


# 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.

## CCS Concepts

• **Hardware** → **Dynamic memory**; **Hardware reliability**; • **Security and privacy** → **Security in hardware**.


*ISCA '23, June 17–21, 2023, Orlando, FL, USA*

ACM ISBN 979-8-4007-0095-8/23/06…$15.00
https://doi.org/10.1145/3579371.3589063

## Keywords

DRAM, Read Disturbance, RowPress, RowHammer, Reliability, Security, Safety, Testing

**ACM Reference Format:**
Haocong Luo, Ataberk Olgun, A. Giray Yağlıkçı, Yahya Can Tuğrul, Steve Rhyner, Meryem Banu Cavlak, Joël Lindegger, Mohammad Sadrosadati, Onur Mutlu. 2023. RowPress: Amplifying Read Disturbance in Modern DRAM Chips. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23), June 17–21, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3579371.3589063

## 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) [1], 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) [2, 3].

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 [2, 4–55]. Prior works [2, 3] 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 [3]. 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

[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 [56, 57], while Samsung calls this "Passing Gate Effect" in a very recent work placed on arXiv while our paper has been under review [58]. 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.

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.

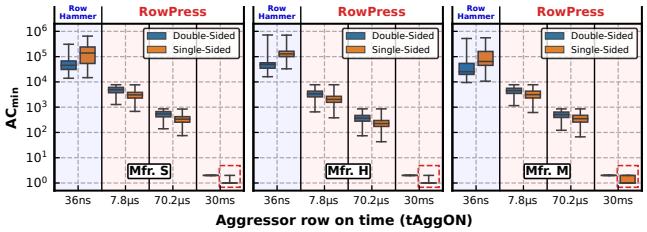


**Figure 1: $AC_{min}$ distributions of conventional RowHammer (RH) and three representative cases of RowPress (RP) at $80°C$ across 164 DDR4 chips from manufacturers S, H, and M.**

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 [59], as done in conventional RowHammer attacks [2, 4–55]. 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 $t_{AggON}$ [59], 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 [2, 3, 27, 35, 62–67]. 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 [2] and Graphene [68]), 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 [69] 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.

[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 [59]. We use a 36 ns minimum $t_{AggON}$ 1) to cover the whole range of $t_{RAS}$ 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) [60].

[4]*Refresh interval* is the time interval between two consecutive refresh commands that a DRAM row can be kept open [59, 61].

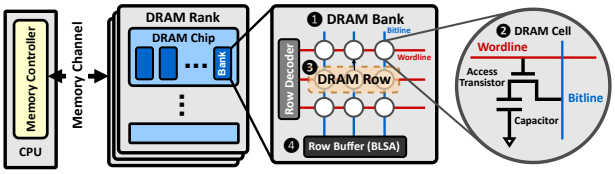


**Figure 2: Hierarchical organization of modern DRAM.**

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 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* [59, 61, 70, 71]. 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.

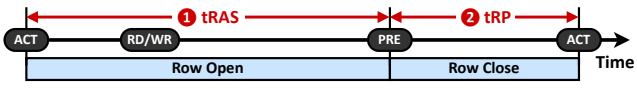


**Figure 3: Timeline of key DRAM access operations.**

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 [72–75]), 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 [59] 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 [76–79], 2) data retention failures due to the volatile and leaky nature of DRAM cells [80–84], and 3) read disturbance (e.g., RowHammer [2, 3, 48, 51, 62–67, 85–93]) 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 [2, 4–55]. 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 [94]), programmed with DRAM Bender [60, 95] (based on SoftMC [96, 97]), 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 [98]) that controls the heaters and keeps the temperature at the desired level.

**Disabling Interference Sources.** To observe RowPress' effects at the circuit level, we disable potential sources of interference following a methodology similar to prior works [3, 44, 66, 88]. 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) [38, 44] so as to observe the DRAM chip's fundamental read disturbance behavior at the circuit level. Second, we bound

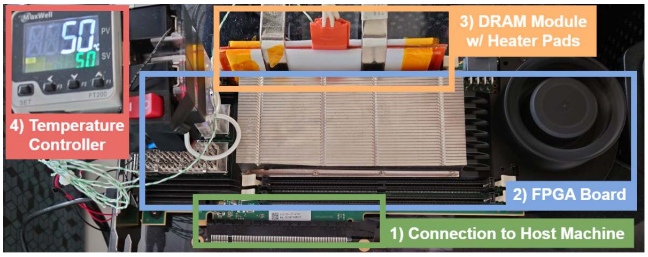


**Figure 4: Our DDR4 DRAM testing infrastructure.**

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 | 2053 |
| | 1 | 8 | 8Gb | C | x8 | N/A |
| | 3 | 8 | 8Gb | D | x8 | 2110 |
| | 2 | 8 | 4Gb | F | x8 | N/A |
| Mfr. H (SK Hynix) | 1 | 8 | 4Gb | A | x8 | 1946 |
| | 1 | 8 | 4Gb | X | x8 | N/A |
| | 2 | 8 | 16Gb | A | x8 | 2051 |
| | 2 | 8 | 16Gb | C | x8 | 2136 |
| Mfr. M (Micron) | 1 | 16 | 8Gb | B | x4 | N/A |
| | 2 | 4 | 16Gb | B | x16 | 2126 |
| | 1 | 16 | 16Gb | E | x4 | 2014 |
| | 2 | 4 | 16Gb | E | x16 | 2046 |
| | 1 | 4 | 16Gb | F | x16 | 2150 |

To account for in-DRAM row address mapping [2, 9, 28, 39, 81, 84, 100–107], we reverse-engineer the physical row address layout, following the methodology of prior works [3, 44, 66, 88].

# 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.

[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 are in Appendix B of [99].

## 4.1 Experiment Methodology

**Metric.** To characterize how RowPress amplifies DRAM's vulnerability to read disturbance, we examine how the minimum number of total aggressor row activations to cause at least one bitflip ($AC_{min}$) 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.

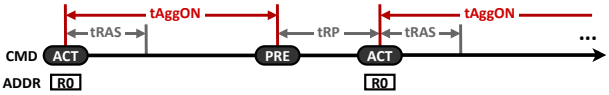


**Figure 5: Single-sided RowPress access pattern used to characterize how $AC_{min}$ changes as $t_{AggON}$ increases.**

**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 [66, 88]. Instead of a fixed $AC_{min}$ accuracy (e.g., 100 in [66] and 512 in [88]), 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 [59]). 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 [108] 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 an extended version of our paper [99].

**Temperature.** We maintain the DRAM chip temperature at a normal operating condition of 50°C.

## 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

[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.

$t_{AggON}$ 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 [59].[7] We mark $AC_{min}$ = 1 on the y-axis. We make three major observations from Fig. 6.

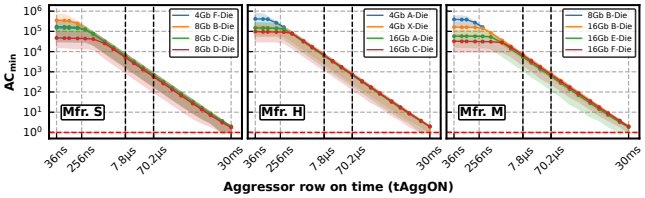


**Figure 6: $AC_{min}$ as $t_{AggON}$ increases; single-sided RowPress at 50°C.**

**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. Our extended paper [99] provides a more detailed analysis.

Fig. 7 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.

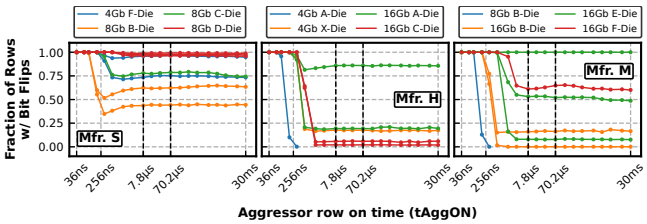


**Figure 7: The fraction of rows that experience at least one bitflip; single-sided RowPress at 50°C.**

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

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. 8 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.

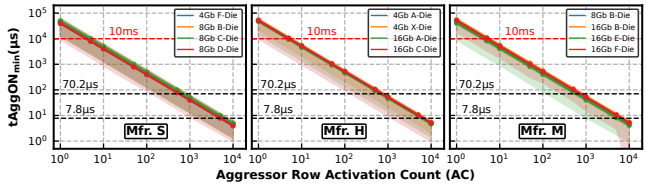


**Figure 8: $t_{AggONmin}$ as aggressor row activation count (AC) increases; single-sided RowPress at 50°C.**

**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

[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 [59] allows *up to* eight refresh commands to be postponed (Section 4.26 in [59]), 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.

[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.

$t_{AggONmin}$ 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. 8.[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 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. 9 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. 7, each curve represents a different DRAM module, color-coded based on its die revision.

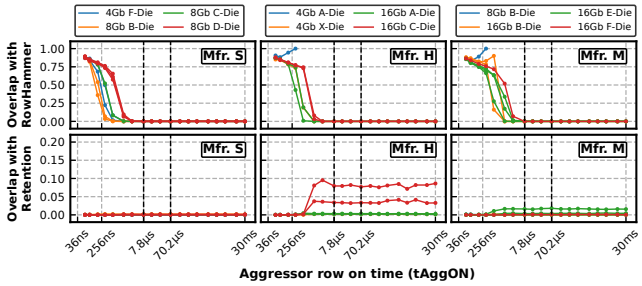


**Figure 9: Overlap ratio of RowPress-vulnerable cells with RowHammer (first row) and retention failures (second row).**

**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.

**Bitflip Direction.** Fig. 10 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. 7, each curve represents a different DRAM module, color-coded based on its die revision.

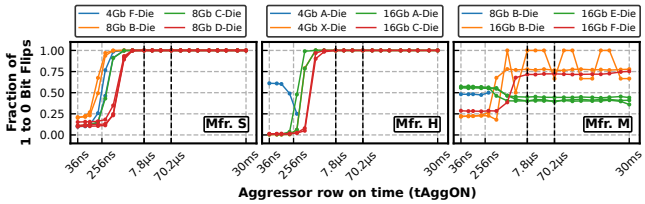


**Figure 10: Fraction of 1 to 0 bitflips.**

**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 $t_{AggON}$ 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 $t_{AggON} \geq 7.8$ µs. 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 ($t_{AggOFF}$).

## 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. 11 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.

**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 µs, the average $AC_{min}$ at 80°$C$ is only 0.55×, 0.32×, and 0.59×

[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. 8).

[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 [82].

[13] Prior works [2, 3] already show that RowHammer bitflips have little overlap with retention failure bitflips.

[14] In a concurrent work [58], 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) [81].

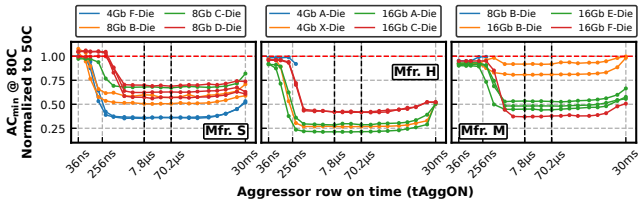


**Figure 11: $AC_{min}$ at 80°C normalized to 50°C; single-sided RowPress.**

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 µs at 80°C. When $t_{AggON}$ increases from 36 ns to 70.2 µs, $AC_{min}$ 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 µs and 190× (up to 537×) when $t_{AggON}$ increases from 36 ns to 70.2 µs. 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. 11) at 80°C (only 13.1% at 50°C).

Fig. 12 shows the fraction of rows that have at least one RowPress bitflip as we sweep $t_{AggON}$ at 80°C.

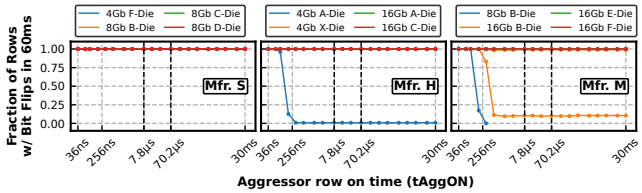


**Figure 12: Fraction of rows that experience at least one bitflip at 80°C; single-sided RowPress.**

**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.

To study the effect of increasing temperature on $t_{AggONmin}$ (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. 13.[16] The error band shows the standard deviation of $t_{AggONmin}$.

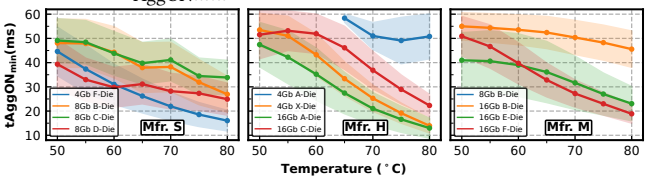


**Figure 13: $t_{AggONmin}$ when $AC = 1$ as we sweep temperature from 50°C to 80°C with 5°C steps; single-sided RowPress.**

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

[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.

We observe that $t_{AggONmin}$ significantly decreases as we gradually increase temperature from 50°C to 80°C. For Mfr. S, H, and M, the average (minimum) $t_{AggONmin}$ 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) $t_{AggONmin}$ 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. 8), we are able to induce RowPress bitflips when $AC$ = 1 at temperatures ≥ 65°C.

**Takeaway 3.** RowPress gets significantly worse as temperature increases. This behavior is very different from how RowHammer bitflips change with temperature [2, 88].

## 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. 14. 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 14: 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. 15. The error band shows the minimum and maximum $AC_{min}$ values.

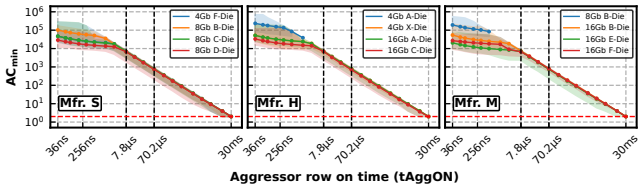


**Figure 15: $AC_{min}$ of double-sided RowPress; 50°C.**

**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. 15 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. 16.

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.

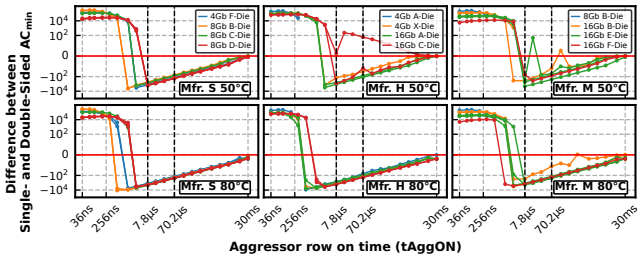


**Figure 16: Single-sided $AC_{min}$ minus double-sided $AC_{min}$ at $50°C$ (first row) and $80°C$ (second row).**

**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.

Note that this behavior is very different from RowHammer, where double-sided RowHammer is strictly more effective at inducing bitflips than single-sided [2]. 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 tAggON vs tAggOFF

Prior works on *device-level* mechanisms of RowHammer [62, 109] 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 [62, 109] is limited because they only test 1) a very small range of $t_{AggON}$ and $t_{AggOFF}$ values (up to 50 ns in [62] and 72.5 ns in [109]), and 2) a single-sided access pattern.

**Access Pattern.** To compare RowPress to the read-disturb mechanisms discussed in prior works [62, 109], we design the RowPress-ONOFF access pattern shown in Fig. 17, based on the pattern proposed in [109]. 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 $\min(t_{AggON}) + \min(t_{AggOFF}) = t_{RAS} + t_{RP} = t_{RC}$.

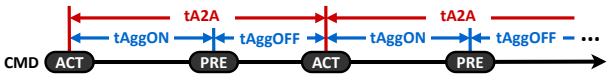


**Figure 17: The RowPress-ONOFF pattern.**

**Methodology.** We fix the activation frequency of a row by fixing $t_{A2A}$. We increase $t_{A2A}$ beyond $t_{RC}$ by $\Delta t_{A2A}$ = {240, 600, 1200, 2400, 6000} ns. For each $t_{A2A}$ value, we sweep the fraction of $\Delta t_{A2A}$ that contributes to $t_{AggON}$ from 0% to 100% (with a step size of 25%). For example, 25% means $t_{AggON}$ = 25% $\Delta t_{A2A}$ + $t_{RAS}$, and $t_{AggOFF}$ = 75% $\Delta t_{A2A}$ + $t_{RP}$. 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. 18 shows the *BER* (y-axis) for both single-sided (top row) and double-sided (bottom row) RowPress-ONOFF pattern for a representative[17] die revision (8Gb D-Die from Mfr. S). We sweep $\Delta t_{A2A}$ (different lines in each plot) and the percentage of $\Delta t_{A2A}$ 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.

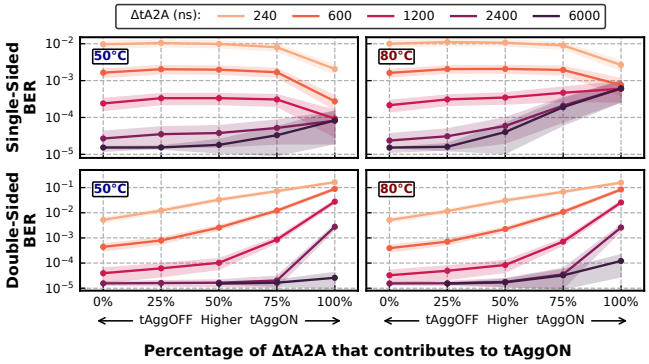


**Figure 18: BER of the representative Mfr. S 8Gb D-Die; single- (top row) and double-sided (bottom row) RowPress-ONOFF pattern at $50°C$ (left column) and $80°C$ (right column).**

**Obsv. 14.** For the single-sided access pattern, increasing $t_{AggON}$ (i.e., decreasing $t_{AggOFF}$) with small (large) $\Delta t_{A2A}$ values mitigates (exacerbates) read disturbance.

[17] We observe a similar trend for almost all other die revisions (see [99] for more details). We show only one representative die revision to illustrate the results more clearly.

For $\Delta t_{A2A}$ values $\leq$ 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 [62, 64][18] that test a small range of $t_{AggON}$/$t_{AggOFF}$ values (up to 50 ns in [62] and 72.5 ns in [109], respectively). As $\Delta t_{A2A}$ 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 [62, 64].

**Obsv. 15.** For the single-sided access pattern, increasing temperature exacerbates read disturbance for large $\Delta t_{A2A}$ 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 almost unchanged) for large (small) $\Delta t_{A2A}$ and $t_{AggON}$ values. For example, the average BER increases by 7.5× (only 1.04×) from 50°$C$ to 80°$C$ when $\Delta t_{A2A}$ = 6000 ns (240 ns) and 100% of $\Delta t_{A2A}$ contributes to $t_{AggON}$. At the inflection point of $\Delta t_{A2A}$ = 1200 ns, when 50% to 100% of $\Delta t_{A2A}$ contributes to $t_{AggON}$, BER *decreases* at 80°$C$, in contrast to *increasing* at 50°$C$. This observation is *not* fully explained by prior device-level works [62, 64] because they do *not* change $\Delta t_{A2A}$, $t_{AggON}$, and $t_{AggOFF}$ when investigating the effect of temperature on read disturbance.

**Obsv. 16.** For the double-sided pattern, read disturbance consistently worsens as $t_{AggON}$ increases and $t_{AggOFF}$ decreases.

For *all* $\Delta t_{A2A}$ 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 $\Delta t_{A2A}$ 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 [62, 64]. 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 [110]) with an Intel i5-10400 (Comet Lake) processor [111] and a 16GB dual rank DDR4 DRAM module [112] from Mfr. S (Samsung). This DRAM module has target row refresh (TRR) [38, 44], 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 [18], similar to prior works (e.g., [38, 43, 45]). We allocate a 1GB page using hugepage support [113] 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.[19]

## 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 [38, 44] and these mechanisms can be bypassed by certain access patterns that access many other dummy aggressor rows (called dummy rows [38, 44]) 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 [59]. 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 [44], 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[20] to prevent the in-DRAM TRR mechanism from detecting the aggressor row activations [38, 43–45]. 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

[18] Injected charge (from diffused channel electrons [64] and charge traps [62]) needs sufficient amount of time to be recombined at the victim cell and fully exhausted *after* the row is closed (i.e., longer $t_{AggOFF}$)

[19] 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 [30, 37, 41, 47] on system-level RowHammer attacks experimentally demonstrate. One can extend our real-system demonstration program to avoid using hugepages.

[20] Dummy rows are placed at least 100 rows away from the victim row [44] to ensure that activating them does not cause bitflips on the victim row.

RowPress with varying $t_{AggON}$), and 2) activates each of the 16 dummy rows four times (line 17) [44].

```
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  for (iter = 0 ; iter < NUM_ITER ; iter++):
7    for (i = 0 ; i < NUM_AGGR_ACTS ; i++):
8      // access multiple cache blocks in each aggressor row
9      // to keep the aggressor row open longer
10     for (j = 0 ; j < NUM_READS ; j++): *AGGRESSOR1[j];
11     for (j = 0 ; j < NUM_READS ; j++): *AGGRESSOR2[j];
12     // flush the cache blocks of each aggressor row
13     for (j = 0 ; j < NUM_READS ; j++):
14       clflushopt (AGGRESSOR1[j]);
15       clflushopt (AGGRESSOR2[j]);
16     mfence ();
17   activate_dummy_rows();
18 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 DRAM chip characterization studies (lines 4–5). We use this data pattern as it is reported [3] 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 [2]. 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}[21] 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 [43, 45], to increase the chance of bypassing TRR.

**Results.** Fig. 19 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 [2] and RowHammer attacks [4–55]), 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).

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

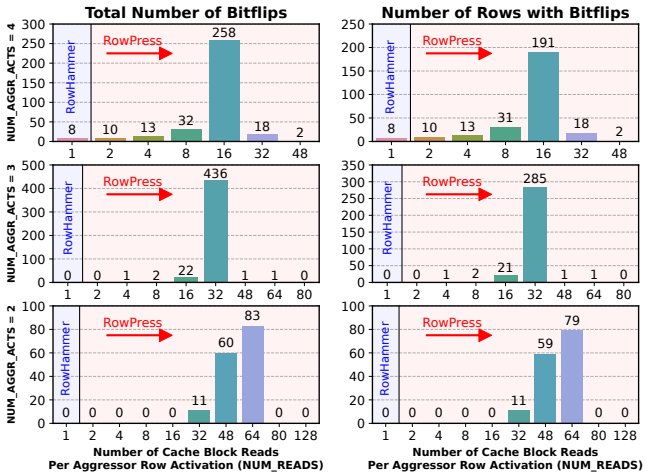


**Figure 19: Number of RowHammer vs. RowPress bitflips (left) and number of rows with bitflips (right) we observe after running our test program with four (top), three (middle), and two (bottom) activations per aggressor row per iteration.**

**Obsv. 17.** Our test program leveraging RowPress induces bitflips when RowHammer cannot.

**Obsv. 18.** 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 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. 19.** 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., [38, 45]) can leverage RowPress to be more effective.

## 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[22], 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. 20 shows the frequency histogram of latency values (observed using Intel time stamp counter [114]) 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.

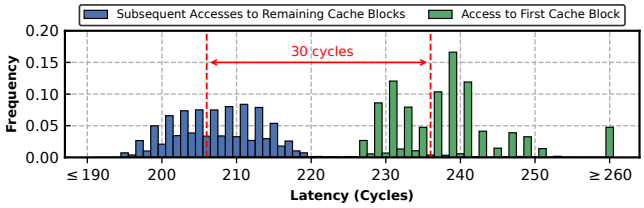


**Figure 20: Histogram of the latency of the first and remaining cache block accesses to the same DRAM row.**

[22]We disable all hardware prefetchers of the processor by modifying model-specific register values [114] 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.

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 [72–74, 115–121]) can facilitate future attacks leveraging RowPress.

# 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 [68] and PARA [2] as examples.

## 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. 21 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).

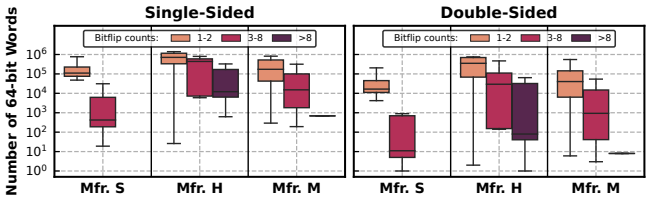


**Figure 21: Number of 64-bit words with different bitflip counts for single-sided (left) and double-sided (right) RowPress.**

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 [122] and Chipkill [123–125][23]) cannot correct or detect *all* RowPress bitflips we observe, which can lead to silent data corruption [126–128]. Even a (7, 4) Hamming code (correcting one bitflip in a 4-bit data word) [122] 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

[23]Chipkill [123–125] 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.

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. 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) [129, 130] since such techniques could render up to 35.77% of storage capacity useless.

## 7.2 Decoupling the Row Buffer from the Row

Prior works [131, 132] 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 [132]. 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. 15, $AC_{min}$ 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 (our results show up to 34.1% performance degradation[24]). 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) [115–121]. 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.

[24]Compared to the open-row baseline, observed in 462.libquantum from [133]. Our extended paper [99] provides detailed results.

## 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 $X$ns, $AC_{min}$ 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 $X$ns 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 [68], a low performance overhead mechanism, and PARA [2], 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 [2, 68], as shown in Table 2. We perform a sensitivity study of their respective performance overheads over Graphene and PARA[25] with these configurations using Ramulator [136, 137] with a realistic baseline system configuration[26] and show the results in Table 2.

**Table 2: 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% |

[25]Measured by the weighted speedup [134, 135] of Graphene-RP (PARA-RP) normalized to Graphene (PARA).

[26]4 GHz out-of-order core, dual-rank DDR4 DRAM [59], FR-FCFS [73, 74] scheduling, open-row policy. 58 four-core multiprogrammed workloads from SPEC CPU2017 [133], TPC-H [138], and YCSB [139]. We find similar performance results for single-core workloads, as shown in our extended version [99].

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 [72, 75]). 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 $T'_{RH}$ [3, 68, 140], 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 provide more evaluations of our proposed mitigation mechanisms in an extended version of this paper [99]. We expect future work to introduce new mitigation mechanisms, as it has been happening analogously for RowHammer.

## 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 [88] and prior device-level studies [62, 64] 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 [88], 50 ns in [62], and 72.5 ns in in [64], 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 [56, 57] 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" [56], and "stress applied to adjacent word lines by a word line being on for an extended duration" [57]. 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). [56] 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). [57] 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 [56, 57] evaluates or analyzes its proposed mitigation mechanisms at the system-level.

**Other DRAM Read Disturbance Mitigation Techniques.** Many works (e.g., [2, 24, 31, 59, 68, 86, 140–166]) 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 [2, 68] (§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 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 [69].

## 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.

[27] Two recent works [167, 168] discuss at a high level how to handle increased vulnerability due to small increases in $t_{AggON}$ (as reported by [88]) by modifying their proposed RowHammer mitigation mechanisms. However, these works do not evaluate their modified mechanisms.

## References


[1] Robert H. Dennard. Field-Effect Transistor Memory. U.S. Patent 3387286, 1968.
[2] 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.
[3] Jeremie S. Kim, Minesh Patel, Abdullah Giray Yağlıkçı, Hasan Hassan, Roknoddin Azizi, Lois Orosa, and Onur Mutlu. Revisiting RowHammer: An Experimental Analysis of Modern Devices and Mitigation Techniques. In *ISCA*, 2020.
[4] Daniel Gruss, Clementine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. *arXiv:1507.06955 [cs.CR]*, 2015.
[5] Apostolos P Fournaris, Lidia Pocero Fraile, and Odysseas Koufopavlou. Exploiting Hardware Vulnerabilities to Attack Embedded System Devices: A Survey of Potent Microarchitectural Attacks. *Electronics*, 2017.
[6] Damian Poddebniak, Juraj Somorovsky, Sebastian Schinzel, Manfred Lochter, and Paul Rösler. Attacking Deterministic Signature Schemes using Fault Attacks. In *EuroS&P*, 2018.
[7] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Throwhammer: Rowhammer Attacks Over the Network and Defenses. In *USENIX ATC*, 2018.
[8] Sebastien Carre, Matthieu Desjardins, Adrien Facon, and Sylvain Guilley. OpenSSL Bellcore's Protection Helps Fault Attack. In *DSD*, 2018.
[9] Alessandro Barenghi, Luca Breveglieri, Niccolò Izzo, and Gerardo Pelosi. Software-Only Reverse Engineering of Physical DRAM Mappings for Rowhammer Attacks. In *IVSW*, 2018.
[10] Zhenkai Zhang, Zihao Zhan, Daniel Balasubramanian, Xenofon Koutsoukos, and Gabor Karsai. Triggering Rowhammer Hardware Faults on ARM: A Revisit. In *ASHES*, 2018.
[11] Sarani Bhattacharya and Debdeep Mukhopadhyay. Advanced Fault Attacks in Software: Exploiting the Rowhammer Bug. *Fault Tolerant Architectures for Cryptography and Hardware Security*, 2018.
[12] Mark Seaborn and Thomas 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.
[13] SAFARI Research Group. RowHammer — GitHub Repository. https://github.com/CMU-SAFARI/rowhammer.
[14] Mark Seaborn and Thomas Dullien. Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges. *Black Hat*, 2015.
[15] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *CCS*, 2016.
[16] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in Javascript. arXiv:1507.06955 [cs.CR], 2016.
[17] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip Feng Shui: Hammering a Needle in the Software Stack. In *USENIX Security*, 2016.
[18] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security*, 2016.
[19] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation. In *USENIX Security*, 2016.
[20] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *S&P*, 2016.
[21] Sarani Bhattacharya and Debdeep Mukhopadhyay. Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis. In *CHES*, 2016.
[22] Wayne Burleson, Onur Mutlu, and Mohit Tiwari. Invited: Who is the Major Threat to Tomorrow's Security? You, the Hardware Designer. In *DAC*, 2016.
[23] Rui Qiao and Mark Seaborn. A New Approach for RowHammer Attacks. In *HOST*, 2016.
[24] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. Can't Touch This: Software-Only Mitigation Against Rowhammer Attacks Targeting Kernel Memory. In *USENIX Security*, 2017.
[25] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. SGX-Bomb: Locking Down the Processor via Rowhammer Attack. In *SOSP*, 2017.
[26] Misiker Tadesse Aga, Zelalem Birhanu Aweke, and Todd Austin. When Good Protections Go Bad: Exploiting Anti-DoS Measures to Accelerate Rowhammer Attacks. In *HOST*, 2017.
[27] Onur Mutlu. The RowHammer Problem and Other Issues We May Face as Memory Becomes Denser. In *DATE*, 2017.
[28] Andrei Tatar, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Defeating Software Mitigations Against Rowhammer: A Surgical Precision Hammer. In *RAID*, 2018.
[29] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another Flip in the Wall of Rowhammer Defenses. In *S&P*, 2018.
[30] Moritz Lipp, Misiker Tadesse Aga, Michael Schwarz, Daniel Gruss, Clémentine Maurice, Lukas Raab, and Lukas Lamster. Nethammer: Inducing Rowhammer Faults Through Network Requests. arXiv:1805.04956 [cs.CR], 2018.
[31] Victor van der Veen, Martina Lindorfer, Yanick Fratantonio, Harikrishnan Padmanabha Pillai, Giovanni Vigna, Christopher Kruegel, Herbert Bos, and Kaveh Razavi. GuardION: Practical Mitigation of DMA-Based Rowhammer Attacks on ARM. In *DIMVA*, 2018.
[32] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In *S&P*, 2018.
[33] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks. In *S&P*, 2019.
[34] Sangwoo Ji, Youngjoo Ko, Saeyoung Oh, and Jong Kim. Pinpoint Rowhammer: Suppressing Unwanted Bit Flips on Rowhammer Attacks. In *ASIACCS*, 2019.
[35] Onur Mutlu and Jeremie S Kim. RowHammer: A Retrospective. *TCAD*, 2019.
[36] Sanghyun Hong, Pietro Frigo, Yiğitcan Kaya, Cristiano Giuffrida, and Tudor Dumitraş. Terminal Brain Damage: Exposing the Graceless Degradation in Deep Neural Networks Under Hardware Fault Attacks. In *USENIX Security*, 2019.
[37] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. RAMBleed: Reading Bits in Memory Without Accessing Them. In *S&P*, 2020.
[38] Pietro Frigo, Emanuele Vannacci, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TRRespass: Exploiting the Many Sides of Target Row Refresh. In *S&P*, 2020.
[39] Lucian Cojocar, Jeremie Kim, Minesh Patel, Lillian Tsai, Stefan Saroiu, Alec Wolman, and Onur Mutlu. Are We Susceptible to Rowhammer? An End-to-End Methodology for Cloud Providers. In *S&P*, 2020.
[40] Zane Weissman, Thore Tiemann, Daniel Moghimi, Evan Custodio, Thomas Eisenbarth, and Berk Sunar. JackHammer: Efficient Rowhammer on Heterogeneous FPGA–CPU Platforms. arXiv:1912.11523 [cs.CR], 2020.
[41] Zhi Zhang, Yueqiang Cheng, Dongxi Liu, Surya Nepal, Zhi Wang, and Yuval Yarom. PThammer: Cross-User-Kernel-Boundary Rowhammer through Implicit Accesses. In *MICRO*, 2020.
[42] Fan Yao, Adnan Siraj Rakin, and Deliang Fan. DeepHammer: Depleting the Intelligence of Deep Neural Networks Through Targeted Chain of Bit Flips. In *USENIX Security*, 2020.
[43] Finn de Ridder, Pietro Frigo, Emanuele Vannacci, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. SMASH: Synchronized Many-Sided Rowhammer Attacks from JavaScript. In *USENIX Security*, 2021.
[44] Hasan Hassan, Yahya Can Tugrul, Jeremie S. Kim, Victor van der Veen, Kaveh Razavi, and Onur Mutlu. Uncovering in-DRAM RowHammer Protection Mechanisms: A New Methodology, Custom RowHammer Patterns, and Implications. In *MICRO*, 2021.
[45] Patrick Jattke, Victor van der Veen, Pietro Frigo, Stijn Gunter, and Kaveh Razavi. Blacksmith: Scalable Rowhammering in the Frequency Domain. In *SP*, 2022.
[46] M Caner Tol, Saad Islam, Berk Sunar, and Ziming Zhang. Toward Realistic Backdoor Injection Attacks on DNNs using RowHammer. arXiv:2110.07683v2 [cs.LG], 2022.
[47] Andreas Kogler, Jonas Juffinger, Salman Qazi, Yoongu Kim, Moritz Lipp, Nicolas Boichat, Eric Shiu, Mattias Nissler, and Daniel Gruss. Half-Double: Hammering From the Next Row Over. In *USENIX Security*, 2022.
[48] Lois Orosa, Ulrich Rührmair, A Giray Yaglikci, Haocong Luo, Ataberk Olgun, Patrick Jattke, Minesh Patel, Jeremie Kim, Kaveh Razavi, and Onur Mutlu. SpyHammer: Using RowHammer to Remotely Spy on Temperature. arXiv:2210.04084, 2022.
[49] Zhi Zhang, Wei He, Yueqiang Cheng, Wenhao Wang, Yansong Gao, Dongxi Liu, Kang Li, Surya Nepal, Anmin Fu, and Yi Zou. Implicit Hammer: Cross-Privilege-Boundary Rowhammer through Implicit Accesses. *IEEE Transactions on Dependable and Secure Computing*, 2022.
[50] Liang Liu, Yanan Guo, Yueqiang Cheng, Youtao Zhang, and Jun Yang. Generating Robust DNN with Resistance to Bit-Flip based Adversarial Weight Attack. *IEEE Transactions on Computers*, 2022.
[51] Yaakov Cohen, Kevin Sam Tharayil, Arie Haenel, Daniel Genkin, Angelos D Keromytis, Yossi Oren, and Yuval Yarom. HammerScope: Observing DRAM Power Consumption Using Rowhammer. In *CCS*, 2022.
[52] Mengxin Zheng, Qian Lou, and Lei Jiang. TrojViT: Trojan Insertion in Vision Transformers. arXiv:2208.13049, 2022.
[53] Michael Fahr Jr, Hunter Kippen, Andrew Kwong, Thinh Dang, Jacob Lichtinger, Dana Dachman-Soled, Daniel Genkin, Alexander Nelson, Ray Perlner, Arkady Yerukhimovich, et al. When Frodo Flips: End-to-End Key Recovery on FrodoKEM via Rowhammer. *CCS*, 2022.
[54] Youssef Tobah, Andrew Kwong, Ingab Kang, Daniel Genkin, and Kang G. Shin. SpecHammer: Combining Spectre and Rowhammer for New Speculative Attacks. In *SP*, 2022.

[55] Adnan Siraj Rakin, Md Hafizul Islam Chowdhuryy, Fan Yao, and Deliang Fan. DeepSteal: Advanced Model Extractions Leveraging Efficient Weight Stealing in Memories. In *SP*, 2022.

[56] Yutaka Ito and Yuan He. Apparatus and methods for refreshing memory. U.S. Patent 11062754B2, 2019.

[57] Gregg D. Wolff. Word line cache mode. U.S. Patent 10366733B1, 2019.

[58] Seungki Hong, Dongha Kim, Jaehyung Lee, Reum Oh, Changsik Yoo, Sangjoon Hwang, and Jooyoung Lee. Dsac: Low-cost rowhammer mitigation using in-dram stochastic and approximate counting algorithm. arXiv:2302.03591, 2023.

[59] JEDEC. *JESD79-4C: DDR4 SDRAM Standard*, 2020.

[60] Ataberk Olgun, Hasan Hassan, A. Giray Yaglikci, Yahya Can Tugrul, Lois Orosa, Haocong Luo, Minesh Patel, Oguz Ergin, and Onur Mutlu. DRAM Bender: An Extensible and Versatile FPGA-based Infrastructure to Easily Test State-of-the-art DRAM Chips. arXiv:2211.05838, 2022.

[61] JEDEC. *JESD79-3: DDR3 SDRAM Standard*, 2012.

[62] Thomas Yang and Xi-Wei Lin. Trap-Assisted DRAM Row Hammer Effect. *EDL*, 2019.

[63] Andrew J. Walker, Sungkwon Lee, and Dafna Beery. On DRAM RowHammer and the Physics on Insecurity. *IEEE TED*, 2021.

[64] Kyungbae Park, Chulseung Lim, Donghyuk Yun, and Sanghyeon Baeg. Experiments and Root Cause Analysis for Active-Precharge Hammering Fault in DDR3 SDRAM under 3xnm Technology. *Microelectronics Reliability*, 2016.

[65] Kyungbae Park, Donghyuk Yun, and Sanghyeon Baeg. Statistical Distributions of Row-Hammering Induced Failures in DDR3 Components. *Microelectronics Reliability*, 2016.

[66] A Giray Yağlıkçı, Haocong Luo, Geraldo Francisco Oliveira, Ataberk Olgun, Minesh Patel, Jisung Park, Hasan Hassan, Jeremie S. Kim, Lois Orosa, and Onur Mutlu. Understanding RowHammer Under Reduced Wordline Voltage: An Experimental Study Using Real DRAM Devices. In *DSN*, 2022.

[67] Onur Mutlu, Ataberk Olgun, and A. Giray Yağlıkçı. Fundamentally Understanding and Solving RowHammer. In *ASP-DAC*, 2023.

[68] Yeonhong Park, Woosuk Kwon, Eojin Lee, Tae Jun Ham, Jung Ho Ahn, and Jae W Lee. Graphene: Strong yet Lightweight Row Hammer Protection. In *MICRO*, 2020.

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

[70] JEDEC. *JESD209-4B: Low Power Double Data Rate 4 (LPDDR4) Standard*, 2017.

[71] JEDEC. *JESD79-5: DDR5 SDRAM Standard*, 2020.

[72] Onur Mutlu and Thomas Moscibroda. Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems. In *ISCA*, 2008.

[73] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory Access Scheduling. In *ISCA*, 2000.

[74] William K Zuravleff and Timothy 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.

[75] Onur Mutlu and Thomas Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *MICRO*, 2007.

[76] R.C. Baumann. Radiation-Induced Soft Errors in Advanced Semiconductor Technologies. *IEEE TDMR*, 2005.

[77] T.C. May and M.H. Woods. Alpha-particle-induced soft errors in dynamic memories. In *IEEE Transactions on Electron Devices*, 1979.

[78] L. Lantz. Soft errors induced by alpha particles. In *IEEE Transactions on Reliability*, 1996.

[79] T.J. O'Gorman. In *The effect of cosmic rays on the soft error rate of a DRAM at ground level*, 1994.

[80] Jamie Liu, Ben Jaiyen, Richard Veras, and Onur Mutlu. RAIDR: Retention-Aware Intelligent DRAM Refresh. In *ISCA*, 2012.

[81] Jamie Liu, Ben Jaiyen, Yoongu Kim, Chris Wilkerson, Onur 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.

[82] Minesh Patel, Jeremie S Kim, and Onur Mutlu. The Reach Profiler (REAPER): Enabling the Mitigation of DRAM Retention Failures via Profiling at Aggressive Conditions. In *ISCA*, 2017.

[83] Samira Khan, Donghyuk Lee, Yoongu Kim, Alaa R Alameldeen, Chris Wilkerson, and Onur Mutlu. The Efficacy of Error Mitigation Techniques for DRAM Retention Failures: A Comparative Experimental Study. In *SIGMETRICS*, 2014.

[84] Samira Khan, Donghyuk Lee, and Onur Mutlu. PARBOR: An Efficient System-Level Technique to Detect Data-Dependent Failures in DRAM. In *DSN*, 2016.

[85] Chulseung Lim, Kyungbae Park, and Sanghyeon Baeg. Active Precharge Hammering to Monitor Displacement Damage Using High-Energy Protons in 3x-nm SDRAM. *TNS*, 2017.

[86] Seong-Wan Ryu, Kyungkyu Min, Jungho Shin, Heimi Kwon, Donghoon Nam, Taekyung Oh, Tae-Su Jang, Minsoo Yoo, Yongtaik Kim, and Sungjoo Hong. Overcoming the Reliability Limitation in the Ultimately Scaled DRAM using Silicon Migration Technique by Hydrogen Annealing. In *IEDM*, 2017.

[87] Donghyuk Yun, Myungsang Park, Chulseung Lim, and Sanghyeon Baeg. Study of TID Effects on One Row Hammering using Gamma in DDR4 SDRAMs. In *IRPS*, 2018.

[88] Lois Orosa, A Giray Yağlıkçı, Haocong Luo, Ataberk Olgun, Jisung Park, Hasan Hassan, Minesh Patel, Jeremie S. Kim, and Onur Mutlu. A Deeper Look into RowHammer's Sensitivities: Experimental Analysis of Real DRAM Chips and Implications on Future Attacks and Defenses. In *MICRO*, 2021.

[89] Mohammad Nasim Imtiaz Khan and Swaroop Ghosh. Analysis of Row Hammer Attack on STTRAM. In *ICCD*, 2018.

[90] 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.

[91] Haitong Li, Hong-Yu Chen, Zhe Chen, Bing Chen, Rui Liu, Gang Qiu, Peng Huang, Feifei Zhang, Zizhen Jiang, Bin Gao, Lifeng Liu, Xiaoyan Liu, Shimeng Yu, H.-S. Philip Wong, and Jinfeng Kang. Write Disturb Analyses on Half-Selected Cells of Cross-Point RRAM Arrays. In *IRPS*, 2014.

[92] Kai Ni, Xueqing Li, Jeffrey A. Smith, Matthew Jerry, and Suman Datta. Write Disturb in Ferroelectric FETs and Its Implication for 1T-FeFET AND Memory Arrays. *IEEE EDL*, 2018.

[93] Paul R. Genssler, Victor M. van Santen, Jörg Henkel, and Hussam Amrouch. On the Reliability of FeFET On-Chip Memory. *TC*, 2022.

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

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

[96] Hasan Hassan, Nandita Vijaykumar, Samira Khan, Saugata Ghose, Kevin Chang, Gennady Pekhimenko, Donghyuk Lee, Oguz Ergin, and Onur Mutlu. SoftMC: A Flexible and Practical Open-Source Infrastructure for Enabling Experimental DRAM Studies. In *HPCA*, 2017.

[97] SAFARI Research Group. SoftMC — GitHub Repository. https://github.com/CMU-SAFARI/softmc.

[98] Maxwell. FT20X. https://www.maxwell-fa.com/upload/files/base/8/m/311.pdf.

[99] Haocong Luo, Ataberk Olgun, A. Giray Yağlıkçı, Yahya Can Tuğrul, Steve Rhyner, Meryem Banu Cavlak, Joël Lindegger, Mohammad Sadrosadati, and Onur Mutlu. RowPress: Amplifying Read Disturbance in Modern DRAM Chips. *arXiv*, 2023.

[100] Robert T. Smith, James D. Chlipala, John F. M. Bindels, Roy G. Nelson, Frederick H. Fischer, and Thomas F. Mantz. Laser Programmable Redundancy and Yield Improvement in a 64K DRAM. *JSSC*, 1981.

[101] Masashi Horiguchi. Redundancy Techniques for High-Density DRAMs. In *ISIS*, 1997.

[102] B. Keeth and R.J. Baker. *DRAM Circuit Design: A Tutorial*. Wiley, 2001.

[103] Kiyoo Itoh. *VLSI Memory Chip Design*. Springer, 2001.

[104] Vivek Seshadri, Thomas Mullins, Amirali Boroumand, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, and Todd C Mowry. Gather-Scatter DRAM: In-DRAM Address Translation to Improve the Spatial Locality of Non-Unit Strided Accesses. In *MICRO*, 2015.

[105] Samira Khan, Chris Wilkerson, Zhe Wang, Alaa R Alameldeen, Donghyuk Lee, and Onur Mutlu. Detecting and Mitigating Data-Dependent DRAM Failures by Exploiting Current Memory Content. In *MICRO*, 2017.

[106] Donghyuk Lee, Samira Khan, Lavanya Subramanian, Saugata Ghose, Rachata Ausavarungnirun, Gennady Pekhimenko, Vivek Seshadri, and Onur Mutlu. Design-Induced Latency Variation in Modern DRAM Chips: Characterization, Analysis, and Latency Reduction Mechanisms. In *SIGMETRICS*, 2017.

[107] Minesh Patel, Jeremie Kim, Taha Shahroodi, Hasan Hassan, and Onur Mutlu. Bit-Exact ECC Recovery (BEER): Determining DRAM On-Die ECC Functions by Exploiting DRAM Data Retention Characteristics (Best Paper). In *MICRO*, 2020.

[108] A.J. van de Goor and I. Schanstra. Address and Data Scrambling: Causes and Impact on Memory Tests. In *DELTA*, 2002.

[109] Kyungbae Park, Sanghyeon Baeg, ShiJie Wen, and Richard Wong. Active-Precharge Hammering on a Row-Induced Failure in DDR3 SDRAMs Under 3x nm Technology. In *IIRW*, 2014.

[110] Launchpad. linux 5.4.0-131.147 source package in Ubuntu. https://launchpad.net/ubuntu/+source/linux/5.4.0-131.147, 2022.

[111] 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.

[112] 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.

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

[114] 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.

[115] James M. Dodd. Adaptive page management. U.S. Patent 7076617B2, 2005.

[116] Manu Awasthi, David W. Nellans, Rajeev Balasubramonian, and Al Davis. Prediction Based DRAM Row-Buffer Management in the Many-Core Era. In *PACT*, 2011.

[117] Tomas G. Rokicki. Method and computer system for speculatively closing pages in memory. U.S. Patent 6389514B1, 2002.

[118] S.-I. Park and I.-C. Park. History-based memory mode prediction for improving memory performance. In *ISCAS*, 2003.

[119] O. Kahn and J. Wilcox. Method for dynamically adjusting a memory page closing policy. 2004.

[120] B. Sander, P. Madrid, and G. Samus. Dynamic idle counter threshold value for use in memory paging policy. 2005.

[121] Y. Xu, A. Agarwal, and B. Davis. Prediction in dynamic sdram controller policies. In *SAMOS*, 2009.

[122] Richard W Hamming. Error Detecting and Error Correcting Codes. *The Bell System Technical Journal*, 1950.

[123] Timothy J Dell. A White Paper on the Benefits of Chipkill-Correct ECC for PC Server Main Memory. *IBM Microelectronics Division*, 1997.

[124] David Locklear. Chipkill Correct Memory Architecture. *Dell Enterprise Systems Group, Technology Brief*, 2000.

[125] IBM Chipkill Memory. Advanced ECC Memory for the IBM Netfinity 7000 M10. *Enhancing IBM Nethnity Server Reliability*.

[126] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. Detection and Correction of Silent Data Corruption for Large-Scale High-Performance Computing. In *SC*, 2012.

[127] George Papadimitriou and Dimitris Gizopoulos. Demystifying the system vulnerability stack: Transient fault effects across the layers. In *ISCA*, 2021.

[128] Michael B. Sullivan, Nirmal R. Saxena, Mike O'Connor, Donghyuk Lee, Paul Racunas, Saurabh Hukerikar, Timothy Tsai, Siva Kumar Sastry Hari, and Stephen W. Keckler. Characterizing and Mitigating Soft Errors in GPU DRAM. *IEEE Micro*, 2022.

[129] Dong Tang, P. Carruthers, Z. Totari, and M.W. Shapiro. Assessment of the Effect of Memory Page Retirement on System RAS Against Hardware Faults. In *DSN*, 2006.

[130] Justin Meza, Qiang Wu, Sanjeev Kumar, and Onur Mutlu. Revisiting Memory Errors in Large-Scale Production Data Centers: Analysis and Modeling of New Trends from the Field. In *DSN*, 2015.

[131] O. Seongil, Young Hoon Son, Nam Sung Kim, and Jung Ho Ahn. Row-buffer decoupling: A case for low-latency DRAM microarchitecture. In *ISCA*, 2014.

[132] Lavanya Subramanian, Kaushik Vaidyanathan, Anant Nori, Sreenivas Subramoney, Tanay Karnik, and Hong Wang. Closed yet Open DRAM: Achieving Low Latency and High Performance in DRAM Memory Systems. In *DAC*, 2018.

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

[134] Allan Snavely and Dean M Tullsen. Symbiotic Jobscheduling for A Simultaneous Multithreaded Processor. In *ASPLOS*, 2000.

[135] Stijn Eyerman and Lieven Eeckhout. System-Level Performance Metrics for Multiprogram Workloads. *IEEE Micro*, 2008.

[136] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A Fast and Extensible DRAM Simulator. *CAL*, 2016.

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

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

[139] Brian Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*, 2010.

[140] A Giray Yağlıkçı, Minesh Patel, Jeremie S. Kim, Roknoddin Azizibarzoki, Ataberk Olgun, Lois Orosa, Hasan Hassan, Jisung Park, Konstantinos Kanellopoullos, Taha Shahroodi, Saugata Ghose, and Onur Mutlu. BlockHammer: Preventing RowHammer at Low Cost by Blacklisting Rapidly-Accessed DRAM Rows. In *HPCA*, 2021.

[141] Barbara Aichinger. DDR Memory Errors Caused by Row Hammer. In *HPEC*, 2015.

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

[143] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin. ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks. In *ASPLOS*, 2016.

[144] Dae-Hyun Kim, Prashant J Nair, and Moinuddin K Qureshi. Architectural Support for Mitigating Row Hammering in DRAM Memories. *CAL*, 2015.

[145] Mungyu Son, Hyunsun Park, Junwhan Ahn, and Sungjoo Yoo. Making DRAM Stronger Against Row Hammering. In *DAC*, 2017.

[146] Eojin Lee, Ingab Kang, Sukhan Lee, G Edward Suh, and Jung Ho Ahn. TWiCe: Preventing Row-Hammering by Exploiting Time Window Counters. In *ISCA*, 2019.

[147] Jung Min You and Joon-Sung Yang. MRLoc: Mitigating Row-Hammering Based on Memory Locality. In *DAC*, 2019.

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

[149] Radhesh Krishnan Konoth, Marco Oliverio, Andrei Tatar, Dennis Andriesse, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. ZebRAM: Comprehensive and Compatible Software Protection Against Rowhammer Attacks. In *OSDI*, 2018.

[150] Ingab Kang, Eojin Lee, and Jung Ho Ahn. CAT-TWO: Counter-Based Adaptive Tree, Time Window Optimized for DRAM Row-Hammer Prevention. *IEEE Access*, 2020.

[151] Kuljit Bains, John Halbert, Christopher Mozak, Theodore Schoenborn, and Zvika Greenfield. Row Hammer Refresh Command. U.S. Patent 9117544, 2015.

[152] Kuljit S Bains and John B Halbert. Distributed Row Hammer Tracking. U.S. Patent 9299400, 2016.

[153] Kuljit S Bains and John B Halbert. Row Hammer Monitoring Based on Stored Row Hammer Threshold Value. U.S. Patent 9384821, 2016.

[154] H. Gomez, A. Amaya, and E. Roa. DRAM Row-Hammer Attack Reduction Using Dummy Cells. In *NORCAS*, 2016.

[155] 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.

[156] Fabrice Devaux and Renaud Ayrignac. Method and Circuit for Protecting a DRAM Memory Device from the Row Hammer Effect. U.S. Patent 10885966, 2021.

[157] Chia Yang, Chen Kang Wei, Yu Jing Chang, Tieh Chiang Wu, Hsiu Pin Chen, and Chao Sung 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.

[158] Chia-Ming Yang, Chen-Kang Wei, Hsiu-Pin Chen, Jian-Shing Luo, Yu Jing Chang, Tieh-Chiang Wu, and Chao-Sung Lai. Scanning Spreading Resistance Microscopy for Doping Profile in Saddle-Fin Devices. *IEEE Transactions on Nanotechnology*, 2017.

[159] SK Gautam, SK Manhas, Arvind Kumar, Mahendra Pakala, and Ellie Yieh. Row Hammering Mitigation Using Metal Nanowire in Saddle Fin DRAM. *IEEE TED*, 2019.

[160] A. Giray Yağlıkçı, Jeremie S. Kim, Fabrice Devaux, and Onur Mutlu. Security Analysis of the Silver Bullet Technique for RowHammer Prevention. arXiv:2106.07084 [cs.CR], 2021.

[161] Moinuddin Qureshi. Rethinking ECC in the Era of Row-Hammer. *DRAMSec*, 2021.

[162] Zvika Greenfield and Tomer Levy. Throttling Support for Row-Hammer Counters. U.S. Patent 9251885, 2016.

[163] Gururaj Saileshwar, Bolin Wang, Moinuddin Qureshi, and Prashant J Nair. Randomized Row-Swap: Mitigating Row Hammer by Breaking Spatial Correlation Between Aggressor and Victim Rows. In *ASPLOS*, 2022.

[164] Moinuddin Qureshi, Aditya Rohan, Gururaj Saileshwar, and Prashant J Nair. Hydra: Enabling Low-Overhead Mitigation of Row-Hammer at Ultra-Low Thresholds via Hybrid Tracking. In *ISCA*, 2022.

[165] Minbok Wi, Jaehyun Park, Seoyoung Ko, Michael Jaemin Kim, Nam Sung Kim, Eojin Lee, and Jung Ho Ahn. SHADOW: Preventing Row Hammer in DRAM with Intra-Subarray Row Shuffling. In *HPCA*, 2023.

[166] Jeonghyun Woo, Gururaj Saileshwar, and Prashant J. Nair. Scalable and Secure Row-Swap: Efficient and Safe Row Hammer Mitigation in Memory Systems. In *HPCA*, 2023.

[167] Michele Marazzi, Patrick Jattke, Flavien Solt, and Kaveh Razavi. ProTRR: Principled yet Optimal In-DRAM Target Row Refresh. In *S&P*, 2022.

[168] Michele Marazzi, Flavien Solt, Patrick Jattke, Kubo Takashi, and Kaveh Razavi. REGA: Scalable Rowhammer Mitigation with Refresh-Generating Activations. In *S&P*, 2023.

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

# A Artifact Description Appendix

## A.1 Abstract

Our artifact [69, 169] 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 [60, 95] program used to perform the characterization, 2) the source code of our real-system demonstration (§6), and 3) the source code of the Ramulator [136, 137] 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<br>Python3 scripts/Jupyter Notebooks<br>Shell scripts |
| Compilation | C++17 compiler (tested with GCC 9) |
| Run-time environment | Ubuntu 20.04 (or similar) Linux<br>Ubuntu 18.04 (with Linux kernel 5.4.0-131-generic [110]), used for reproducing *Demonstration* results<br>Python 3.9+<br>DRAM Bender [60]<br>Boost 1.71+<br>Xilinx Vivado 2020.2+<br>Slurm 20+ |
| Hardware | x86 machine w/ PCIe 3.0 x16 slot<br>FPGA development board supported by DRAM Bender (e.g., Xilinx Alveo U200)<br>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)<br>3 to 4 weeks per DRAM module (Replicate characterization results)<br>≈ 5 days (Demonstration)<br>≈ 1 day (Mitigation) |
| Publicly available? | Zenodo (https://doi.org/10.5281/zenodo.7750890)<br>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 [60, 95] (e.g., Xilinx Alveo U200 [94]).
- Heater pads attached to the DRAM module under test.
- A temperature controller (e.g., MaxWell FT200 [98]) 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) [111] processor and a Samsung `M378A2K43CB1-CTD` DDR4 DRAM module with the 8Gb C-Dies from Mfr. S (`K4A8G085WC-BCTD`) [112]. 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 [136, 137] 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 [110]) 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 [95] 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. 19.
(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. 20.

### 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 2.

## 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. 13, Fig. 15, Fig. 16, Fig. 18, and Fig. 21), 2) real-system demonstration of RowPress (Fig. 19 and Fig. 20), and 3) simulation results of our proposed RowPress mitigation (Table 2).

## 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 3, and encourage the reader to refer to the help messages of the program for all options and their explanations.

**Table 3: 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)<br>1 (ACmin for given access pattern)<br>3 (Retention failures for given refresh-idle time)<br>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 [44]) 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`.