.

[7] Whether 7.8 µs or 70.2 µs is the upper bound for t_{AggON} depends on the memory controller's implementation. If the memory controller does *not* allow any refresh commands to be postponed, the upper bound is 7.8 µs. Otherwise, because the JEDEC DDR4 standard [56] allows *up to* eight refresh commands to be postponed (Section 4.26 in [56]), the upper bound can be as high as 70.2 µs.

[8] The only exceptions are Mfr. H's 4Gb A-Dies and Mfr. M's 8Gb B-Dies, none of which exhibit any bitflips when t_{AggON} is larger than 336 ns with the single-sided RowPress pattern at 50°C.

In general, the more advanced the technology node[9] (as indicated by the die revision), the more rows are vulnerable to RowPress. For example, for the three 8Gb Dies from Mfr. S, as t_{AggON} increases, almost 100% of the tested rows of the D-Dies experience RowPress bitflips, which drops to below 80% for the C-Dies and below 60% for the B-Dies.

Takeaway 1. RowPress 1) is a common read-disturb phenomenon in DRAM chips that exacerbates DRAM's vulnerability to read disturbance and 2) gets worse as DRAM technology scales down to smaller node sizes.

To further understand the relationship between t_{AggON} and aggressor row activation count (*AC*) of RowPress, we examine the *minimum* t_{AggON} ($t_{AggONmin}$) to induce at least one bitflip for a given activation count using the single-sided RowPress pattern. Fig. 9 shows how $t_{AggONmin}$ changes as we sweep activation count from 1 to 10K. The error band shows the minimum and maximum $t_{AggONmin}$ values. We highlight the two potential upper-bound t_{AggON} values of 7.8 µs (t_{REFI}) and 70.2 µs (9×t_{REFI}) on the y-axis.

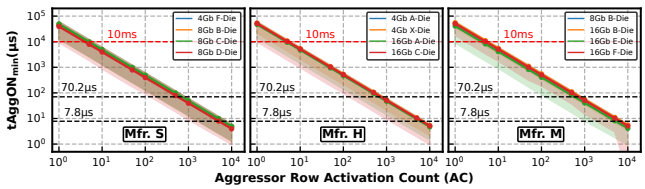

Figure 9: $t_{AggONmin}$ as aggressor row activation count (AC) increases; single-sided RowPress at 50°C.

Obsv. 5. $t_{AggONmin}$ significantly decreases as *AC* increases.

As *AC* increases from 1 to 10000, the average $t_{AggONmin}$ decreases from 43.3 ms to 4.3 µs, from 48.3 ms to 4.8 µs, and from 44.5 ms to 4.5 µs for Mfr. S, H, and M, respectively.[10] The decreasing $t_{AggONmin}$ trend lines are very similar across all three manufacturers. Their slopes are -1.000, -0.999, and -1.000 for Mfr. S, H, and M, respectively, in Fig. 9.[11]

Obsv. 6. In extreme cases, RowPress can induce bitflips for t_{AggON} values less than 10 ms with only a single aggressor row activation (i.e., $AC = 1$).

We observe that, for the Mfr. S 8Gb D-Dies, the Mfr. H 16Gb C-Dies, and the Mfr. M 16Gb E-Dies, there are one, two, and two rows out of the 3072 rows we test experience bitflips with $AC = 1$ at a $t_{AggONmin}$ value less than 10 ms (highlighted with dashed red lines). The minimum $t_{AggONmin}$ observed for these three dies are 9.2 ms, 9.8 ms, and 9.0 ms, respectively.

4.3 Distinguishing Characteristics of RowPress

Cells Vulnerable to RowPress vs. RowHammer and Retention Failure. We compare the set of DRAM cells that experience bitflips from our search for AC_{min} as we sweep t_{AggON} beyond 36 ns (i.e., for each t_{AggON}, the set of cells that experience bitflips with the minimum number of aggressor row activations that causes bitflips for that t_{AggON}) with two other sets of cells: 1) the set of cells that experience RowHammer bitflips (i.e., when t_{AggON} equals t_{RAS} 36 ns), and 2) the set of cells that exhibit bitflips in a data retention failure test.[12] Fig. 10 shows how increasing t_{AggON} (x-axis) changes the fraction of RowPress-vulnerable cells (y-axis) that also experience RowHammer (retention) failure in the first (second) row of subplots. Similar to Fig. 8, each curve represents a different DRAM module, color-coded based on its die revision.

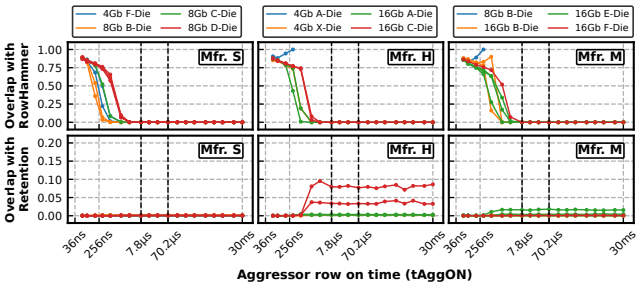

Figure 10: Overlap ratio of RowPress-vulnerable cells @ AC_{min} with RowHammer-vulnerable cells @ AC_{min} (first row of plots) and retention failures (second row of plots).

Obsv. 7. An overwhelming majority of the DRAM cells vulnerable to RowPress are *not* vulnerable to RowHammer or data retention failures.

For $t_{AggON} \geq 7.8$ µs, on average, only less than 0.013% of DRAM cells vulnerable to RowPress overlap with those vulnerable to RowHammer, and less than 0.34% overlap with retention failures. Therefore, an overwhelming majority of RowPress bitflips are different from those caused by RowHammer and retention failures.[13] These results suggest that different failure mechanisms lead to RowPress and RowHammer bitflips.

Fig. 11 shows the overlap ratio of the set of cells that experience bitflips when we activate the aggressor row as many times as possible (i.e., at AC_{max}) for each t_{AggON} value with the RowHammer-vulnerable cells (also at AC_{max}, first row of plots) and retention failures (second row of plots). Similar to Fig. 10, we observe that the overlap between RowPress-vulnerable cells and RowHammer vulnerable cells significantly decreases as t_{AggON} increases.

Bitflip Direction. Fig. 12 shows the fraction of 1 to 0 bitflips across all the bitflips we observe (y-axis) as we sweep t_{AggON} (x-axis). Similar to Fig. 8, each curve represents a different DRAM module, color-coded based on its die revision.

Obsv. 8. RowPress and RowHammer bitflips have opposite directions.

With the checkerboard data pattern we test, the dominant bitflip direction for RowHammer (i.e., when t_{AggON} is 36 ns) is 0 to 1. As

[9] For a given manufacturer and die density, the later in the alphabetical order the die revision code is, the more likely the chip has a more advanced technology node.

[10] We observe no bitflips in modules with Mfr. H 4Gb A-Die and Mfr. M 8Gb B-Die in this experiment.

[11] Note that for Mfr. M's 16Gb F-Die (colored red), when $AC = 10^4$, we observe a minimum $t_{AggONmin}$ of only 66 ns (cropped in Fig. 9).

[12] We initialize the DRAM rows with the same checkerboard data pattern as in §4.2, and disable auto-refresh for four seconds at 80 °C to induce retention-failure bitflips, similar to prior work [111].

[13] Prior works [67, 68] already show that RowHammer bitflips have little overlap with retention failure bitflips.

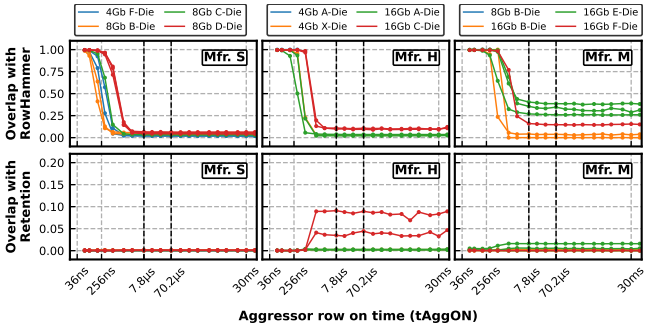

Figure 11: Overlap ratio of RowPress-vulnerable cells @ AC_{max} with RowHammer-vulnerable cells @ AC_{max} (first row of plots) and retention failures (second row of plots).

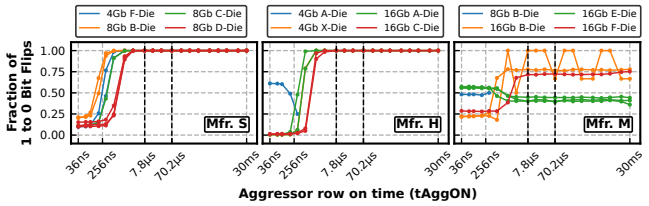

Figure 12: Fraction of 1 to 0 bitflips.

t_{AggON} increases (i.e., for RowPress), for almost all die revisions from Mfr. S and H (except for Mfr. H's 4Gb A-Die chips that do not show any bitflip), the dominant bitflip direction changes to 1 to 0. For example, the fraction of 1 to 0 bitflips reaches 100% for $t_{AggON} \geq 7.8\,\mu s$. Similarly, the fraction of 1 to 0 bitflips in Mfr. M's 16Gb B-Die and F-Die chips reaches 75% in this region of t_{AggON}.[14] As an exception, Mfr. M's 16Gb E-Die chips show an opposite trend: the fraction of 1 to 0 bitflips decreases as t_{AggON} increases. The reason for this opposite behavior could be a different layout of true- and anti-cells compared to that in other chips.[15]

Takeaway 2. RowPress has a different failure mechanism from RowHammer and data retention failures in DRAM. There is almost no overlap between RowPress, RowHammer, and data retention bitflips, and the directionality of RowHammer and RowPress bitflips show opposite trends.

5 RowPress Sensitivity Study

We examine the sensitivity of RowPress bitflips to 1) temperature, 2) access pattern, and 3) aggressor row off time (t_{AggOFF}). We study the repeatability of RowPress bitflips in Appendix E.

5.1 Temperature

Methodology. To investigate how RowPress bitflips change as DRAM chip temperature changes, we repeat the AC_{min} experiments (as described in 4.1) except we increase the temperature from $50°C$ to $80°C$. Fig. 13 shows the mean AC_{min} values we observe at $80°C$ normalized to $50°C$ as we sweep t_{AggON} at $80°C$ in linear (y-axis) - log (x-axis) scale.

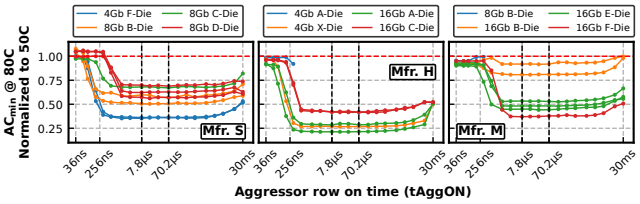

Figure 13: AC_{min} at $80°C$ normalized to $50°C$; single-sided RowPress.

Obsv. 9. As temperature increases, RowPress reduces AC_{min} more.

We observe that for all die revisions vulnerable to RowPress, AC_{min} consistently reduces for the same t_{AggON} value as temperature increases from $50°C$ to $80°C$. For example, when t_{AggON} is $7.8\,\mu s$, the average AC_{min} at $80°C$ is only 0.55×, 0.32×, and 0.59× of that at $50°C$, for Mfr. S, H, and M, respectively. Across all manufacturers, AC_{min} reduces by 48× on average (up to 122×, observed in 8Gb B-Dies from Mfr. S) when t_{AggON} increases from 36 ns to $7.8\,\mu s$ at $80°C$. When t_{AggON} increases from 36 ns to $70.2\,\mu s$, AC_{min} reduces by 438× on average (up to 1106×) at $80°C$. In contrast, at $50°C$, the reduction in AC_{min} is only 21× on average (up to 59×) when t_{AggON} increases from 36 ns to $7.8\,\mu s$ and 190× (up to 537×) when t_{AggON} increases from 36 ns to $70.2\,\mu s$. For a t_{AggON} of 30 ms, 82.8% of the rows with bitflips experience an AC_{min} of *only one* (not shown in Fig. 13) at $80°C$ (only 13.1% at $50°C$). We provide more results involving AC_{min} at $65°C$ in Appendix F.

Fig. 14 shows the fraction of rows that have at least one RowPress bitflip as we sweep t_{AggON} at $80°C$.

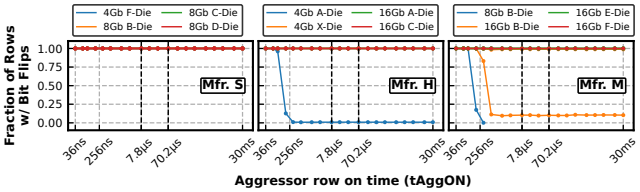

Figure 14: Fraction of rows that experience at least one bitflip at $80°C$; single-sided RowPress.

Obsv. 10. Fraction of rows that have at least one RowPress bitflip significantly increases as temperature increases.

We observe that almost all die revisions from all three manufacturers that are vulnerable to RowPress have their fractions of rows with at least one bitflip increase to almost 100% at $80°C$. Note that, for 4Gb A-Die from Mfr. H where we observe *no bitflips at all* for $t_{AggON} > 336$ ns at $50°C$, we are able to observe bitflips in a small fraction of rows (on average, 0.86% of all tested rows) with larger t_{AggON} values up to 30 ms at $80°C$.

[14] In a concurrent work [46], DRAM engineers from Samsung claim that the bitflips caused by RowHammer and the passing gate effect (caused by increased t_{AggON}) have opposite directionality because RowHammer *injects* electrons into the victim cell while the passing gate effect *attracts* electrons from the victim cell. We call for more detailed device-level modeling and analysis on this topic.

[15] A fully charged (discharged) DRAM cell does not necessarily imply that the stored value is 1 (0). A cell is called true (anti) cell if a fully charged state represents a value of 1 (0) [82].

To study the effect of increasing temperature on $t_{AggON}min$ (i.e., the minimum t_{AggON} to induce at least one bitflip) when $AC = 1$, we sweep temperature from $50°C$ to $80°C$ with a step size of $5°C$ and show the results in Fig. 15.[16] The error band shows the standard deviation of $t_{AggON}min$.

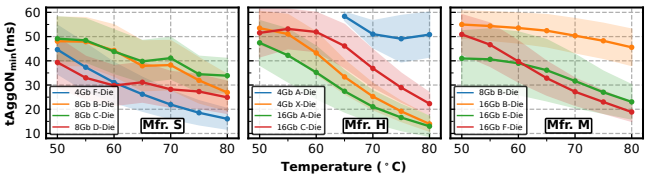

Figure 15: $t_{AggON}min$ when $AC = 1$ as we sweep temperature from $50°C$ to $80°C$ with $5°C$ steps; single-sided RowPress.

Obsv. 11. As temperature increases, $t_{AggONmin}$ significantly decreases.

We observe that $t_{AggON}min$ significantly decreases as we gradually increase temperature from $50°C$ to $80°C$. For Mfr. S, H, and M, the average (minimum) $t_{AggON}min$ reduces by 1.78× (1.90×), 2.84× (3.24×), and 1.64× (1.95×), respectively, going from $50°C$ to $80°C$. For example, for 16Gb A-Dies from Mfr. H, across all tested rows, the average (minimum) $t_{AggON}min$ is 47.4 ms (14.3 ms) at $50°C$, and reduces to only 13.0 ms (3.0 ms) at $80°C$. Note that for Mfr. H's 4Gb A-Die, where we could not induce any bitflip even when $AC = 10000$ at $50°C$ (Fig. 9), we are able to induce RowPress bitflips when $AC = 1$ at temperatures $\geq 65°C$.

Takeaway 3. RowPress gets significantly worse as temperature increases. This behavior is very different from how RowHammer bitflips change with temperature [68, 103].

5.2 Access Pattern

Methodology. To investigate how the bitflips induced by RowPress change as access pattern changes, we repeat the AC_{min} experiments (described in §4.1) except we use a *double-sided* RowPress pattern involving two aggressor rows, as shown in Fig. 16. In the double-sided RowPress pattern, we replace the row address of every other aggressor row activation in the single-sided access pattern (shown in Fig. 5) from R0 to R2. We treat the row R1 between R0 and R2 and three adjacent rows before R0 (i.e., R-1, R-2, R-3) and after R2 (i.e., R3, R4, R5) as the victim rows. We conduct the test at both $50°C$ and $80°C$.

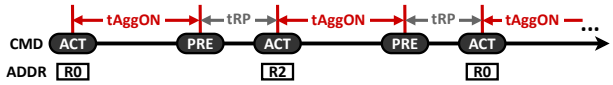

Figure 16: Double-sided RowPress access pattern.

We show how AC_{min} changes with the double-sided RowPress pattern at $50°C$ as we sweep t_{AggON} in Fig. 17. The error band shows the minimum and maximum AC_{min} values.

[16]We do not sweep the temperature with the fine-grained step size $5°C$ for the other experiments because of the prohibitively long experiment times.

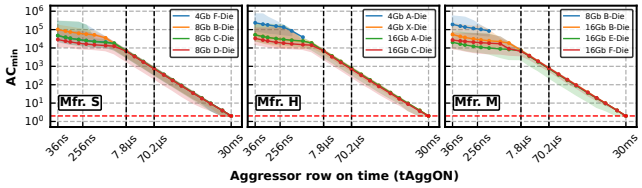

Figure 17: AC_{min} of double-sided RowPress; $50°C$.

Obsv. 12. As t_{AggON} increases, double-sided RowPress exhibits a similar decreasing AC_{min} trend as single-sided.

As t_{AggON} increases, AC_{min} significantly decreases with the double-sided RowPress pattern. The slopes of the overlapping AC_{min} trend lines in Fig. 17 for $t_{AggON} \geq 7.8$ µs of Mfr. S, H, M are −1.015, −1.010, and −1.011, respectively. Compared to the single-sided RowPress pattern, the decrease in AC_{min} is much larger with the double-sided RowPress pattern. For example, on average, when t_{AggON} increases from 36 ns to 186 ns, AC_{min} reduces by 1.62×, 1.56×, and 1.64× for Mfr. S, H, and M, respectively, with the double-sided pattern, compared to only 1.17×, 1.04×, and 1.08× of the single-sided pattern.

To comprehensively investigate how the access pattern and the temperature of the DRAM chip affect AC_{min}, we plot the difference between single- and double-sided AC_{min} (i.e., $AC_{min}(single)$ - $AC_{min}(double)$) at $50°C$ (first row) and $80°C$ (second row) in Fig. 18. A data point below 0 means that the single-sided RowPress pattern needs fewer aggressor row activations in total to induce a bitflip compared to double-sided.

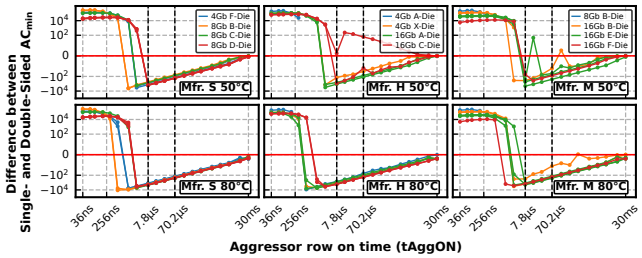

Figure 18: Single-sided AC_{min} minus double-sided AC_{min} at $50°C$ (first row) and $80°C$ (second row).

Obsv. 13. Single-sided RowPress becomes more effective at inducing bitflips as t_{AggON} increases beyond a certain value compared to double-sided RowPress.

We observe that, as t_{AggON} increases, double-sided RowPress is initially more effective compared to single-sided at $50°C$ (e.g., the single-sided pattern requires at least 10^4 more aggressor row activations to cause bitflips for almost all die revisions when $t_{AggON} <$ 1536 ns). However, as t_{AggON} continues to increase beyond 1536 ns, single-sided RowPress becomes more effective compared to double-sided for some die revisions. For example, for $t_{AggON} = 1536$ ns, single-sided RowPress requires 4210 less aggressor row activations on average to induce bitflips compared to double-sided for the 8Gb B-Dies from Mfr. S at $50°C$. As temperature increases from $50°C$ to $80°C$, we observe that: 1) single-sided RowPress becomes

even more effective, for example, for the 8Gb B-Dies from Mfr. S, the single-sided RowPress pattern needs 8699 less aggressor row activations on average for AC_{min} = 1536 ns compared to the double-sided RowPress pattern, and 2) for almost all die revisions from all manufacturers, single-sided AC_{min} is *consistently* smaller than double-sided for t_{AggON} values larger than 7.8 µs. We provide more results involving AC_{min} at 65°C in Appendix F.

Note that this behavior is very different from RowHammer, where double-sided RowHammer is strictly more effective at inducing bitflips than single-sided [68]. Fig. 1 summarizes the AC_{min} results we observe for single-sided and double-sided patterns for RowHammer and RowPress at 80°C.

Takeaway 4. RowPress behaves very differently from RowHammer as we change the access pattern from single-sided to double-sided. As t_{AggON} increases beyond a certain value, RowPress needs fewer aggressor row activations to induce bitflips with the single-sided pattern compared to the double-sided pattern.

5.3 Data Pattern

Methodology. To investigate how RowPress bitflips are affected by the data pattern of the victim and aggressor rows (i.e., what is the most effective data pattern to induce RowPress bitflips?), we repeat the AC_{min} experiments with more data patterns, summarized in Table 2. We denote the inverse of a data pattern with the suffix "I". Due to the large search space of all t_{AggON} values, we test a set of representative t_{AggON} values: 36 ns (=t_{RAS}), 66 ns, 636 ns, 7.8 μs (=t_{REFI}), 9×7.8 μs, 300 μs, and 6 ms.

Table 2: Tested data patterns

Row Type	Data Pattern		
	CheckerBoard (**I**)	**R**ow**S**tripe (**I**)	**C**ol**S**tripe (**I**)
Aggressor	0xAA (0x55)	0xFF (0x00)	0x55 (0xAA)
Victim	0x55 (0xAA)	0x00 (0xFF)	0x55 (0xAA)

Metric. To quantify the effectiveness of different data patterns for a die revision, we normalize their average AC_{min} (across all rows we test) to the average AC_{min} value of the CheckerBoard (CB) pattern. A value lower (higher) than 1.00 means the data pattern is more (less) effective than the CB pattern at inducing bitflips.

Fig. 19 shows the normalized AC_{min} values of different data patterns (y-axis) at different t_{AggON} values (x-axis) from three representative die revisions from the three manufacturers[17] using a single-sided access pattern at 50°C (left column) and 80°C (right column). A red (blue) cell means at a given x t_{AggON}, the y data pattern is less (more) effective at inducing bitflips compared to the baseline CheckerBoard pattern. Certain data patterns could not induce any bitflip at certain t_{AggON} values, even with the maximum possible activation count (within 60ms, which is strictly smaller than the 64ms refresh window). We mark these cases as "No Bitflip" (white cell) in the figure. We make the following two observations.

Obsv. 14. CheckerBoard pattern is in general the most effective RowPress data pattern among the ones tested.

[17]We find that the remaining die revisions behave similarly to one of the three representative die revisions.

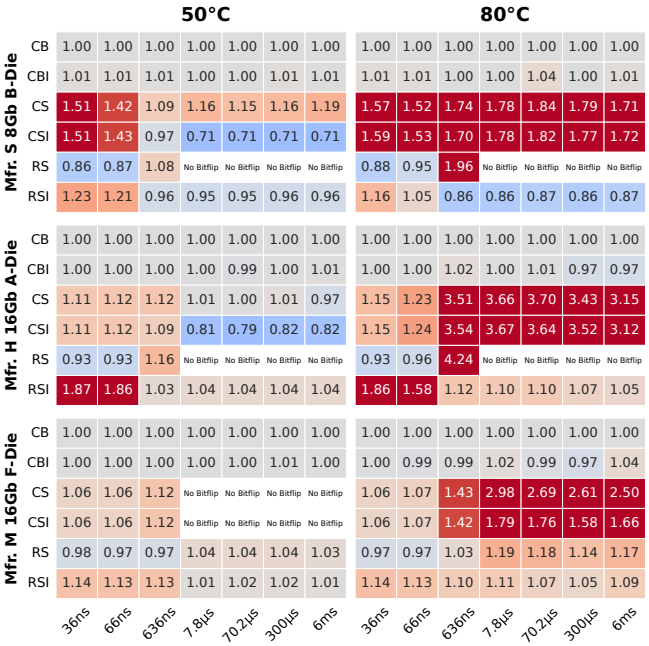

Figure 19: AC_{min} of different data patterns normalized to the CB data pattern at different t_{AggON} values from three representative die revisions from the three manufacturers; Single-sided access pattern; 50°C (left column) and 80°C (right column); A value lower (higher) than 1.00 means the data pattern is more (less) effective than the CB pattern at inducing bitflips, colored as blue (red).

We observe that, in most cases, the CheckerBoard pattern is the most effective at inducing RowPress bitflips among the tested data patterns for the following two reasons. First, we can *always* induce bitflips with the CheckerBoard pattern as we increase t_{AggON}. In comparison, although the RowStripe pattern in Mfr. S 8Gb B-Die and Mfr. H 16Gb A-Die is more effective with low t_{AggON} values (i.e., up to 13% smaller AC_{min} when t_{AggON} is 66 ns), it cannot induce *any* bitflip for t_{AggON} larger than 636 ns. Second, compared to the other data patterns, the CheckerBoard pattern is less affected by the increase in temperature. For example, although the ColumnStripeI pattern is the most effective for large t_{AggON} values (≥ 7.8 µs) for Mfr. S 8Gb B-Die and Mfr. H 16Gb A-Die at 50°C (up to 29% smaller AC_{min}), it becomes the least effective (up to 267% increase in AC_{min}) at 80°C.

Obsv. 15. The most effective RowHammer data pattern is not necessarily the most effective RowPress pattern.

For all three representative die revisions shown in Fig. 19, RowStripe is the most effective data pattern to induce Rowhammer bitflips (i.e., t_{AggON} = 36 ns). However, as we increase t_{AggON}, it becomes significantly less effective compared to the other patterns. For Mfr. S 8Gb B-Die and Mfr. H 16Gb A-Die, the RowStripe pattern *cannot* induce any bitflip for t_{AggON} > 636 ns, even at 80°C.

Fig. 20 shows the normalized AC_{min} values of different data patterns from Mfr. S 8Gb B-die using a double-sided access pattern

at 50°C and 80°C. We observe that the effectiveness of ColumnStripe and ColumnStripeI patterns increases as t_{AggON} increases in the double-sided access pattern, in contrast to the decreasing effectiveness as we show in Fig. 19. Note that this is the only case where we observe any major difference comparing single-sided to double-sided. The other die revisions behave similarly for the double-sided access pattern compared to single-sided.

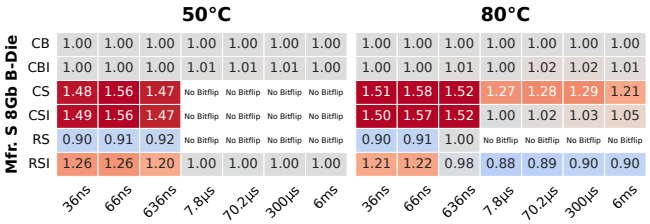

Figure 20: Normalized AC_{min} of different data patterns of Mfr. S 8Gb B-Die; Double-sided access pattern; 50°C (left column) and 80°C (right column).

We believe the data pattern dependence of RowPress and RowHammer require more and deeper study to fully understand and model the effect on the two read disturbance phenomena.

5.4 tAggON vs tAggOFF

Prior works on *device-level* mechanisms of RowHammer [105, 169] show that increasing t_{AggON} has little impact on DRAM read disturbance, while doing the opposite, increasing t_{AggOFF} (i.e., the aggressor row off time), worsens read disturbance. This seems to contradict our results in §4.2 and §5.2. However, the methodology of those prior works [105, 169] is limited because they only test 1) a very small range of t_{AggON} and t_{AggOFF} values (up to 50 ns in [169] and 72.5 ns in [105]), and 2) a single-sided access pattern.

Access Pattern. To compare RowPress to the read-disturb mechanisms discussed in prior works [105, 169], we design the RowPress-ONOFF access pattern shown in Fig. 21, based on the pattern proposed in [105]. In this pattern, we can adjust t_{AggON} and t_{AggOFF} by changing: 1) when we issue the PRE command to close the aggressor row, and 2) when we issue the ACT command to open the aggressor row. We denote the time interval between two consecutive ACT commands as t_{A2A}. Notice that since $t_{A2A} = t_{AggON} + t_{AggOFF}$, the minimum possible value of t_{A2A} is $\min(t_{AggON}) + \min(t_{AggOFF}) = t_{RAS} + t_{RP} = t_{RC}$.

Figure 21: The RowPress-ONOFF pattern.

Methodology. We fix the activation frequency of a row by fixing t_{A2A}. We increase t_{A2A} beyond t_{RC} by Δt_{A2A} = {240, 600, 1200, 2400, 6000} ns. For each t_{A2A} value, we sweep the fraction of Δt_{A2A} that contributes to t_{AggON} from 0% to 100% (with a step size of 25%). For example, 25% means t_{AggON} = 25% Δt_{A2A} + t_{RAS}, and t_{AggOFF} = 75% Δt_{A2A} + t_{RP}. For all configurations, we activate the aggressor row(s) as many times as possible to induce the most number of bitflips without exceeding the experiment time limit of 60 ms. We conduct the experiments at 50°C and 80°C.

Metric. We measure the bit error rate (*BER*), i.e., the fraction of DRAM cells in a DRAM row that experience bitflips. We repeat the experiment five times and report the highest *BER* to evaluate the worst-case scenario.

Fig. 22 shows the *BER* (y-axis) for both single-sided (top row) and double-sided (bottom row) RowPress-ONOFF pattern for a representative[18] die revision (8Gb D-Die from Mfr. S). We sweep Δt_{A2A} (different lines in each plot) and the percentage of Δt_{A2A} that contributes to t_{AggON} (x-axis) at 50°C (left column) and 80°C (right column). The error band shows the standard deviation of *BER*. We make the following three observations.

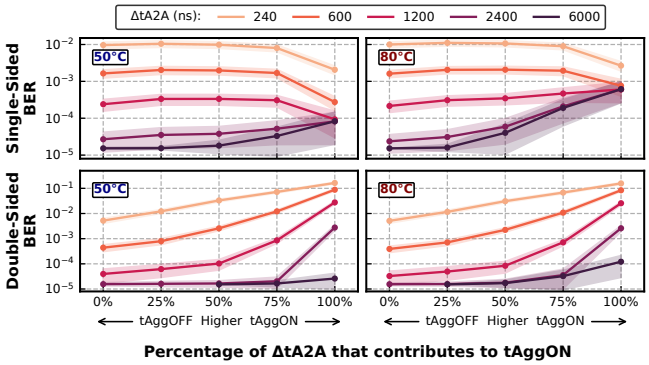

Figure 22: BER of the representative Mfr. S 8Gb D-Die; single- (top row) and double-sided (bottom row) RowPress-ONOFF pattern at 50°C (left column) and 80°C (right column).

Obsv. 16. For the single-sided access pattern, increasing t_{AggON} (i.e., decreasing t_{AggOFF}) with small (large) Δt_{A2A} values mitigates (exacerbates) read disturbance.

For Δt_{A2A} values ≤ 1200 ns (i.e., the upper three lines in the top two plots), we observe that BER decreases as we increase t_{AggON} (and thus decrease t_{AggOFF}) with the single-sided pattern. This agrees with prior device-level works [106, 169][19] that test a small range of t_{AggON}/t_{AggOFF} values (up to 50 ns in [169] and 72.5 ns in [105], respectively). As Δt_{A2A} takes larger values (e.g., 2400 ns and 6000 ns), we observe an *opposite trend* to what we observe with smaller t_{A2A} values: BER increases as we increase t_{AggON} (and thus decrease t_{AggOFF}). This is neither observed nor explained by prior device-level works [106, 169].

Obsv. 17. For the single-sided access pattern, increasing temperature exacerbates read disturbance for large Δt_{A2A} and t_{AggON} values.

For the single-sided pattern, we observe that as temperature increases from 50°C to 80°C, BER significantly increases (remains

[18]We observe a similar trend for almost all other die revisions. We show only one representative die revision to illustrate the results more clearly. We show all other die revisions in Appendix §C.1.

[19]Injected charge (from diffused channel electrons [106] and charge traps [169]) needs sufficient amount of time to be recombined at the victim cell and fully exhausted *after* the row is closed (i.e., longer t_{AggOFF})

almost unchanged) for large (small) Δt_{A2A} and t_{AggON} values. For example, the average BER increases by 7.5× (only 1.04×) from $50^{\circ}C$ to $80^{\circ}C$ when Δt_{A2A} = 6000 ns (240 ns) and 100% of Δt_{A2A} contributes to t_{AggON}. At the inflection point of Δt_{A2A} = 1200 ns, when 50% to 100% of Δt_{A2A} contributes to t_{AggON}, BER *decreases* at $80^{\circ}C$, in contrast to *increasing* at $50^{\circ}C$. This observation is *not* fully explained by prior device-level works [106, 169] because they do *not* change Δt_{A2A}, t_{AggON}, and t_{AggOFF} when investigating the effect of temperature on read disturbance.

Obsv. 18. For the double-sided pattern, read disturbance consistently worsens as t_{AggON} increases and t_{AggOFF} decreases.

For *all* Δt_{A2A} values we test with the double-sided access pattern, we observe that BER consistently increases as t_{AggON} increases (i.e., as t_{AggOFF} decreases), unlike the single-sided case where we observe *opposite* trends for small and large Δt_{A2A} values. Such a difference in the bit error rate behavior of single-sided and double-sided access patterns is *not* covered by prior device-level works [106, 169]. Our observations indicate that access pattern plays an important role in RowPress's device-level failure mechanisms and further device-level investigation is necessary to develop a better understanding of RowPress.

Takeaway 5. RowPress is a read-disturb phenomenon that existing device-level studies do not fully explain. We call for more device-level research to provide fundamental lower-level understanding of the RowPress phenomenon.

6 Real System Demonstration of RowPress

We experimentally demonstrate that a simple user-level C++ program can induce RowPress bitflips on a real DDR4-based system despite the existence of periodic auto-refresh and in-DRAM target row refresh (TRR) mechanisms employed by the manufacturer.

6.1 Experimental Setup

System Configuration. We use an Ubuntu 18.04 system (Linux kernel 5.4.0-131-generic [76]) with an Intel i5-10400 (Comet Lake) processor [48] and a 16GB dual rank DDR4 DRAM module [129] from Mfr. S (Samsung). This DRAM module has target row refresh (TRR) [32, 43], a widely adopted in-DRAM RowHammer mitigation mechanism employed by DRAM manufacturers.

Memory Address Mapping. We reverse engineer the processor's address mapping from physical memory addresses to DRAM rank, bank, row, and column addresses using DRAMA [112], similar to prior works (e.g., [22, 32, 53]). We allocate a 1GB page using hugepage support [147] to directly manipulate the least significant 30 physical address bits that contain all of the DRAM rank and bank address bits and part of the row address bits. We carefully generate pointers to aggressor and victim rows within the 1GB page to precisely place them in physically adjacent DRAM rows.[20]

6.2 RowPress on Real Systems

Challenges. We face two challenges in inducing RowPress bitflips in a real system. First, TRR can detect aggressor rows in a RowPress access pattern and prevent us from inducing bitflips by refreshing the victim rows. However, TRR mechanisms typically keep track of *only* a few aggressor rows [32, 43] and these mechanisms can be bypassed by certain access patterns that access many other dummy aggressor rows (called dummy rows [32, 43]) besides the real aggressor rows. Such access patterns aim to trick a TRR mechanism into detecting *only* the dummy rows and allow the real aggressor rows to remain undetected.

Second, the memory controller needs to keep the aggressor row on for a long duration (i.e., large t_{AggON}) such that we can perform RowPress. Ensuring that a DRAM row remains open for a large t_{AggON} value is not straightforward because we do not have fine-grained control over the timing parameters used and the command sequences scheduled by the memory controller in a real system (in contrast to our real chip characterization setup in §3.1). However, carefully-designed access patterns can make the memory controller keep the DRAM row open for a long duration. For example, if a DRAM row is open, the memory controller can serve memory requests that target different cache blocks in the row at high data transfer rates [56]. Therefore, if an access pattern issues memory requests to different cache blocks in the *same* DRAM row, we hypothesize that the memory controller will keep the DRAM row open to serve subsequent memory requests in the access pattern (we verify this hypothesis in §6.3).

Test Program. Algorithm 1 shows the key part of our test program. We mark the input parameters of the program in red. To overcome the first challenge, the program is based on an access pattern described in [43], which can induce RowHammer bitflips in the presence of TRR. This access pattern uses 16 dummy rows that are activated shortly after the aggressor rows[21] to prevent the in-DRAM TRR mechanism from detecting the aggressor row activations [22, 32, 43, 53]. To overcome the second challenge and use large t_{AggON} values, we access multiple (i.e., NUM_READS) cache blocks in each aggressor row. In every iteration, the access pattern 1) activates the two aggressor rows adjacent to a victim row multiple (i.e., NUM_AGGR_ACTS in line 7) times (i.e., performs double-sided RowPress with varying t_{AggON}), and 2) activates each of the 16 dummy rows four times (line 17) [43].

```
1  // find two neighboring aggressor rows based on physical address mapping
2  AGGRESSOR1, AGGRESSOR2 = find_aggressor_rows(VICTIM);
3  // initialize the aggressor and the victim rows
4  initialize(VICTIM, 0x55555555);
5  initialize(AGGRESSOR1, AGGRESSOR2, 0xAAAAAAAA);
6  // Synchronize with refresh
7  for (iter = 0 ; iter < NUM_ITER ; iter++):
8    for (i = 0 ; i < NUM_AGGR_ACTS ; i++):
9      // access multiple cache blocks in each aggressor row
10     // to keep the aggressor row open longer
11     for (j = 0 ; j < NUM_READS ; j++): *AGGRESSOR1[j];
12     for (j = 0 ; j < NUM_READS ; j++): *AGGRESSOR2[j];
13     // flush the cache blocks of each aggressor row
14     for (j = 0 ; j < NUM_READS ; j++):
15       clflushopt (AGGRESSOR1[j]);
16       clflushopt (AGGRESSOR2[j]);
17     mfence ();
18   activate_dummy_rows();
19 record_bitflips[VICTIM] = check_bitflips(VICTIM);
```

Algorithm 1: RowPress test program.

The test program first initializes the victim and the aggressor rows using the same checkerboard data pattern we evaluated in our

[20] Although we leverage a 1GB hugepage for this real-system demonstration of RowPress, hugepages are not necessary for allocating physically adjacent DRAM rows and inducing bitflips, as prior works [72, 74, 81, 174] on system-level RowHammer attacks experimentally demonstrate. One can extend our real-system demonstration program to avoid using hugepages.

[21] Dummy rows are placed at least 100 rows away from the victim row [43] to ensure that activating them does not cause bitflips on the victim row.

DRAM chip characterization studies (lines 4–5). We use this data pattern as it is reported [67] to have the highest average read disturbance error coverage across DDR4 chips from three manufacturers. Second, the test program executes one or multiple (depending on the NUM_READS parameter) memory load instructions targeting different cache blocks of each aggressor row (lines 10, 11). Executing multiple memory load instructions to different cache blocks keeps an aggressor row open for a long time, whereas switching between different aggressor rows opens and closes the two aggressor rows as they are in the same bank (§2). Third, the program executes one or multiple clflushopt instructions to flush the cache blocks of each aggressor row to DRAM (lines 13–15). Doing so ensures that subsequent memory accesses (i.e., using load instructions) to the aggressor rows will access DRAM instead of processor caches. Fourth, the program executes an mfence instruction (line 16) to ensure that the data is fully flushed before any subsequent memory load instruction is executed [68]. Fifth, the program accesses the 16 dummy rows, four times each, to bypass TRR (line 17). For every victim row, we execute this access pattern for 800K iterations (i.e., NUM_ITER=800K in line 6) to gather statistically significant results and record the bitflips in the victim row (line 18).

Methodology. We run our program using NUM_AGGR_ACTS ={1,2,3,4}, and NUM_READS={1,2,4,16,32,48,64,80,128}[22] on 1500 arbitrarily selected victim rows. To reduce experiment time, we do not test NUM_READS>48(80) for NUM_AGGR_ACTS=4(3) because the access pattern would not fit in a t_{REFI} window. We synchronize our access pattern with the refresh commands, similarly to prior works [22, 53], to increase the chance of bypassing TRR.

Results. Fig. 23 shows the total number of bitflips (left) and the number of rows with bitflips (right) for different number of cache blocks read per aggressor row activation (NUM_READS; x-axis) when we activate each aggressor row four (top plots), three (middle plots), and two (bottom plots) times per iteration. We do not plot NUM_AGGR_ACTS=1 because we do not observe any bitflips for all NUM_READS we test. The leftmost bar in each graph shows the number of *conventional RowHammer-induced* bitflips, where we read *only a single* cache block per aggressor row activation, as done in prior works that induce RowHammer bitflips (e.g., via proof-of-concept programs [68] and RowHammer attacks [1, 10, 12–20, 22, 28, 30–32, 38–40, 43, 45, 52, 53, 58, 72, 74, 81, 84, 94, 95, 102, 112–114, 117, 118, 124, 131, 132, 145, 146, 148, 149, 153, 154, 156, 160, 170, 173–176]), such that the aggressor row is kept open for a short time. Remaining bars in each graph show results for RowPress-induced bitflips (with an increasing number of cache block reads from left to right, such that the aggressor row is kept open for an increasing amount of time).

Obsv. 19. Our test program leveraging RowPress induces bitflips when RowHammer cannot.

Obsv. 20. Our test program leveraging RowPress induces many more bitflips compared to RowHammer, at the same aggressor row activation count.

Our test program leveraging RowPress induces a significant number of bitflips in many DRAM rows while RowHammer *cannot* induce *any* bitflip when NUM_AGGR_ACTS={2,3} (i.e., the program

[22] A DRAM row in the module we test has 128 cache blocks.

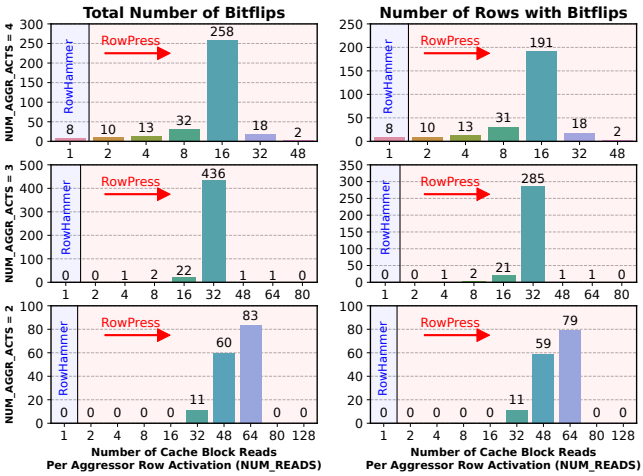

Figure 23: Number of RowHammer vs. RowPress bitflips (left) and number of rows with bitflips (right) we observe after running our test program with four (top), three (middle), and two (bottom) activations per aggressor row per iteration.

activates each aggressor row two/three times per iteration). The program induces up to 83 bitflips in 79 rows when NUM_AGGR_ACTS=2 and NUM_READS=64 (i.e., the program reads 64 cache blocks per aggressor row activation), and up to 436 bitflips in 285 rows when NUM_AGGR_ACTS=3 and NUM_READS=32.

When NUM_AGGR_ACTS=4, our test program leveraging RowPress induces significantly more bitflips compared to RowHammer. For example, the program induces up to 258 bitflips in 191 rows when NUM_READS=16. In comparison, RowHammer induces only 8 bitflips in 8 rows with the same aggressor row activation count.

Takeaway 6. Leveraging RowPress, a user-level program 1) induces bitflips when RowHammer cannot, and 2) induces many more bitflips compared to RowHammer, at the same aggressor row activation count.

Obsv. 21. In a real system, our test program does not always induce more bitflips as the number of cache blocks read per aggressor row activation increases.

We observe that the number of bitflips and DRAM rows with bitflips first increases significantly as we increase NUM_READS, but then decreases significantly after NUM_READS reaches a certain point. For example, when NUM_AGGR_ACTS=4, the number of bitflips (rows with bitflips) keeps increasing from 8 (8) to 258 (191) as NUM_READS increases from 1 to 16, but then decreases to 18 (18) when NUM_READS is 32, and only 2 (2) when NUM_READS is 48.

We attribute the increase in the number of bitflips and rows with bitflips when NUM_READS increases to two reasons. First, the increase in NUM_READS causes the memory controller keep the DRAM row open for a longer period of time, which leads to an increase in t_{AggON}. Second, the increase of NUM_READS reduces the activation frequency of the real aggressor rows compared to the dummy rows, which reduces the probability of real aggressor rows being detected by the TRR mechanism. We hypothesize that the reasons for the decrease in the number of bitflips and rows with bitflips after NUM_READS increases beyond a certain value are that 1) the access pattern

becomes too long, making it difficult to synchronize with the refresh commands, and 2) the activation frequency of the aggressor rows becomes too low to induce a large number of bitflips.

We conclude that, with a user-level program on a real DDR4-based Intel system with TRR protection, 1) RowPress induces bitflips when RowHammer cannot, 2) RowPress induces many more bitflips than RowHammer, and 3) increasing t_{AggON} up to a certain value increases RowPress-induced bitflips and number of rows with such bitflips. Thus, read-disturb-based attacks on real systems (e.g., [32, 53]) can leverage RowPress to be more effective.

We investigate a variant of our RowPress test program that induces even more bitflips in more rows in Appendix §G.

6.3 Verifying t_{AggON} Increase

We assumed in our real system experiment in the previous section that accessing different cache blocks in a DRAM row can keep the row open for a long time. We now briefly describe how we verify that this is indeed the case. We develop a simple program that 1) flushes all cache blocks of a tested DRAM row from the processor's caches using `clflushopt` instructions[23], 2) accesses a different row in the same bank as the tested row to ensure that the memory controller sends a precharge command to close the open row, and 3) records how many processor cycles it takes to access each cache block in the tested DRAM row. We run this program 100K times to collect statistically significant results.

Fig. 24 shows the frequency histogram of latency values (observed using Intel time stamp counter [49]) for 1) accessing the first cache block (green bars) and 2) accessing the subsequent (i.e., the remaining 127) cache blocks (blue bars). We mark the median latency values for these two types of accesses with dashed red lines.

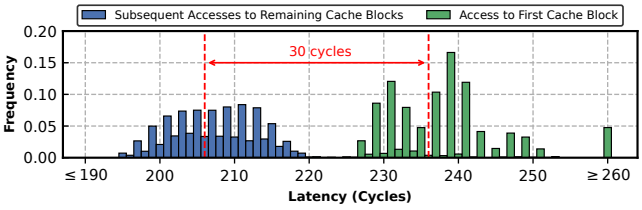

Figure 24: Histogram of the latency of the first and remaining cache block accesses to the same DRAM row.

We observe that the median latency values of accessing the first cache block and the other cache blocks are 30 cycles apart. Accessing the first cache block takes significantly longer than accessing other cache blocks. This happens because the first access requires activation of the DRAM row but the remaining ones do not. We conclude that, in the system we test, accessing consecutive cache blocks in an activated row causes the memory controller to keep the DRAM row open. Thus, existing memory controllers that behave similarly (e.g., using adaptive row buffer management policies [5, 26, 59, 97, 108, 119, 120, 130, 162, 177]) can facilitate future attacks leveraging RowPress.

[23]We disable all hardware prefetchers of the processor by modifying model-specific register values [49] before running the verification program. Doing so, together with the `clflushopt` instructions that flushes all cache blocks in the tested DRAM row in the program, makes sure subsequent accesses to the remaining cache blocks (i.e., after accessing the first cache block) of the row are served from DRAM.

7 Mitigating RowPress

We examine four potential ways to mitigate RowPress bitflips: 1) using error correcting codes (ECC), 2) decoupling the row buffer from the opened DRAM row, 3) limiting the maximum row-open time, and 4) adapting existing RowHammer mitigations to account for RowPress. We believe the fourth way is the most effective among the four. §7.1, §7.2, and §7.3 explain why the first three approaches are either ineffective or undesirable mitigations for RowPress. §7.4 describes and evaluates our proposed adaptations of RowHammer mitigations, using Graphene [109] and PARA [68] as examples. Appendix §D provides detailed evaluation results with more benchmarks, analyses, and graphs.

7.1 Error Correcting Codes (ECC)

We examine the capability of ECC, which is widely used in modern memory systems to correct memory errors, in mitigating RowPress. We analyze the number of bitflips in every 64-bit word for both single- and double-sided RowPress for a t_{AggON} of 7.8 µs. To maximize the number of bitflips at this t_{AggON}, we activate the aggressor row(s) as many times as possible within 60 ms at 80°C. Fig. 25 is a box-and-whiskers plot that shows the distribution of the number of erroneous 64-bit words with 1) at most two bitflips (1–2), 2) at least three and at most eight bitflips (3–8), and 3) more than eight bitflips (>8) across all tested modules from every manufacturer (x-axis).

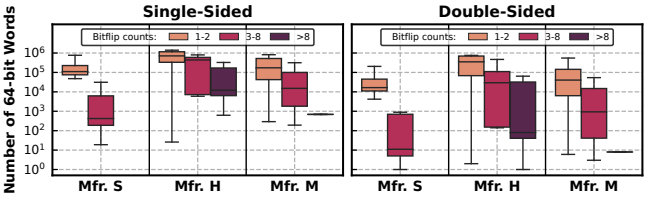

Figure 25: Number of 64-bit words with different bitflip counts for single-sided (left) and double-sided (right) RowPress.

We make two key observations from our analysis. First, there are up to 25 RowPress bitflips (not shown) in a 64-bit data word. ECC schemes that are widely used in memory systems (e.g., SECDED [41] and Chipkill [23, 85, 91][24]) cannot correct or detect *all* RowPress bitflips we observe, which can lead to silent data corruption [29, 104, 143]. Even a (7, 4) Hamming code (correcting one bitflip in a 4-bit data word) [41] with 75% DRAM storage overhead (3 parity bits for every 4 data bits), is not capable of correcting 25 bitflips in a 64-bit data word. Other ECC schemes that can correct *all* RowPress bitflips require prohibitively large storage overheads. Thus, relying on ECC alone to prevent *all* RowPress bitflips is a very expensive solution. Second, for all three manufacturers (Mfrs.

[24]Chipkill [23, 85, 91] can correct one-symbol errors and detect two-symbol errors. Because we observe up to 25 bitflips in a 64-bit data word, at least seven (four, two), symbols (i.e., data from seven, four, two DRAM chips, for x4, x8, and x16 chips, respectively) will be erroneous. Therefore, Chipkill *cannot* provide guaranteed mitigation against RowPress.

A, B, and C), a significant fraction (up to 0.99%, 35.77%, and 10.08% for t_{AggON} = 7.8 µs, respectively) of 64-bit data words exhibit at least three RowPress bitflips. This makes RowPress bitflips costly to prevent using techniques like *memory page retirement* (where erroneous DRAM rows are not used in the system) [92, 144] since such techniques could render up to 35.77% of storage capacity useless.

Fig. 26 shows the same distribution of the number of erroneous 64-bit words as Fig. 25 for t_{AggON} = 70.2 µs. We make similar observations and conclusions as for Fig. 25.

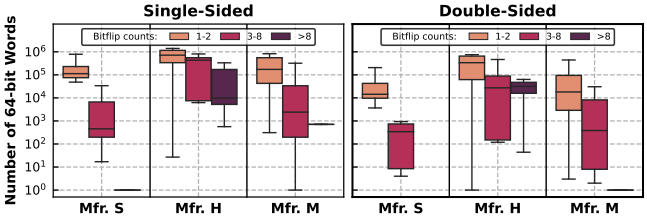

Figure 26: Number of 64-bit words with different bitflip counts for single-sided (left) and double-sided (right) RowPress when t_{AggON} is 70.2 µs.

7.2 Decoupling the Row Buffer from the Row

Prior works [133, 142] on improving DRAM performance and energy efficiency propose to decouple the row buffer from the DRAM row by disconnecting the DRAM row from the row buffer and de-asserting the wordline once the charge restoration process is completed after row activation. Doing so can *potentially* aid with RowPress mitigation because it limits t_{AggON} to the minimum possible value (t_{RAS}) regardless of the number of read requests sent to the DRAM row. However, there are at least three issues with this solution. First, it requires non-trivial changes in cost-sensitive DRAM chips. Second, to prevent write requests from increasing t_{AggON}, the row needs to be reconnected to the row buffer (by re-asserting the wordline) only for the last write request, which further complicates DRAM chip design and memory controller request scheduling [142]. Third, row buffer decoupling does *not* provide mitigation against *RowHammer*. We leave a detailed evaluation of using row buffer decoupling to mitigate RowPress to future works.

7.3 Limiting the Maximum Row-Open Time

Since RowPress is caused by keeping a DRAM row open for a long period of time, limiting the *maximum row-open time* (t_{mro}) by modifying the memory controller's row policy (i.e., forcing the closing of a row after t_{mro} even if there are requests in the memory controller ready to be served from the opened row) may appear to be a mitigation for RowPress. However, it is *not* effective because closing the row does *not* mitigate the read disturbance already caused by the longer activation, unless t_{mro} is set to its minimum possible value, t_{RAS} (as we show in Fig. 17, AC_{min} decreases as soon as t_{AggON} is higher than t_{RAS}). Having such a row policy that immediately closes an opened row after t_{RAS} causes two issues. First, it may turn a benign workload with high row-buffer locality to a potential RowHammer attack as the same DRAM row may have to be activated more times. Second, it can cause large slowdown as it increases the average memory access latency by reducing the row buffer hit rate (up to 34.1% single-core performance degradation, as we show in Appendix §D.1). We show in §7.4 that mitigating RowPress is possible by co-designing a row policy that enforces t_{mro} together with an enhanced RowHammer mitigation technique.

Some existing row policy proposals adapt t_{mro} based on row access patterns (e.g., keep the row open for longer when the row is predicted to be accessed soon in the future) [5, 26, 59, 108, 120, 130, 162]. Such row policies cannot mitigate RowPress as t_{mro} can be controlled by an attacker to be set to larger values than t_{RAS}, as we show in §6.

7.4 Adapting Existing RowHammer Mitigations

Adaptation Methodology. We propose a simple yet effective methodology to adapt existing RowHammer mitigation mechanisms to also mitigate RowPress with low *additional* area overhead. The key idea is, based on device characterization (§4, §5), to 1) quantify the worst-case (across different temperatures, access patterns, and data patterns) read disturbance caused by longer row-open time and translate it into an equivalent reduction in the RowHammer threshold (T_{RH}), defined as the minimum number of aggressor row activations needed to cause a RowHammer bitflip, and 2) also limit the maximum row-open time (t_{mro}). For example, if the device characterization shows that for a t_{AggON} of Xns, AC_{min} reduces by a maximum of Y%, then the adapted RowHammer mitigation mechanism will have $T'_{RH} = (1 - Y\%)T_{RH}$, and the memory controller must close the opened row after Xns even if there are requests ready to be served by the row.

Security Analysis. Assuming the original RowHammer mitigation is secure (i.e., it issues preventive refreshes to the victim rows before any DRAM row is activated T_{RH} times within the refresh window) and the DRAM device is properly characterized to uncover the worst-case RowPress vulnerability, our adapted mitigation mechanism 1) still mitigates RowHammer because the adapted mitigation is more aggressive than the original mitigation (i.e., T'_{RH} is strictly smaller than T_{RH}), and 2) mitigates RowPress because the limited maximum row-open time ensures that at least T'_{RH} activations to a DRAM row are needed to induce RowPress bitflips, which the adapted mitigation already properly prevents (i.e., a preventive refresh is issued before a row is activated T'_{RH} times).

Configuration and Evaluation. Our adaptation methodology is applicable to a wide range of RowHammer mitigations. We demonstrate this by applying it to two major ones: Graphene [109], a low performance overhead mechanism, and PARA [68], a low area overhead mechanism. We denote the adapted versions of Graphene and PARA as Graphene-RowPress (RP) and PARA-RowPress (RP), respectively. We use the characterization results of the 8Gb B-Die from Mfr. S to configure Graphene-RP and PARA-RP with a baseline T_{RH} of 1K using the methodology provided in [68, 109], as shown in Table 3. We perform a sensitivity study of their respective performance overheads over Graphene and PARA[25] with these

[25]Measured by the weighted speedup [27, 137] of Graphene-RP (PARA-RP) normalized to Graphene (PARA).

configurations using Ramulator [71, 123] with a realistic baseline system configuration[26] and show the results in Table 3.

Table 3: Graphene-RP and PARA-RP evaluations.

t_{mro} (ns)	36 (=t_{RAS})	66	96	186	336	636
T'_{RH}	1000 (=T_{RH})	809	724	619	555	419
Graphene-RP T	333	269	241	206	185	139
Avg. Perf. Overhead	1.3%	-0.43%	-0.63%	-0.49%	-0.14%	0.60%
Max. Perf. Overhead	10.2%	6.6%	6.4%	5.0%	5.0%	4.6%
PARA-RP p	0.034	0.042	0.047	0.054	0.061	0.079
Avg. Perf. Overhead	3.2%	3.6%	4.5%	6.0%	7.9%	12.9%
Max. Perf. Overhead	23.8%	13.4%	13.1%	14.7%	19.4%	31.6%

We make two major observations from the results. First, Graphene-RP and PARA-RP can mitigate RowPress at low *additional* performance overhead. Compared to Graphene (PARA), Graphene-RP (PARA-RP) has an average slowdown of only −0.63% (3.2%) when t_{mro} is 96ns (36ns). When t_{mro} is 636ns (96ns), Graphene-RP (PARA-RP) causes a maximum slowdown of only 4.6% (13.1%) over Graphene (PARA). The reason for the small negative slowdowns (i.e., speedups) is that some t_{mro} values improve fairness between cores in a way that increases weighted speedups (similar to [96, 97]). Second, the performance overheads of Graphene-RP and PARA-RP change differently with different t_{mro} configurations. For Graphene-RP, having a t_{mro} value that is either smaller or larger than 96ns increases the performance overhead. This is because row-buffer locality reduces at a smaller t_{mro}, and more preventive refreshes are issued at a larger t_{mro}. For PARA-RP, any t_{mro} value larger than 36ns increases the performance overhead. The reason is that PARA's performance overhead does *not* scale well with smaller T'_{RH} [67, 109, 166], and thus the benefit of longer row-open time is outweighed by the performance overhead of more preventive refreshes. We conclude that existing RowHammer mitigations can be relatively easily adapted to mitigate RowPress at low additional performance overhead. We expect future work to introduce new mitigation mechanisms, as it has been happening analogously for RowHammer.

We provide more evaluations and analyses of our proposed mitigation mechanisms in Appendix §D.2.

8 Related Work

To our knowledge, this is the first work to experimentally demonstrate and characterize RowPress, *a widespread read-disturb phenomenon in real DRAM chips*. Our analysis of RowPress (especially in §4.3, §5.1 and §5.2) shows that RowPress is different from RowHammer. This section highlights the most relevant works.

RowHammer with Increased t_{AggON}. A recent experimental characterization of real DRAM chips [103] and prior device-level studies [106, 169] provide preliminary results on how increasing t_{AggON} *by small amounts* affects RowHammer bitflips. These works treat this phenomenon the same as RowHammer and do *not* identify a DRAM read-disturb phenomenon *different* from RowHammer because they do *not*: 1) test a wide range of t_{AggON} values (only up to 154.5 ns in [103], 50 ns in [169], and 72.5 ns in in [106], as opposed to up to 30 ms in our work), 2) study sensitivity of increased t_{AggON} to temperature and access pattern, and 3) study the properties of the bitflips they induce. As such, these works attribute the bitflips to RowHammer. In contrast, our work clearly shows that RowPress bitflips have almost no overlap with RowHammer bitflips and thus RowPress is a different phenomenon from RowHammer.

RAS Clobber. Two patents from Micron [50, 158] very briefly mention a "RAS Clobber" effect similar to RowPress. They only describe RAS Clobber as "the selected word line is driven to the active level continuously for a considerably long period" [50], and "stress applied to adjacent word lines by a word line being on for an extended duration" [158]. These patents do *not* provide any evaluation, analysis or demonstration of this effect, and they do *not* clearly distinguish this effect from RowHammer. We show through detailed real DRAM chip characterization that RowPress is different from RowHammer (§4, §5), and demonstrate that RowPress can be leveraged to induce bitflips in a real system (§6). [50] describes a sampling-based read disturbance mitigation mechanism which they claim can handle both RowHammer and RAS Clobber. We introduce a general methodology that adapts existing RowHammer mitigation mechanisms to also mitigate RowPress (§7.4). [158] proposes to lower the wordline voltage after row activation and charge restoration to mitigate RAS Clobber. However, it does *not* demonstrate that reduced wordline voltage eliminates the read disturbance effect of increased t_{AggON}. Neither patent [50, 158] evaluates or analyzes its proposed mitigation mechanisms at the system-level.

One-Location RowHammer. A prior work [38] proposes a single-sided RowHammer technique called "One-Location Hammering" that "continuously re-opens the same DRAM row." However, it is unclear whether the bitflips this work observes are caused by increased t_{AggON} or conventional single-sided RowHammer. The access pattern in this work does not consider on-die RowHammer mitigations (e.g., TRR [32, 43]), unlike our real-system demonstration (§6).

Other DRAM Read Disturbance Mitigation Techniques. Many works (e.g., [3, 4, 6–9, 15, 25, 33, 36, 37, 42, 56, 60, 66, 68, 73, 78, 109, 115, 116, 121, 128, 135, 138, 154, 157, 159, 163, 166–168, 171]) propose techniques to mitigate RowHammer bitflips. None of these take RowPress into account.[27] We describe a methodology to adapt such techniques to mitigate both RowHammer and RowPress and evaluate it on two example prior works [68, 109] (§7.4).

9 Conclusion

We demonstrated and experimentally analyzed a widespread read-disturb phenomenon called RowPress in modern DRAM chips: keeping a row open for a long time disturbs physically nearby rows enough to cause bitflips. Our experimental characterization of 164 real DRAM chips reveals that RowPress 1) has a different underlying mechanism from the well-studied RowHammer phenomenon, 2) greatly amplifies DRAM's vulnerability to read disturbance by reducing the number of activations to induce a bitflip by one to two orders of magnitude (and in extreme cases to only a single

[26] 4 GHz out-of-order core, dual-rank DDR4 DRAM [56], FR-FCFS [119, 177] scheduling, open-row policy. 58 four-core multiprogrammed workloads from SPEC CPU2017 [140], TPC-H [150], and YCSB [21]. We find similar performance results for single-core workloads, as shown in our extended version [86].

[27] Two recent works [87, 88] discuss at a high level how to handle increased vulnerability due to small increases in t_{AggON} (as reported by [103]) by modifying their proposed RowHammer mitigation mechanisms. However, these works do not evaluate their modified mechanisms.

activation), and 3) becomes worse as DRAM technology node size reduces. We demonstrate that a user-level program causes RowPress bitflips in a real system, even in the presence of in-DRAM read-disturb mitigation mechanisms, much more so than the bitflips RowHammer can induce. We describe a methodology to adapt existing read-disturb mitigation mechanisms that only consider RowHammer to also mitigate RowPress, enabling strong protection against RowPress with low *additional performance overhead*. We hope that the findings reported in this work lead to further examination of and new solutions to the RowPress phenomenon at multiple levels of the computing stack. To this end, we open source all our infrastructure, test programs, and raw data at [125].

Acknowledgments

We thank the anonymous reviewers of ISCA 2023 for feedback. We thank the SAFARI Research Group members for valuable feedback and the stimulating intellectual environment they provide. We acknowledge the generous gift funding provided by our industrial partners (especially Google, Huawei, Intel, Microsoft, VMware), which has been instrumental in enabling the decade-long research we have been conducting on read disturbance in DRAM in particular and memory systems in general. This work was in part supported by the a Google Security and Privacy Research Award and the Microsoft Swiss Joint Research Center.

References

[1] M. T. Aga, Z. B. Aweke, and T. Austin, "When Good Protections Go Bad: Exploiting Anti-DoS Measures to Accelerate Rowhammer Attacks," in *HOST*, 2017.
[2] S. Agarwal, H. Dixit, D. Datta, M. Tran, D. Houssameddine, D. Shum, and F. Benistant, "Rowhammer for Spin Torque based Memory: Problem or not?" in *INTERMAG*, 2018.
[3] B. Aichinger, "DDR Memory Errors Caused by Row Hammer," in *HPEC*, 2015.
[4] Apple Inc., "About the Security Content of Mac EFI Security Update 2015-001," https://support.apple.com/en-us/HT204934, 2015.
[5] M. Awasthi, D. W. Nellans, R. Balasubramonian, and A. Davis, "Prediction Based DRAM Row-Buffer Management in the Many-Core Era," in *PACT*, 2011.
[6] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. Oren, and T. Austin, "ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks," in *ASPLOS*, 2016.
[7] K. Bains and J. Halbert, "Row Hammer Monitoring Based on Stored Row Hammer Threshold Value," U.S. Patent 9384821, 2016.
[8] K. Bains, J. Halbert, C. Mozak, T. Schoenborn, and Z. Greenfield, "Row Hammer Refresh Command," U.S. Patent 9117544, 2015.
[9] K. S. Bains and J. B. Halbert, "Distributed Row Hammer Tracking," U.S. Patent 9299400, 2016.
[10] A. Barenghi, L. Breveglieri, N. Izzo, and G. Pelosi, "Software-Only Reverse Engineering of Physical DRAM Mappings for Rowhammer Attacks," in *IVSW*, 2018.
[11] R. Baumann, "Radiation-Induced Soft Errors in Advanced Semiconductor Technologies," *IEEE TDMR*, 2005.
[12] S. Bhattacharya and D. Mukhopadhyay, "Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis," in *CHES*, 2016.
[13] Bhattacharya, Sarani and Mukhopadhyay, Debdeep, "Advanced Fault Attacks in Software: Exploiting the Rowhammer Bug," *Fault Tolerant Architectures for Cryptography and Hardware Security*, 2018.
[14] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida, "Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector," in *S&P*, 2016.
[15] F. Brasser, L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "Can't Touch This: Software-Only Mitigation Against Rowhammer Attacks Targeting Kernel Memory," in *USENIX Security*, 2017.
[16] W. Burleson, O. Mutlu, and M. Tiwari, "Invited: Who is the Major Threat to Tomorrow's Security? You, the Hardware Designer," in *DAC*, 2016.
[17] S. Carre, M. Desjardins, A. Facon, and S. Guilley, "OpenSSL Bellcore's Protection Helps Fault Attack," in *DSD*, 2018.
[18] Y. Cohen, K. S. Tharayil, A. Haenel, D. Genkin, A. D. Keromytis, Y. Oren, and Y. Yarom, "HammerScope: Observing DRAM Power Consumption Using Rowhammer," in *CCS*, 2022.
[19] L. Cojocar, J. Kim, M. Patel, L. Tsai, S. Saroiu, A. Wolman, and O. Mutlu, "Are We Susceptible to Rowhammer? An End-to-End Methodology for Cloud Providers," in *S&P*, 2020.
[20] L. Cojocar, K. Razavi, C. Giuffrida, and H. Bos, "Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks," in *S&P*, 2019.
[21] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *SoCC*, 2010.
[22] F. de Ridder, P. Frigo, E. Vannacci, H. Bos, C. Giuffrida, and K. Razavi, "SMASH: Synchronized Many-Sided Rowhammer Attacks from JavaScript," in *USENIX Security*, 2021.
[23] T. J. Dell, "A White Paper on the Benefits of Chipkill-Correct ECC for PC Server Main Memory," *IBM Microelectronics Division*, 1997.
[24] R. H. Dennard, "Field-Effect Transistor Memory," U.S. Patent 3387286, 1968.
[25] F. Devaux and R. Ayrignac, "Method and Circuit for Protecting a DRAM Memory Device from the Row Hammer Effect," U.S. Patent 10885966, 2021.
[26] J. M. Dodd, "Adaptive page management," U.S. Patent 7076617B2, 2005.
[27] S. Eyerman and L. Eeckhout, "System-Level Performance Metrics for Multiprogram Workloads," *IEEE Micro*, 2008.
[28] M. Fahr Jr, H. Kippen, A. Kwong, T. Dang, J. Lichtinger, D. Dachman-Soled, D. Genkin, A. Nelson, R. Perlner, A. Yerukhimovich *et al.*, "When Frodo Flips: End-to-End Key Recovery on FrodoKEM via Rowhammer," *CCS*, 2022.
[29] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell, "Detection and Correction of Silent Data Corruption for Large-Scale High-Performance Computing," in *SC*, 2012.
[30] A. P. Fournaris, L. Pocero Fraile, and O. Koufopavlou, "Exploiting Hardware Vulnerabilities to Attack Embedded System Devices: A Survey of Potent Microarchitectural Attacks," *Electronics*, 2017.
[31] P. Frigo, C. Giuffrida, H. Bos, and K. Razavi, "Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU," in *S&P*, 2018.
[32] P. Frigo, E. Vannacci, H. Hassan, V. van der Veen, O. Mutlu, C. Giuffrida, H. Bos, and K. Razavi, "TRRespass: Exploiting the Many Sides of Target Row Refresh," in *S&P*, 2020.
[33] S. Gautam, S. Manhas, A. Kumar, M. Pakala, and E. Yieh, "Row Hammering Mitigation Using Metal Nanowire in Saddle Fin DRAM," *IEEE TED*, 2019.
[34] P. R. Genssler, V. M. van Santen, J. Henkel, and H. Amrouch, "On the Reliability of FeFET On-Chip Memory," *TC*, 2022.
[35] S. Ghose, T. Li, N. Hajinazar, D. S. Cali, and O. Mutlu, "Demystifying Complex Workload–DRAM Interactions: An Experimental Study," in *SIGMETRICS*, 2019.
[36] H. Gomez, A. Amaya, and E. Roa, "DRAM Row-Hammer Attack Reduction Using Dummy Cells," in *NORCAS*, 2016.
[37] Z. Greenfield and T. Levy, "Throttling Support for Row-Hammer Counters," U.S. Patent 9251885, 2016.
[38] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom, "Another Flip in the Wall of Rowhammer Defenses," in *S&P*, 2018.
[39] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A Remote Software-Induced Fault Attack in Javascript," arXiv:1507.06955 [cs.CR], 2016.
[40] Gruss, Daniel and Maurice, Clementine and Mangard, Stefan, "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript," *arXiv:1507.06955 [cs.CR]*, 2015.
[41] R. W. Hamming, "Error Detecting and Error Correcting Codes," *The Bell System Technical Journal*, 1950.
[42] H. Hassan, M. Patel, J. S. Kim, A. G. Yağlıkçı, N. Vijaykumar, N. Mansouri Ghiasi, S. Ghose, and O. Mutlu, "CROW: A Low-Cost Substrate for Improving DRAM Performance, Energy Efficiency, and Reliability," in *ISCA*, 2019.
[43] H. Hassan, Y. C. Tugrul, J. S. Kim, V. v. d. Veen, K. Razavi, and O. Mutlu, "Uncovering in-DRAM RowHammer Protection Mechanisms: A New Methodology, Custom RowHammer Patterns, and Implications," in *MICRO*, 2021.
[44] H. Hassan, N. Vijaykumar, S. Khan, S. Ghose, K. Chang, G. Pekhimenko, D. Lee, O. Ergin, and O. Mutlu, "SoftMC: A Flexible and Practical Open-Source Infrastructure for Enabling Experimental DRAM Studies," in *HPCA*, 2017.
[45] S. Hong, P. Frigo, Y. Kaya, C. Giuffrida, and T. Dumitraş, "Terminal Brain Damage: Exposing the Graceless Degradation in Deep Neural Networks Under Hardware Fault Attacks," in *USENIX Security*, 2019.
[46] S. Hong, D. Kim, J. Lee, R. Oh, C. Yoo, S. Hwang, and J. Lee, "DSAC: Low-Cost Rowhammer Mitigation Using In-DRAM Stochastic and Approximate Counting Algorithm," arXiv:2302.03591, 2023.
[47] M. Horiguchi, "Redundancy Techniques for High-Density DRAMs," in *ISIS*, 1997.
[48] Intel, "Intel Core i5-10400 Processor," https://ark.intel.com/content/www/us/en/ark/products/199271/intel-core-i510400-processor-12m-cache-up-to-4-30-ghz.html.
[49] Intel, "Intel 64 and IA-32 Architectures Software Developer's Manual – Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4," https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html, 2022.
[50] Y. Ito and Y. He, "Apparatus and Methods for Refreshing Memory," U.S. Patent 11062754B2, 2019.
[51] K. Itoh, *VLSI Memory Chip Design*. Springer, 2001.
[52] Y. Jang, J. Lee, S. Lee, and T. Kim, "SGX-Bomb: Locking Down the Processor via Rowhammer Attack," in *SOSP*, 2017.

[53] P. Jattke, V. van der Veen, P. Frigo, S. Gunter, and K. Razavi, "Blacksmith: Scalable Rowhammering in the Frequency Domain," in *SP*, 2022.
[54] JEDEC, *JESD79-3: DDR3 SDRAM Standard*, 2012.
[55] JEDEC, *JESD209-4B: Low Power Double Data Rate 4 (LPDDR4) Standard*, 2017.
[56] JEDEC, *JESD79-4C: DDR4 SDRAM Standard*, 2020.
[57] JEDEC, *JESD79-5: DDR5 SDRAM Standard*, 2020.
[58] S. Ji, Y. Ko, S. Oh, and J. Kim, "Pinpoint Rowhammer: Suppressing Unwanted Bit Flips on Rowhammer Attacks," in *ASIACCS*, 2019.
[59] O. Kahn and J. Wilcox, "Method for Dynamically Adjusting a Memory Page Closing Policy," 2004.
[60] I. Kang, E. Lee, and J. H. Ahn, "CAT-TWO: Counter-Based Adaptive Tree, Time Window Optimized for DRAM Row-Hammer Prevention," *IEEE Access*, 2020.
[61] B. Keeth and R. Baker, *DRAM Circuit Design: A Tutorial*. Wiley, 2001.
[62] M. N. I. Khan and S. Ghosh, "Analysis of Row Hammer Attack on STTRAM," in *ICCD*, 2018.
[63] S. Khan, D. Lee, Y. Kim, A. R. Alameldeen, C. Wilkerson, and O. Mutlu, "The Efficacy of Error Mitigation Techniques for DRAM Retention Failures: A Comparative Experimental Study," in *SIGMETRICS*, 2014.
[64] S. Khan, D. Lee, and O. Mutlu, "PARBOR: An Efficient System-Level Technique to Detect Data-Dependent Failures in DRAM," in *DSN*, 2016.
[65] S. Khan, C. Wilkerson, Z. Wang, A. R. Alameldeen, D. Lee, and O. Mutlu, "Detecting and Mitigating Data-Dependent DRAM Failures by Exploiting Current Memory Content," in *MICRO*, 2017.
[66] D.-H. Kim, P. J. Nair, and M. K. Qureshi, "Architectural Support for Mitigating Row Hammering in DRAM Memories," *CAL*, 2015.
[67] J. S. Kim, M. Patel, A. G. Yağlıkçı, H. Hassan, R. Azizi, L. Orosa, and O. Mutlu, "Revisiting RowHammer: An Experimental Analysis of Modern Devices and Mitigation Techniques," in *ISCA*, 2020.
[68] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," in *ISCA*, 2014.
[69] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers," in *HPCA*, 2010.
[70] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *MICRO*, 2010.
[71] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A Fast and Extensible DRAM Simulator," *CAL*, 2016.
[72] A. Kogler, J. Juffinger, S. Qazi, Y. Kim, M. Lipp, N. Boichat, E. Shiu, M. Nissler, and D. Gruss, "Half-Double: Hammering From the Next Row Over," in *USENIX Security*, 2022.
[73] R. K. Konoth, M. Oliverio, A. Tatar, D. Andriesse, H. Bos, C. Giuffrida, and K. Razavi, "ZebRAM: Comprehensive and Compatible Software Protection Against Rowhammer Attacks," in *OSDI*, 2018.
[74] A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, "RAMBleed: Reading Bits in Memory Without Accessing Them," in *S&P*, 2020.
[75] L. Lantz, "Soft Errors Induced by Alpha Particles," in *IEEE Transactions on Reliability*, 1996.
[76] Launchpad, "linux 5.4.0-131.147 source package in Ubuntu," https://launchpad.net/ubuntu/+source/linux/5.4.0-131.147, 2022.
[77] D. Lee, S. Khan, L. Subramanian, S. Ghose, R. Ausavarungnirun, G. Pekhimenko, V. Seshadri, and O. Mutlu, "Design-Induced Latency Variation in Modern DRAM Chips: Characterization, Analysis, and Latency Reduction Mechanisms," in *SIGMETRICS*, 2017.
[78] E. Lee, I. Kang, S. Lee, G. Edward Suh, and J. Ho Ahn, "TWiCe: Preventing Row-Hammering by Exploiting Time Window Counters," in *ISCA*, 2019.
[79] H. Li, H.-Y. Chen, Z. Chen, B. Chen, R. Liu, G. Qiu, P. Huang, F. Zhang, Z. Jiang, B. Gao, L. Liu, X. Liu, S. Yu, H.-S. P. Wong, and J. Kang, "Write Disturb Analyses on Half-Selected Cells of Cross-Point RRAM Arrays," in *IRPS*, 2014.
[80] C. Lim, K. Park, and S. Baeg, "Active Precharge Hammering to Monitor Displacement Damage Using High-Energy Protons in 3x-nm SDRAM," *TNS*, 2017.
[81] M. Lipp, M. T. Aga, M. Schwarz, D. Gruss, C. Maurice, L. Raab, and L. Lamster, "Nethammer: Inducing Rowhammer Faults Through Network Requests," arXiv:1805.04956 [cs.CR], 2018.
[82] J. Liu, B. Jaiyen, Y. Kim, C. Wilkerson, O. Mutlu, J. Liu, B. Jaiyen, Y. Kim, C. Wilkerson, and O. Mutlu, "An Experimental Study of Data Retention Behavior in Modern DRAM Devices," in *ISCA*, 2013.
[83] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu, "RAIDR: Retention-Aware Intelligent DRAM Refresh," in *ISCA*, 2012.
[84] L. Liu, Y. Guo, Y. Cheng, Y. Zhang, and J. Yang, "Generating Robust DNN with Resistance to Bit-Flip based Adversarial Weight Attack," *IEEE Transactions on Computers*, 2022.
[85] D. Locklear, "Chipkill Correct Memory Architecture," *Dell Enterprise Systems Group, Technology Brief*, 2000.
[86] H. Luo, A. Olgun, A. G. Yağlıkçı, Y. C. Tuğrul, S. Rhyner, M. B. Cavlak, J. Lindegger, M. Sadrosadati, and O. Mutlu, "RowPress: Amplifying Read Disturbance in Modern DRAM Chips," *arXiv*, 2023.
[87] M. Marazzi, P. Jattke, F. Solt, and K. Razavi, "ProTRR: Principled yet Optimal In-DRAM Target Row Refresh," in *S&P*, 2022.
[88] M. Marazzi, F. Solt, P. Jattke, K. Takashi, and K. Razavi, "REGA: Scalable Rowhammer Mitigation with Refresh-Generating Activations," in *S&P*, 2023.
[89] Maxwell, "FT20X," https://www.maxwell-fa.com/upload/files/base/8/m/311.pdf.
[90] T. May and M. Woods, "Alpha-Particle-Induced Soft Errors in Dynamic Memories," in *IEEE Transactions on Electron Devices*, 1979.
[91] I. C. Memory, "Advanced ECC Memory for the IBM Netfinity 7000 M10," *Enhancing IBM Nethnity Server Reliability*.
[92] J. Meza, Q. Wu, S. Kumar, and O. Mutlu, "Revisiting Memory Errors in Large-Scale Production Data Centers: Analysis and Modeling of New Trends from the Field," in *DSN*, 2015.
[93] T. Moscibroda and O. Mutlu, "Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems," in *USENIX Security*, 2007.
[94] O. Mutlu, "The RowHammer Problem and Other Issues We May Face as Memory Becomes Denser," in *DATE*, 2017.
[95] O. Mutlu and J. S. Kim, "RowHammer: A Retrospective," *TCAD*, 2019.
[96] O. Mutlu and T. Moscibroda, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors," in *MICRO*, 2007.
[97] ——, "Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems," in *ISCA*, 2008.
[98] O. Mutlu, A. Olgun, and A. G. Yağlıkçı, "Fundamentally Understanding and Solving RowHammer," in *ASP-DAC*, 2023.
[99] K. Ni, X. Li, J. A. Smith, M. Jerry, and S. Datta, "Write Disturb in Ferroelectric FETs and Its Implication for 1T-FeFET AND Memory Arrays," *IEEE EDL*, 2018.
[100] T. O'Gorman, in *The Effect of Cosmic Rays on the Soft Error Rate of a DRAM at Ground Level*, 1994.
[101] A. Olgun, H. Hassan, A. G. Yaglikci, Y. C. Tugrul, L. Orosa, H. Luo, M. Patel, O. Ergin, and O. Mutlu, "DRAM Bender: An Extensible and Versatile FPGA-based Infrastructure to Easily Test State-of-the-art DRAM Chips," arXiv:2211.05838, 2022.
[102] L. Orosa, U. Rührmair, A. G. Yaglikci, H. Luo, A. Olgun, P. Jattke, M. Patel, J. Kim, K. Razavi, and O. Mutlu, "SpyHammer: Using RowHammer to Remotely Spy on Temperature," arXiv:2210.04084, 2022.
[103] L. Orosa, A. G. Yağlıkçı, H. Luo, A. Olgun, J. Park, H. Hassan, M. Patel, J. S. Kim, and O. Mutlu, "A Deeper Look into RowHammer's Sensitivities: Experimental Analysis of Real DRAM Chips and Implications on Future Attacks and Defenses," in *MICRO*, 2021.
[104] G. Papadimitriou and D. Gizopoulos, "Demystifying the system vulnerability stack: Transient fault effects across the layers," in *ISCA*, 2021.
[105] K. Park, S. Baeg, S. Wen, and R. Wong, "Active-Precharge Hammering on a Row-Induced Failure in DDR3 SDRAMs Under 3x nm Technology," in *IIRW*, 2014.
[106] K. Park, C. Lim, D. Yun, and S. Baeg, "Experiments and Root Cause Analysis for Active-Precharge Hammering Fault in DDR3 SDRAM under 3xnm Technology," *Microelectronics Reliability*, 2016.
[107] K. Park, D. Yun, and S. Baeg, "Statistical Distributions of Row-Hammering Induced Failures in DDR3 Components," *Microelectronics Reliability*, 2016.
[108] S.-I. Park and I.-C. Park, "History-Based Memory Mode Prediction For Improving Memory Performance," in *ISCAS*, 2003.
[109] Y. Park, W. Kwon, E. Lee, T. J. Ham, J. H. Ahn, and J. W. Lee, "Graphene: Strong yet Lightweight Row Hammer Protection," in *MICRO*, 2020.
[110] M. Patel, J. Kim, T. Shahroodi, H. Hassan, and O. Mutlu, "Bit-Exact ECC Recovery (BEER): Determining DRAM On-Die ECC Functions by Exploiting DRAM Data Retention Characteristics (Best Paper)," in *MICRO*, 2020.
[111] M. Patel, J. S. Kim, and O. Mutlu, "The Reach Profiler (REAPER): Enabling the Mitigation of DRAM Retention Failures via Profiling at Aggressive Conditions," in *ISCA*, 2017.
[112] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks," in *USENIX Security*, 2016.
[113] D. Poddebniak, J. Somorovsky, S. Schinzel, M. Lochter, and P. Rösler, "Attacking Deterministic Signature Schemes using Fault Attacks," in *EuroS&P*, 2018.
[114] R. Qiao and M. Seaborn, "A New Approach for RowHammer Attacks," in *HOST*, 2016.
[115] M. Qureshi, "Rethinking ECC in the Era of Row-Hammer," *DRAMSec*, 2021.
[116] M. Qureshi, A. Rohan, G. Saileshwar, and P. J. Nair, "Hydra: Enabling Low-Overhead Mitigation of Row-Hammer at Ultra-Low Thresholds via Hybrid Tracking," in *ISCA*, 2022.
[117] A. S. Rakin, M. H. I. Chowdhuryy, F. Yao, and D. Fan, "DeepSteal: Advanced Model Extractions Leveraging Efficient Weight Stealing in Memories," in *SP*, 2022.
[118] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, "Flip Feng Shui: Hammering a Needle in the Software Stack," in *USENIX Security*, 2016.
[119] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory Access Scheduling," in *ISCA*, 2000.
[120] T. G. Rokicki, "Method and computer system for speculatively closing pages in memory," U.S. Patent 6389514B1, 2002.

[121] S.-W. Ryu, K. Min, J. Shin, H. Kwon, D. Nam, T. Oh, T.-S. Jang, M. Yoo, Y. Kim, and S. Hong, "Overcoming the Reliability Limitation in the Ultimately Scaled DRAM using Silicon Migration Technique by Hydrogen Annealing," in *IEDM*, 2017.

[122] SAFARI Research Group, "DRAM Bender — GitHub Repository," https://github.com/CMU-SAFARI/DRAM-Bender.

[123] SAFARI Research Group, "Ramulator — GitHub Repository," https://github.com/CMU-SAFARI/ramulator.

[124] SAFARI Research Group, "RowHammer — GitHub Repository," https://github.com/CMU-SAFARI/rowhammer.

[125] SAFARI Research Group, "RowPress Artifact — GitHub Repository," https://github.com/CMU-SAFARI/RowPress.

[126] ——, "SoftMC — GitHub Repository," https://github.com/CMU-SAFARI/softmc.

[127] SAFARI Research Group, "RowPress Artifact — Zenodo Repository," https://doi.org/10.5281/zenodo.7750890, 2023.

[128] G. Saileshwar, B. Wang, M. Qureshi, and P. J. Nair, "Randomized Row-Swap: Mitigating Row Hammer by Breaking Spatial Correlation Between Aggressor and Victim Rows," in *ASPLOS*, 2022.

[129] Samsung Electronics, "288pin Unbuffered DIMM based on 8Gb C-die," https://download.semiconductor.samsung.com/resources/data-sheet/DDR4_8Gb_C_die_Unbuffered_DIMM_Rev1.4_Apr.18.pdf.

[130] B. Sander, P. Madrid, and G. Samus, "Dynamic Idle Counter Threshold Value for Use in Memory Paging Policy," 2005.

[131] M. Seaborn and T. Dullien, "Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges," http://googleprojectzero.blogspot.com.tr/2015/03/exploiting-dram-rowhammer-bug-to-gain.html, 2015.

[132] Seaborn, Mark and Dullien, Thomas, "Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges," *Black Hat*, 2015.

[133] O. Seongil, Y. H. Son, N. S. Kim, and J. H. Ahn, "Row-buffer Decoupling: A Case for Low-Latency DRAM Microarchitecture," in *ISCA*, 2014.

[134] V. Seshadri, T. Mullins, A. Boroumand, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Gather-Scatter DRAM: In-DRAM Address Translation to Improve the Spatial Locality of Non-Unit Strided Accesses," in *MICRO*, 2015.

[135] S. M. Seyedzadeh, A. K. Jones, and R. Melhem, "Mitigating Wordline Crosstalk Using Adaptive Trees of Counters," in *ISCA*, 2018.

[136] R. T. Smith, J. D. Chlipala, J. F. M. Bindels, R. G. Nelson, F. H. Fischer, and T. F. Mantz, "Laser Programmable Redundancy and Yield Improvement in a 64K DRAM," *JSSC*, 1981.

[137] A. Snavely and D. M. Tullsen, "Symbiotic Job Scheduling for A Simultaneous Multithreaded Processor," in *ASPLOS*, 2000.

[138] M. Son, H. Park, J. Ahn, and S. Yoo, "Making DRAM Stronger Against Row Hammering," in *DAC*, 2017.

[139] Standard Performance Evaluation Corp., "SPEC CPU 2006," http://www.spec.org/cpu2006/.

[140] Standard Performance Evaluation Corp., "SPEC CPU 2017," http://www.spec.org/cpu2017/.

[141] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu, "BLISS: Balancing Performance, Fairness and Complexity in Memory Access Scheduling," *TPDS*, 2016.

[142] L. Subramanian, K. Vaidyanathan, A. Nori, S. Subramoney, T. Karnik, and H. Wang, "Closed yet Open DRAM: Achieving Low Latency and High Performance in DRAM Memory Systems," in *DAC*, 2018.

[143] M. B. Sullivan, N. R. Saxena, M. O'Connor, D. Lee, P. Racunas, S. Hukerikar, T. Tsai, S. K. S. Hari, and S. W. Keckler, "Characterizing and Mitigating Soft Errors in GPU DRAM," *IEEE Micro*, 2022.

[144] D. Tang, P. Carruthers, Z. Totari, and M. Shapiro, "Assessment of the Effect of Memory Page Retirement on System RAS Against Hardware Faults," in *DSN*, 2006.

[145] A. Tatar, C. Giuffrida, H. Bos, and K. Razavi, "Defeating Software Mitigations Against Rowhammer: A Surgical Precision Hammer," in *RAID*, 2018.

[146] A. Tatar, R. K. Konoth, E. Athanasopoulos, C. Giuffrida, H. Bos, and K. Razavi, "Throwhammer: Rowhammer Attacks Over the Network and Defenses," in *USENIX ATC*, 2018.

[147] The Linux Kernel Archives, "Summary of Hugetlbpage Support," https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt, 2022.

[148] Y. Tobah, A. Kwong, I. Kang, D. Genkin, and K. G. Shin, "SpecHammer: Combining Spectre and Rowhammer for New Speculative Attacks," in *SP*, 2022.

[149] M. C. Tol, S. Islam, B. Sunar, and Z. Zhang, "Toward Realistic Backdoor Injection Attacks on DNNs using RowHammer," arXiv:2110.07683v2 [cs.LG], 2022.

[150] Transaction Processing Performance Council, "TPC-H," https://www.tpc.org/tpch.

[151] D. Tullsen and J. Brown, "Handling Long-Latency Loads in a Simultaneous Multithreading Processor," in *MICRO*, 2001.

[152] A. van de Goor and I. Schanstra, "Address and Data Scrambling: Causes and Impact on Memory Tests," in *DELTA*, 2002.

[153] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic Rowhammer Attacks on Mobile Platforms," in *CCS*, 2016.

[154] V. van der Veen, M. Lindorfer, Y. Fratantonio, H. P. Pillai, G. Vigna, C. Kruegel, H. Bos, and K. Razavi, "GuardION: Practical Mitigation of DMA-Based Rowhammer Attacks on ARM," in *DIMVA*, 2018.

[155] A. J. Walker, S. Lee, and D. Beery, "On DRAM RowHammer and the Physics on Insecurity," *IEEE TED*, 2021.

[156] Z. Weissman, T. Tiemann, D. Moghimi, E. Custodio, T. Eisenbarth, and B. Sunar, "JackHammer: Efficient Rowhammer on Heterogeneous FPGA–CPU Platforms," arXiv:1912.11523 [cs.CR], 2020.

[157] M. Wi, J. Park, S. Ko, M. J. Kim, N. Sung Kim, E. Lee, and J. H. Ahn, "SHADOW: Preventing Row Hammer in DRAM with Intra-Subarray Row Shuffling," in *HPCA*, 2023.

[158] G. D. Wolff, "Word Line Cache Mode," U.S. Patent 10366733B1, 2019.

[159] J. Woo, G. Saileshwar, and P. J. Nair, "Scalable and Secure Row-Swap: Efficient and Safe Row Hammer Mitigation in Memory Systems," in *HPCA*, 2023.

[160] Y. Xiao, X. Zhang, Y. Zhang, and R. Teodorescu, "One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation," in *USENIX Security*, 2016.

[161] Xilinx, "Xilinx Alveo U200 FPGA Board," https://www.xilinx.com/products/boards-and-kits/alveo/u200.html, 2021.

[162] Y. Xu, A. Agarwal, and B. Davis, "Prediction in Dynamic SDRAM Controller Policies," in *SAMOS*, 2009.

[163] A. G. Yağlıkçı, J. S. Kim, F. Devaux, and O. Mutlu, "Security Analysis of the Silver Bullet Technique for RowHammer Prevention," arXiv:2106.07084 [cs.CR], 2021.

[164] A. G. Yağlıkçı, H. Luo, G. F. Oliveira, A. Olgun, M. Patel, J. Park, H. Hassan, J. S. Kim, L. Orosa, and O. Mutlu, "Understanding RowHammer Under Reduced Wordline Voltage: An Experimental Study Using Real DRAM Devices," in *DSN*, 2022.

[165] A. G. Yağlikçi, A. Olgun, M. Patel, H. Luo, H. Hassan, L. Orosa, O. Ergin, and O. Mutlu, "HiRA: Hidden Row Activation for Reducing Refresh Latency of Off-the-Shelf DRAM Chips," in *MICRO*, 2022.

[166] A. G. Yağlıkçı, M. Patel, J. S. Kim, R. Azizibarzoki, A. Olgun, L. Orosa, H. Hassan, J. Park, K. Kanellopoullos, T. Shahroodi, S. Ghose, and O. Mutlu, "BlockHammer: Preventing RowHammer at Low Cost by Blacklisting Rapidly-Accessed DRAM Rows," in *HPCA*, 2021.

[167] C. Yang, C. K. Wei, Y. J. Chang, T. C. Wu, H. P. Chen, and C. S. Lai, "Suppression of RowHammer Effect by Doping Profile Modification in Saddle-Fin Array Devices for Sub-30-nm DRAM Technology," *IEEE Transactions on Device and Materials Reliability*, 2016.

[168] C.-M. Yang, C.-K. Wei, H.-P. Chen, J.-S. Luo, Y. J. Chang, T.-C. Wu, and C.-S. Lai, "Scanning Spreading Resistance Microscopy for Doping Profile in Saddle-Fin Devices," *IEEE Transactions on Nanotechnology*, 2017.

[169] T. Yang and X.-W. Lin, "Trap-Assisted DRAM Row Hammer Effect," *EDL*, 2019.

[170] F. Yao, A. S. Rakin, and D. Fan, "DeepHammer: Depleting the Intelligence of Deep Neural Networks Through Targeted Chain of Bit Flips," in *USENIX Security*, 2020.

[171] J. M. You and J.-S. Yang, "MRLoc: Mitigating Row-Hammering Based on Memory Locality," in *DAC*, 2019.

[172] D. Yun, M. Park, C. Lim, and S. Baeg, "Study of TID Effects on One Row Hammering using Gamma in DDR4 SDRAMs," in *IRPS*, 2018.

[173] Z. Zhang, Z. Zhan, D. Balasubramanian, X. Koutsoukos, and G. Karsai, "Triggering Rowhammer Hardware Faults on ARM: A Revisit," in *ASHES*, 2018.

[174] Z. Zhang, Y. Cheng, D. Liu, S. Nepal, Z. Wang, and Y. Yarom, "PThammer: Cross-User-Kernel-Boundary Rowhammer through Implicit Accesses," in *MICRO*, 2020.

[175] Z. Zhang, W. He, Y. Cheng, W. Wang, Y. Gao, D. Liu, K. Li, S. Nepal, A. Fu, and Y. Zou, "Implicit Hammer: Cross-Privilege-Boundary Rowhammer through Implicit Accesses," *IEEE Transactions on Dependable and Secure Computing*, 2022.

[176] M. Zheng, Q. Lou, and L. Jiang, "TrojViT: Trojan Insertion in Vision Transformers," arXiv:2208.13049, 2022.

[177] W. K. Zuravleff and T. Robinson, "Controller for a Synchronous DRAM That Maximizes Throughput by Allowing Memory Requests and Commands to Be Issued Out of Order," U.S. Patent 5630096, 1997.

A Artifact Description Appendix

A.1 Abstract

Our artifact [125, 127] contains the data, source code, and scripts needed to reproduce our results, including all figures in the paper. We provide: 1) original characterization data from our real-chip characterization (§4, §5) and source code of the DRAM Bender [101, 122] program used to perform the characterization, 2) the source code of our real-system demonstration (§6), and 3) the source code of the Ramulator [71, 123] implementation of our proposed RowPress mitigation (§7.4). We provide Python scripts and Jupyter Notebooks to analyze and plot the results for all three parts (referred to as *Characterization*, *Demonstration*, and *Mitigation*, respectively).

A.2 Artifact Check-list (Meta-information)

Parameter	Value
Program	C++ program Python3 scripts/Jupyter Notebooks Shell scripts
Compilation	C++17 compiler (tested with GCC 9)
Run-time environment	Ubuntu 20.04 (or similar) Linux Ubuntu 18.04 (with Linux kernel 5.4.0-131-generic [76]), used for reproducing *Demonstration* results Python 3.9+ DRAM Bender [101] Boost 1.71+ Xilinx Vivado 2020.2+ Slurm 20+
Hardware	x86 machine w/ PCIe 3.0 x16 slot FPGA development board supported by DRAM Bender (e.g., Xilinx Alveo U200) Temperature control setup for DRAM modules under test (e.g., Maxwell FT200)
Output	Data and execution logs in plain text and plots in pdf and png format
Experiment workflow	Perform characterizations (simulations), aggregate results, and run analysis scripts on the results
Experiment Customization	Possible. See §A.7.1 for details
Disk space requirement	≈ 1TB
Workflow preparation time	≈ 1 day
Experiment completion time	≈ 3 hours (Reproduce characterization figures with provided raw data) 3 to 4 weeks per DRAM module (Replicate characterization results) ≈ 5 days (Demonstration) ≈ 1 day (Mitigation)
Publicly available?	Zenodo (https://doi.org/10.5281/zenodo.7750890) Github (https://github.com/CMU-SAFARI/RowPress)
Code licenses	MIT

A.3 Description

A.3.1 How to Access

The artifact is available on Zenodo with DOI https://doi.org/10.5281/zenodo.7750890. The live repository is at https://github.com/CMU-SAFARI/RowPress.

A.3.2 Hardware Dependencies

Characterization. To reproduce our real-DRAM characterization results (figures) using the provided raw data from our experiments, a Linux workstation with 1TB free disk space is required (the data size is about 800GB before compression). To replicate our results, the reader needs a similar setup as shown in Fig. 4:

- A host x86 machine with a PCIe 3.0 x16 slot.
- An FPGA board with a DIMM/SODIMM slot supported by DRAM Bender [101, 122] (e.g., Xilinx Alveo U200 [161]).
- Heater pads attached to the DRAM module under test.
- A temperature controller (e.g., MaxWell FT200 [89]) connected to the heater pads and programmable by the host machine.

Demonstration. To reproduce our real-system demonstration of RowPress, the reader needs a system with an Intel Core i5 10400 (Comet Lake-S) [48] processor and a Samsung `M378A2K43CB1-CTD` DDR4 DRAM module with the 8Gb C-Dies from Mfr. S (`K4A8G085WC-BCTD`) [129]. We describe how to adapt our demonstration program to replicate our results on systems with a different processor and DRAM module in §A.7.2.

Mitigation. The Ramulator [71, 123] implementation of our proposed RowPress mitigation can be run on a Linux workstation. We recommend using a machine or a compute cluster with many CPU cores and large main memory to parallelize the simulation tasks.

A.3.3 Software Dependencies

- `GNU Make`, `CMake 3.10+`
- `C++17` build toolchain (tested with `GCC 9`)
- `boost 1.71+`
- Xilinx `Vivado 2020.2+`
- `pigz` for fast decompression of raw characterization data
- `Python 3.9+` with Jupyter Notebook
- `pip` packages: `pandas`, `scipy`, `matplotlib`, and `seaborn`
- Slurm 20+
- Ubuntu 18.04 (Linux kernel 5.4.0-131-generic [76]) for reproducing *Demonstration*

A.4 Installation

To reproduce our results, no system-level installation is needed for *Characterization* and *Mitigation*. For *Demonstration*, 1GB hugepage support is required to simplify the process of finding neighboring DRAM rows in a real system.

To replicate our real-DRAM characterization, please follow the instructions in DRAM Bender's Github repository [122] to install all dependencies to run DRAM Bender programs.

A.5 Experiment Workflow

A.5.1 Characterization (Reproducing Figures)

We describe how to reproduce all figures related to our real-DRAM characterization using the raw data from the artifact. For readers who wish to replicate our characterization results using their own infrastructure and DRAM modules, please see §A.7.1 for details.

(1) Extract raw characterization data (≈ 800GB):
```
$ tar -I pigz -pxvf rowpress_characterization_data.tar.gz
```
(2) Process the raw data into pandas dataframes:
```
$ cd characterization/analysis/scripts
$ DATA_ROOT=<path-to-data>
$ ./process_data_slurm.sh ${DATA_ROOT}
```

The processed characterization data will be placed at `characterization/analysis/processed_data/`. To reproduce all figures related to *Characterization*, open `characterization/analysis/-plots/paper_plots.ipynb` and run all code blocks. We use Markdown blocks in the notebook to clearly mark and explain all figures. The generated figures can be viewed both in the notebook and in `characterization/analysis/plots/output/`.

A.5.2 Demonstration

(1) Build the demonstration program:
```
$ cd demonstration/
$ make
```
(2) Run the program with root privilege (required only for accessing the hugepage) and analyze the bitflip results:
```
$ sudo ./mount_hugepage.sh # Should print 1 if successful
$ sudo demo --num_victims 1500 > bitflips.txt
$ python3 analyze.py bitflips.txt > parsed_results.txt
```
Open `real_system_bitflips.ipynb` and run all code blocks to analyze the results and reproduce Fig. 23.

(3) Verify that t_{AggON} increases (§6.3):
```
$ sudo ./disable_prefetching.sh
$ sudo demo --verify
```
Open `real_system_access.ipynb` and run all code blocks to reproduce Fig. 24.

A.5.3 Mitigation

Our artifact contains: 1) a modified version of Ramulator where we implement our proposed RowPress mitigation, 2) traces used to form workloads, and 3) scripts that automatically generate simulation configurations. The following instructions assume the reader is using Slurm to schedule a large number of parallelizable simulation jobs. Alternatively, readers can find the command lines for individual simulation jobs in the form of `mitigation/run_cmds/<config>-<workload>.sh` after executing step 2 to be used for their own job scheduler.

(1) Build ramulator:
```
$ cd mitigation/ramulator/
$ ./build.sh
```
(2) Generate simulation configurations and submit jobs:
```
$ python3 gen_jobs.py
$ ./run.sh
```

Executing the above generates Ramulator statistics files from the simulations in `mitigation/results`. The reader can then open the `mitigation/analyze.ipynb` Jupyter notebook and run all code blocks to reproduce our results in Table 3.

A.6 Evaluation and Expected Results

Running each of the experiments described in §A.5 is sufficient to reproduce all of 1) our real-chip characterization results (Fig. 1, Fig. 6 to Fig. 15, Fig. 17 to Fig. 20, Fig. 22, and Fig. 25), 2) real-system demonstration of RowPress (Fig. 23 and Fig. 24), and 3) simulation results of our proposed RowPress mitigation (Table 3).

A.7 Experiment Customization

A.7.1 Characterization

The source code of our RowPress characterization program is at `characterization/DRAM-Bender/sources/apps/RowPress/`. A python script `characterization/run.py` automates the experiments. Note that this script is tightly coupled to our internal DRAM testing infrastructure to provide ad-hoc functionalities (e.g., experiment and infrastructure status book-keeping, communicating with the temperature controller). Readers who wish to replicate our characterization on their own infrastructure can modify `characterization/run_bare.py`, which includes the infrastructure-independent experiment parameters, with `characterization/run.py` as a reference to perform the experiments on their own testing infrastructure. Performing all experiments for a single DRAM module takes about three to four weeks.

Our RowPress characterization program is highly configurable to test different DRAM modules, data and access patterns, aggressor row activation counts, t_{AggON}/t_{AggOFF} values, etc. Note that it is the responsibility of the reader's own DRAM testing infrastructure, not our characterization program, to control the temperature of the DRAM chips. We explain some key options in Table 4, and encourage the reader to refer to the help messages of the program for all options and their explanations.

Table 4: Key Options of RowPress Characterization Program

Option	Explanation
--help	Print all available options and their explanations.
--experiment	0 (Bitflips for given access pattern and activation count) 1 (ACmin for given access pattern) 3 (Retention failures for given refresh-idle time) 5 (Bitflips for given RowPress-ONOFF pattern and activation count)
--pattern_file	Path to a file specifying the data pattern and spatial layout of the aggressor and victim rows.
--hammer_count	The number of activations per aggressor row.
--RAS_scale	The increase in t_{AggON} beyond t_{RAS} (1 unit = 30ns).
--extra_cycles	$\Delta\ t_{A2A}$ for the RowPress-ONOFF pattern (1 unit = 6ns).
--RAS_ratio	Fraction of $\Delta\ t_{A2A}$ that contributes to t_{AggON}

A.7.2 Demonstration

On the system described in §A.3.2, the reader can change the number of victim rows to be tested using the demonstration program with the command line option `--num_victims`. The number of cache blocks accessed per aggressor row activation can be configured by modifying the `no_reads_arr` array in line 635 of `main.cpp`.

To successfully run the demonstration program on a different system (i.e., different processor and/or DRAM module) from that described in §A.3.2, the reader needs to perform the following:

(1) Reverse engineer the DRAM address mapping of the memory controller of the processor.
(2) Obtain a baseline access pattern (e.g., using U-TRR [43]) that can bypass the existing on-die RowHammer mitigation mechanism.
(3) Profile the system to obtain a threshold memory access latency that can be used to decide whether a DRAM refresh is happening (used to synchronize the access pattern with DRAM refresh).

We explain these steps and how to modify the demonstration program in `demonstration/README.md`.

A.7.3 Mitigation

The provided configurations can be evaluated with user-provided Ramulator traces. To include more traces in the job generation script, please modify the list of traces in `mitigation/gen_jobs.py`.

B Summary Tables of RowPress and RowHammer Characteristics of All Tested DRAM Modules

Table 5: Summary of all tested DDR4 modules and their RowHammer/RowPress vulnerabilities in terms of AC_{min} ($t_{AggON}{}_{min}$). We report the smaller AC_{min} ($t_{AggON}{}_{min}$) we observe from single-sided and double-sided versions of RowPress and RowHammer. Recall that AC_{min} is the minimum number of total aggressor row activations needed to induce at least one bitflip, and $t_{AggON}{}_{min}$ is the *minimum aggressor row on time (i.e., t_{AggON})* to induce at least one bitflip for a given aggressor row activation count.

Mfr.	DIMM Part	DRAM Part	Die Rev.[28]	Die Density	DQ	Date Code[29]	ID	RowHammer Vulnerability	RowPress Vulnerability		RowHammer Vulnerability	RowPress Vulnerability					
								AC_{min} @ Representative t_{AggON} Avg. (Min.)						$t_{AggON}{}_{min}$ (@ Representative AC) Avg. (Min.)			
								50°C			80°C			50°C		80°C	
								t_{AggON}=36ns (tRAS)	t_{AggON}=7.8us (tREFI)	t_{AggON}=70.2us (9xtREFI)	t_{AggON}=36ns (tRAS)	t_{AggON}=7.8us (tREFI)	t_{AggON}=70.2us (9xtREFI)	AC=1	AC=10K	AC=1	AC=10K
Samsung (Mfr. S)	M393A1K43BB1-CTD	K4A8G085WB-BCTD	B	8 Gb	x8	20-53	S0	279K (47K)	6.1K (1.6K)	682 (176)	295K (46K)	3.9K (776)	427 (87)	47.3 (12.4) ms	4.7 (1.3) us	24.8 (6.2) ms	2.5 (0.6) us
							S1	262K (38K)	6.3K (1.7K)	700 (187)	284K (41K)	4.5K (808)	486 (89)	49.4 (14.1) ms	4.9 (1.4) us	29.0 (6.6) ms	2.9 (0.8) us
	M378A2K43CB1-CTD	K4A8G085WC-BCTD	C	8 Gb	x8	N/A	S2	110K (24K)	6.4K (1.6K)	708 (179)	108K (23K)	5.3K (1.0K)	590 (107)	49.1 (13.0) ms	4.9 (1.3) us	33.9 (7.9) ms	3.4 (0.8) us
	M378A1K43DB2-CTD	K4A8G085WD-BCTD	D	8 Gb	x8	21-10	S3	41K (12K)	5.7K (1.3K)	627 (147)	43K (15K)	4.0K (835)	447 (79)	40.7 (11.4) ms	4.1 (1.2) us	23.4 (6.8) ms	2.4 (0.7) us
							S4	42K (13K)	5.5K (1.0K)	606 (107)	42K (13K)	4.5K (721)	493 (81)	38.7 (9.6) ms	3.9 (1.0) us	26.9 (6.6) ms	2.7 (0.7) us
							S5	41K (15K)	5.5K (932)	607 (98)	43K (13K)	4.2K (712)	470 (77)	38.7 (9.2) ms	3.9 (1.0) us	24.4 (5.5) ms	2.5 (0.6) us
	(G.Skill) F4-2400C17S-8GNT	K4A4G085WF-BCTD	F	4 Gb	x8	Mar. 21	S6	116K (20K)	6.4K (1.4K)	703 (147)	117K (21K)	3.0K (450)	328 (51)	48.5 (15.0) ms	4.9 (1.7) us	17.7 (5.7) ms	1.8 (0.7) us
							S7	129K (22K)	5.9K (1.5K)	651 (166)	130K (22K)	2.6K (682)	291 (75)	41.8 (13.5) ms	4.2 (1.4) us	14.4 (4.8) ms	1.5 (0.5) us
SK Hynix (Mfr. H)	HMAA4GU6AJR8N-XN	H5ANAG8NAJR-XN	A	16 Gb	x8	20-51	H0	119K (21K)	6.1K (1.8K)	680 (200)	112K (26K)	1.7K (380)	190 (43)	46.2 (14.3) ms	4.6 (1.4) us	10.0 (3.0) ms	1.0 (0.3) us
							H1	115K (24K)	6.9K (2.4K)	759 (268)	108K (25K)	2.7K (527)	299 (65)	53.5 (28.2) ms	5.4 (2.9) us	15.9 (5.6) ms	1.6 (0.6) us
	HMAA4GU7CJR8N-XN	H5ANAG8NCJR-XN	C	16 Gb	x8	21-36	H2	77K (14K)	6.7K (2.8K)	736 (316)	75K (17K)	3.8K (959)	422 (105)	51.9 (25.4) ms	5.1 (2.5) us	22.0 (7.5) ms	2.3 (0.8) us
							H3	78K (17K)	6.7K (1.3K)	7.8 (135)	76K (16K)	3.9K (739)	426 (87)	51.3 (9.8) ms	5.1 (1.0) us	22.6 (6.0) ms	2.3 (0.6) us
	(Kingston) KVR24R17S8/4	H5AN4G8NAFR-UHC	A	4 Gb	x8	19-46	H4	382K (83K)	No Bitflip	No Bitflip	373K (85K)	6.5K (2.7K)	719 (305)	No Bitflip	No Bitflip	50.8 (28.2) ms	5.1 (2.8) us
	(Corsair) CMV4GX4M1A2133C15	N/A	X	4 Gb	x8	N/A	H5	119K (20K)	6.8K (2.4K)	754 (259)	116K (21K)	2.3K (469)	259 (53)	53.5 (21.8) ms	5.3 (2.2) us	13.9 (4.1) ms	1.4 (0.5) us
Micron (Mfr. M)	MTA18ASF2G72PZ-2G3B1	MT40A2G4WE-083E:B	B	8 Gb	x4	N/A	M0	386K (87K)	No Bitflip	No Bitflip	367K (80K)	No Bitflip	No Bitflip	No Bitflip	No Bitflip	No Bitflip	No Bitflip
	MTA4ATF1G64HZ-3G2B2	MT40A1G16RC-062E:B	B	16 Gb	x16	21-26	M1	114K (24K)	7.1K (3.7K)	784 (403)	105K (23K)	6.3K (2.4K)	689 (259)	55.0 (35.2) ms	5.5 (3.4) us	44.5 (21.8) ms	4.5 (1.8) us
							M2	118K (26K)	7.0K (5.5K)	785 (621)	110K (22K)	7.0K (3.5K)	781 (379)	58.4 (56.8) ms	5.9 (5.8) us	55.0 (28.2) us	5.5 (2.8) us
	MTA36ASF8G72PZ-2G9E1	MT40A4G4JC-062E:E	E	16 Gb	x4	20-14	M3	41K (10K)	7.2K (2.4K)	770 (310)	39K (11K)	4.8K (815)	545 (91)	53.3 (28.1) ms	5.3 (2.8) us	28.3 (9.8) ms	2.8 (1.0) us
	MTA4ATF1G64HZ-3G2E1	MT40A1G16KD-062E:E	E	16 Gb	x16	20-46	M4	36K (12K)	7.0K (2.2K)	746 (245)	34K (10K)	4.1K (925)	468 (102)	52.3 (17.1) ms	5.2 (1.7) us	25.1 (7.2) ms	2.5 (0.8) us
							M5	40K (11K)	5.6K (1.2K)	610 (127)	37K (11K)	2.6K (616)	289 (67)	34.6 (9.0) ms	3.5 (0.9) us	15.8 (4.6) ms	1.6 (0.5) us
	MTA4ATF1G64HZ-3G2F1	MT40A1G16TB-062E:F	F	16 Gb	x16	21-50	M6	31K (8.7K)	6.7K (1.4K)	737 (181)	30K (8.2K)	3.4K (611)	381 (67)	50.9 (17.9) ms	5.1 (0.1) us	18.9 (6.4) us	1.9 (0.1) us

[28] We report the die revision marked on the DRAM chip package (if available). A die revision of "X" means the original markings on the DRAM chips package are removed by the DRAM module vendor, and thus the die revision could not be identified.

[29] In most cases, we report the date code of a DRAM module in the WW-YY format (i.e., 20-53 means the module is manufactured in the 53rd week of year 2020) as marked on the label of the module. We report the date codes of modules S6 and S7 in the "month-year" format because it is the only date marked on the labels. We report "N/A" if no date is marked on the label of a module.

Table 6: Summary of all tested DDR4 modules and their RowHammer/RowPress vulnerabilities in terms of maximum bit error rate. We report the maximum bit error rate at representative t_{AggON} values with the maximum activation count within 60ms.

Mfr.	DIMM Part	DRAM Part	Die Rev.[30]	Die Density	DQ	Date Code[31]	ID	RowHammer Vulnerability	RowPress Vulnerability		RowHammer Vulnerability	RowPress Vulnerability	
								Maximum Bit Error Rate @ Representative t_{AggON} and Maximum Activation Count Single-Sided (Double-Sided)					
								50°C			80°C		
								t_{AggON}=36ns (tRAS)	t_{AggON}=7.8us (tREFI)	t_{AggON}=70.2us (9xtREFI)	t_{AggON}=36ns (tRAS)	t_{AggON}=7.8us (tREFI)	t_{AggON}=70.2us (9xtREFI)
Samsung (Mfr. S)	M393A1K43BB1-CTD	K4A8G085WB-BCTD	B	8 Gb	x8	20-53	S0	0.1% (3.8%)	0.009% (0.005%)	0.009% (0.005%)	0.1% (3.6%)	0.09% (0.04%)	0.09% (0.04%)
							S1	0.2% (4.3%)	0.008% (0.003%)	0.008% (0.003%)	0.2% (3.8%)	0.07% (0.03%)	0.07% (0.03%)
	M378A2K43CB1-CTD	K4A8G085WC-BCTD	C	8 Gb	x8	N/A	S2	0.7% (9.5%)	0.02% (0.003%)	0.02% (0.003%)	0.8% (9.0%)	0.1% (0.02%)	0.1% (0.02%)
	M378A1K43DB2-CTD	K4A8G085WD-BCTD	D	8 Gb	x8	21-10	S3	7.7% (33.1%)	0.07% (0.01%)	0.08% (0.01%)	8.2% (32.5%)	0.6% (0.2%)	0.6% (0.1%)
							S4	5.2% (30.0%)	0.04% (0.02%)	0.04% (0.02%)	5.9% (30.0%)	0.2% (0.05%)	0.2% (0.04%)
							S5	7.8% (33.9%)	0.06% (0.01%)	0.06% (0.01%)	8.0% (33.0%)	0.3% (0.07%)	0.3% (0.07%)
	(G.Skill) F4-2400C17S-8GNT	K4A4G085WF-BCTD	F	4 Gb	x8	Mar. 21	S6	0.5% (7.9%)	0.02% (0.01%)	0.02% (0.01%)	0.6% (7.6%)	0.7% (0.3%)	0.8% (0.3%)
							S7	0.5% (7.6%)	0.03% (0.01%)	0.02% (0.01%)	0.6% (7.2%)	0.9% (0.3%)	1.0% (0.3%)
SK Hynix (Mfr. H)	HMAA4GU6AJR8N-XN	H5ANAG8NAJR-XN	A	16 Gb	x8	20-51	H0	1.0% (9.3%)	0.03% (0.01%)	0.03% (0.01%)	2.8% (10.7%)	9.4% (5.7%)	9.4% (5.7%)
							H1	1.1% (9.6%)	0.008% (0.002%)	0.006% (0.002%)	1.7% (10.8%)	3.9% (1.4%)	3.9% (1.3%)
	HMAA4GU7CJR8N-XN	H5ANAG8NCJR-XN	C	16 Gb	x8	21-36	H2	2.2% (14.0%)	0.002% (0.002%)	0.002% (0.002%)	2.6% (15.0%)	0.5% (0.1%)	0.5% (0.1%)
							H3	2.0% (13.0%)	0.003% (0.002%)	0.003% (0.002%)	2.0% (14.0%)	0.4% (0.1%)	0.4% (0.1%)
	(Kingston) KVR24R17S8/4	H5AN4G8NAFR-UHC	A	4 Gb	x8	19-46	H4	0.2% (1.1%)	No Bitflip	No Bitflip	0.2% (1.2%)	0.003% (0.002%)	0.003% (0.002%)
	(Corsair) CMV4GX4M1A2133C15	N/A	X	4 Gb	x8	N/A	H5	0.9% (9.0%)	0.005% (0.002%)	0.005% (0.002%)	1.7% (9.8%)	4.0% (1.6%)	3.8% (1.5%)
Micron (Mfr. M)	MTA18ASF2G72PZ-2G3B1	MT40A2G4WE-083E:B	B	8 Gb	x4	N/A	M0	0.3% (2.6%)	No Bitflip	No Bitflip	0.3% (3.0%)	No Bitflip	No Bitflip
	MTA4ATF1G64HZ-3G2B2	MT40A1G16RC-062E:B	B	16 Gb	x16	21-26	M1	1.2% (12.0%)	0.005% (0.002%)	0.005% (0.002%)	1.7% (13.2%)	0.03% (0.006%)	0.03% (0.005%)
							M2	1.3% (12.0%)	0.002% (No Bitflip)	0.002% (No Bitflip)	1.6% (12.8%)	0.003% (0.002%)	0.005% (0.002%)
	MTA36ASF8G72PZ-2G9E1	MT40A4G4JC-062E:E	E	16 Gb	x4	20-14	M3	7.4% (39.2%)	0.003% (0.009%)	0.003% (0.002%)	9.3% (41.3%)	0.1% (0.04%)	0.1% (0.03%)
	MTA4ATF1G64HZ-3G2E1	MT40A1G16KD-062E:E	E	16 Gb	x16	20-46	M4	9.0% (41.0%)	0.001% (0.001%)	0.009% (0.005%)	11.3% (43.7%)	0.4% (0.1%)	0.4% (0.06%)
							M5	8.6% (39.8%)	0.06% (0.02%)	0.06% (0.01%)	11.4% (43.2%)	2.6% (1.2%)	2.6% (1.0%)
	MTA4ATF1G64HZ-3G2F1	MT40A1G16TB-062E:F	F	16 Gb	x16	21-50	M6	7.1% (23.2%)	0.01% (0.02%)	0.01% (0.001%)	8.6% (24.3%)	1.0% (0.4%)	1.0% (0.3%)

[30] We report the die revision marked on the DRAM chip package (if available). A die revision of "X" means the original markings on the DRAM chips package are removed by the DRAM module vendor, and thus the die revision could not be identified.

[31] In most cases, we report the date code of a DRAM module in the WW-YY format (i.e., 20-53 means the module is manufactured in the 53rd week of year 2020) as marked on the label of the module. We report the date codes of modules S6 and S7 in the "month-year" format because it is the only date marked on the labels. We report "N/A" if no date is marked on the label of a module.

C Extended Characterization Results

C.1 Extended Characterization Results of the RowPress-ONOFF Pattern

We plot the bit error rate (*BER*) for both single-sided (top row of plots) and double-sided (bottom row of plots) RowPress-ONOFF pattern for all die revisions using the same methodology in §5.4 in the following figures (i.e., Fig. 27 to Fig. 37). We sweep Δt_{A2A} (different lines in each plot) and the percentage of Δt_{A2A} that contributes to t_{AggON} (x-axis) at 50°C (left column) and 80°C (right column). The error band shows the standard deviation of *BER*.

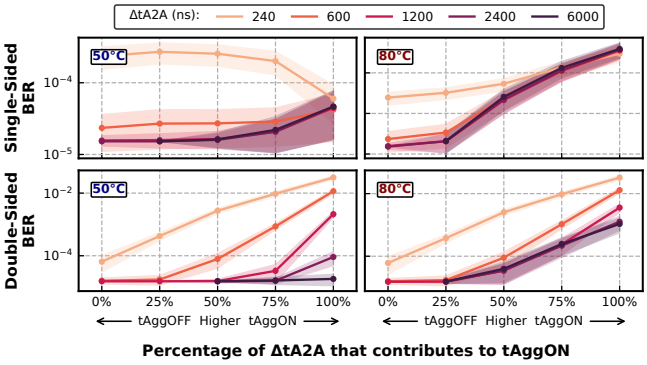

Figure 27: Mfr. S 4Gb F-Die.

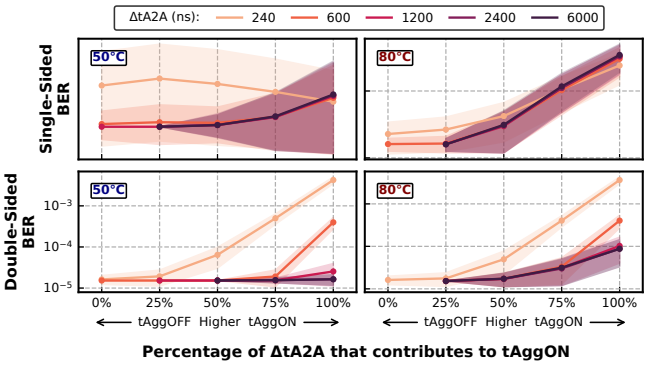

Figure 28: Mfr. S 8Gb B-Die.

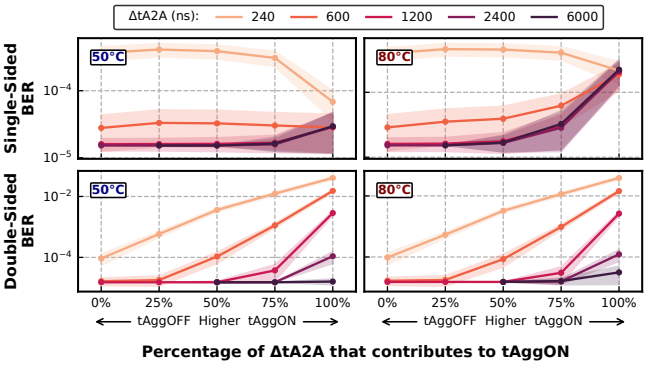

Figure 29: Mfr. S 8Gb C-Die.

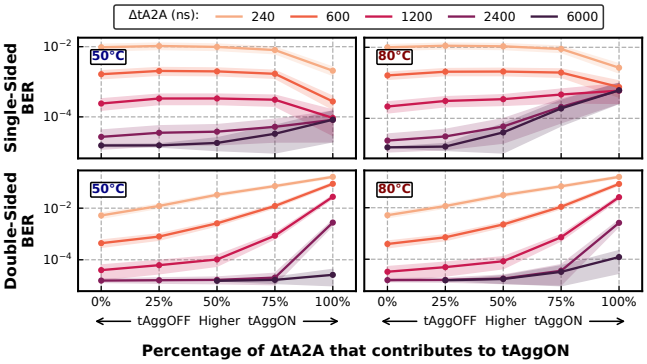

Figure 30: Mfr. S 8Gb D-Die.

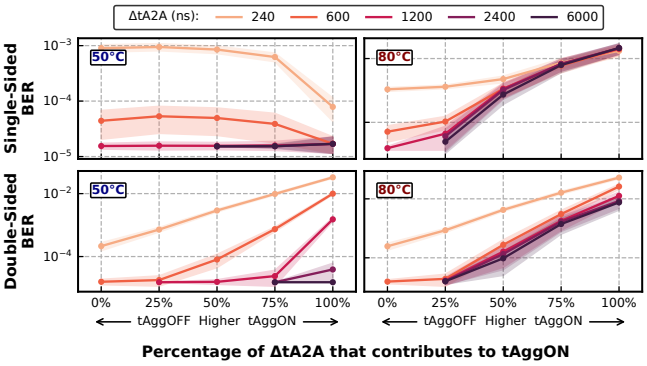

Figure 31: Mfr. H 4Gb X-Die.

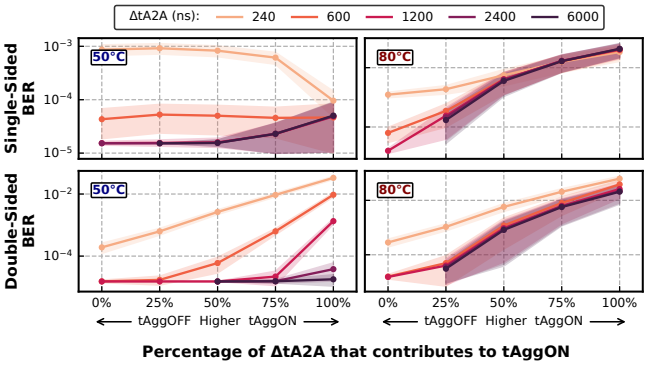

Figure 32: Mfr. H 16Gb A-Die.

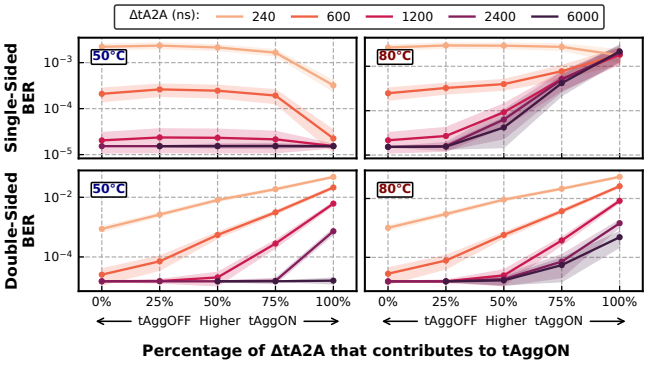

Figure 33: Mfr. H 16Gb C-Die.

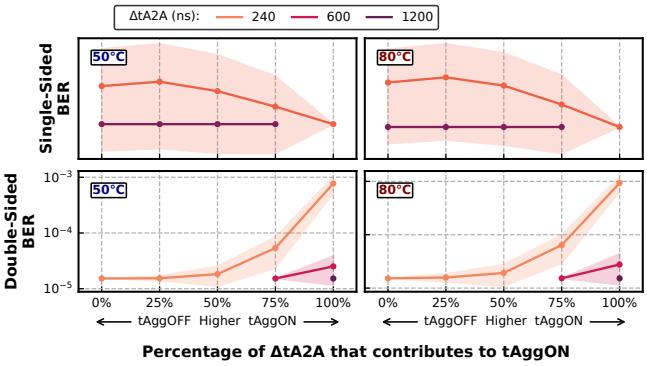

Figure 34: Mfr. M 8Gb B-Die.

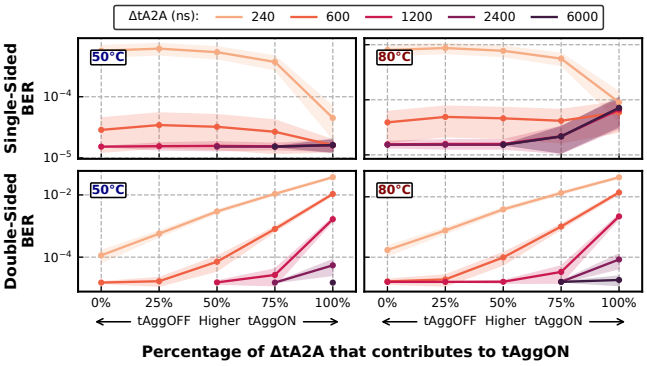

Figure 35: Mfr. M 16Gb B-Die.

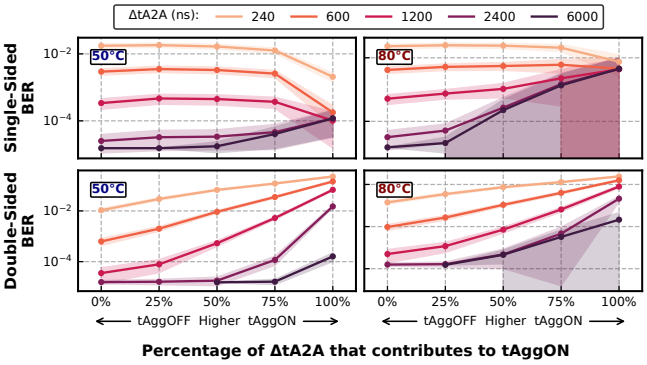

Figure 36: Mfr. M 16Gb E-Die.

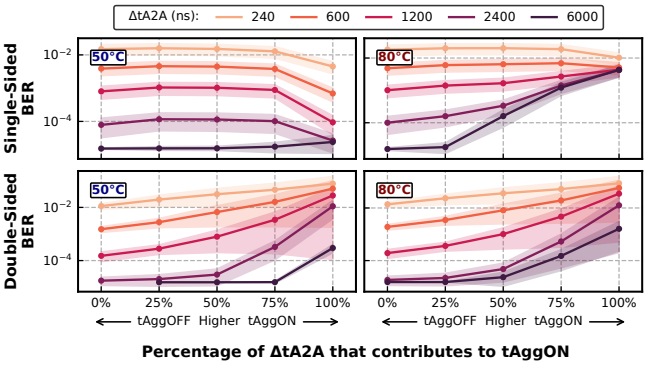

Figure 37: Mfr. M 16Gb F-Die.

D Extended Evaluation Results

D.1 Limiting the Maximum Row-Open Time

We evaluate 61 workloads from SPEC CPU2006 [139], SPEC CPU-2017 [140], TPC-H [150], and YCSB [21] using Ramulator [71, 123] with a realistic baseline system configuration, as shown in Table 7. We compare the increase in the number of activations to each individual DRAM row in a 64 ms (t_{REFW}) time window and the performance overhead with the minimally-open-row policy to those of the baseline open-row policy [119].

Table 7: Simulated system configuration.

Processor	4 GHz Out-of-Order core, 4-wide issue, 128-entry instruction window. 8 MSHRs per core, 2MiB LLC per core
Memory Controller	64-entry read/write request buffer, FR-FCFS request scheduling [119, 177], open-row policy [119]
DRAM	DDR4 [56] 3200MT/s, 1 channel, 2 rank, 4 bankgroups, 4 banks per bankgroup JEDEC DDR4-3200W Speedbin [56]

Fig. 38 shows the maximum increase in the number of activations to each individual DRAM row within t_{REFW}. For clarity, we only plot the workloads with a maximum increase over 50×.

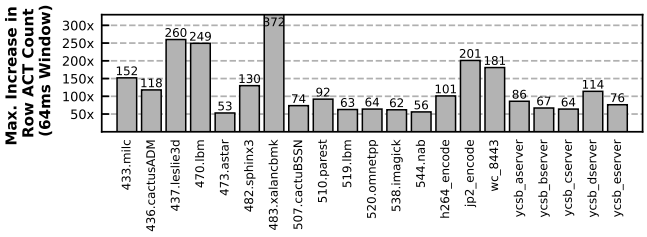

Figure 38: Maximum increase in the number of activations to each individual DRAM row in t_{REFW} with the minimally-open-row policy, compared to an open-row policy baseline.

We observe that using a minimally-open-row policy significantly increases the number of activations to a single DRAM row within t_{REFW} for a large group of workloads (i.e., 21 out of 58 workloads have at least 50× increase), by up to 372× (from only 1 to 372 activations for 483.xalancbmk). We also observe that, across all the rows accessed by the workloads, using a minimally-open-row policy significantly increases the number of activations to the most activated DRAM row. For example, the most activated row in 510.parest is only activated 497 times within t_{REFW} for the open-row policy, but this increases to 3808 times for the minimally-open-row policy. We find that 436.cactusADM, jp2_decode, 505.mcf, 471.omnetpp, and 483.xalancbmk also have their most-activated row activation count increased from less than 1000 to over 1000, which is used by many prior works (e.g., [67, 109, 116, 165, 166]) as a projected RowHammer threshold (T_{RH}), defined as the minimum number of aggressor row activations needed to cause a RowHammer bitflip, in the near future. We conclude that using a minimally-open-row policy can potentially turn benign workloads into a RowHammer attack [1, 10, 12–20, 22, 28, 30–32, 38–40, 43, 45, 52, 53, 58, 68, 72, 74, 81, 84, 94, 95, 102, 112–114, 117, 118, 124, 131, 132, 145, 146, 148, 149, 153, 154, 156, 160, 170, 173–176].

Fig. 39 shows the instruction per cycle (IPC) of the workloads with the minimally-open-row policy, normalized to the open-row policy baseline. For clarity, we only plot the workloads with a normalized IPC smaller than 0.95. We do *not* observe any workload with a normalized IPC higher than 1.0.

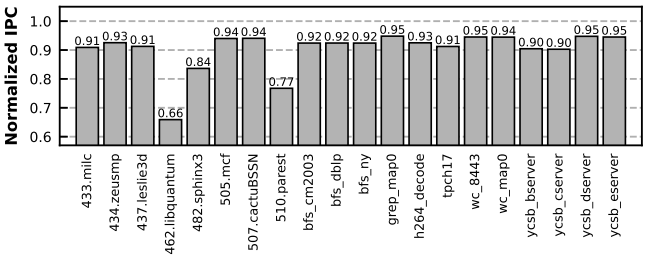

Figure 39: IPC of workloads when using the minimally-open-row policy, normalized to the baseline open-row policy.

We observe that using the minimally-open-row policy can significantly reduce the performance of workloads with high row buffer locality. For example, the IPC of 462.libquantum reduces by 34.1%, as its row buffer misses per kilo instructions (RBMPKI) increases by 110% from only 0.91 to 1.90. The performance of 510.parest reduces by 23.2%, as its RBMPKI increases by 62%. We conclude that using the minimally-open-row policy can significantly reduce system performance by reducing the row buffer hit rate.

Some existing row policy proposals adapt t_{mro} based on row access patterns (e.g., keep the row open for longer when the row is predicted to be accessed soon in the future) [5, 26, 59, 108, 120, 130, 162]. Such row policies *cannot* securely mitigate RowPress as t_{mro} can be controlled by an attacker to be set to larger values than t_{RAS}, as we show in §6. We believe securely mitigating RowPress requires co-designing existing RowHammer mitigations with a row policy that enforces t_{mro}, as §7.4 describes and evaluates.

D.2 Adapting Existing RowHammer Mitigations

Evaluation Methodology. We perform a sensitivity study of the performance overheads of Graphene-RP and PARA-RP over Graphene and PARA with the configurations shown in Table 8 using Ramulator [71, 123] with the same baseline system configuration in §7.3 on both single- and four-core multiprogrammed workloads. We evaluate both 1) homogeneous four-core workloads where we run four copies of each single-core workload on four cores, and 2) heterogeneous four-core workloads where we run different workloads on each core. To create the heterogeneous four-core workloads, we categorize the memory-intensity of the single-core workloads using two metrics: last-level cache misses per kilo instructions, LLC-MPKI, *and* row buffer misses per kilo instructions, RBMPKI. We group the single-core workloads into high-memory-intensity (i.e., LLC-MPKI ≥ 1 *and* RBMPKI ≥ 1), denoted as "H", and low-memory-intensity, (i.e., LLC-MPKI < 1 *and* RBMPKI < 1), denoted as "L". We evaluate five different groups

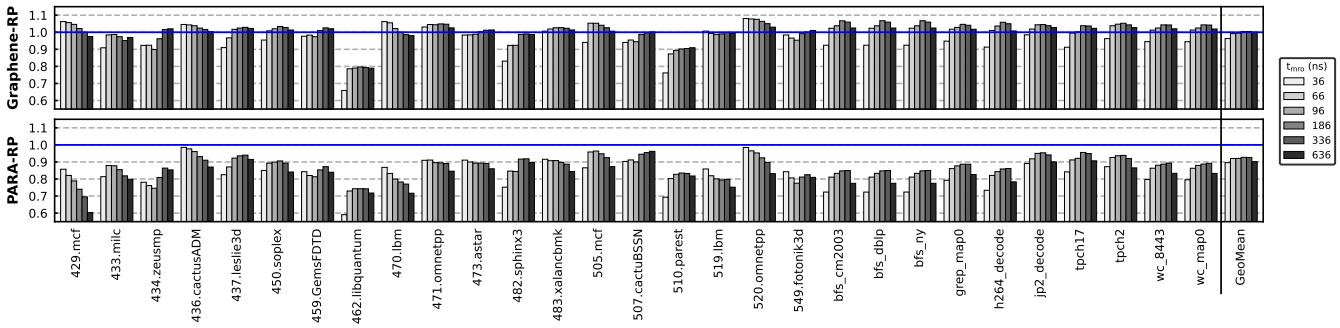

Figure 40: IPC of Graphene-RP and PARA-RP of single-core workloads (LLC-MPKI > 5) with different t_{mro} configurations, normalized to Graphene and PARA, respectively.

of heterogeneous workloads, denoted as HHHH, HHHL, HHLL, HLLL, and LLLL. For example, HHHH means all four workloads are from the "H" category, and HHLL means two workloads are from "H" and the other two are from "L". For each group, we evaluate eight different randomly picked workload mixes for a total of 40 heterogeneous four-core workloads. We use instruction per cycle (IPC) as the performance metric for single-core workloads, and weighted speedup [151] for four-core workloads.

Table 8: Graphene-RP and PARA-RP configurations for different t_{mro} values.

t_{mro} (ns)	36 (=t_{RAS})	66	96	186	336	636
T'_{RH}	1000 (=T_{RH})	809	724	619	555	419
Graphene-RP T	333	269	241	206	185	139
PARA-RP p	0.034	0.042	0.047	0.054	0.061	0.079

Evaluation Results. Fig. 40 shows the IPC of different Graphene-RP (top row) and PARA-RP (bottom row) configurations on the single-core workloads, normalized to Graphene and RP, respectively. For clarity, we only plot the workloads with more than five last-level-cache misses per kilo-instruction (LLC-MPKI > 5). We show the average (geometric mean) normalized IPC across *all* single-core workloads. We make the following observations.

First, Graphene-RP can mitigate RowPress with a slightly higher performance compared to Graphene. For single-core workloads, Graphene-RP slightly outperforms Graphene for all $t_{mro} \geq 66ns$, by up to 0.46% on average when $t_{mro} = 336ns$. The reason for the small speedups of Graphene-RP over Graphene is that enforcing a t_{mro} increases the performance of workloads with low row buffer hit rate. For example, the performance of 429.mcf (baseline RBMPKI = 68.6) increases significantly (normalized IPC increases from 0.97 to 1.06) as t_{mro} decreases from 636ns to 36ns. On the other hand, enforcing a t_{mro} reduces the performance of workloads with high row buffer hit rate. For example, the performance of 462.libquantum (baseline RBMPKI of only 0.91) decreases significantly (as low as 0.66 when t_{mro} is 36ns) over Graphene for all of the t_{mro} values we evaluate.

Second, PARA-RP can mitigate RowPress at low *additional* performance overhead for single-core workloads. For example, PARA-RP performs the best when $t_{mro} = 186ns$, with an average slowdown of only 7.3%. The reason for PARA-RP's consistent slowdown across different t_{mro} values over PARA is that PARA does *not* track aggressor rows deterministically (like Graphene). As a result, *any* extra row activations (i.e., even if the activations are *not* concentrated on a small number of rows) caused by the enforced t_{mro} will increase the number of (false-positive) preventive refreshes[32] issued by PARA. For example, when t_{mro} is 636ns, the number of preventive refreshes issued by PARA-RP (427074) is 17.6× more than Graphene-RP (23006), even though both PARA-RP and Graphene-RP have similar numbers of extra row activations (191447 for PARA-RP and 117229 for Graphene-RP) compared to the open-row baseline.

Fig. 41 shows the geometric means of the normalized weighted speedups of different Graphene-RP (top row) and PARA-RP (bottom row) configurations on homogeneous (left column) and heterogeneous (right column) four-core workloads. The error bars mark the lowest and highest normalized weighted speedups observed within a workload group. We make the following two observations.

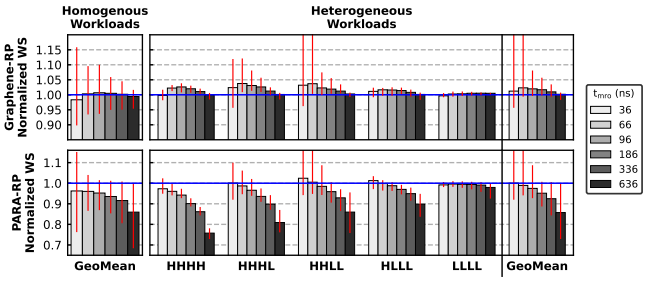

Figure 41: Geometric means of the normalized Weighted speedups of Graphene-RP and PARA-RP for homogeneous and heterogeneous four-core workloads with different t_{mro} configurations.

[32] A DRAM row is *preventively* refreshed to prevent bitflips before its adjacent row is activated too many times (i.e., T_{RH} times).

First, both Graphene-RP and PARA-RP have small performance overhead compared to Graphene and PARA, respectively. For homogeneous workloads, Graphene-RP outperforms Graphene by 0.67% when t_{mro} is 96ns, and PARA-RP performs the best with only 3.8% slowdown over PARA when t_{mro} is 36ns. Across all heterogeneous workloads, Graphene-RP outperforms Graphene by 2.3% when t_{mro} is 66ns, and PARA-RP can perform PARA by 0.03% when t_{mro} is 36ns. We notice that when t_{mro} is 36ns, both Graphene-RP and PARA-RP significantly improve the performance of certain workloads. For example, PARA-RP (Graphene-RP) has a speedup of 31.3% (28.8%) over PARA (Graphene) for a HHLL workload containing h264_encode. This is because h264_encode has a very high row buffer hit rate (87.0%) in the baseline, and thus gets unfairly prioritized by the memory controller's FR-FCFS [119, 177] scheduling and open-row policy.[33] A low t_{mro} value thus improves fairness between cores by allowing other workloads to progress and increases the weighted speedup.

Second, in contrast to single-core workloads, the performance overhead of PARA-RP always reduces as t_{mro} increases beyond 36ns. The reason is that PARA's performance overhead does *not* scale well with reducing T'_{RH} [67, 109, 166], and thus the performance benefits of longer row-open time and incrased row-buffer hit rate is outweighed by the performance overheads of the increased preventive refreshes.

We summarize our performance evaluation results of Graphene-RP and PARA-RP in Table 9. We conclude that existing RowHammer mitigations can be easily adapted to mitigate RowPress at low additional performance overhead. We expect future work to introduce and discuss new mitigation mechanisms in detail, as it has been happening analogously with RowHammer.

Table 9: Additional performance overhead of Graphene-RP and PARA-RP over Graphene and PARA for single-core and multi-core workloads.

t_{mro} (ns)	36(=t_{RAS})	66	96	186	336	636
Average Graphene-RP Perf. Overhead Over Graphene						
Single-core	3.7%	0.8%	0.5%	-0.4%	-0.5%	-0.05%
Homogeneous Multi-core	1.7%	-0.3%	-0.7%	-0.5%	-0.2%	0.6%
Heterogeneous Multi-core	-1.2%	-2.3%	-2.0%	-1.7%	-1.0%	-0.2%
Average PARA-RP Perf. Overhead Over PARA						
Single-core	10.4%	8.0%	7.9%	7.3%	7.4%	9.9%
Homogeneous Multi-core	3.8%	4.0%	4.8%	6.5%	8.4%	14.0%
Heterogeneous Multi-core	-0.0%	1.1%	2.5%	4.9%	7.5%	14.3%

[33] Such (un)fairness and resulting performance issues are well-studied by prior works [35, 69, 70, 93, 96, 97, 141].

E Repeatability of RowPress Bitflips

We study the *repeatability* of RowPress bitflips across all five iterations of our experiments. We define repeatability as the number of occurrences of a bitflip across all five iterations (i.e., ranges from 1 to 5, the higher the number of occurrences, the higher the repeatability). Fig. 42 is a histogram of the distribution of the repeatability of RowPress bitflips from our experiments described in §4.2. The y-axis shows the percentage of bitflips with different repeatability (from 1 to 5, x-axis). We plot representative t_{AggON} values in different rows of plots.

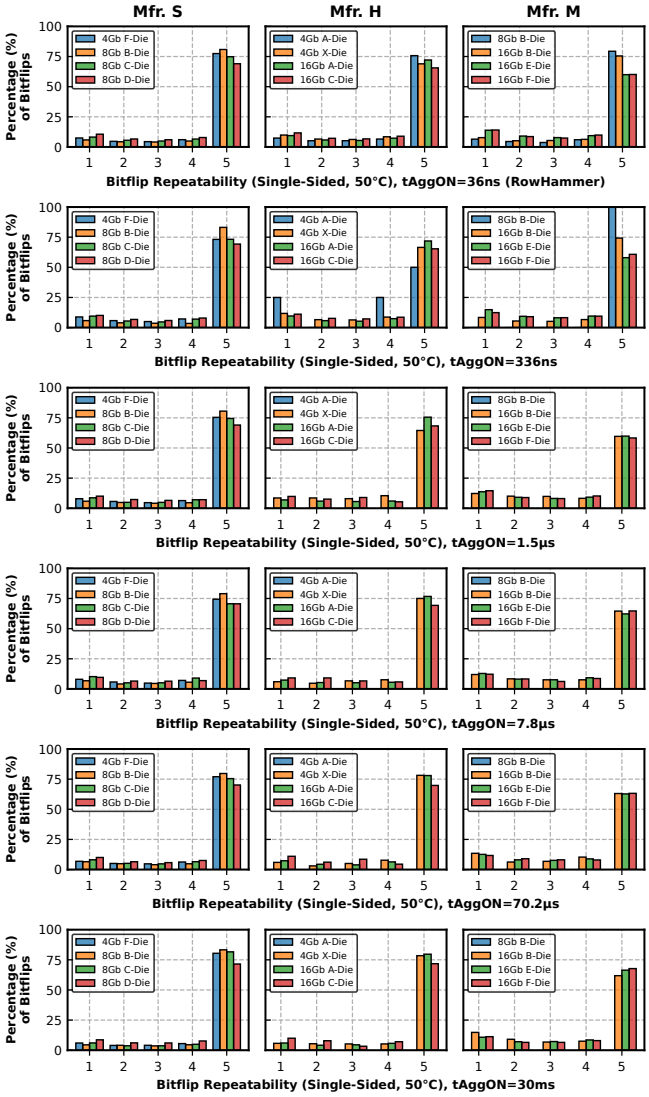

Figure 42: Repeatability of the single-sided RowPress (RowHammer) bitflips; $50°C$

We observe that the majority of the RowPress bitflips are repeatable across all five iterations. For all t_{AggON} values we test and for almost all die revisions from all three major DRAM manufacturers, at least 50% of the bitflips occurs in all five iterations. Even when

t_{AggON} is 30ms, the lowest percentage of bitflips that occur in all five iterations is still 61.9% (observed from 16Gb B-Dies from Mfr. M). We conclude that RowPress bitflips are repeatable, similar to RowHammer bitflips [68].

Obsv. 22. RowPress bitflips are repeatable, i.e., if they occur once in a cell, they are likely to occur again and again.

Fig. 43, Fig. 44, and Fig. 45 show the percentage of bitflips (y-axis) with different repeatability (x-axis) based on the single-sided access pattern at $80°C$, double-sided access pattern at $50°C$, and double-sided access pattern at $80°C$, respectively. We plot representative t_{AggON} values in different rows of the plots. The lowest percentage of bitflips that occur in all five iterations is 56.8% (observed from 16Gb C-Dies from Mfr. H) for the single-sided pattern at $80°C$. For the double-sided pattern, the lowest percentage of bitflips that occur in all five iterations is 33.3% at $50°C$ (observed from 16Gb B-Dies from Mfr. M) and 47.2% at $80°C$ (observed from 16Gb E-Dies from Mfr. M). We conclude that RowPress bitflips are repeatable with both single-sided and double-sided access patterns and at a higher temperature of $80°C$.

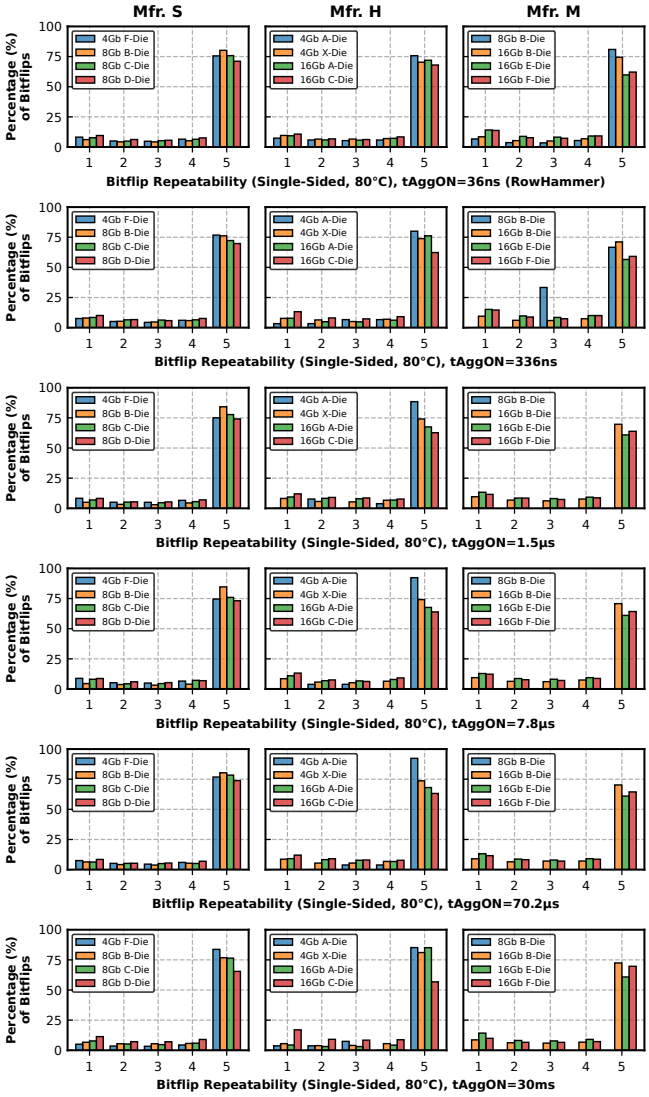

Figure 43: Repeatability of the single-sided RowPress (RowHammer) bitflips; $80°C$

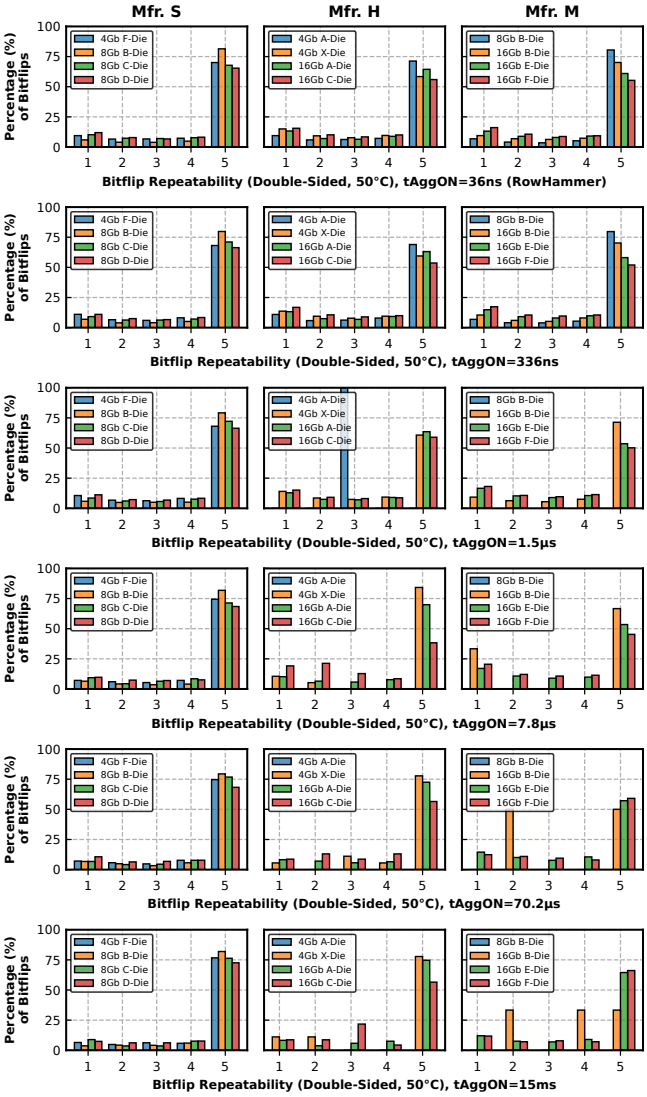

Figure 44: Repeatability of the double-sided RowPress (RowHammer) bitflips; $50°C$

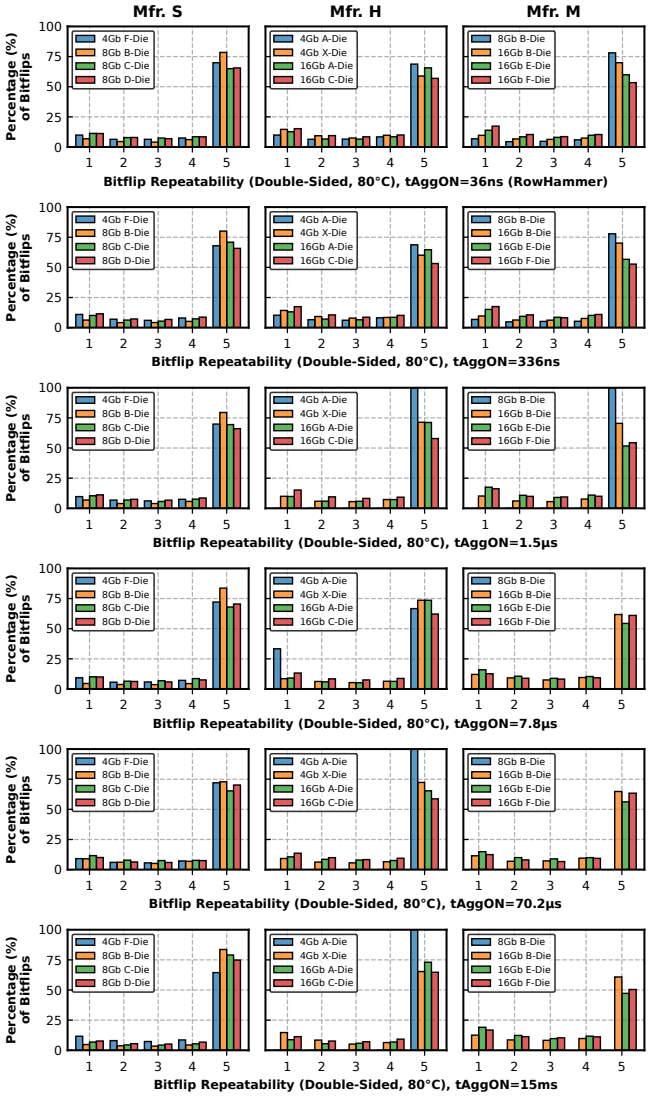

Figure 45: Repeatability of the double-sided RowPress (RowHammer) bitflips; $80°C$

F Extended Results on the Effect of Temperature on RowPress Bitflips

We conduct further experiments to characterize RowPress bitflips at 65°C to strengthen our observations that RowPress gets worse as temperature increases (Obsv. 9), and behaves differently compared to RowHammer as temperature and access pattern changes (Obsv. 13). Fig. 46 (Fig. 47) shows the mean AC_{min} values we observe at 65°C (80°C) normalized to 50°C (65°C) as we sweep t_{AggON} in linear (y-axis) - log (x-axis) scale, using the same experimental methodology as described in §5.1. A data point below AC_{min} = 1 (highlighted with dashed red lines) means that for a given t_{AggON}, it requires less aggressor row activations to induce at least one bitflip at a higher temperature.

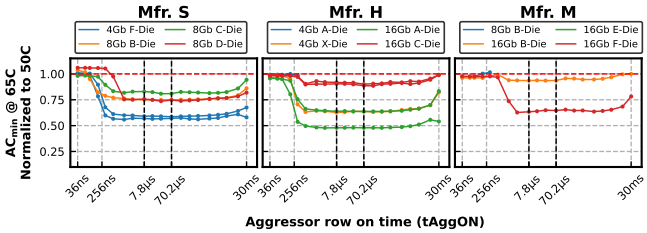

Figure 46: AC_{min} at 65°C normalized to 50°C; single-sided RowPress.

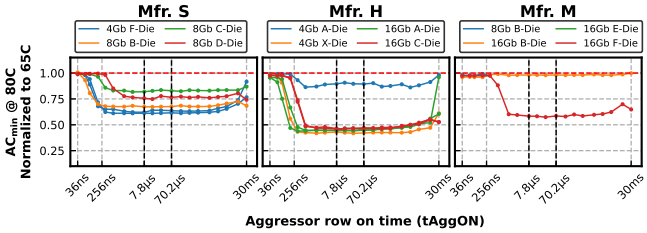

Figure 47: AC_{min} at 80°C normalized to 65°C; single-sided RowPress.

We observe that for all die revisions vulnerable to RowPress, AC_{min} *consistently* reduces for the same t_{AggON} value as temperature increases from 50°C to 65°C, and then from 65°C to 80°C (i.e., Obsv. 9 still holds when we consider three different temperatures, 50°C, 65°C, and 80°C).

Fig. 48 shows the difference between single- and double-sided AC_{min} (i.e., $AC_{min}(single)$ - $AC_{min}(double)$) at 50°C (first row), 65°C (second row) and 80°C (third row), using the same experimental methodology as described in §5.2. A data point below 0 means that the single-sided RowPress pattern needs fewer aggressor row activations in total to induce a bitflip compared to double-sided. We observe that, at 65°C, the single-sided RowPress pattern still needs fewer aggressor row activations in total to induce a bitflip compared to double-sided (i.e., Obsv. 13 still holds when we consider three different temperatures, 50°C, 65°C, and 80°C).

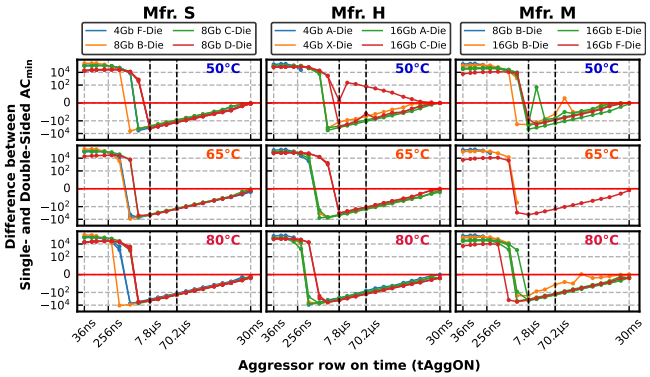

Figure 48: Single-sided AC_{min} minus double-sided AC_{min} at 50°C (first row), 65°C (second row) and 80°C (third row).

G Inducing Even More Bitflips on the Real System

Algorithm 2 shows a variant of our real system RowPress test program (i.e., Algorithm 1 in §6) that changes the *program order* of the accesses to the cache blocks and the `clflushopt` instructions to flush them from the cache. In the original Algorithm 1, we flush the cache blocks *only after* accessing *all* cache blocks from *both* aggressor rows (in program order, lines 11-16 in Algorithm 1). In Algorithm 2, we *immediately* flush each cache block *after* each cache block access (in program order, lines 13-18 in Algorithm 2).

```
1  // find two neighboring aggressor rows based on physical address mapping
2  AGGRESSOR1, AGGRESSOR2 = find_aggressor_rows(VICTIM);
3  // initialize the aggressor and the victim rows
4  initialize(VICTIM, 0x55555555);
5  initialize(AGGRESSOR1, AGGRESSOR2, 0xAAAAAAAA);
6  // Synchronize with refresh
7  for (iter = 0 ; iter < NUM_ITER ; iter++):
8    for (i = 0 ; i < NUM_AGGR_ACTS ; i++):
9      // access multiple cache blocks in each aggressor row
10     // to keep the aggressor row open longer
11     // ************************************
12     // *** MODIFIED PART START ***
13     for (j = 0 ; j < NUM_READS ; j++):
14         *AGGRESSOR1[j];
15         clflushopt (AGGRESSOR1[j]);
16     for (j = 0 ; j < NUM_READS ; j++):
17         *AGGRESSOR2[j];
18         clflushopt (AGGRESSOR2[j]);
19     // *** MODIFIED PART END ***
20     // ************************************
21     mfence ();
22   activate_dummy_rows();
23 record_bitflips[VICTIM] = check_bitflips(VICTIM);
```

Algorithm 2: A variant of our RowPress test program that can induce many more bitflips than the test program in Algorithm 1.

We run this variant of the test program (i.e., Algorithm 2) using the same methodology on the same system as described in §6. We plot the total number of bitflips (left) and the number of rows with bitflips (right) from both Algorithm 2 (purple bars) and Algorithm 1 (blue bars)[34] for different numbers of cache blocks read per aggressor row activation (`NUM_READS`; x-axis) when we activate each aggressor row four (top plots), three (middle plots), and two (bottom plots) times per iteration in Fig. 49. We do not plot `NUM_AGGR_ACTS=1` because we do not observe any bitflips for all `NUM_READS` we test. The leftmost bar in each graph shows the number of *conventional RowHammer-induced* bitflips, where we read *only a single* cache block per aggressor row activation, such that the aggressor row is kept open for a short time. Remaining bars in each graph show results for RowPress-induced bitflips (with an increasing number of cache block reads from left to right, such that the aggressor row is kept open for an increasing amount of time). We make the following major observation from Fig. 49.

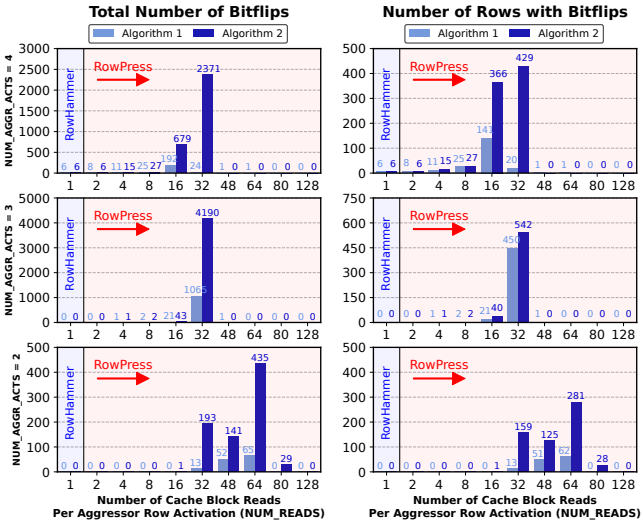

Figure 49: Number of RowHammer vs. RowPress bitflips (left) and number of rows with bitflips (right) we observe after running our proof of concept test programs with Algorithm 1 (blue bars) and Algorithm 2 (purple bars) with four (top), three (middle), and two (bottom) activations per aggressor row per iteration.

Obsv. 23. With Algorithm 2, the proof of concept test program induces significantly more bitflips in many more DRAM rows in a real system.

With Algorithm 2, our test program induces significantly more bitflips in significantly more DRAM rows. For example, when `NUM_AGGR_ACTS=4` and `NUM_READS=32`, with Algorithm 2, the test program induces 2371 bitflips in 429 DRAM rows, compared to only 24 bitflips in 20 rows with Algorithm 1, amounting to an increase of 98.79× and 21.45×, respectively. When `NUM_AGGR_ACTS=3` and `NUM_READS=32`, Algorithm 2 induces 4190 bitflips in 542 DRAM rows, compared to 1065 bitflips in 450 DRAM rows with Algorithm 1, amounting to an increase of 3.93× and 1.20×, respectively. We hypothesize that the memory access pattern of Algorithm 2 causes the aggressor rows to be open longer than that of Algorithm 1, leading to many more bitflips in many more DRAM rows. Our results call for more investigation of how DRAM row open time is (and should be) handeled in modern memory controllers. To aid such research, we open source all our proof of concept programs (including Algorithm 2) in our Github repository at [125].

[34] The number of bitflips and the number of DRAM rows with bitflips from Algorithm 1 depicted in Fig. 2 differ slightly from what we show in Fig. 23 in §6 because these figures depict results from different runs of our test program with Algorithm 1 *on the real system*. Low-level events that are transparent to the program (e.g., the dynamic process scheduling decisions by the operating system and different synchronization points with the DRAM refresh commands) cause slight variations in the experimental results across different runs of the same program.