Uncovering In-DRAM RowHammer Protection Mechanisms: A New Methodology, Custom RowHammer Patterns, and Implications

Hasan Hassan[†] Yahya Can Tuğrul[†‡] Jeremie S. Kim[†] Victor van der Veen[σ]
Kaveh Razavi[†] Onur Mutlu[†]
[†]*ETH Zürich* [‡]*TOBB University of Economics & Technology* [σ]*Qualcomm Technologies Inc.*

Abstract

The RowHammer vulnerability in DRAM is a critical threat to system security. To protect against RowHammer, vendors commit to security-through-obscurity: modern DRAM chips rely on undocumented, proprietary, on-die mitigations, commonly known as *Target Row Refresh* (TRR). At a high level, TRR detects and refreshes potential RowHammer-victim rows, but its exact implementations are not openly disclosed. Security guarantees of TRR mechanisms cannot be easily studied due to their proprietary nature.

To assess the security guarantees of recent DRAM chips, we present *Uncovering TRR* (U-TRR), an experimental methodology to analyze in-DRAM TRR implementations. U-TRR is based on the new observation that data retention failures in DRAM enable a side channel that leaks information on how TRR refreshes potential victim rows. U-TRR allows us to (i) understand how logical DRAM rows are laid out physically in silicon; (ii) study undocumented on-die TRR mechanisms; and (iii) combine (i) and (ii) to evaluate the RowHammer security guarantees of modern DRAM chips. We show how U-TRR allows us to craft RowHammer access patterns that successfully circumvent the TRR mechanisms employed in 45 DRAM modules of the three major DRAM vendors. We find that the DRAM modules we analyze are vulnerable to RowHammer, having bit flips in up to 99.9% of all DRAM rows.

CCS Concepts

• **Hardware → Dynamic memory**; • **Security and privacy → Hardware reverse engineering**.

Keywords

DRAM, RowHammer, Reliability, Security, Testing

ACM Reference Format:
Hasan Hassan, Yahya Can Tugrul, Jeremie S. Kim, Victor van der Veen, Kaveh Razavi, and Onur Mutlu. 2021. Uncovering In-DRAM RowHammer Protection Mechanisms: A New Methodology, Custom RowHammer Patterns, and Implications. In *MICRO'21: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3466752.3480110

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MICRO '21, October 18–22, 2021, Virtual Event, Greece
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8557-2/21/10…$15.00
https://doi.org/10.1145/3466752.3480110

1 Introduction

DRAM [21] has long been the dominant memory technology used in almost all computing systems due to its low latency and low cost per bit. DRAM vendors still push DRAM technology scaling forward to continuously shrink DRAM cells to further reduce the cost of DRAM [81]. Unfortunately, DRAM technology scaling leads to increasingly important DRAM reliability problems as DRAM cells get smaller and the distance between the cell reduce. Kim et al. [56] report that most DRAM chips manufactured since 2010 are susceptible to disturbance errors that are popularly referred to as RowHammer. Recent works [25, 45, 56, 83, 90, 103, 126, 133–135] explain the circuit-level charge leakage mechanisms that lead to the RowHammer vulnerability. A security attack exploits the RowHammer vulnerability by hammering (i.e., repeatedly activating and precharging) an *aggressor* row many times (e.g., 139K in DDR3 [56], 10K in DDR4 [54], and 4.8K in LPDDR4 [54])[1] to cause bit flips in the cells of the *victim* rows that are physically adjacent to the hammered row. Since the discovery of RowHammer, researchers have proposed many techniques that take advantage of the RowHammer vulnerability to compromise operating systems [7, 15, 16, 29, 38, 44, 62, 96, 98, 109, 123, 124, 128, 140], web browsers [8, 19, 23, 24, 28], cloud virtual machines [100, 129], remote servers [71, 122], and deep neural networks [34, 136].[2] Thus, RowHammer is a clear major threat to system security.

To prevent RowHammer, DRAM vendors equip their DRAM chips with a *mitigation mechanism* known as Target Row Refresh (TRR) [16, 24, 54, 78]. The main idea of TRR is to detect an aggressor row (i.e., a row that is being rapidly activated) and refresh its victim rows (i.e., neighboring rows that are physically adjacent to the aggressor row). TRR refreshes the victim rows separately from the *regular* refresh operations [6, 17, 72, 73, 125] that must periodically (e.g., once every 64 ms) refresh each DRAM row in the entire DRAM chip. Some of the major DRAM vendors advertise *RowHammer-free* DDR4 DRAM chips [24, 69, 77]. However, none of the DRAM vendors have so far disclosed the implementation details let alone proved the protection guarantees of their TRR mechanisms. It is long understood that security cannot be achieved only through obscurity [3, 106, 107]. Yet, such is the current state of practice when it comes to DRAM.

The recent TRRespass work [24] shows that in certain DRAM chips, the TRR mechanism keeps track of only a few aggressor rows. Hence, an access pattern that hammers many aggressor rows can circumvent the TRR protection and induce RowHammer bit flips. While TRRespass already demonstrates the flaws of certain TRR

[1] For DDR3 chips, [56] reports the minimum number of row activations on a *single* aggressor row (i.e., single-sided RowHammer) to cause a RowHammer bit flip. For DDR4 and LPDDR4 chips, [54] reports the minimum number of row activations to *each of the two* immediately-adjacent aggressor rows (i.e., double-sided RowHammer).
[2] A review of many RowHammer attacks and defenses is provided in [83].

implementations, it does *not* propose a methodology for *discovering* these flaws. According to [24], simply increasing the number of aggressor rows is *not* sufficient to induce bit flips on 29 out of 42 of the DDR4 modules that were tested by [24]. However, it is unclear whether such DDR4 chips are *fully* secure due to the lack of insight into the detailed operation of TRR in these chips. Thus, we need new methods for identifying whether or not DRAM chips are fully secure against RowHammer.

U-TRR. We develop U-TRR, a new and practical methodology that uncovers the inner workings of TRR mechanisms in modern DRAM chips. To develop U-TRR, we formulate TRR as a function that takes a number of DRAM accesses including aggressor rows as input and refreshes a number of rows that are detected as victims. The goal of U-TRR is to enable the observation (i.e., uncovering) of all refreshes generated by TRR after inducing a carefully crafted sequence of DRAM accesses. To make this possible, we make a key observation that retention failures that occur on a DRAM row can be used as a side channel to detect *when* the row is refreshed, due to either TRR or periodic refresh operations. We easily distinguish TRR-induced refreshes from regular refreshes as the latter occur at fixed time intervals independently of the access pattern.

We build two new tools, *Row Scout* (*RS*) and *TRR Analyzer* (*TRR-A*), that make use of this new observation to construct U-TRR and thus enable a deep analysis of TRR. The goal of *RS* is to find a set of DRAM rows that meet certain requirements as needed by a *TRR-A* experiment and identify the data retention times of these rows. The goal of *TRR-A* is to use the *RS*-provided rows to determine when a TRR mechanism refreshes a victim row by exactly distinguishing between TRR refreshes and regular refreshes, and thus build an understanding of the underlying TRR operation.

Row Scout (*RS*) profiles the data retention time of DRAM rows and passes to *TRR-A* a list of rows that satisfy two key requirements. First, a DRAM row should have a *consistent* retention time that does *not* vary over time based on effects such as Variable Retention Time (VRT) [72, 99, 102, 132]. *RS* should not supply to *TRR-A* a row with an inconsistent retention time since it would not be possible for *TRR-A* to distinguish whether the row has been refreshed or it simply retained its data correctly for longer than the profiled retention time. Second, to enable observing how many and which DRAM rows TRR treats as victim rows, *RS* should provide *multiple* DRAM rows that have similar retention times and that are located at certain *configurable* distances with respect to each other. It is crucial to find rows with similar retention times in order to observe whether or not TRR can refresh multiple rows at the same time. These two requirements enable reliable and precise analysis of TRR-induced refreshes by *TRR-A*.

TRR Analyzer (*TRR-A*) discovers 1) access patterns that cause the TRR mechanism to treat a certain row as an aggressor row and 2) when a TRR-induced refresh targets a victim row. At a high level, *TRR-A* infers the occurrence of a TRR-induced refresh operation to an *RS*-provided row if the row does *not* contain any bit flips after disabling regular refreshes and accesses to the row for its *RS*-profiled retention time. Only a TRR-induced refresh can prevent the *RS*-provided row from experiencing bit flips by refreshing the row before it experiences a retention failure. Thus, *TRR-A* attributes *not* observing a bit flip in an *RS*-provided row to TRR-induced refresh.

TRR-A operates in three main steps. First, it initializes to known data (e.g., all *ones*) i) the *RS*-provided rows and ii) the rows it selects as aggressor rows in the experiment. Then, *TRR-A* lets the *RS*-provided rows leak their charge for half of their profiled retention time. Second, *TRR-A* hammers (i.e., repeatedly activates and precharges) the aggressor rows. After the hammers, *TRR-A* issues a small number of DRAM refresh commands (originally used for only periodic refresh) so that the TRR-induced row refresh can take place.[3] *TRR-A* then waits for the remaining half of the profiled retention time. If an *RS*-provided row was not refreshed by TRR-induced refresh operations as a result of the hammers issued by *TRR-A*, the *RS*-provided row would now have not been refreshed for its full profiled retention time and will contain bit flips. Third, *TRR-A* reads back the data stored in the *RS*-provided rows and checks for bit flips. *Observing no bit flips* in an *RS*-provided row indicates that either a TRR-induced or regular refresh, which *TRR-A* can easily distinguish between since the regular refreshes happen periodically, targeted the same row during step two. In contrast, *observing bit flips* indicates that the *RS*-provided row was never refreshed. These three steps constitute the core of the experiments that we conduct to understand different in-DRAM TRR implementations.

Security analysis of TRR. We use the *RS* and *TRR-A* on 45 DDR4 modules from the three major DRAM chip vendors (i.e., Micron, Samsung, SK Hynix). Our methodology uncovers important details of the in-DRAM TRR designs of all vendors. We demonstrate the usefulness of these insights for developing effective RowHammer attacks on DRAM chips from each vendor by crafting specialized DRAM access patterns that hammer a row enough times to cause a RowHammer bit flip *without* alerting the TRR protection mechanism (i.e., by redirecting potential TRR refreshes *away from* the victim rows). In our evaluation, we find that all tested DRAM modules with different manufacturing dates (from 2016 to 2020) are vulnerable to the new access patterns we can craft via U-TRR. Further, we find that 1) over 99.9% of the DRAM rows are vulnerable (i.e., have at least one bit flip) to the new access patterns and 2) the new access patterns cause up to 9.4 million bit flips per DRAM bank. The large number of RowHammer bit flips caused by our specialized access patterns has significant implications for systems protected by Error Correction Codes (ECC) [47, 92, 93, 95]. Our analysis shows that the U-TRR-discovered access patterns can cause up to 7 bit flips at arbitrary locations in one 8-byte dataword, suggesting that typical ECC schemes capable of correcting one error/symbol and detecting two errors/symbols (e.g., SECDED ECC [10, 37, 43, 60, 61, 79, 87, 118] and Chipkill [2, 20, 86]) *cannot* provide sufficient protection against RowHammer even in the presence of TRR mechanisms.

Contributions. We expect that U-TRR will help future research on both evaluating the security of existing RowHammer protections and the design of more secure RowHammer mitigation mechanisms. In summary, we make the following major contributions:

- We develop U-TRR, a new methodology for reverse-engineering Target Row Refresh (TRR) and regular refresh mechanisms.
- We use U-TRR to understand and uncover the TRR implementations of 45 DDR4 modules from the three major DRAM vendors.

[3] The current TRR implementations avoid changing the DDR interface by piggybacking TRR-induced refreshes to regular refresh commands [24, 131].

This evaluation shows that our new methodology is broadly applicable to any DRAM chip.

- Leveraging the TRR implementation details uncovered by U-TRR, we craft specialized RowHammer access patterns that make existing TRR protections ineffective.
- Our specialized U-TRR-discovered access patterns are significantly more effective than patterns from the state-of-the-art [24]: we show that our new RowHammer access patterns cause 1) bit flips in 45 DDR4 modules we comprehensively examine, 2) bit flips in up to 99.9% of the all rows in a DRAM bank, and 3) two and more (up to 7) bit flips in a single 8-byte dataword, enabling practical RowHammer attacks in systems that employ ECC.

2 Background

We provide background on DRAM and the RowHammer phenomenon that is required for the reader to understand how U-TRR can precisely uncover the behavior of an in-DRAM RowHammer mitigation mechanism. For more detailed descriptions of DRAM organization and operation, we refer the reader to many prior works [6, 11–15, 24, 31, 32, 52–54, 56–58, 63–67, 72–74, 88, 93, 111–114, 127, 130, 138, 139].

2.1 DRAM Organization

Fig. 1 shows the typical organization of a modern DRAM system. DRAM is organized into a hierarchical array of billions of DRAM cells, each holding one bit of data. In modern systems, a CPU chip implements a set of memory controllers, where each memory controller interfaces with a DRAM *channel* to perform read, write, and maintenance operations (e.g., refresh) via a dedicated I/O bus that is independent of other channels in the system. A DRAM channel can host one or more *DRAM modules*, where each module consists of one or more *DRAM ranks*. A rank is comprised of multiple *DRAM chips* that operate in lock step and ranks in the same channel time-share the channel's I/O bus.

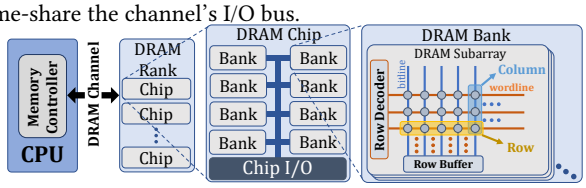

Figure 1: Typical DRAM system organization.

A DRAM chip consists of multiple DRAM *banks*, which share an internal data/command bus. Within a DRAM bank, DRAM cells are organized into multiple (e.g., 128) dense two-dimensional arrays of DRAM cells called *subarrays* [12, 57, 111] and corresponding peripheral circuitry for manipulating the data within the subarray. A row of cells (i.e., DRAM *row*) within a subarray share a wire (i.e., *wordline*), which is driven by a *row decoder* to *open* (i.e., select) the row of cells to be read or written. A column of cells (i.e., DRAM *column*) within a subarray share a wire (i.e., *bitline*), which is used to read and write to the cells with the help of a *row buffer* (consisting of *sense amplifiers*). This hierarchical layout of DRAM cells enables any data in the DRAM system to be accessed and updated using unique rank, bank, row, and column addresses.

2.2 DRAM Operation

The memory controller interfaces with DRAM using a series of commands sent over the I/O bus. To read or write data, the memory controller must first issue an *activate* (ACT) command to *open* a row corresponding to the provided memory address. When a row is activated, the data within the DRAM row is copied into the row buffer of the subarray. The memory controller then issues *READ* (RD) or *WRITE* (WR) commands to read or update the data in the row buffer. Changes to the data in the row buffer propagate to the DRAM cells in the opened row. When data from another row is required, the memory controller must first issue a *precharge* (PRE) command that *closes* the open row and prepares the bank for the activation of another row.

DRAM Refresh. A DRAM cell stores a data value in the form of charge in its capacitor (e.g., a charged cell can represent 0 or 1 and vice versa). Since the capacitor naturally loses charge over time, the capacitor charge must be actively and periodically *refreshed* to prevent information loss due to a data retention failure [39, 40, 48–50, 72, 73, 94]. To enable such periodic refresh of all DRAM cells, the memory controller must periodically issue a *refresh* (REF) command (e.g., every 7.8 µs) to ensure that every DRAM cell is refreshed once at a fixed *refresh interval* (i.e., typically 32 or 64 ms) [39, 40, 72, 73].

2.3 RowHammer

Modern DRAM chips suffer from disturbance errors that occur when a high number of activations (within a refresh interval) to one DRAM row unintentionally affects the values of cells in nearby rows [56]. This phenomenon, popularly called *RowHammer* [56, 83], stems from electromagnetic interference between circuit elements. RowHammer becomes exacerbated as manufacturing process technology node size (and hence DRAM cell size) shrinks and circuit elements are placed closer together [54, 82]. As demonstrated in prior work [54, 56], the RowHammer effect is strongest between immediately physically-adjacent rows. RowHammer bit flips are most likely to appear in neighboring rows physically adjacent to a *hammered row* that is activated many times (e.g., 139*K* in DDR3 [56], 10*K* in DDR4 [54], and 4.8*K* in LPDDR4 [54])[1]. A hammered row is also called an *aggressor row* and a nearby row that is affected by the hammered row is called a *victim row*, regardless of whether or not the victim row actually experiences RowHammer bit flips.

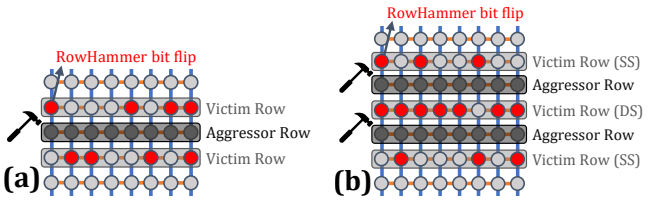

Figure 2: Typical Single-sided (SS) and Double-sided (DS) RowHammer access patterns.

To most effectively exploit the RowHammer phenomenon, attackers typically perform *i*) single-sided RowHammer (i.e., repeatedly activate one aggressor row that is physically adjacent to the victim row, as we show in Fig. 2a) [56] or *ii*) double-sided RowHammer (i.e., repeatedly activate in an alternating manner two aggressor rows that are both physically adjacent to the victim row, as we show in Fig. 2b) [109, 110]. Prior works have shown that double-sided RowHammer leads to more bit flips and does so more quickly than single-sided RowHammer [54, 56, 83, 109, 110].

2.4 RowHammer Mitigation Mechanisms

To combat attacks that exploit the RowHammer phenomenon, various RowHammer mitigation mechanisms have been proposed in literature [4, 5, 9, 22, 27, 55, 56, 59, 68, 91, 115, 117, 121, 124, 130, 131, 137].[2] Yaglikci et al. [130] classify these mitigation mechanisms into four groups: *i*) increasing the refresh rate to reduce the number of activations that can be performed within a refresh interval [4, 56], *ii*) isolating sensitive data from DRAM rows that an attacker can potentially hammer [9, 59, 124], *iii*) keeping track of row activations and refreshing potential victim rows [5, 22, 55, 56, 68, 91, 115, 117, 121, 131, 137], and *iv*) throttling row activations to limit the times a row can be activated within a refresh interval [27, 56, 130]. Many of these research proposals describe the details of their proposed mechanisms and discuss their security guarantees [56, 91, 130].

Unfortunately, DRAM vendors currently implement different *proprietary* in-DRAM RowHammer mitigation mechanisms, which they broadly refer to as Target Row Refresh (TRR) [16, 24, 54, 78]. TRR detects a potential aggressor row and refreshes its neighbor rows. The vendors have so far not disclosed the implementation details of their TRR mechanisms, and thus the security guarantees of such TRR mechanisms *cannot* be properly and openly evaluated.

In fact, a recent work, TRRespass [24], shows that existing proprietary in-DRAM TRR mechanisms can be circumvented via many-sided RowHammer attacks, which aim to overflow the internal tables that TRR uses to detect aggressor rows. As such, it is critical to develop a rigorous methodology to understand the weaknesses of TRR mechanisms and develop more secure alternatives.

Our goal is to study in-DRAM TRR mechanisms so that we can understand how they operate, assess their security, and enable fully-secure DRAM against RowHammer.

3 Overview of U-TRR

U-TRR is a new methodology for gaining visibility into Target Row Refresh (TRR) operations. It enables system designers and researchers to understand how TRR detects an aggressor row, when it refreshes the victim rows of the aggressor row, and how many potential victim rows it refreshes. U-TRR enables users to easily conduct experiments that uncover the inner workings of the TRR mechanism in an off-the-shelf DRAM module.

Fig. 3 illustrates the two components of U-TRR: *Row Scout* (*RS*) and *TRR Analyzer* (*TRR-A*). *RS* finds a set of DRAM rows that meet certain requirements as needed by *TRR-A* and identifies the data retention times of these rows. *TRR-A* uses the *RS*-provided rows to distinguish between TRR refreshes and regular refreshes, and thus builds an understanding of the underlying TRR mechanism.

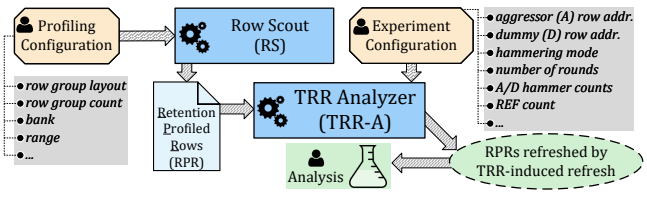

Figure 3: Overview of U-TRR.

3.1 Overview of *Row Scout* (*RS*)

The goal of *RS* is to identify a list of useful DRAM rows and their retention times, and provide this list to *TRR-A*. *RS* profiles the retention time of a DRAM row by writing a data pattern (e.g., all ones) [72, 73] to the entire row and measuring the time interval for which the row can correctly retain its data without being refreshed.

A useful DRAM row must satisfy two key requirements. First, the retention time of the row should be consistent and *not* vary over time based on effects such as Variable Retention Time (VRT) [47, 48, 72, 80, 99, 102, 132]. A consistent retention time is essential for *TRR-A* to accurately infer whether or not a row has been refreshed after a specific time interval, based on whether or not the row contains retention failures. *RS* validates the retention time of a row one thousand times to ensure its consistency over time.

Second, to observe exactly which DRAM rows the TRR mechanism treats as victim rows for each aggressor row it detects, *RS* should provide *multiple* DRAM rows that have the *same* retention times and that are located at certain *configurable* distances with respect to each other (we call this a *row group*). It is crucial to find rows with the same retention times in order to observe whether or not TRR can refresh *multiple* rows at the same time. Enforcing a particular distance between rows is useful when the user wants to specify an aggressor row at a specific distance from the *RS*-provided rows. These two requirements enable reliable and precise analysis of TRR-induced refreshes by *TRR-A*.

As shown in Fig. 3, the number of row groups (i.e., row group count) and the relative distances of the rows within the group (i.e., row group layout) is specified in the *profiling configuration*. U-TRR user also specifies a certain DRAM bank and row range within the bank for the *RS* to search for the desired row groups. We discuss the operation and capabilities of *RS* in greater detail in §4.

3.2 Overview of *TRR Analyzer* (*TRR-A*)

The goal of *TRR-A* is to use *RS*-provided rows to determine when a TRR mechanism refreshes a victim row by exactly distinguishing between TRR refreshes and regular refreshes, and thus build an understanding of the underlying TRR operation. *TRR-A* runs a RowHammer attack and monitors retention failures in *RS*-provided rows to determine *when* TRR refreshes any of these rows. As Fig. 3 shows, *TRR-A* operates based on an *experiment configuration*, which includes several parameters we discuss in §5.2. Fig. 4 shows the three steps a *TRR-A* experiment generally follows:

(1) *TRR-A* uses *RS*-provided rows as victim rows and initializes ❶ them by writing into them the same data pattern that is used during retention profiling with *RS*. Since the RowHammer vulnerability greatly depends on the data values stored in an aggressor row [54, 56], *TRR-A* also initializes aggressor rows to the data values that the user specifies in the experiment configuration. *TRR-A* waits for half of the victim rows' retention time ($\frac{T}{2}$) without performing any refreshes or accesses.

(2) *TRR-A* hammers ❷ the aggressor rows and issues REF ❸ commands based on the experiment configuration.

(3) After again waiting for the half of the victim rows' retention time ($\frac{T}{2}$), excluding the time spent on hammering and refresh during step (2), *TRR-A* reads the victim rows and compares ❹ the data stored in them against the initial data value

pattern written to them in step (1). A victim row with no bit flips indicates that either a TRR-induced or a regular refresh operation targeted the victim row while serving the REF commands in step (2). *TRR-A* easily distinguishes between a TRR-induced and a regular refresh as the latter refreshes a row periodically (e.g., once in every 8*K* REF commands). The user can then examine the TRR-induced refresh patterns to gain insight into the underlying TRR implementation.

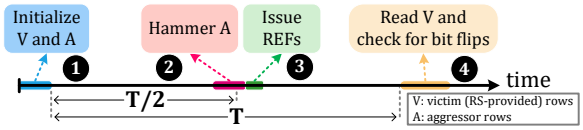

Figure 4: General approach for detecting a TRR-induced refresh using *TRR-A*.

TRR-A offers flexibility in experiments via parameters in the profiling and experiment configurations, which enable analyzing different TRR implementations with low effort. §5 provides details.

3.3 Required Experimental Setup for U-TRR

Both the *RS* and *TRR-A* tools require a way to directly interface with a DRAM module at a DDR-command level. This is because the tools need to accurately control when an individual DDR command (e.g., ACT, REF) is issued to the DRAM module. However, existing systems based on commodity CPUs can access DRAM *only* using load/store instructions. Therefore, we implement *RS* and *TRR-A* using SoftMC [33, 105], an FPGA-based DRAM testing infrastructure that provides precise control on the DDR commands issued to a DRAM module. We modify SoftMC to support testing DDR4 modules, as also done in [24, 54, 88, 89]. Fig. 5 shows our experimental SoftMC setup. Table 1 provides a list of the 45 DDR4 DRAM modules we analyze in this paper.

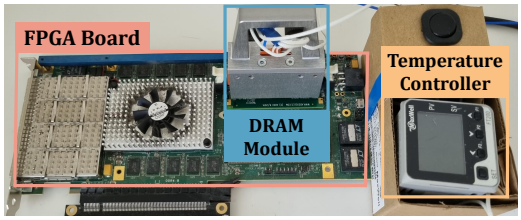

Figure 5: DDR4 SO-DIMM SoftMC [33] experimental setup.

4 Row Scout (*RS*)

U-TRR uses data retention failures as a side channel to determine if and when a row receives a (TRR-induced or regular) refresh. As such, to know how long a row can retain its data correctly without being refreshed, U-TRR requires a mechanism for profiling the retention time of a DRAM row. We define *DRAM row retention time* as the maximum time interval for which all cells in the row can correctly retain their data without being refreshed.

Unlike existing DRAM retention-time profiling techniques [18, 72, 73, 94, 99], U-TRR does *not* require a profiler that finds the retention time of *all* rows in a DRAM chip. Instead, U-TRR needs to search for a small set of DRAM rows (i.e., tens of rows depending on the experiment) that match certain criteria (§4.1) as specified by the U-TRR user based on the desired experiment.

4.1 Row Scout (*RS*) Requirements

Depending on the experiment that the U-TRR user conducts, *TRR-A* needs the data retention time of DRAM rows that match different criteria. We identify the following general requirements for *RS* to enable it to search for DRAM rows suitable for *TRR-A*.

Rows with uniform retention time. A TRR mechanism may refresh multiple victim rows. For instance, during a single-sided RowHammer attack, a TRR mechanism may refresh the row on either side of the aggressor *or* both at the same time. To examine whether or not TRR can refresh multiple victim rows at the same time (i.e., with a single REF), *RS* must provide multiple rows (i.e., a *row group*) that have the *same* retention time.

Relative positions of profiled rows. The location of a victim row depends on the location of the aggressor rows that the U-TRR user specifies for an experiment. For example, for a double-sided RowHammer attack (see Fig. 2b), *RS* must provide three rows with the same retention time that are exactly one row apart from each other. *TRR-A* can then analyze which of the three victim rows get refreshed by TRR when hammering the two aggressor rows that are placed between the victim rows. We represent the relative positions of rows in a row group (i.e., the *row group layout*) using a notation such as R-R-R, where 'R' indicates a retention-profiled row and '-' indicates a distance of one DRAM row. *RS* must find a row group based on the row group layout that the user specifies.

Rows in specific DRAM regions. TRR may treat rows in different parts of a DRAM chip differently by operating independently at different granularities. For example, TRR may operate independently at the granularity of a DRAM bank or a region of DRAM bank. To identify the granularity at which TRR operates, *RS* must find DRAM rows within a *specific region* of a DRAM chip.

Rows with consistent retention time. An *RS*-provided row must have a consistent retention time such that U-TRR can accurately infer the occurrence of a TRR-induced refresh operation based on whether the row contains retention failures after a time period equivalent to the row's retention time. The main difficulty is a phenomenon known as Variable Retention Time (VRT) [47, 48, 72, 80, 99, 102, 132], which causes the retention time of certain DRAM cells to change over time. If an *RS*-provided row has an inconsistent retention time that was initially measured to be T, U-TRR will not be able to correctly infer the occurrence of a TRR-induced refresh operation.[4] To ensure consistency of a row's retention time, *RS* validates the retention time of a row *one thousand times* in order to rule out inconsistencies (that are due to VRT).

Rows with short retention times. The time it takes to finish a single U-TRR experiment depends on the retention time of the rows *RS* finds. This is because even retention-weak DRAM rows typically retain their data correctly for tens or hundreds of milliseconds [72, 73, 94], whereas other *TRR-A* operations (e.g., reading from or writing to a row, hammering a row, performing refresh) often take much less than a millisecond. Thus, as Fig. 4 shows, the

[4] U-TRR fails to correctly infer a TRR-induced refresh when a row retains its data for significantly longer or shorter than T. If the row retains its data for longer than T, U-TRR will *always* infer the occurrence of a TRR-induced refresh operation. If the row fails too soon (i.e., before $\frac{T}{2}$ or during Step 1 in §3.2), U-TRR will always observe retention failures, since even a TRR-induced refresh will not be able to prevent the bit flip (in Step 2 in §3.2). Consequently, U-TRR will *always* infer that a TRR-induced refresh operation was not issued to the row.

duration of a *TRR-A* experiment is dominated by retention times (T) of the profiled rows. To reduce the overall experiment time, it is critical for *RS* to identify rows with short data retention times.

4.2 Row Scout (*RS*) Operation

We design and implement *Row Scout* (*RS*), a DRAM retention time profiler, such that it satisfies the requirements listed in §4.1. We implement *RS* using a modified version of SoftMC [33, 105] with DDR4 support (described in §3.3).

We illustrate the operation of *RS* in Fig. 6. ❶ *RS* scans a full range of DRAM rows within a DRAM bank, as specified in the profiling configuration (Fig. 3), and collects the addresses of rows that experience retention failures if not refreshed over the time interval *T*. *RS* initially sets *T* to a small value (e.g., 100 ms) in order to identify rows with small retention times as we discuss in the requirements of *RS* (§4.1). ❷ *RS* creates candidate row groups by combining the appropriate row addresses (with retention time *T*) that match the row group layout specified in the profiling configuration. If the number of candidate row groups is less than the number of row groups to find according to the profiling configuration, ❸ *RS* increases *T* by a certain amount (e.g., 50 ms) and starts over from ❶. Otherwise, ❹ *RS* tests each row in a candidate row group one thousand times to ensure that all rows in the candidate row group have a consistent retention time that is equal to *T*. If the number of candidate row groups that pass the retention time consistency test is less than the number of row groups to find according to the profiling configuration, ❺ *RS* increases *T* by a certain amount (e.g., 50 ms) and starts over from ❶. Otherwise, ❻ *RS* provides a list of retention time-profiled rows to be used by *TRR-A*.

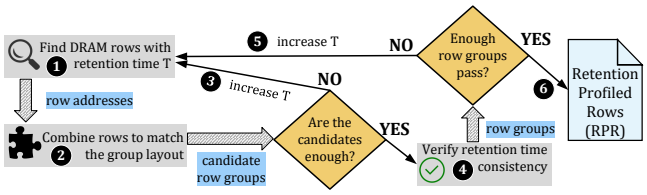

Figure 6: Detailed *RS* operation.

5 Analyzing TRR-induced Refresh

TRR-A is a configurable and extensible component in U-TRR for analyzing TRR-induced as well as regular refresh operations. We use the *TRR-A* to inspect in-DRAM RowHammer mitigations in modules from the three major DRAM vendors. We implement *TRR-A* on top of a modified version of SoftMC [33, 105] with DDR4 support. In §3, we discuss the general operation of *TRR-A* using Fig. 4.

5.1 TRR Analyzer Requirements

We identify and discuss four key requirements needed to enable reverse engineering a RowHammer mitigation mechanism. First, to analyze the capability of TRR in detecting multiple aggressor rows, *TRR-A* should allow the user to specify one or more aggressor rows, their corresponding hammer counts, and the order in which to hammer the aggressor rows.

Requirement 1. *Ability to hammer multiple aggressor rows with individually configurable hammer counts in a configurable order.*

The user should be able to specify dummy rows[5] that can be hammered to divert the TRR mechanism to refresh the neighbors of a dummy row instead of victims of an aggressor row.

Requirement 2. *Ability to specify dummy rows that are hammered in addition to the aggressor rows.*

To force the TRR mechanism to perform an additional refresh operation when desired during the experiment, *TRR-A* should allow flexibly issuing an any number of REF commands at arbitrary times.

Requirement 3. *Ability to flexibly issue* REF *commands.*

The TRR mechanism under study may retain its state beyond a single experiment, potentially causing the TRR mechanism to detect different rows as aggressors depending on previous experiments. For example, in a *counter-based TRR* (§6.1.2), the TRR mechanism's internal counter values updated due to a previous experiment might affect the outcome of future experiments. To isolate an experiment from the past experiments, *TRR-A* should reset TRR's internal state to a consistent state after each experiment.

Requirement 4. *Ability to reset TRR mechanism's internal state.*

5.2 TRR Analyzer Operation

We explain how *TRR Analyzer* (*TRR-A*) satisfies all of the requirements described in §5.1 to enable detailed experiments that uncover the implementation details of in-DRAM RowHammer mitigation mechanisms. Fig. 7 illustrates a typical *TRR-A* experiment and provides a list of the experiment configuration parameters.

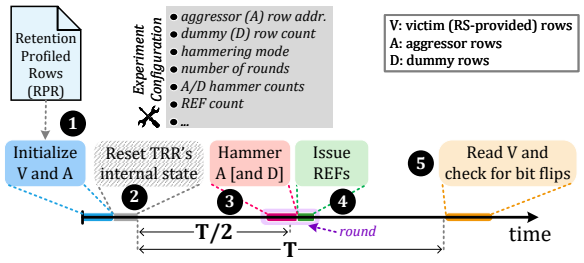

Figure 7: Detailed *TRR-A* experiment.

Initializing rows' data. *TRR-A* initializes ❶ the *RS*-provided rows by writing into them the same data pattern that that *RS* used for profiling these rows. *TRR-A* also initializes the aggressor rows, whose addresses are specified in the experiment configuration, as the RowHammer vulnerability greatly depends on the data values stored in an aggressor row [54, 56].

Resetting internal TRR state. To reset the TRR mechanism's internal state ❷ so as to satisfy Requirement 4, *TRR-A* performs refresh for multiple refresh periods while hammering a set of dummy rows. *TRR-A* issues REF commands at the default refresh rate (i.e., one REF every 7.8 µs) for several (e.g., 10) 64 ms refresh periods. During these refresh operations, the *RS*-provided rows do *not* experience bit flips as they get refreshed by the regular refresh operations. Between two refresh commands, *TRR-A* hammers a large number (e.g., 128) of dummy rows as many times as the DRAM timing parameters (i.e., tRAS and tRP [57, 63, 66, 77, 116]) permit. *TRR-A* automatically selects dummy rows from the same bank as the *RS*-provided and aggressor rows, since a TRR mechanism may

[5] A dummy row operates similarly to an aggressor row, but it can be implemented more efficiently in a SoftMC program since a dummy row does not need to be initialized with specific data unlike an aggressor row.

operate independently in each bank. To ensure that hammering the dummy rows does *not* cause RowHammer bit flips on the *RS*-provided rows, *TRR-A* enforces a minimum row distance of 100 between a selected dummy row and the *RS*-provided rows. We find that the operations we perform in ❷ make the TRR mechanism clear any internal state that is relevant to rows activated in past experiments or during ❶. Resetting TRR's internal state is an optional step since the user may sometimes want to examine how TRR operates across multiple experiments.

Hammering aggressor and dummy rows. The experiment configuration specifies the addresses and individual hammer counts of aggressor rows that *TRR-A* accordingly hammers ❸. Optionally, the configuration specifies a number of dummy rows that can be hammered to divert the TRR mechanism away from the actual aggressor rows. *TRR-A* automatically selects dummy rows based on the criteria for selecting dummy rows explained above. The experiment configuration specifies a single hammer count for all dummy rows; *TRR-A* hammers each dummy row by that count.

Hammering modes. The order with which DRAM rows get hammered can affect both the magnitude of the disturbance each hammer causes and how the TRR mechanism detects an aggressor row. *TRR-A* supports two *hammering modes*. *Interleaved* mode hammers each aggressor row one at a time until all aggressors accumulate their corresponding hammer count. *Cascaded* mode repeatedly hammers one aggressor row until it accumulates its corresponding hammer count and then does the same for the other aggressor rows. In our experiments, we observe that interleaved hammering generally causes more bit flips (up to four orders of magnitude) compared to cascaded hammering for a given hammer count. In contrast, we find that cascaded hammering is more effective at evading the TRR mechanism than interleaved hammering. Therefore, it is critical to support both hammering modes.

Issuing `REF`s. To cause the TRR mechanism to perform TRR-induced refresh operations on the victim rows, *TRR-A* issues a number of `REF` commands ❸ according to the experiment configuration. As shown in Fig. 7, *TRR-A* issues the `REF` commands *after* hammering the aggressor/dummy rows and waiting for half of the retention time T of the *RS*-provided rows. This is to 1) allow TRR to potentially detect the aggressor rows hammered in ❸ and 2) ensure that a victim row refreshed in ❹ does *not* experience retention failures until its data is read in ❺.

Hammering rounds. To allow distributing hammers between multiple `REF` commands, *TRR-A* performs hammers in multiple *rounds*. A round consists of 1) hammering the aggressor and dummy rows and 2) issuing `REF` commands as the last operation of the round. The experiment configuration specifies the aggressor/dummy row hammer counts and the number of `REF`s to issue per round.

5.3 Determining Physical Row Mapping

DRAM rows that have consecutive *logical* row addresses (from the perspective of the memory controller) may not be physically adjacent inside a DRAM chip [15, 46, 57] due to two main reasons. First, post-manufacturing row repair techniques (e.g., [10, 35, 41, 42, 47, 75, 85, 118]) may repair a faulty row by remapping the logical row address that points to the faulty row to a spare row at a different physical location inside a DRAM chip. Second, a row address decoder in the DRAM chip may not necessarily map consecutive row addresses to adjacent wordlines [15, 46, 57]. The row address decoder can maintain the logical row address order in physical row space but it may scramble the logical-to-physical row mapping as well depending on the circuit implementation.

Since a TRR mechanism should refresh rows that are physically adjacent to an aggressor row, we need to ensure that *TRR-A* uses physically adjacent rows in the experiments. For this purpose, we use two methods. First, before we run *RS*, we reverse engineer the logical-to-physical row address mapping of a DRAM chip by disabling refresh and performing double-sided RowHammer.[6] We analyze the rows at which RowHammer bit flips appear, so as to determine the physical adjacency of rows and hence reconstruct the physical row mapping. If the logical row address order is preserved in the physical space, we simply observe RowHammer bit flips on logical row addresses $R-1$ and $R+1$ as a result of hammering R. Otherwise, bit flips occur in other logical rows depending on the logical-to-physical row address mapping of the DRAM chip. Second, before an experiment, *TRR-A* verifies that the given aggressor row can actually successfully hammer the *RS*-provided rows by hammering the aggressor row a large number of times (i.e., $300K$) with refresh disabled. Doing so ensures that the *RS*-provided or aggressor rows are *not* remapped due to post-manufacturing repair and are still physically adjacent to each other.

6 Reverse-Engineering TRR

We use `U-TRR`, the components of which we describe in §4 and §5, to gain insights about TRR implementations by analyzing DDR4 modules from three major DRAM vendors. We follow a systematic approach in our reverse engineering to gain these insights. First, we discover which refresh commands perform TRR. Second, we observe how many rows are concurrently refreshed by TRR. Third, we try to understand the strategy that the TRR mechanism employs for detecting a DRAM row as an aggressor row so as to refresh its neighboring victim rows. Unless stated otherwise, we conduct all experiments at 85 °C DRAM temperature.

Our approach leads to a number of new insights specific to each DRAM vendor. Table 1 summarizes our key findings regarding the TRR implementations of the 45 modules we test. We describe the experiments that lead us to these insights in more detail.

6.1 Vendor A

Using `U-TRR`, we find that vendor A uses two slightly different TRR implementations in their modules as we show in Table 1. We explain how to use `U-TRR` to understand the operation of the A_{TRR1} mechanism but our methodology is also applicable to A_{TRR2}.

6.1.1 TRR-capable `REF` Commands.

We first run an experiment to determine whether all of the `REF` commands issued to DRAM can perform TRR-induced refresh in addition to the regular refresh operations or only certain `REF` commands are responsible for TRR-induced refresh.

To uncover TRR-capable `REF` commands, we perform experiments that follow the general template that we present in Fig. 7. We use *RS* to find N row groups that match the `R-R` layout. Among the profiled rows in each row group, we designate an aggressor row, which we hammer H times. We choose H so that it does not

[6] Other works [46, 51, 65, 89, 96] also propose various methods for reverse engineering DRAM physical layout and their methods can also be used for our purposes.

Table 1: Summary of our key observations and results on TRR implementations of 45 DDR4 DRAM modules.

Module	Date (yy-ww)	Chip Density (Gbit)	Organization			HC_{first}†	Our Key TRR Observations and Results							
			Ranks	Banks	Pins		Version	Aggressor Detection	Aggressor Capacity	Per-Bank TRR	TRR-to-REF Ratio	Neighbors Refreshed	% Vulnerable DRAM Rows†	Max. Bit Flips per Row per Hammer†
A0	19-50	8	1	16	8	16K	A_{TRR1}	Counter-based	16	✓	1/9	4	73.3%	1.16
A1-5	19-36	8	1	8	16	13K-15K	A_{TRR1}	Counter-based	16	✓	1/9	4	99.2% - 99.4%	2.32 - 4.73
A6-7	19-45	8	1	8	16	13K-15K	A_{TRR1}	Counter-based	16	✓	1/9	4	99.3% - 99.4%	2.12 - 3.86
A8-9	20-07	8	1	16	8	12K-14K	A_{TRR1}	Counter-based	16	✓	1/9	4	74.6% - 75.0%	1.96 - 2.96
A10-12	19-51	8	1	16	8	12K-13K	A_{TRR1}	Counter-based	16	✓	1/9	4	74.6% - 75.0%	1.48 - 2.86
A13-14	20-31	8	1	8	16	11K-14K	A_{TRR2}	Counter-based	16	✓	1/9	2	94.3% - 98.6%	1.53 - 2.78
B0	18-22	4	1	16	8	44K	B_{TRR1}	Sampling-based	1	✗	1/4	2	99.9%	2.13
B1-4	20-17	4	1	16	8	159K-192K	B_{TRR1}	Sampling-based	1	✗	1/4	2	23.3% - 51.2%	0.06 - 0.11
B5-6	16-48	4	1	16	8	44K-50K	B_{TRR1}	Sampling-based	1	✗	1/4	2	99.9%	1.85 - 2.03
B7	19-06	8	2	16	8	20K	B_{TRR1}	Sampling-based	1	✗	1/4	2	99.9%	31.14
B8	18-03	4	1	16	8	43K	B_{TRR1}	Sampling-based	1	✗	1/4	2	99.9%	2.57
B9-12	19-48	8	1	16	8	42K-65K	B_{TRR2}	Sampling-based	1	✗	1/9	2	36.3% - 38.9%	16.83 - 24.26
B13-14	20-08	4	1	16	8	11K-14K	B_{TRR3}	Sampling-based	1	✓	1/2	4	99.9%	16.20 - 18.12
C0-3	16-48	4	1	16	x8	137K-194K	C_{TRR1}	Mix	Unknown	✓	1/17	2	1.0% - 23.2%	0.05 - 0.15
C4-6	17-12	8	1	16	x8	130K-150K	C_{TRR1}	Mix	Unknown	✓	1/17	2	7.8% - 12.0%	0.06 - 0.08
C7-8	20-31	8	1	8	x16	40K-44K	C_{TRR1}	Mix	Unknown	✓	1/17	2	39.8% - 41.8%	9.66 - 14.56
C9-11	20-31	8	1	8	x16	42K-53K	C_{TRR2}	Mix	Unknown	✓	1/9	2	99.7%	9.30 - 32.04
C12-14	20-46	16	1	8	x16	6K-7K	C_{TRR3}	Mix	Unknown	✓	1/8	2	99.9%	4.91 - 12.64

†We report the minimum and maximum HC_{first}, *% Vulnerable DRAM Rows*, and *Max. Bit Flips per Row per Hammer* for table rows containing *multiple* DRAM modules.
HC_{first}: Minimum activation count per aggressor row in double-sided RowHammer to cause a bit flip. | *Version*: Unique identifier for different TRR implementations we observe across DRAM vendors.
Aggressor Detection: Main method used to detect an aggressor row. | *Aggressor Capacity*: Maximum number of potential aggressor rows TRR can track.
Per-Bank TRR: Indicates whether a TRR mechanism operates independently in each bank or is shared across banks. | *TRR-to-REF Ratio*: Fraction of TRR-capable REFs out of all REFs.
Neighbors Refreshed: Number of neighboring victim rows refreshed by a TRR-induced refresh. | *% Vulnerable DRAM Rows*: Fraction of DRAM rows vulnerable to our custom access patterns.
Max. Bit Flips per Row per Hammer: Maximum number of bit flips observed in any victim row per each hammer to an aggressor row between two REFs.

cause RowHammer bit flips on the profiled rows but at the same time is large enough to potentially trigger the TRR mechanism to consider the hammered row as a potential aggressor. To verify that H hammers do not cause RowHammer bit flips, we simply run a separate experiment where we 1) initialize the profiled rows, 2) immediately after initialization, we hammer the profiled rows H times each, without performing any refresh, and 3) read back the profiled rows and verify that there are no bit flips.

We issue only one REF command to individually analyze each refresh operation. Without a TRR mechanism, we expect to see retention failures in *all* of the profiled rows in *almost every* iteration of the experiment. This is because each row is not refreshed for a long enough period of time (i.e., for T as in Fig. 4) such that retention failures occur. Retention failures may not be observed during the *very few* iterations that a regular refresh operation refreshes one or more of the profiled rows. Since regular refreshes happen periodically (i.e., a row is refreshes by a regular refresh at a fixed REF command interval (§6.1.3)), U-TRR easily determines when a row is refreshed by a regular refresh. When we observe a profiled row with no bit flips when regular refresh is *not* expected, we attribute that to a TRR-induced refresh operation.

When we run the experiment in Fig. 7 with $N \geq 16$ and $H = 5K$, we find an interesting pattern where we see a row group with no bit flips *only* in every 9^{th} iteration of the experiment, i.e., for every 9^{th} REF command issued consecutively. This shows that, for this particular TRR design, not all REF commands perform a TRR-induced refresh but only every 9^{th} of them have this capability.

Vendor A | Observation 1. *Every 9^{th} REF command performs a TRR-induced refresh.*

We also find with this experiment that, when TRR detects an aggressor row, it simultaneously refreshes *both* victim rows on each side of the detected aggressor row with a single REF. To check if the refreshes are limited to these two rows, we repeat the experiment using three profiled rows on each side of the row that we hammer (i.e., we use row group layout RRR-RRR). We observe that the TRR mechanism refreshes *four* of the victim rows closest to the detected aggressor row, i.e., two victims on each side of the aggressor. This is likely done to protect against the probability that RowHammer bit flips can occur in victim rows that are two rows apart from the aggressor rows, as demonstrated by prior works [54, 56, 130, 131].

Vendor A | Observation 2. *TRR refreshes four rows that are physically closest to the detected aggressor row. When row address A is detected as an aggressor, TRR refreshes rows $A \mp 1$ and $A \mp 2$.*

We next perform a slightly different experiment to understand in what sequence TRR detects the hammered rows as aggressor rows in consecutive TRR-capable REF commands. We use two R-R row groups and hammer the aggressor rows H_0 and H_1 times, where $H_0 << H_1$ (e.g., typical values we use are $H_0 = 50$ and $H_1 = 5K$). This experiment uncovers that there are two different types of TRR-induced refresh operations that alternate on every 9^{th} REF. These two TRR-induced refresh operations differ in how they detect an aggressor row to refresh its neighbors. The first type ($TREF_a$) always detects the row that has accumulated the most hammers since the time TRR previously detected the same row (e.g., the row that we hammer H_1 times in this experiment). This suggests that this particular TRR mechanism might use a counter table to keep track of activation counts of the accessed DRAM rows. The second type ($TREF_b$) detects the same row periodically every 16^{th} instance of $TREF_b$. We anticipate that $TREF_b$ uses a pointer that refers to an entry in the counter table that has 16 entries. $TREF_b$ refreshes the neighbor rows of the row address associated with the table entry that the pointer refers to. After performing a TRR-induced refresh, $TREF_b$ increments the pointer to refer to the next entry in the table. This is our hypothesis as to why $TREF_b$ repeatedly detects the same row once every 16 instances of $TREF_b$, and detects other activated rows that are in the counter table during other instances of $TREF_b$. In §6.1.2, we uncover the exact reason why we see the neighbors of the same row refreshed every 16^{th} $TREF_b$ operation.

Vendor A | Observation 3. *The TRR mechanism performs two types of TRR-induced refresh operations ($TREF_a$ and $TREF_b$) that both use a 16-entry counter table to detect aggressor rows.*
$TREF_a$: Detects the row that corresponds to the table entry with the

highest counter value.
$TREF_b$: Traverses the counter table by detecting a row that corresponds to one table entry at each of its instances.

6.1.2 Counter-based TRR. Observation 3 indicates that the TRR mechanism is capable of determining which single row is activated (i.e., hammered) more than the others. This suggests that the TRR mechanism implements a set of counters it associates with the accessed rows and increments the corresponding counter upon a DRAM row activation. We perform a set of experiments using the U-TRR methodology to understand more about this counter-based TRR implementation we hypothesize about.

To find the maximum number of rows that the TRR mechanism can keep track of, we perform an experiment where we use N R-R row groups and hammer the rows between the profiled rows H times each in cascaded hammering mode (§5.2). We use $H = 1K$ and vary N in the range $1 \leq N \leq 32$. When we repeatedly run the experiment, we observe that all profiled rows are eventually refreshed by $TREF_a$ or $TREF_b$ when $1 \leq N \leq 16$. However, with $N \geq 16$, we start observing profiled rows that are never refreshed (except when they are refreshed due to regular refresh operations as we discuss in §6.1.3). Thus, we infer that this particular TRR mechanism has a counter table capacity for 16 different row addresses. We also observe that the $TREF_b$ operations detect rows by repeatedly iterating over the counter table entries, such that a $TREF_b$ detects a row associated with one entry and the next $TREF_b$ detects the row associated with the next entry. We find this to be the reason for why every 16^{th} $TREF_b$ detects the same aggressor row when the same set of aggressor rows are repeatedly hammered. Using row groups from different banks, we uncover that the TRR mechanism keeps track of 16 different rows in *each* bank, suggesting that each bank implements a separate counter table.

Vendor A | Observation 4. *The TRR mechanism counts how many times DRAM rows are activated using a per-bank counter table, which can keep track of activation counts to 16 different row addresses.*

We next try to find how TRR decides which row to evict from the counter table when a new row is to be inserted. We perform an experiment where we check if the TRR mechanism evicts the entry with the smallest counter value from the counter table. In the experiment, we use 17 R-R row groups and hammer the row between the two retention-profiled rows in each group. We hammer the aggressors in the following order. First, we hammer one of the aggressors H_0 times. Next, we hammer the remaining 16 aggressors H_1 times, where $H_0 < H_1$ (e.g., $H_0 = 50$ and $H_1 = 100$). Even after running the experiment for thousands of iterations, we observe that the TRR mechanism never identifies the row that is hammered H_0 times as an aggressor row. This indicates that the counter table entry with the smallest counter value is evicted from the table upon inserting a new row address (i.e., the last row that we hammer H_1 times in this case) into the table.

Vendor A | Observation 5. *When inserting a new row into the counter table, TRR evicts the row with the smallest counter value.*

We have already observed that a DRAM row activation increments the corresponding counter in the table. However, we do not yet know whether or not a TRR-induced refresh operation updates the corresponding counter value of the detected aggressor row (e.g., resets the counter to 0). To check if a TRR-induced refresh updates the corresponding counter, we conduct another experiment with two R-R row groups, where we hammer the two aggressors H_0 and H_1 times. When we run the experiment multiple times with $H_0 < H_1$, we notice that $TREF_a$ detects an aggressor row based on how many hammers the aggressor row accumulated since the last time it is detected by $TREF_a$ or $TREF_b$. For example, with $H_0 = 2K$ and $H_1 = 3K$, the corresponding counters accumulate $36K$ and $54K$ hammers, respectively, assuming the 18^{th} REF performs $TREF_a$.[7] Thus, $TREF_a$ detects the aggressor row that is hammered $54K$ times and resets the corresponding counter. Until the subsequent $TREF_a$ operation, the two counters reach $72K$ and $54K$ hammers, respectively, and $TREF_a$ detects the first aggressor row as its counter value is higher than that of the second aggressor row since the latter counter was reset earlier. This experiment shows that a TRR-induced refresh operation resets the counter that corresponds to the aggressor row detected to refresh the neighbors of.

Vendor A | Observation 6. *When TRR detects an aggressor row, TRR resets the counter corresponding to the detected row to zero.*

We next question whether, once inserted, a row address remains indefinitely in the counter table or TRR periodically clears out the counter table. To answer this question, we run *only once* an experiment with one R-R row group and hammer the row between the profiled rows several times to insert the aggressor rows into the counter table. Then, we repeat the experiment many times *without* hammering the aggressor row. After running these experiments, we observe that the aggressor row is detected by a $TREF_a$ operation only once. This is expected since we do not access the row except in the first experiment, and once reset by the first $TREF_a$, the corresponding counter value remains reset and never becomes a target for $TREF_a$ again. However, we observe that every 16^{th} $TREF_b$ detects the same aggressor row and refreshes its neighbors. We keep observing the same even after repeating the experiment $32K$ times (i.e., issuing $32K$ REF commands that equal the number of refreshes issued within four 64 ms nominal refresh periods). This shows that the aggressor row remains in the counter table and keeps getting periodically detected by $TREF_b$ operations. The TRR mechanism does *not* seem to periodically clear the counter table, for example, based on time or the number of issued REF commands.

Vendor A | Observation 7. *After an entry corresponding to a row is inserted into the counter table, the entry remains in the table indefinitely until it is evicted due to insertion of a different row.*

6.1.3 Analyzing Regular Refresh. To refresh every DRAM cell at the default 64 ms period, the memory controller issues a REF command once every 7.8 µs according to the DDR4 specification [41, 77, 116]. In total, the memory controller issues $\approx 8K$ (64 ms/7.8 µs) REF commands every 64 ms. Therefore, it is expected that $\approx 8K$ REF commands refresh each row in the DRAM chip once to prevent a row from leaking charge for more than 64 ms. In our experiments, we observe that the DRAM chips of vendor A internally refresh more rows with each REF such that a row receives a regular refresh once every 3758 (instead of $\approx 8K$) REF commands. Thus, the DRAM chip internally refreshes its rows with a period even smaller than 32 ms instead of the specified 64 ms. We suspect this could be an additional measure vendor A takes 1) to protect against RowHammer [56] or 2) in response to the decreasing retention time as DRAM technology node size becomes smaller [12, 73, 81, 84].

[7] Since $TREF_a$ and $TREF_b$ happen every 9^{th} REF in an interleaved manner, $TREF_a$ happens every 18^{th} REF.

Vendor A | Observation 8. *Periodic DRAM refresh leads to internally refreshing the DRAM chip with a period smaller than half of the specified* 64 ms *refresh period.*

In §7, we exploit the insights we present in this section to craft a new DRAM access pattern that effectively circumvents the protection of the TRR mechanism. This new custom access pattern induces a significantly higher number of RowHammer bit flips than the state-of-the-art access patterns presented in [24].

6.2 Vendor B

Using U-TRR, we find that vendor B uses three slightly different TRR implementations in their modules, as Table 1 shows. We explain how to use U-TRR to understand the operation of the B_{TRR1} mechanism. Our methodology is also applicable to B_{TRR2} and B_{TRR3}.

6.2.1 TRR-capable REF commands. Similar to vendor A, we again start with uncovering which REF commands can perform TRR-induced refresh. When we repeatedly run the experiment with one or more row groups (i.e., $N \geq 16$) while hammering each aggressor row $5K$ times in each iteration of the experiment, we observe that not all REF commands perform TRR-induced refresh. Instead, we find that, in B_{TRR1} only every 4^{th} REF command is used for TRR-induced refresh. Similar experiments on modules that implement B_{TRR2} and B_{TRR3} uncover that every 9^{th} and 2^{nd} REF command, respectively, is used for TRR-induced refresh.

Vendor B | Observation 1. *Every 4^{th}, 9^{th}, and 2^{nd} REF command performs a TRR-induced refresh in the three TRR mechanisms of vendor B.*

From the same experiment, we also observe that a TRR-induced refresh operation refreshes only the two neighboring rows that are immediately adjacent to the hammered row as opposed to vendor A's TRR implementation, which refreshes the four physically closest rows to the hammered row.

Vendor B | Observation 2. *The TRR mechanism refreshes the two rows physically closest to the detected aggressor row. When row address A is detected as an aggressor, TRR refreshes rows $A \mp 1$.*

6.2.2 Sampling-based TRR. We perform experiments to show how the TRR mechanism detects the potential aggressor rows. When we perform the experiments that we use for vendor A's modules (described in §6.1.2), we do *not* observe obvious patterns in the rows detected by TRR so as to indicate a counter-based TRR implementation. Instead, we observe that the aggressor row that is last hammered before a REF command is more likely to be detected. In particular, when we hammer two aggressor rows $H_0 = 5K$ and $H_1 = 3K$ times, respectively, we find that the 4^{th} REF *always* refreshes the neighbors of the second aggressor row, which we hammer $2K$ times less than the first aggressor row. We perform experiments with different H_0 and H_1 values and find that, when we hammer the second row at least $2K$ times and issue a REF, the TRR mechanism consistently refreshes the neighbors of the second row on every 4^{th} REF. However, as we reduce H_1, the first aggressor row gets detected by TRR with an increasing probability.

With further analysis, we determine that B_{TRR1} operates by sampling the row addresses provided along with ACT commands. This sampling of ACT commands happens with a certain probability such that $2K$ consecutive activations to a particular row consistently causes the row to be detected for TRR-induced refresh. We did not analyze this aspect of TRR further; we suspect (based on some experiments) that the sampling does not happen truly randomly but is likely based on pseudo-random sampling of an incoming ACT.

Vendor B | Observation 3. *TRR probabilistically detects aggressor rows by sampling row addresses of* ACT *commands.*

To determine how many rows the TRR mechanism can sample and refresh at the same time, we repeat the previous experiment with the same $H_0 = 5K$ and $H_1 = 3K$ hammer counts but by issuing M REF commands, instead of just one, after performing the hammers. Even when we use a large M (e.g., to 100) such that it contains multiple TRR-capable REF commands (e.g., 25), we *never* see the neighbors of the first aggressor row to be refreshed but *always* the neighbors of the *second* aggressor row. This suggests that, a newly-sampled row overwrites the previously-sampled one. Therefore, we conclude that the TRR mechanism has a capacity to sample *only* one row address. Further, we find that this sampling capacity is shared across *all banks* in a DRAM chip that implements B_{TRR1} and B_{TRR2}. When TRR samples row R_1 from DRAM bank B_1, it overwrites a previously-sampled row R_2 from bank B_0 even though R_2's neighbors may not have been refreshed yet.

Vendor B | Observation 4. *The TRR mechanism has a sampling capacity of only one row that is shared across all banks in a DRAM chip (except for B_{TRR3}).*

Our experiments also uncover that a previously-sampled row address is *not* cleared when the TRR mechanism performs a TRR-induced refresh on the neighbors of this aggressor row. Instead, when a new TRR-enabled REF is issued, TRR refreshes (again) the neighbors of the same row.

Vendor B | Observation 5. *A TRR-induced refresh does not clear the sampled row, and therefore the same row keeps getting detected until TRR samples another aggressor row.*

6.3 Vendor C

Using U-TRR, we find that vendor C uses three slightly different TRR implementations in their modules, as Table 1 shows. For brevity, we omit the details of the experiments as they are largely similar to the experiments for the modules of vendors A and B (§ 6.1 and §6.2). Instead, we only describe our key observations.

We start with running experiments to find which REF commands are TRR-capable. Different from the modules of vendors A and B, we find that vendor C's modules implement a TRR mechanism that *can* perform a TRR-induced refresh during the execution of *any* REF command. The TRR mechanism performs a TRR-induced refresh once every 17 consecutive REF commands during a likely RowHammer attack. When likely not under an attack (i.e., when a small number of row activations happen), TRR can defer a TRR-induced refresh to any of the subsequent REF commands until it detects an aggressor row. We do not observe TRR-induced refresh more frequently than once in every 17 REF commands. For C_{TRR2} and C_{TRR3}, we find that every 9^{th} and 8^{th} REF command, respectively, performs a TRR-induced refresh.

Vendor C | Observation 1. *Every 17^{th}, 9^{th}, and 8^{th} REF command normally performs a TRR-induced refresh in the three TRR mechanisms of vendor C. A TRR-induced refresh can be deferred to a later* REF *if no potential aggressor row is detected.*

To uncover the logic behind how a potential aggressor row is detected, we run experiments similar to those we use for the modules

from vendors A and B. We find that vendor C's TRR mechanism detects aggressor rows only from the set of rows targeted by the first 2K ACT commands (per bank) following a TRR-induced refresh operation. We also find that 1) TRR probabilistically detects one of the rows activated within the first 2K ACT commands and 2) the rows that are activated earlier have a higher chance to be targeted by the subsequent TRR-induced refresh operation. Discovering that TRR detects aggressor rows based on only the first 2K ACT commands helped us to craft an effective access pattern (§7.1); thus we did not further analyze vendor C modules to uncover the maximum number of potential aggressor rows TRR tracks.

Vendor C | Observation 2. *TRR detects an aggressor row only among the first 2K* ACT[8] *(to each bank) following a TRR-induced refresh. Rows activated earlier are more likely to be detected by TRR.*

Our experiments uncover a unique DRAM row organization in modules $C0$-8. It appears that two consecutively addressed rows (i.e., physical row addresses R and $R+1$ where R is an even row address) are isolated in pairs such that hammering one row (e.g., R) can induce RowHammer bit flips *only* in its *pair row* (e.g., $R+1$), and not in any other row in the bank. As expected, we also observe that TRR issues refresh operations *only* to the pair row of each aggressor row that it identifies.

Vendor C | Observation 3. *Given any two rows, R and R + 1, where R is an even number, TRR refreshes* only *one of the rows (e.g., R) upon detecting the other (e.g., R + 1) as an aggressor row.*

7 Bypassing TRR Using U-TRR Observations

U-TRR uncovers critical characteristics of the TRR mechanisms different DRAM vendors implement in their chips. We leverage those characteristics to craft custom DRAM access patterns that hammer an aggressor row such that TRR cannot refresh the aggressor row's neighbors (i.e., victim rows) in a timely manner. Our results show that these new custom access patterns greatly increase RowHammer bit flips on the 45 DDR4 modules we test.

7.1 Custom RowHammer Access Patterns

Vendor A. Using U-TRR, we find that vendor A's modules implement a counter-based TRR (A_{TRRx}[9]), the details of which are in §6.1. Since A_{TRRx} evicts the entry with the lowest counter value when inserting a new entry to the table, a custom RowHammer access pattern that takes advantage of our U-TRR analysis should first hammer two aggressor rows in a double-sided manner and then evict the two aggressor rows from the table by hammering other rows (i.e., *dummy rows*) within the same bank during the remaining time until the memory controller issues a REF command.[10]

We show how we can hammer two aggressor rows (A_0 and A_1) in a double-sided manner without allowing A_{TRRx} to refresh their victim rows. First, the attacker should synchronize the memory accesses with the periodic REF commands[11] in order to hammer A_0 and A_1 *right after* the memory controller issues a REF. After hammering the two aggressor rows, the attacker should then use the remaining time until the next REF to hammer dummy rows in order to steer A_{TRRx} to identify one of the dummy rows (and *not* rows A_0 and A_1) as potential aggressors and refresh the dummy rows' neighboring victim rows. The particular access pattern that leads to the largest number of bit flips is hammering A_0 and A_1 24 times each, followed by hammering 16 dummy rows 6 times each. We discover the access pattern that maximizes the bit flip count by sweeping the number of hammers to A_0 and A_1 and adjusting the number of hammers to the 16 dummy rows based on the time that remains until the next REF after hammering the aggressors.

Vendor B. B_{TRRx} operates by probabilistically sampling a single row address from all ACT (§6.2) commands issued (across all banks for B_{TRR1} and B_{TRR2}) to DRAM. Rows neighboring the sampled row are refreshed during a TRR-induced refresh operation that happens once in every 4, 9, and 2 REF commands for B_{TRR1}, B_{TRR2}, and B_{TRR3}, respectively. To maximize the probability of B_{TRRx} detecting a dummy row instead of the aggressor row, our custom access pattern maximizes the number of hammers to dummy rows after hammering the aggressor rows and before every TRR-induced refresh operation. Our custom access pattern first hammers rows A_0 and A_1 immediately following a TRR-induced refresh. Then, it simultaneously hammers a single dummy row in each of four banks[12] to perform a large number of dummy row activations within the limited time until the next TRR-induced refresh.[13] We find that 220 hammers per aggressor row (leaving 156 hammers for each dummy row in the four banks) within a window of four consecutive REF commands causes RowHammer bit flips even in the least RowHammer-vulnerable module of the 15 vendor B modules that we use in our experiments.

Vendor C. C_{TRR1}, C_{TRR2}, and C_{TRR3} have the ability to perform a TRR-induced refresh once in every 17, 9, and 8 REF commands, respectively, and they can defer the TRR-induced refresh to a later REF until a potential aggressor row is detected (§6.3). C_{TRRx} does not keep track of more than 2K ACT commands that follow a TRR-induced refresh operation and rows activated earlier in the set of 2K ACT commands are more likely to be detected. Therefore, we craft a custom RowHammer access pattern that follows a TRR-induced refresh operation with a large number (e.g., 2K) of dummy row activations and then hammers the aggressor rows A_0 and A_1 until the next TRR-induced refresh operation. To properly execute this access pattern, it is critical to synchronize the dummy and aggressor row hammers with TRR-enabled REF commands.

7.2 Effect on RowHammer Bit Flip Count

We implement and evaluate the different custom DRAM access patterns that are used to circumvent A_{TRRx}, B_{TRRx}, and C_{TRRx} on our FPGA-based SoftMC platform [33] (§3.3). The SoftMC program executes each custom access pattern for a fixed interval of time (determined by each chip's TRR-induced refresh frequency), while also issuing REF commands once every 7.8 µs to comply with the vendor-specified default refresh rate.

[8] Except for the modules that implement C_{TRR3} (Table 1). C_{TRR3} detects an aggressor row only among the first 1K activations to each bank.

[9] We refer to all versions of TRR mechanisms that vendor A's modules implement (i.e., A_{TRR1} and A_{TRR2}) as A_{TRRx}. We use a similar terminology for other vendors.

[10] The memory controller issues a REF once every 7.8 µs when using the default 64 ms refresh period. This allows at most 149 hammers to a single DRAM bank assuming typical activation (35 ns), precharge (15 ns), and refresh (350 ns) latencies [12, 72, 77].

[11] A recent work [19] shows how to detect when a memory controller issues a periodic REF from an unprivileged process and from a web browser using JavaScript.

[12] We do not hammer a dummy row in more than four different banks due to the Four-Activation-Window (tFAW [41, 77, 116]) DRAM timing constraint.

[13] For B_{TRR3}, which separately samples ACT commands to each bank, we hammer a dummy row from the aggressor row's bank.

Fig. 8 shows the distribution of number of bit flips per row as box-and-whisker plots[14] in modules $A5$, $B8$, and $C7$[15] when sweeping the number of hammers issued to aggressor rows in each custom access pattern. The x-axis is normalized so that it shows the average number of hammers performed between two REFs to a single aggressor row.[16] We show different hammers per aggressor per REF for each module as the number of hammers we can fit between two REFs depend on the custom RowHammer patterns we craft (§7.1). Each access pattern uses a fixed number of dummy rows as described in §7.1. We perform the maximum number of hammers that fit between two REFs. Therefore, a lower number of aggressor row hammers translates to a higher number of dummy row hammers.

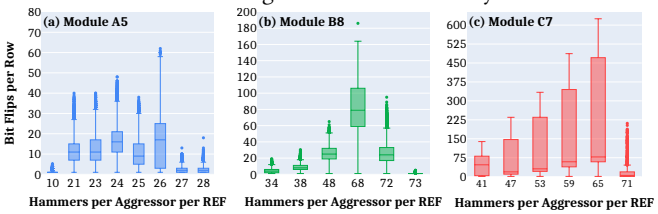

Figure 8: Distribution of bit flips per DRAM row for different aggressor hammer counts in three representative modules.

Vendor A. We observe the highest bit flip count (i.e., up to 62 bit flips in a row) when using 26 hammers per aggressor row, in which case each of the 16 dummy rows are hammered 6 times. The number of bit flips decreases as we hammer the aggressors more than 26 times. This is because the aggressors become less likely to be evicted from the counter table as more activations to them increment the corresponding counters to higher values. In contrast, the aggressor rows become more likely to be evicted from the counter table when they are hammered fewer than 26 times each. However, we still observe a smaller bit flip count with fewer than 26 hammers per aggressor because fewer hammers are insufficient for many victim rows to experience RowHammer bit flips.

Vendor B. The number of bit flips gradually increases with the number of hammers per aggressor row. This increases to a point where too many aggressor row activations leave an insufficient time to perform enough dummy row activations to ensure that a dummy row is sampled to replace an aggressor row for the subsequent TRR-induced refresh. According to our experiments, at least 12 total dummy row activations simultaneously performed in four banks (leaving enough time to perform 73 hammers per aggressor) are needed to induce RowHammer bit flips. We observe the maximum number of bit flips with 68 hammers per aggressor row.

Vendor C. We observe bit flips appear when a dummy row is initially hammered a large number of times to make the subsequent aggressor row activations less likely to be tracked by C_{TRRx}. The access pattern causes bit flips when performing at least 252 dummy hammers (right after a TRR-enabled REF) prior to continuously hammering the aggressor rows until the next TRR-enabled REF. This leaves time to perform 71 hammers per aggressor row per REF on average. We observe the maximum number of bit flips with 65 hammers per aggressor row.

7.3 Effect on Individual Rows

To mount a successful system-level RowHammer attack, it is critical to force the operating system to place sensitive data in vulnerable rows. To make this task easier, it is important to induce RowHammer bit flips in as many rows as possible. Ideally, all rows should be vulnerable to RowHammer from an attacker's perspective.

Fig. 9 shows the percentage of vulnerable DRAM rows, i.e., rows that experience at least one RowHammer bit flip with our custom RowHammer access patterns (§7.1), as a fraction of all rows in a bank of the tested 45 modules. We report data for a single bank[17] from each module. For each DRAM module, we use a different number of hammers per aggressor that results in the highest percentage of vulnerable rows in the corresponding module (see §7.2).[18]

In many (i.e., 8, 7, and 6, respectively) modules from vendor A, B, and C we see bit flips in more than 99.9% of the rows. This shows that the custom access patterns we use are effective at circumventing the A_{TRRx}, B_{TRRx}, and C_{TRRx} implementations. The other modules from vendors A and B have a smaller yet still a very significant (i.e., >23% in all cases) fraction of vulnerable rows. We believe that modules $A0$, $A8$-12 are slightly more resistant to our access pattern than the other modules of vendor A due to having more banks (i.e., 16 vs. 8) and smaller banks (i.e., $32K$ vs. $64K$ rows per bank). $B1$-4 have stronger rows that can endure more hammers than the other modules of vendor B (as shown in HC_{first} column of Table 1), and therefore they have a lower fraction of vulnerable rows. $B9$-12 implement a different TRR version (B_{TRR2}), for which our custom access patterns are not as effective.

Modules from vendor C that implement C_{TRR1} (i.e., $C0$-8) are less vulnerable to our access patterns than the other modules of the same vendor. We believe this is due to two main reasons. First, these modules use a unique row organization that pairs every two consecutive DRAM rows, as we explain in §6.3. We only observe bit flips when hammering two aggressor rows that have odd-numbered addresses but not when the two aggressor have even-numbered addresses. This essentially halves the number of victim rows where our access patterns can cause bit flips. Second, $C0$-6 have stronger rows that are less vulnerable to RowHammer than the other vendor C modules (as Table 1 shows), and therefore $C0$-6 have even lower fraction of vulnerable rows than $C7$-8.

Overall, even though our custom RowHammer access patterns cause bit flips in 45 DRAM modules, we could not explore the entire space of both TRR implementations and custom RowHammer patterns. Therefore, we believe future work can lead to even better RowHammer patterns via more exhaustive analysis and testing.

7.4 Bypassing System-Level ECC Using U-TRR

Although we clearly show that the custom access patterns we craft induce RowHammer bit flips in a very large fraction of DRAM rows

[14] The lower and upper bounds of the box represent the first quartile (i.e., the median of the first half of a sorted dataset) and the second quartile (i.e., the median of the second half of a sorted dataset), respectively. The median line is within the box. The size of the box represents the inter-quartile range (IQR). The whiskers are placed at $1.5 * IQR$ on both sides of the box. The outliers are represented with dots.

[15] We analyze $A5$, $B8$, and $C7$ as they are the modules that experience the most RowHammer bit flips and implement A_{TRR1}, B_{TRR1}, and C_{TRR1}, respectively.

[16] The x-axis shows the number of hammers per aggressor row per REF to enable easy comparison of the effectiveness of different RowHammer patterns across different modules. Our actual experiments perform the aggressor and dummy row hammers as required by each custom RowHammer access pattern described in §7.1.

[17] We test a single bank to reduce the experiment time. To ensure that the results are similar across different banks, we tested multiple banks from several modules.

[18] When using the conventional single- and double-sided RowHammer, we do *not* observe RowHammer bit flips in any of the 45 DDR4 modules (as expected from our understanding of the TRR implementations and from [24]).

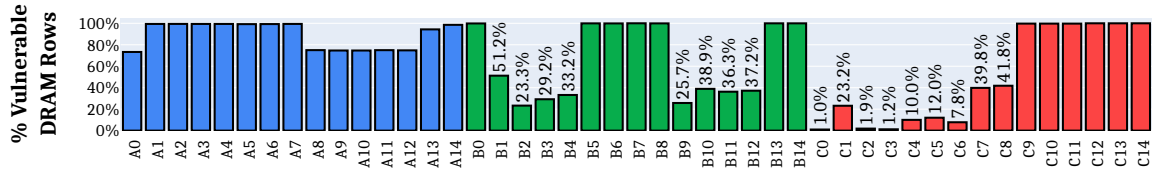

Figure 9: Percentage of rows that experience at least one RowHammer bit flip using our custom RowHammer access patterns.

(§7.3), a system that uses Error Correction Codes (ECC) [20, 26, 30, 47, 70, 76, 79, 86, 93, 108, 119, 120] can potentially protect against RowHammer bit flips if those bit flips are distributed such that no ECC codeword contains more bit flips than ECC can correct.

Fig. 10 shows the distribution of RowHammer bit flips that our custom access patterns induce across 8-byte data chunks as box-and-whisker plots[14] for *all* 45 DRAM modules we test across three vendors. We use 8-byte data chunks as DRAM ECC typically uses 8-byte or larger datawords [10, 37, 43, 60, 61, 79, 87, 118].

The majority of the 8-byte data chunks are those that have only a single RowHammer bit flip (i.e., up to 6.9 million 8-byte data chunks with a single bit flip in one bank of module *B*13), which can be corrected using typical SECDED ECC [10, 37, 43, 60, 61, 79, 87, 118]. However, our RowHammer access patterns can cause at least 3 (up to 7) bit flips in many single datawords, which the SECDED ECC *cannot* correct or detect, in all three vendor's modules.

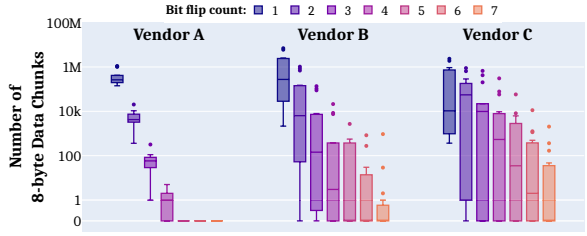

Figure 10: Distribution of 8-byte data chunks (log scale) with different RowHammer bit flip counts in a single DRAM bank from each of the 45 tested DDR4 modules.

Chipkill [2, 20, 86] is a symbol-based code conventionally designed to correct errors in one symbol (i.e., one DRAM chip failure) and detect errors in two symbols (i.e., two DRAM chip failures). Because our access patterns cause more than two bit flips in *arbitrary* locations (i.e., different DRAM chips), and thus in arbitrary symbols within an 8-byte data chunk, Chipkill does *not* provide guaranteed protection. Reed-Solomon codes [101] can be designed to provide stronger correction/detection capability at the cost of additional parity- check symbols [36, 70]. To detect (and correct half of) the maximum number of bit flips (i.e., 7) that our access patterns can cause in an 8-byte data chunk, a Reed-Solomon code would incur a large overhead by requiring at least 7 parity-check symbols [101].

We conclude that 1) conventional DRAM ECC *cannot* protect against our new custom RowHammer patterns and 2) an ECC scheme that can protect against our custom patterns requires a large number of parity-check symbols, i.e., large overheads.

8 Related Work

Kim et al. [56] are the first to introduce and analyze the RowHammer phenomenon. Numerous later works develop RowHammer attacks to compromise various systems in various ways [1, 7, 8, 15, 16, 19, 23, 24, 28, 29, 34, 38, 44, 54, 62, 71, 82, 83, 96, 98, 100, 104, 109, 122–124, 128, 129, 136, 140] and analyze RowHammer further [15, 16, 28, 54, 89, 97, 98, 122, 126, 135]. To our knowledge, this is the first work to 1) propose an experimental methodology to understand the inner workings of commonly-implemented in-DRAM RowHammer protection (i.e., TRR) mechanisms and 2) use this understanding to create custom access patterns that circumvent the TRR mechanisms of modern DDR4 DRAM chips.

In-DRAM TRR. We already provided extensive descriptions of TRR and TRRespass in §1, §2.4, and §6. TRRespass [24] is the most relevant prior work to ours in understanding and circumventing TRR mechanisms, yet it is not effective enough. While TRRespass can incur RowHammer bit flips in 13 of 42 DDR4 modules (and 5 of 13 LPDDR4 modules), TRRespass does not uncover many implementation details of the TRR mechanisms, which are important to circumvent TRR mechanisms. For example, in 29 out of 42 DDR4 modules (and 8 out of 13 LPDDR4 modules), TRRespass fails to find an access pattern that can circumvent TRR. In contrast, our new U-TRR methodology can be used to understand different aspects of a TRR mechanism in great detail and use this understanding to generate specific RowHammer access patterns that effectively incur a large number of bit flips (as we show on 45 real DRAM modules).

System-level RowHammer Mitigation Techniques. A number of studies propose system-level RowHammer mitigation techniques [4, 5, 9, 22, 27, 55, 56, 59, 68, 91, 115, 117, 121, 124, 130, 131, 137]. Recent works [23, 28, 54, 130] show that some of these mechanisms are insecure, inefficient, or do not scale well in chips with higher vulnerability to RowHammer. We believe the fundamental principles of U-TRR can be useful for improving the security of these works as well as potentially combining them with in-DRAM TRR. We leave examining such directions to future work.

9 Conclusion

We propose U-TRR, a novel experimental methodology for reverse-engineering the main RowHammer mitigation mechanism, Target Row Refresh (TRR), implemented in modern DRAM chips. Using U-TRR, we 1) provide insights into the inner workings of existing proprietary and undocumented TRR mechanisms and 2) develop custom DRAM access patterns to efficiently circumvent TRR in 45 DDR4 DRAM modules from three major vendors. We conclude that TRR does *not* provide security against RowHammer and can be easily circumvented using the new understanding provided by U-TRR. We believe and hope that U-TRR will facilitate future research by enabling rigorous and open analysis of RowHammer mitigation mechanisms, leading to the development of both new RowHammer attacks and more secure RowHammer protection mechanisms.

Acknowledgments

We thank the anonymous reviewers of MICRO 2021 for feedback. We thank the SAFARI Research Group members for valuable feedback and the stimulating intellectual environment they provide. We acknowledge the generous gifts provided by our industrial partners, especially Google, Huawei, Intel, Microsoft, and VMware. This work was also supported in part by the Netherlands Organisation for Scientific Research through grant NWO 016.Veni.192.262.

References

[1] M. T. Aga, Z. B. Aweke, and T. Austin, "When Good Protections Go Bad: Exploiting Anti-DoS Measures to Accelerate Rowhammer Attacks," in *HOST*, 2017.

[2] AMD, "BKDG for AMD NPT Family 0Fh Processors," 2009.

[3] R. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley & Sons, 2020.

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

[5] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. Oren, and T. Austin, "ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks," in *ASPLOS*, 2016.

[6] I. Bhati, Z. Chishti, S.-L. Lu, and B. Jacob, "Flexible Auto-Refresh: Enabling Scalable and Energy-Efficient DRAM Refresh Reductions," in *ISCA*, 2015.

[7] S. Bhattacharya and D. Mukhopadhyay, "Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis," in *CHES*, 2016.

[8] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida, "Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector," in *S&P*, 2016.

[9] F. Brasser, L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "Can't Touch This: Practical and Generic Software-only Defenses Against RowHammer Attacks," *USENIX Security*, 2017.

[10] S. Cha, O. Seongil, H. Shin, S. Hwang, K. Park, S. J. Jang, J. S. Choi, G. Y. Jin, Y. H. Son, H. Cho, J. H. Ahn, and N. S. Kim, "Defect Analysis and Cost-Effective Resilience Architecture for Future DRAM Devices," in *HPCA*, 2017.

[11] K. K. Chang, A. Kashyap, H. Hassan, S. Ghose, K. Hsieh, D. Lee, T. Li, G. Pekhimenko, S. Khan, and O. Mutlu, "Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization," in *SIGMETRICS*, 2016.

[12] K. K. Chang, D. Lee, Z. Chishti, A. R. Alameldeen, C. Wilkerson, Y. Kim, and O. Mutlu, "Improving DRAM Performance by Parallelizing Refreshes with Accesses," in *HPCA*, 2014.

[13] K. K. Chang, P. J. Nair, D. Lee, S. Ghose, M. K. Qureshi, and O. Mutlu, "Low-Cost Inter-Linked Subarrays (LISA): Enabling Fast Inter-Subarray Data Movement in DRAM," in *HPCA*, 2016.

[14] K. K. Chang, A. G. Yağlikçi, S. Ghose, A. Agrawal, N. Chatterjee, A. Kashyap, D. Lee, M. O'Connor, H. Hassan, and O. Mutlu, "Understanding Reduced-Voltage Operation in Modern DRAM Devices: Experimental Characterization, Analysis, and Mechanisms," in *SIGMETRICS*, 2017.

[15] L. Cojocar, J. Kim, M. Patel, L. Tsai, S. Saroiu, A. Wolman, and O. Mutlu, "Are We Susceptible to Rowhammer? An End-to-End Methodology for Cloud Providers," *S&P*, 2020.

[16] L. Cojocar, K. Razavi, C. Giuffrida, and H. Bos, "Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against RowHammer Attacks," in *S&P*, 2019.

[17] Z. Cui, S. A. McKee, Z. Zha, Y. Bao, and M. Chen, "DTail: A Flexible Approach to DRAM Refresh Management," in *SC*, 2014.

[18] A. Das, H. Hassan, and O. Mutlu, "VRL-DRAM: Improving DRAM Performance via Variable Refresh Latency," in *DAC*, 2018.

[19] F. de Ridder, P. Frigo, E. Vannacci, H. Bos, C. Giuffrida, and K. Razavi, "SMASH: Synchronized Many-sided Rowhammer Attacks from JavaScript," in *USENIX Security*, 2021.

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

[21] R. H. Dennard, "Field-Effect Transistor Memory," 1968, US Patent 3,387,286.

[22] F. Devaux and R. Ayrignac, "Method and Circuit for Protecting a DRAM Memory Device from the Row Hammer Effect," Jan. 5 2021, US Patent 10,885,966.

[23] P. Frigo, C. Giuffrida, H. Bos, and K. Razavi, "Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU," in *S&P*, 2018.

[24] P. Frigo, E. Vannacci, H. Hassan, V. van der Veen, O. Mutlu, C. Giuffrida, H. Bos, and K. Razavi, "TRRespass: Exploiting the Many Sides of Target Row Refresh," in *S&P*, 2020.

[25] S. Gautam, S. Manhas, A. Kumar, M. Pakala, and E. Yieh, "Row Hammering Mitigation Using Metal Nanowire in Saddle Fin DRAM," *TED*, 2019.

[26] S.-L. Gong, J. Kim, and M. Erez, "DRAM Scaling Error Evaluation Model Using Various Retention Time," in *DSN-W*, 2017.

[27] Z. Greenfield and L. Tomer, "Throttling Support for Row-Hammer Counters," 2016, US Patent 9,251,885.

[28] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom, "Another Flip in the Wall of RowHammer Defenses," in *S&P*, 2018.

[29] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A Remote Software-Induced Fault Attack in Javascript," in *DIMVA*, 2016.

[30] R. W. Hamming, "Error Detecting and Error Correcting Codes," in *Bell Labs Technical Journal*, 1950.

[31] H. Hassan, M. Patel, J. S. Kim, A. G. Yağlıkçı, N. Vijaykumar, N. M. Ghiasi, S. Ghose, and O. Mutlu, "CROW: A Low-Cost Substrate for Improving DRAM Performance, Energy Efficiency, and Reliability," in *ISCA*, 2019.

[32] H. Hassan, G. Pekhimenko, N. Vijaykumar, V. Seshadri, D. Lee, O. Ergin, and O. Mutlu, "ChargeCache: Reducing DRAM Latency by Exploiting Row Access Locality," in *HPCA*, 2016.

[33] H. Hassan, N. Vijaykumar, S. Khan, S. Ghose, K. Chang, G. Pekhimenko, D. Lee, O. Ergin, and O. Mutlu, "SoftMC: A Flexible and Practical Open-Source Infrastructure for Enabling Experimental DRAM Studies," in *HPCA*, 2017.

[34] S. Hong, P. Frigo, Y. Kaya, C. Giuffrida, and T. Dumitraş, "Terminal Brain Damage: Exposing the Graceless Degradation in Deep Neural Networks under Hardware Fault Attacks," in *USENIX Security*, 2019.

[35] M. Horiguchi and K. Itoh, *Nanoscale Memory Repair*. Springer SBM, 2011.

[36] W. C. Huffman and V. Pless, *Fundamentals of Error-Correcting Codes*. Cambridge University Press, 2003.

[37] Intelligent Memory, "IM ECC DRAM with Integrated Error Correcting Code," 2016, Product Brief.

[38] Y. Jang, J. Lee, S. Lee, and T. Kim, "SGX-Bomb: Locking Down the Processor via Rowhammer Attack," in *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, 2017.

[39] JEDEC, "Double Data Rate 3 (DDR3) SDRAM Specification," 2012.

[40] JEDEC, "Low Power Double Data Rate 4 (LPDDR4) SDRAM Specification," 2014.

[41] JEDEC, "DDR4 SDRAM Standard - JESD79-4C," 2020.

[42] JEDEC, "DDR5 SDRAM - JESD79-5," 2020.

[43] S. Jeong, S. Kang, and J.-S. Yang, "PAIR: Pin-aligned In-DRAM ECC Architecture using Expandability of Reed-Solomon Code," in *DAC*, 2020.

[44] S. Ji, Y. Ko, S. Oh, and J. Kim, "Pinpoint Rowhammer: Suppressing Unwanted Bit Flips on Rowhammer Attacks," in *ASIACCS*, 2019.

[45] Y. Jiang, H. Zhu, D. Sullivan, X. Guo, X. Zhang, and Y. Jin, "Quantifying Rowhammer Vulnerability for DRAM Security." DAC, 2021.

[46] M. Jung, C. C. Rheinländer, C. Weis, and N. Wehn, "Reverse Engineering of DRAMs: Row Hammer with Crosshair," in *MEMSYS*, 2016.

[47] U. Kang, H. S. Yu, C. Park, H. Zheng, J. Halbert, K. Bains, S. Jang, and J. S. Choi, "Co-Architecting Controllers and DRAM to Enhance DRAM Process Scaling," in *The Memory Forum*, 2014.

[48] S. Khan, D. Lee, Y. Kim, A. R. Alameldeen, C. Wilkerson, and O. Mutlu, "The Efficacy of Error Mitigation Techniques for DRAM Retention Failures: A Comparative Experimental Study," in *SIGMETRICS*, 2014.

[49] S. Khan, D. Lee, and O. Mutlu, "PARBOR: An Efficient System-Level Technique to Detect Data-Dependent Failures in DRAM," in *DSN*, 2016.

[50] S. Khan, C. Wilkerson, Z. Wang, A. R. Alameldeen, D. Lee, and O. Mutlu, "Detecting and Mitigating Data-Dependent DRAM Failures by Exploiting Current Memory Content," in *MICRO*, 2017.

[51] J. S. Kim, M. Patel, H. Hassan, and O. Mutlu, "Solar-DRAM: Reducing DRAM Access Latency by Exploiting the Variation in Local Bitlines," in *ICCD*, 2018.

[52] J. S. Kim, M. Patel, H. Hassan, and O. Mutlu, "The DRAM Latency PUF: Quickly Evaluating Physical Unclonable Functions by Exploiting the Latency-Reliability Tradeoff in Modern Commodity DRAM Devices," in *HPCA*, 2018.

[53] J. S. Kim, M. Patel, H. Hassan, L. Orosa, and O. Mutlu, "D-RaNGe: Using Commodity DRAM Devices to Generate True Random Numbers with Low Latency and High Throughput," in *HPCA*, 2019.

[54] J. S. Kim, M. Patel, A. G. Yaglikci, H. Hassan, R. Azizi, L. Orosa, and O. Mutlu, "Revisiting RowHammer: An Experimental Analysis of Modern DRAM Devices and Mitigation Techniques," in *ISCA*, 2020.

[55] M. J. Kim, J. Park, Y. Park, W. Doh, N. Kim, T. J. Ham, J. W. Lee, and J. H. Ahn, "Mithril: Cooperative Row Hammer Protection on Commodity DRAM Leveraging Managed Refresh," *arXiv preprint arXiv:2108.06703*, 2021.

[56] 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.

[57] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu, "A Case for Exploiting Subarray-level Parallelism (SALP) in DRAM," in *ISCA*, 2012.

[58] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A Fast and Extensible DRAM Simulator," in *CAL*, 2016.

[59] R. K. Konoth, M. Oliverio, A. Tatar, D. Andriesse, H. Bos, C. Giuffrida, and K. Razavi, "ZebRAM: Comprehensive and Compatible Software Protection Against Rowhammer Attacks," in *OSDI*, 2018.

[60] N. Kwak, S. Kim, K. H. Lee, C. Baek, M. S. Jang, Y. Joo, S. Lee, W. Y. Lee, E. Lee, D. Han, J. Kang, J. H. Lim, J. Park, K. Kim, S. Cho, S. W. Han, J. Y. Keh, J. H. Chun, J. Oh, and S. H. Lee, "A 4.8 Gb/s/pin 2Gb LPDDR4 SDRAM with Sub-100μA Self-Refresh Current for IoT Applications," in *ISSCC*, 2017.

[61] H. Kwon, E. Seo, C. Lee, Y. Seo, G. Han, H. Kim, J. Lee, M. Jang, S. Do, S. Cho, J. Park, S. Doo, J. Shin, S. Jung, H. Kim, I. Im, B. Cho, J. Lee, J. Lee, K. Yu, H. Kim, C. Jeon, H. Park, S. Kim, S. Lee, J. Park, S. Lee, B. Lim, J. Park, Y. Park, H. Kwon, S. Bae, J. Choi, K. Park, S. Jang, and G. Jin, "An Extremely Low-Standby-Power 3.733 Gb/s/pin 2Gb LPDDR4 SDRAM for Wearable Devices," in *ISSCC*, 2017.

[62] A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, "RAMBleed: Reading Bits in Memory Without Accessing Them," in *S&P*, 2020.

[63] D. Lee, Y. Kim, G. Pekhimenko, S. Khan, V. Seshadri, K. Chang, and O. Mutlu, "Adaptive-Latency DRAM: Optimizing DRAM Timing for the Common-Case," in *HPCA*, 2015.

[64] D. Lee, S. Ghose, G. Pekhimenko, S. Khan, and O. Mutlu, "Simultaneous Multi-Layer Access: Improving 3D-Stacked Memory Bandwidth at Low Cost," in *TACO*,

2016.
[65] D. Lee, S. Khan, L. Subramanian, S. Ghose, R. Ausavarungnirun, G. Pekhimenko, V. Seshadri, and O. Mutlu, "Design-Induced Latency Variation in Modern DRAM Chips: Characterization, Analysis, and Latency Reduction Mechanisms," in *SIGMETRICS*, 2017.
[66] D. Lee, Y. Kim, V. Seshadri, J. Liu, L. Subramanian, and O. Mutlu, "Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture," in *HPCA*, 2013.
[67] D. Lee, L. Subramanian, R. Ausavarungnirun, J. Choi, and O. Mutlu, "Decoupled Direct Memory Access: Isolating CPU and IO Traffic by Leveraging a Dual-Data-Port DRAM," in *PACT*, 2015.
[68] E. Lee, I. Kang, S. Lee, G. Edward Suh, and J. Ho Ahn, "TWiCe: Preventing Row-Hammering by Exploiting Time Window Counters," in *ISCA*, 2019.
[69] J.-B. Lee, "Green Memory Solution," in *Samsung Electronics, Investor's Forum*, 2014.
[70] S. Lin and D. J. Costello, *Error Control Coding: Fundamentals and Applications*, 2002.
[71] M. Lipp, M. T. Aga, M. Schwarz, D. Gruss, C. Maurice, L. Raab, and L. Lamster, "Nethammer: Inducing RowHammer Faults Through Network Requests," *Euro S&PW*, 2020.
[72] J. Liu, B. Jaiyen, Y. Kim, C. Wilkerson, and O. Mutlu, "An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms," in *ISCA*, 2013.
[73] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu, "RAIDR: Retention-Aware Intelligent DRAM Refresh," in *ISCA*, 2012.
[74] H. Luo, T. Shahroodi, H. Hassan, M. Patel, A. G. Yağlıkçı, L. Orosa, J. Park, and O. Mutlu, "CLR-DRAM: A Low-Cost DRAM Architecture Enabling Dynamic Capacity-Latency Trade-Off," in *ISCA*, 2020.
[75] J. A. Mandelman, R. H. Dennard, G. B. Bronner, J. K. DeBrosse, R. Divakaruni, Y. Li, and C. J. Radens, "Challenges and Future Directions for the Scaling of Dynamic Random-access Memory (DRAM)," in *IBM JRD*, 2002.
[76] J. Meza, Q. Wu, S. Kumar, and O. Mutlu, "Revisiting Memory Errors in Large-scale Production Data Centers: Analysis and Modeling of New Trends from the Field," in *DSN*, 2015.
[77] Micron, "DDDR4 SDRAM Datasheet," 2016.
[78] Micron, "8Gb: x4, x8, x16 DDR4 SDRAM Features - Excessive Row Activation," 2020.
[79] Micron Technology inc., "ECC Brings Reliability and Power Efficiency to Mobile Devices," Micron Technology inc., Tech. Rep., 2017.
[80] Y. Mori, K. Ohyu, K. Okonogi, and R. I. Yamada, "The Origin of Variable Retention Time in DRAM," in *IEDM*, 2005.
[81] O. Mutlu, "Memory Scaling: A Systems Architecture Perspective," in *IMW*, 2013.
[82] O. Mutlu, "The RowHammer Problem and Other Issues We may Face as Memory Becomes Denser," in *DATE*, 2017.
[83] O. Mutlu and J. S. Kim, "RowHammer: A Retrospective," *TCAD*, 2019.
[84] O. Mutlu and L. Subramanian, "Research Problems and Opportunities in Memory Systems," in *SUPERFRI*, 2014.
[85] P. J. Nair, D.-H. Kim, and M. K. Qureshi, "ArchShield: Architectural Framework for Assisting DRAM Scaling by Tolerating High Error Rates," in *ISCA*, 2013.
[86] P. J. Nair, V. Sridharan, and M. K. Qureshi, "XED: Exposing On-Die Error Detection Information for Strong Memory Reliability," in *ISCA*, 2016.
[87] T.-Y. Oh, H. Chung, J.-Y. Park, K.-W. Lee, S. oh, S.-Y. Doo, H.-J. Kim, C. Lee, H.-R. Kim, J.-H. Lee, J.-I. Lee, K.-S. Ha, Y. Choi, Y.-C. Cho, Y.-C. Bae, T. Jang, C. Park, K. Park, S. Jang, and J. Choi, "A 3.2Gbps/pin 8Gbit 1.0V LPDDR4 SDRAM with Integrated ECC Engine for Sub-1V DRAM Core Operation," *JSSC*, 2014.
[88] A. Olgun, M. Patel, A. G. Yağlıkçı, H. Luo, J. S. Kim, N. Bostancı, N. Vijaykumar, O. Ergin, and O. Mutlu, "QUAC-TRNG: High-Throughput True Random Number Generation Using Quadruple Row Activation in Commodity DRAM Chips," in *ISCA*, 2021.
[89] L. Orosa, A. G. Yağlıkçı, H. Luo, A. Olgun, J. Park, H. Hassan, M. Patel, J. S. Kim, and O. Mutlu, "A Deeper Look into RowHammer's Sensitivities: Experimental Analysis of Real DRAM Chips and Implications on Future Attacks and Defenses," in *MICRO*, 2021.
[90] K. Park, C. Lim, D. Yun, and S. Baeg, "Experiments and Root Cause Analysis for Active-Precharge Hammering Fault in DDR3 SDRAM under 3× nm Technology," *MR*, 2016.
[91] Y. Park, W. Kwon, E. Lee, T. J. Ham, J. H. Ahn, and J. Lee, "Graphene: Strong yet Lightweight Row Hammer Protection," in *MICRO*, 2020.
[92] M. Patel, J. Kim, T.-M. Shahroodi, H. Hassan, and O. Mutlu, "Bit-Exact ECC Recovery (BEER): Determining DRAM On-Die ECC Functions by Exploiting DRAM Data Retention Characteristics," in *MICRO*, 2020.
[93] M. Patel, J. S. Kim, H. Hassan, and O. Mutlu, "Understanding and Modeling On-Die Error Correction in Modern DRAM: An Experimental Study Using Real Devices," in *DSN*, 2019.
[94] M. Patel, J. S. Kim, and O. Mutlu, "The Reach Profiler (REAPER): Enabling the Mitigation of DRAM Retention Failures via Profiling at Aggressive Conditions," in *ISCA*, 2017.
[95] M. Patel, G. F. Oliveira, and O. Mutlu, "HARP: Practically and Effectively Identifying Uncorrectable Errors in Memory Chips That Use On-Die Error-Correcting Codes," in *MICRO*, 2021.
[96] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks," in *USENIX*, 2016.
[97] S. Qazi, Y. Kim, N. Boichat, E. Shiu, and M. Nissler, "Introducing Half-Double: New Hammering Rechnique for DRAM Rowhammer Bug," http://googleprojectzero.blogspot.com.tr/2015/03/exploiting-dram-rowhammer-bug-to-gain.html, 2021.
[98] R. Qiao and M. Seaborn, "A New Approach for RowHammer Attacks," in *HOST*, 2016.
[99] M. K. Qureshi, D. Kim, S. Khan, P. J. Nair, and O. Mutlu, "AVATAR: A Variable-Retention-Time (VRT) Aware Refresh for DRAM Systems," in *DSN*, 2015.
[100] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, "Flip Feng Shui: Hammering a Needle in the Software Stack," in *USENIX Security*, 2016.
[101] I. S. Reed and G. Solomon, "Polynomial Codes Over Certain Finite Fields," *SIAM*, 1960.
[102] P. J. Restle, J. Park, and B. F. Lloyd, "DRAM Variable Retention Time," in *IEDM*, 1992.
[103] S.-W. Ryu, K. Min, J. Shin, H. Kwon, D. Nam, T. Oh, T.-S. Jang, M. Yoo, Y. Kim, and S. Hong, "Overcoming the Reliability Limitation in the Ultimately Scaled DRAM using Silicon Migration Technique by Hydrogen Annealing," in *IEDM*, 2017.
[104] SAFARI Research Group, "RowHammer — GitHub Repository," https://github.com/CMU-SAFARI/rowhammer.
[105] SAFARI Research Group, "SoftMC Source Code," https://github.com/CMU-SAFARI/SoftMC.
[106] J. H. Saltzer and M. D. Schroeder, "The Protection of Information in Computer Systems," *Proceedings of the IEEE*, 1975.
[107] K. Scarfone, W. Jansen, and M. Tracy, "Guide to General Server Security," *NIST Special Publication*, 2008.
[108] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM Errors in the Wild: A Large-scale Field Study," in *SIGMETRICS*, 2009.
[109] M. Seaborn and T. Dullien, "Exploiting the DRAM RowHammer Bug to Gain Kernel Privileges," *Black Hat*, 2015.
[110] M. Seaborn and T. Dullien, "Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges," http://googleprojectzero.blogspot.com.tr/2015/03/exploiting-dram-rowhammer-bug-to-gain.html, 2015.
[111] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. Mowry, "RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization," in *MICRO*, 2013.
[112] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology," in *MICRO*, 2017.
[113] V. Seshadri, T. Mullins, A. Boroumand, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Gather-scatter DRAM: In-DRAM Address Translation to Improve the Spatial Locality of Non-unit Strided Accesses," in *MICRO*, 2015.
[114] V. Seshadri and O. Mutlu, "In-DRAM Bulk Bitwise Execution Engine," *Advances in Computers*, 2020.
[115] S. M. Seyedzadeh, A. K. Jones, and R. Melhem, "Counter-based Tree Structure for Row Hammering Mitigation in DRAM," *CAL*, 2017.
[116] SK Hynix, "DDR4 SDRAM Device Operation," https://pdf.directindustry.com/pdf/hynix/ddr4-sdram-device-operation/34497-773768.html.
[117] M. Son, H. Park, J. Ahn, and S. Yoo, "Making DRAM Stronger Against Row Hammering," in *DAC*, 2017.
[118] Y. H. Son, S. Lee, O. Seongil, S. Kwon, N. S. Kim, and J. H. Ahn, "CiDRA: A Cache-Inspired DRAM Resilience Architecture," in *HPCA*, 2015.
[119] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi, "Memory Errors in Modern Systems: The Good, the Bad, and the Ugly," in *ASPLOS*, 2015.
[120] V. Sridharan and D. Liberty, "A Study of DRAM Failures in the Field," in *SC*, 2012.
[121] M. Taouil, C. Reinbrecht, S. Hamdioui, and J. Sepúlveda, "LightRoAD: Lightweight Rowhammer Attack Detector," in *ISVLSI*, 2021.
[122] A. Tatar, C. Giuffrida, H. Bos, and K. Razavi, "Defeating Software Mitigations Against Rowhammer: A Surgical Precision Hammer," in *RAID*, 2018.
[123] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic Rowhammer Attacks on Mobile Platforms," in *CCS*, 2016.
[124] V. van der Veen, M. Lindorfer, Y. Fratantonio, H. P. Pillai, G. Vigna, C. Kruegel, H. Bos, and K. Razavi, "GuardION: Practical Mitigation of DMA-Based Rowhammer Attacks on ARM," in *DIMVA*, 2018.
[125] R. K. Venkatesan, S. Herr, and E. Rotenberg, "Retention-Aware Placement in DRAM (RAPID): Software Methods for Quasi-Non-Volatile DRAM," in *HPCA*, 2006.
[126] A. J. Walker, S. Lee, and D. Beery, "On DRAM Rowhammer and the Physics of Insecurity," *TED*, 2021.

[127] Y. Wang, L. Orosa, X. Peng, Y. Guo, S. Ghose, M. Patel, J. Kim, J. Gómez-Luna, M. Sadrosadati, N. Ghiasi, and O. Mutlu, "FIGARO: Improving System Performance via Fine-Grained In-DRAM Data Relocation and Caching," in *MICRO*, 2020.

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

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

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

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

[132] D. S. Yaney, C. Lu, R. A. Kohler, M. J. Kelly, and J. T. Nelson, "A Meta-stable Leakage Phenomenon in DRAM Charge Storage-Variable Hold Time," in *IEDM*, 1987.

[133] C.-M. Yang, C.-K. Wei, Y. J. Chang, T.-C. Wu, H.-P. Chen, and C.-S. Lai, "Suppression of Row Hammer Effect by Doping Profile Modification in Saddle-Fin Array Devices for Sub-30-nm DRAM Technology," *TDMR*, 2016.

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

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

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

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

[138] T. Zhang, K. Chen, C. Xu, G. Sun, T. Wang, and Y. Xie, "Half-DRAM: A High-Bandwidth and Low-Power DRAM Architecture from the Rethinking of Fine-Grained Activation," in *ISCA*, 2014.

[139] X. Zhang, Y. Zhang, B. R. Childers, and J. Yang, "Restore Truncation for Performance Improvement in Future DRAM Systems," in *HPCA*, 2016.

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