PROTRR: Principled yet Optimal In-DRAM Target Row Refresh

Michele Marazzi, Patrick Jattke, Flavien Solt and Kaveh Razavi
Computer Security Group, ETH Zürich

***Abstract*—The DRAM substrate is becoming increasingly more vulnerable to Rowhammer as we move to smaller technology nodes. We introduce PROTRR, the first principled in-DRAM Target Row Refresh mitigation with formal security guarantees and low bounds on overhead. Unlike existing proposals that require changes to the memory controllers, the in-DRAM nature of PROTRR enables its seamless integration. However, this means that PROTRR must respect the synchronous nature of the DRAM protocol, which limits the number of DRAM rows that can be protected at any given time. To overcome this challenge, PROTRR *proactively* refreshes each row that is most likely to observe bit flips in the future. While this strategy catches the rows that are hammered the most, some others may still *fly under the radar*. We use this observation to construct FEINTING, a new Rowhammer attack that we formally prove to be optimal in this setting. We then configure PROTRR to be secure against FEINTING. To achieve this, PROTRR should keep track of accesses to each row, which is prohibitively expensive to implement in hardware. Instead, PROTRR uses a new frequent item counting scheme that leverages FEINTING to provide a provably optimal yet flexible trade-off between the tolerated DRAM vulnerability, the number of counters, and the number of additional refreshes. Our extensive evaluation using an ASIC implementation of PROTRR and cycle-accurate simulation shows that PROTRR can provide principled protection for current and future DRAM technologies with a negligible performance, power, and area impact. PROTRR is fully compatible with DDR4 and the new Refresh Management (RFM) extension in DDR5.**

I. INTRODUCTION

Despite numerous mitigation attempts under Target Row Refresh (TRR), Rowhammer is still an unsolved problem in practice [1]–[3], threatening systems security in many different scenarios [4]–[14]. Existing proposals attempt to mitigate Rowhammer in the memory controller [15]–[20], but CPU vendors have little incentive to introduce expensive mitigations for a problem in the products of DRAM vendors. The natural place to fix Rowhammer is inside DRAM itself, but mitigations with strong security guarantees are currently lacking.

We present PROTRR, the first principled in-DRAM Rowhammer mitigation that is secure against FEINTING, a novel Rowhammer attack that we mathematically prove to be optimal. PROTRR uses the bounds given by FEINTING in the design of a new frequent item counting scheme, called PROMG (Proactive Misra-Gries), with a provably optimal yet flexible trade-off between the number of required counters and additional refreshes. Our extensive evaluation of PROTRR using an ASIC implementation and cycle-accurate simulation shows the feasibility of principled in-DRAM Rowhammer protection for current and future DRAM technologies.

Rowhammer. In their seminal work, Kim et al. [15] showed that by repeatedly activating a DRAM row (i.e., aggressor), it is possible to flip bits in its adjacent rows (i.e., victims) before these rows have a chance to be refreshed as part of the background DRAM refresh operation. This effect is present in most DDR3 devices and has only worsened in DDR4 devices deployed on more recent systems [1]–[3], [21]. In essence, Rowhammer is compromising the isolation of data on DRAM. A plethora of attacks followed, showing that it is possible to abuse these bit flips to escalate privileges [8], [9], [14], compromise browsers [4]–[7], break into co-located virtual machines in the cloud [10], [11], and even attack servers over the network [12], [13]. These attacks highlight the urgent need for strong mitigations against Rowhammer.

Mitigations. Originally, two practical countermeasures were believed to stop Rowhammer: doubling the DRAM's refresh rate and error-correcting code (ECC) DRAM. Unfortunately, neither can fully protect systems [22], [23]. There are also proposals to mitigate Rowhammer in software [9], [22], [24], [25], but these solutions have security and performance issues [5], [8], [26]. To mitigate Rowhammer in hardware, previous work mostly proposes to modify the memory controller to detect potential aggressors and refresh their victims [15], [16], [18]–[20]. Unfortunately, due to their substantial cost, CPU vendors are reluctant to deploy these mitigations given the promise of Rowhammer-free devices by the DRAM vendors [27], [28]. However, without carefully analyzing the security implications of performing TRRs inside DRAM, there will be gaps in the protection, as evident in recent work [1]–[4], [19]. These gaps will only worsen with the increasing Rowhammer vulnerability in newer DRAM generations with smaller technology nodes.

FEINTING. In this paper, we advocate for a principled approach for designing secure in-DRAM mitigations. In-DRAM mitigations allow for seamless system integration, but they need to strictly adhere to the synchronous DRAM timing specifications defined in the DDRx standard [29], [30]. For example, a DRAM refresh command cannot suddenly take longer when the system is under attack. This means that any in-DRAM mitigation can only protect a handful of victim rows at any given point in time. Consequently, even with an ideal in-DRAM TRR scheme that always protects rows that are hammered the most, an attacker can use *decoy* rows to slowly increase the number of times a victim is hammered without it ever being subject to the mitigation. We use this observation to construct FEINTING, a novel Rowhammer attack that we mathematically prove to be

optimal against an ideal in-DRAM mitigation. FEINTING enables us to calculate strict bounds on the degree of Rowhammer vulnerability that can be tolerated on any compliant DDR4 device and future DDR5 devices that use Refresh Management (RFM), a new extension that is primarily introduced in the DDR5 standard to address Rowhammer [30]. To the best of our knowledge, this is the first work to define and calculate these crucial bounds.

PROTRR. Counting the activations of each row for an ideal in-DRAM mitigation is too expensive to implement in hardware. Existing frequent item counting schemes can reduce the number of necessary counters when frequent items need to be identified over an arbitrary sequence of row activations [19]. Unfortunately, these schemes are unsuitable for in-DRAM TRR which needs to proactively protect target rows based on the information that is available at short intervals. We develop Principled yet Optimal Target Row Refresh (PROTRR), a new in-DRAM Rowhammer mitigation that we prove is both secure and optimal in this setting. PROTRR makes use of a new frequent item counting scheme, called PROMG, that adapts FEINTING to right-size Misra-Gries summaries [31] for secure in-DRAM operation. Our calculations show that the insights from FEINTING enable PROTRR to significantly reduce the required number of counters with slight changes to the Rowhammer tolerance. This property provides PROTRR with an unprecedented flexibility: depending on the degree of Rowhammer vulnerability, a DRAM vendor can decide how to balance the number of counters and in-DRAM refreshes for keeping its DRAM devices secure. Furthermore, we provide a proof that PROTRR is optimal in terms of counters and the required refreshes at any given configuration; fixing the number of refreshes, any in-DRAM mitigation that uses fewer counters than PROTRR will be vulnerable to Rowhammer. Similarly, fixing the number of counters, any in-DRAM mitigation that uses fewer refreshes will also be vulnerable.

Our extensive evaluation using an ASIC implementation and cycle-accurate simulation shows that PROTRR provides principled protection with a negligible performance, area, and power impact. For example, PROTRR can protect a DDR5 device where bits flip after only 3,200 activations, with less than 0.2% performance overhead, while increasing the area by 1.78% and energy consumption of DRAM by 2.35%.

Contributions. We make the following contributions:

1. The construction of FEINTING and a mathematical proof of its optimality against an ideal in-DRAM TRR.
2. The design of PROTRR, a principled in-DRAM TRR that is secure against FEINTING while providing a provably-optimal yet flexible trade-off between the required counters and refreshes.
3. A comprehensive evaluation of PROTRR using (i) an ASIC implementation in a popular 12 nm technology for measuring its area and power requirements in DDR4 and DDR5 devices, and (ii) cycle-accurate simulation for measuring its performance overhead when using the recently introduced RFM extension in DDR5.

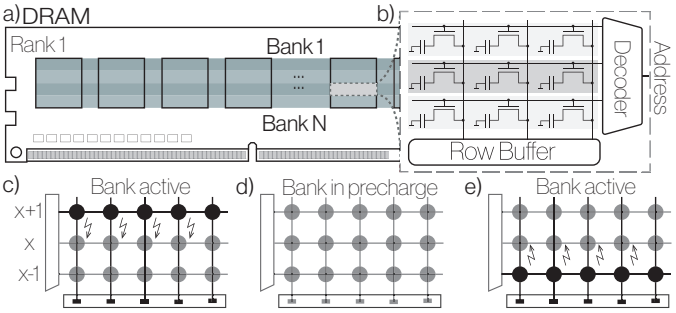

Fig. 1: **DRAM architecture and relevant DRAM operations.** (a) the rank/bank hierarchy in a DRAM device, (b) row addressing after rank/bank selection, (c) activating a row $X+1$ in a bank using `ACT` to bring its content to the row buffer, (d) deactivating the row in the row buffer using `PRE`, (e) activating another row $X-1$. Repeated activation of rows $X+1$ and $X-1$ can potentially trigger Rowhammer bit flips in row X.

II. BACKGROUND

We briefly discuss the architecture and operation of a DRAM device (§II-A) before discussing the Rowhammer vulnerability (§II-B). We then introduce the current proposals for mitigating Rowhammer and discuss their limitations (§II-C). We kindly refer the reader to Table IV (Appendix E) for a summary of all symbols introduced in this and following sections.

A. DRAM architecture

The architecture of DRAM and its basic operation is depicted in Figure 1. Like most memory devices, a principal abstraction in DRAM is the association of data with its address. A DRAM address traverses a hierarchy, starting with a *channel* and continuing to a specific connected DRAM device. Once a device is selected, the data address is further used to identify a *rank* and then a specific *bank* within that rank (Figure 1-a). Each bank is a matrix of cells that stores information using a capacitor (Figure 1-b). When data has to be read or written, its associated row has to be activated using the DRAM `ACTIVATE` (`ACT`) command, which connects the row to the *row buffer* (Figure 1-c), making the bank *active*. To deactivate a bank, the DRAM `PRECHARGE` (`PRE`) command is used. The memory controller can decide when to send the `PRE` command based on a policy. With a closed-page policy, the memory controller sends the `PRE` command right after or with the DRAM access. In contrast, with an open-page policy, the memory controller can delay the `PRE` command. Internally and transparently to the outside world, banks can further be divided into subarrays [32]. Each subarray has its own local row buffer, which is connected to the bank's row buffer. Subarrays allow for parallelization of certain DRAM operations such as the `REFRESH` (`REF`) command. Because of the physical nature of capacitors, their charge constantly leaks. To preserve their value, the CPU's memory controller periodically sends `REF` commands to DRAM, which triggers an internal refresh mechanism. Each issued `REF` only covers a fraction of the addresses. The JEDEC DRAM standard requires each row to be refreshed at least once in a `tREFW` and the memory controller to issue `REF`s at intervals defined by `tREFI` [29], [30]. As an example, if `tREFW` equals 64 ms and `tREFI`

equals 7.8125 µs, the memory controller needs to send a total of 8192 REF commands in a tREFW.

B. Rowhammer

Thanks to continuous improvements in process technology, we observe an increased DRAM chip density each year. Unfortunately, this comes at a reliability cost [33]. As DRAM rows get closer to each other, their electrical isolation gets compromised. Rowhammer is an attack based on repeated row activations [15] that causes cells in nearby rows to leak charge and eventually change their stored values (i.e., bits flip). The row with repeated activations is commonly referred to as the *aggressor* row. The repeated activations of an aggressor row affect its neighboring rows, which are commonly referred to as *victim* rows. A variant of this attack where a victim row is sandwiched between two aggressor rows, known as *double-sided Rowhammer*, is depicted in Figure 1 (c-e). Recently, it has been shown that an aggressor row can influence victims that are two rows apart from the aggressor [34]. This means that in certain DRAM devices, an aggressor can have a *blast diameter* (B) of 4, affecting up to four victim rows.

Seaborn [14] showed for the first time that Rowhammer bit flips could severely compromise security by building a native privilege-escalation exploit. Plenty of other attacks followed [35]–[42], where researchers showed that it is possible to use these bit flips to compromise browsers [5]–[7], cloud virtual machines [10], [11], mobile phones [8], [9] and even remote machines over the network [12], [13].

C. Rowhammer mitigations

In response to these attacks, many solutions have attempted to mitigate Rowhammer in software or hardware. The ones implemented in software, usually inside the operating system's kernel, try to detect aggressor accesses and refresh their victims [22], isolating sensitive data from bit flips [9], [24], [25], or using certain pages to store sensitive information [43]. Unfortunately, these solutions require adoption by operating systems, which has not happened to date. They are also often vulnerable to more advanced attacks [8], [26], [44].

At the hardware level, Rowhammer can be mitigated either at the CPU's memory controller or inside the DRAM itself. Over the years, there have been many proposals by academia to modify the memory controller to detect aggressor rows either deterministically [16], [17], [19], [20], [45] or probabilistically [15], [17] and to refresh their victims under the Target Row Refresh (TRR) scheme. Except for a low-cost solution that was briefly adopted by Intel [1], [15], [46], the remaining ones require extensive modifications to the CPU's memory controller with non-trivial area or performance overhead. As a result, they have not seen any adoption [1]. It is unlikely that all CPU vendors will deploy an expensive mitigation to fix a problem that is in the products of DRAM vendors. Perhaps, the only enabled mitigation in the CPU is the memory controller-based Error-Correction Code (ECC) in server systems. This covers only a fraction of existing computer systems that use DRAM, and even then, ECC does not provide an adequate level of protection against Rowhammer attacks [23], [47].

Rowhammer is a DRAM vulnerability, and arguably the best place to address it is inside the DRAM itself. In fact, this is exactly what DRAM vendors have done [27], [28]. Unfortunately, these in-DRAM TRR mitigations are undocumented and lack formal security guarantees. Recent work shows that there are indeed gaps in currently deployed mitigations and slight changes to existing Rowhammer patterns result in bit flips to resurface [1]–[3]. The only existing academic work on in-DRAM TRR [48] similarly suffers from slightly more advanced patterns [19]. Hence, we urgently need an in-DRAM TRR mechanism with formal security guarantees. In this paper, we show not only that this is possible, but it can be done in a way that is optimal in terms of the number of required counters and the introduced refresh overhead.

III. Threat Model

We consider a DRAM device that is affected by the Rowhammer vulnerability. At the time of this writing, Rowhammer is present in all recent DRAM technologies [3], [21]. We assume that bits start to flip after R_{thresh} cumulative accesses to aggressor rows and that each aggressor row can influence up to B victim rows. We assume an adversary that is capable of sending requests to the DRAM device either through local code execution [8]–[11], [14], [44], [47], from the Web [4]–[7], or even over the network [12], [13] through a CPU that deploys a memory controller that is compliant with the respective DRAM standard [29], [30]. The aim of the adversary is to craft an access pattern that triggers Rowhammer bit flips to compromise the system by ensuring that a victim is hammered at least R_{thresh} times. Our mitigation should provide a formal guarantee that no row can be hammered R_{thresh} times before it is protected by TRR.

IV. Refresh Management in DDR5

Recent (LP)DDR4 devices internally perform TRRs on potential victim rows, whenever they receive REF commands [1]. In theory, it is possible to perform TRRs during the execution of other DRAM commands such as ACT or read/write. However, as these commands are latency-critical, it would adversely affect the performance. As such, the REF is shared between regular refreshes and TRRs. Consequently, TRRs are scarcely performed and can only refresh a limited number of rows each time. Performing multiple TRRs overloads the REF command, and moving to smaller technology nodes with increasing Rowhammer vulnerability [21] only exacerbates this problem. As a remedy, the DDR5 standard [30] introduces a new DRAM command called Refresh Management (RFM) that provides additional time for TRRs.

RFM mechanisms. An RFM command either targets the same bank address in each bankgroup (RFMsb) or all banks (RFMab). Each bank has a counter called *Rolling Accumulated ACT* (RAA) that tracks the number of received ACTs. Once RAA reaches a maximum value defined as *RAA Maximum Management Threshold* (RAAMMT), no more ACTs are accepted

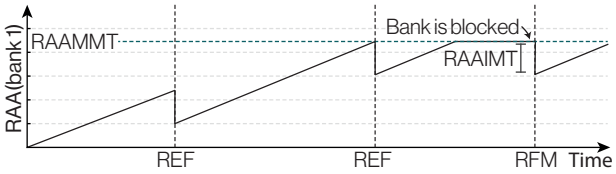

Fig. 2: **RFM example.** Activations are sent to the same bank, increasing RAA. At each REF, RAA is decremented by RAAIMT. Once the RAA reaches RAAMMT, the bank does not accept any ACTs anymore. In this case, issuing a RFM can reduce the counter by RAAIMT to unblock it before the next REF, which will also reduce the RAA counter value.

by the bank until the RAA counter is decremented. There are two possibilities to decrement this counter: RFM and REF commands. Every time an RFM is received, the target banks' RAA is reduced by the value set in the *Initial Management Threshold* (RAAIMT). Instead, REF reduces RAA either by 0.5× or 1× of the RAAIMT, depending on the value of the MR59 OP[7:6] DRAM register. Figure 2 summarizes these concepts with an example. In the current DDR5 standard, valid values for RAAIMT range from 32 to 80, in steps of 8. Since the RFM command can be postponed by the memory controller, in practice RAAIMT defines the average number of activations received by a bank before an RFM is issued. Instead, RAAMMT = $m \times$ RAAIMT defines the maximum number of activations before an RFM or a REF must be issued, where m is an integer between 3 and 6 set by the DRAM. This gives the memory controller flexibility for scheduling RFM and REF commands as long as a bank's RAA count remains below RAAMMT.

V. Feinting

As stated in §II, the design of a secure and working in-DRAM TRR is still an open problem. The operations of such a mitigation are fundamentally different from those implemented inside the memory controller. In particular, (i) the points at which TRRs can be performed in a tREFW are limited, and (ii) only a small number of rows can be refreshed at each point.

In other words, performing in-DRAM TRR means occasionally refreshing a bounded number of rows. Therefore, to successfully protect against Rowhammer, the mitigation has to use the available TRRs effectively. Given these conditions, the only way to implement a secure mitigation is to *proactively* refresh rows. To provide deterministic guarantees, a proactive TRR scheme must keep track of row activations. This can be achieved by storing a list of victims or aggressors. Additionally, we define a Rowhammer mitigation to be *proactive* if (i) rows are refreshed without using a fixed hammering threshold, and (ii) the TRR mechanism is triggered periodically. In a proactive mitigation, every time the mechanism is triggered (*TRR event*), the most hammered V victim rows (*TRR volume*) are refreshed. Because this happens periodically, we consider two consecutive TRR events to be interleaved by T activations (*interval*).

In this section, we consider an ideal TRR scheme, TRR$_{ideal}$, which has a hammer counter for each victim row. The victim row's counter increases every time one of its aggressor rows is activated, TRRed or refreshed by the regular REF. The victim row's counter is reset to zero every time the victim row is activated, TRRed, or refreshed by the regular REF. For clarity, we define REF_I as the refresh where a specific row is regularly refreshed (i.e., not TRRed). In §VI, we show how we can relax these requirements to build an in-DRAM TRR scheme that is both counter- and TRR-optimal while providing the same guarantees as TRR$_{ideal}$.

A. Security analysis of TRR$_{ideal}$

Any proactive TRR mitigation can protect up to a specific degree of Rowhammer vulnerability (R_{thresh}). In an ideal proactive mitigation with unlimited counters, this limit depends on V, T and B. Selecting V and T (B is technology-dependent), there exists a maximum count ($Hammer_{max}$) that a victim row can reach before getting refreshed either by REF_I or TRR.

Definition 1 (Victim hammering)**.** *A victim row $\tilde{x}$ is hammered each time one of its aggressor rows $\tilde{r}$ is activated (i.e., $\tilde{x}$ is one of the $B/2$ rows on each side of $\tilde{r}$). We denote by $x(\alpha)$ the hammer count of row $\tilde{x}$ after the α-th ACT of the attack. $x(\alpha)$ becomes zero every time $\tilde{x}$ is subject to REF_I, TRR, or an activation.*

Definition 2 (Rowhammer attack)**.** *We define a Rowhammer attack $\mathcal{A}$ on a victim $\tilde{x}$, as a finite sequence of L_{attk} activations to a bank's rows. $\mathcal{A}$ is successful against $\tilde{x}$ iff $\exists \alpha \geq 1 \mid x(\alpha) \geq R_{thresh}$. We denote by $\mathscr{A}$ the set of all attacks, which is the set of finite sequences over $[\![1, N_{rows}]\!]$, with N_{rows} being the number of rows in a bank.*

Definition 3 (Optimal Rowhammer attack)**.** *For a given $(V, T, B, mitigation)$, we define $Hammer_{max} = \max_{\mathscr{A}} \max_{1 \leq x \leq N_{rows}} \max_{\alpha \geq 1}[x(\alpha)]$. An attack $\mathcal{A} \in \mathscr{A}$ is optimal against a victim $\tilde{x}$ iff $\mathcal{A}$ reaches $Hammer_{max}$.*

Following, we express the security requirement for TRR$_{ideal}$:

Requirement (Security of TRR$_{ideal}$)**.** *For a given DRAM technology (B, R_{thresh}) and configuration (V, T), TRR$_{ideal}$ is secure if $Hammer_{max} < R_{thresh}$.*

Identifying $Hammer_{max}$ corresponds to finding the *optimal Rowhammer attack* against the mitigation. In what follows, we present and prove the best attack against TRR$_{ideal}$.

Assumptions. In our analysis, (i) we consider a memory controller with a closed-page policy (i.e., no bank collisions are required to induce a PRECHARGE); (ii) if during a TRR event, more than V rows have the same highest count, we consider an attacker that is able to influence which are refreshed; (iii) we assume an attacker that knows when the rows are refreshed by the REF_I — including the victim $\tilde{x}$. These assumptions constitute the worst possible conditions for the defender.

Without TRR, all the activations in a tREFW (L_{tREFW}) can be used against the victim. However, this approach would quickly fail against a proactive mitigation: the mechanism would refresh the victim at the first TRR event, as the victim row would have the highest count. We will demonstrate that by using a specific activation pattern, the TRR event will never refresh the target victim before its REF_I. Moreover, we will

show how this pattern can be used to build the best possible attack against TRR_{ideal}, which we refer to as *FEINTING*[1].

Decoy rows. Given a target victim row $\tilde{x}$, the attacker aims at activating the aggressor rows while protecting the victim from refreshes. During a TRR event, the only case where $\tilde{x}$ is not refreshed is if there are at least V different victim rows (decoys) with a greater or equal hammer count. When this happens, we say that the victim "survives" the TRR event.

Definition 4 (Conditions for victim survival)**.** *A victim row $\tilde{x}$ is not refreshed during a TRR event, after activation α, iff there exist V distinct rows $\tilde{d}_1...\tilde{d}_V$, each different from $\tilde{x}$, such that $\min_j[d_j(\alpha)] \geq x(\alpha)$. We refer to the rows $\tilde{d}_1..\tilde{d}_V$ as "decoys".*

Every time a victim is hammered, its counter is incremented by one. Given Definition 4, it follows that decoy rows must be incremented concurrently. Unfortunately for the attacker, when decoys are refreshed, their counters reset to zero and become lower than the victim's count. At the next event, different decoys will have to be higher or equal to the victim count for the victim to survives again. Generally, to survive n TRR events, a victim needs a total of $n \times V$ decoys. Note that each time an aggressor row is activated, it influences up to B victim rows. That is, for a single aggressor row activation, B decoys are hammered and their counters increase by one.

Problem formalization. This condition creates an optimization problem: before a TRR event, part of the activations should be used to hammer the victim and the remaining to hammer the decoy rows. However, if too many activations target the victim, the decoys cannot protect it from being refreshed. On the opposite, if just a few activations hammer the victim, it will reach a lower hammer count than possible since the extra activations used for the decoys are "wasted" (i.e., not used against the victim). Hence, the number of decoys and their hammer count should be minimized. We formalize this problem as follows: *Considering all activations in an attack (L_{attk}), then $L_{attk} - k$ activations must be used to build and maintain a set of decoys. The remaining **k** activations can be used for hammering the victim row and thus should be maximized.* We solve this problem by answering the following questions:

1) What is the optimal hammer ratio between the different rows? [*optimal distribution*]
2) How many times should the rows be hammered in each step? [*optimal intensity*]
3) How many TRR events should the attack last? [*optimal duration*]

Answering these questions will lead us to the FEINTING attack. We start by obtaining FEINTING for DDR4 devices before adapting it to handle RFM on DDR5. In § VI, we will adapt FEINTING to securely design PROTRR, and we will discuss how FEINTING can be further refined to handle protocol optimizations such as `REF` and `RFM` postponing, and certain DRAM architectural optimizations such as subarray parallelism (Appendix A, Appendix B, and Appendix C). Figure 3 provides a summary of these different FEINTING variations.

[1]FEINTING refers to maneuvers that distract or mislead the opponent.

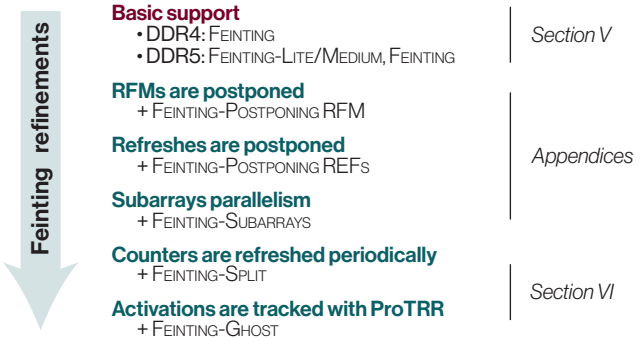

Fig. 3: **Overview of FEINTING variations. The final attack is a combination of the listed refinements, depending on the DDR technology.**

B. *FEINTING on DDR4*

We consider an attack that lasts n TRR events (*intervals*). In the last TRR event, the victim can be refreshed (as the attack ends), so no further decoy is needed. Thus the minimum number of rows hammered in the attack is $D_T = (n-1) \times V + 1$, i.e. $(n-1) \times V$ decoy rows plus the victim row. Generalizing, the victim row can be seen as the last decoy that is refreshed. We refer by $D(\alpha)$ to the number of decoy rows that have not been refreshed yet before the activation α. We define $\tilde{d}_i$ as the i-th decoy, and α_i as the moment it is refreshed.

Theorem 1 (Optimal distribution and intensity)**.** *For a generic TRR event $i \in [\![1, n]\!]$ happening after activation α_i, with $D(\alpha_i)$ decoys ($\tilde{d}_1...\tilde{d}_{D_T-(i-1)\times V}$), an attack $\mathcal{A}$ can only be optimal if all decoys' hammer count ($d_1(\alpha_i)...d_{D_T-(i-1)\times V}(\alpha_i)$) is the same.*

▸ *Intuition.* To maximize ***k*** (the activations that hammer the victim), we must minimize the total activations used to hammer decoys during the attack. Decoys should not be hammered more than the victim because this is unnecessary for the victim to survive. Likewise, a decoy that is hammered insufficient times is useless for the victim's survival. Practically, this translates to *steps* in which all the decoys and the victim increase their hammer counts together and in unison, as shown in Figure 4.

▸ *Proof.* First, we prove that no decoy should be refreshed with a hammer count higher than the victim $\tilde{x}$. We consider any TRR event i, after an activation α_i, in which a decoy $\tilde{d}_i$ is refreshed. We define Δ as the difference of hammer counts between decoy and victim: $d_i(\alpha_i) = y + \Delta$ for $x(\alpha_i) = y$. Given Definition 4, the victim already survives if $d_i(\alpha_i) = x(\alpha_i)$. This means that Δ hammerings are wasted by not spreading them equally over all remaining $D(\alpha_i)$ rows. In other words, the victim can survive with a count of $x(\alpha_i) = y + \frac{\Delta}{D(\alpha_i)}$, which creates a better attack.

Similarly, we now prove that it is not optimal to have decoys hammered less than the one refreshed at the TRR event. We

```
1 nr_intervals = 8192
2 A_T = nr_intervals*166 // T=166
3 nr_decoys = nr_intervals*V
4 aggressors = GetDifferentRows(nr_decoys/B)
5 for ACT = 1 ; ACT ≤ A_T ; ACT++ do
6 |   ACTIVATE GetLeastActivated(aggressors)
7 |   if ACT%T is 0 then // TRR event
  |   |   // remove the TRRed aggressors
8 |   |   RemoveHighest(aggressors, V/B)
```

Algorithm 1: **The pseudocode for FEINTING on DDR4.**

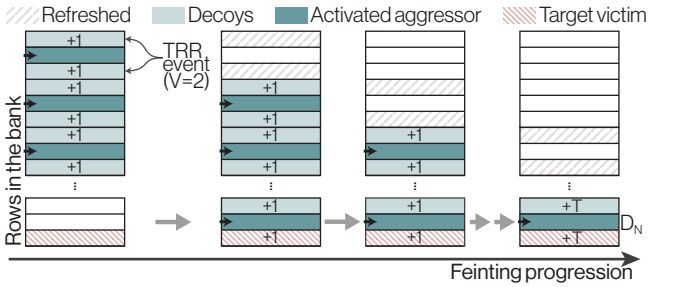

Fig. 4: **FEINTING strategy.** As the attack progresses, decoys get refreshed. In the last round, only the target victim (D_N) is left to be refreshed and all the activations are used against that victim, hammering it T times.

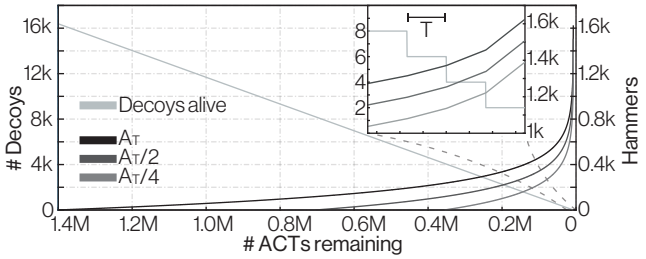

Fig. 5: **Different durations of FEINTING.** Example for DDR4, $\{V; B\} = 2$.

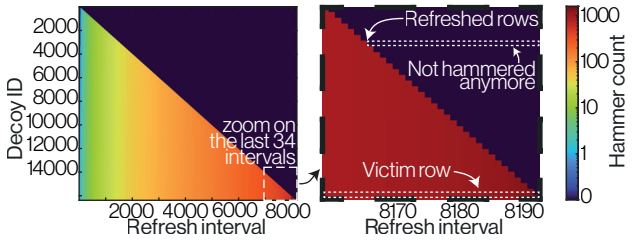

Fig. 6: **Impact of FEINTING against TRR$_{ideal}$.** Decoys' hammer count over time. After a decoy has been refreshed, it is never hammered again.

consider any TRR event i (after activation α_i) in which a decoy $\tilde{d}_i$ is refreshed. In this case, Δ is the difference of hammer counts between a lower decoy ($\tilde{d}_l$) and $\tilde{d}_i$, with $d_i(\alpha_i) = y + \Delta$ for $d_l(\alpha_i) = y$. Decoy $\tilde{d}_i$ is refreshed with an excess of hammer counts: $\tilde{x}$ would have already survived with $d_i(\alpha_i) = x_i(\alpha_i) = y + \Delta'$, where $\Delta' = \frac{\Delta \times (D(\alpha_i)-1)}{D(\alpha_i)}$. The extra hammers $(\Delta - \Delta')$ are wasted, as they could have been used to hammer decoy d_l, which has to be hammered to make the victim survive in a future interval. Concluding, the optimal distribution and intensity minimizes the difference between all decoys and the victim by hammering them in steps and in each step, in unison.

Theorem 2 (Optimal duration)**.** *Given n TRR events happening in a* `tREFW`*, an attack $\mathcal{A}$ is optimal if, given $\mathcal{A}$, $D_T = (n-1) \times V + 1$ and $L_{attk} = L_{tREFW}$.*

▸ *Intuition.* The last intervals of two attacks of different lengths are equivalent. In the last interval, in both cases, only one row survives (the victim), while in the previous interval, there were $V + 1$ rows alive (the decoys and the victim), and so on. In other words, the longer attack extends the shorter attack by more intervals. An attacker can use these extra intervals to hammer the victim and the necessary decoys. As a result, using a fewer number of intervals only leads to a lower $Hammer_{max}$.

▸ *Proof.* Independently from the attack duration, the victim is refreshed after the last interval. Thus, according to Theorem 1 attacks of lengths n_1 and n_2 (with $n_1 < n_2$) will share the same pattern for the corresponding last interval. As such, they will also share the previous intervals (i.e., $n_1 - 1$ and $n_2 - 1$) and so on, until the first $n_2 - n_1$ intervals of the longer attack. We now prove that having $n_2 - n_1 \geq 1$ is beneficial for the attack of length n_2.

Consider an attack that requires $D_T^{(1)}$ rows hammered and lasts n intervals. Adding one interval at the beginning results in hammering each row by $\Delta\epsilon = \frac{B \times T}{D_T^{(1)} + V}$ more and consequently must have V more decoys. However, because $B \times T > V$, it is beneficial for the attack to have this extra interval. Generally, for j intervals added, the victim row is increased by $\Delta\epsilon_{tot}(j) = \sum_{\phi=0}^{\phi=j-1} \frac{B \times T}{D_T - \phi \times V}$, where $D_T = D_T^{(1)} + j \times V$. Thus, the optimal duration of an attack is the maximum number of activations in a `tREFW` (i.e., L_{tREFW}), from which follows $L_{attk} = L_{tREFW}$. This covers all n TRR events in a `tREFW`, which means having $D_T = (n-1) \times V + 1$ decoys. Note that for simplicity, we consider that the available $B \times T$ hammering in each interval can be used flexibly for any victim without loss of generality. In reality, when $D(\alpha) < B$ (last interval(s)), rows are hammered at maximum T times per interval which is what consider for all the plots, evaluation, and calculations.

FEINTING. To summarize, the optimal attack (FEINTING) is an attack that starts immediately after the victim row has been refreshed internally. The attack lasts a `tREFW` (for a total of L_{tREFW} activations), where $D(\alpha)$ rows are alternately hammered once. As TRR events happen, $D(\alpha)$ decreases, up to having only the victim row in the last interval. The number of aggressors needed is $\frac{D_T}{B}$, each associated with unique B decoys. Algorithm 1 presents the implementation of FEINTING according to these three theorems, and Figure 5 shows how the increased duration of the attack allows for a higher $Hammer_{max}$. Figure 6 shows the hammer count of the victim in FEINTING over one `tREFW`.

Number of TRR events. In DDR4, a `REF` is sent every `tREFI` and may trigger a TRR event [1]. The distance d identifies after how many `REFs` one triggers a TRR event. This means that the total number of TRR events is $\frac{8192}{d}$ *regardless of the number of activations used in a* `tREFW`. In contrast, DDR5 introduces the new `RFM` command (§IV), which, depending on the number of activations, allows for a higher number of TRR events. Next, we look at the impact of `RFM` on FEINTING.

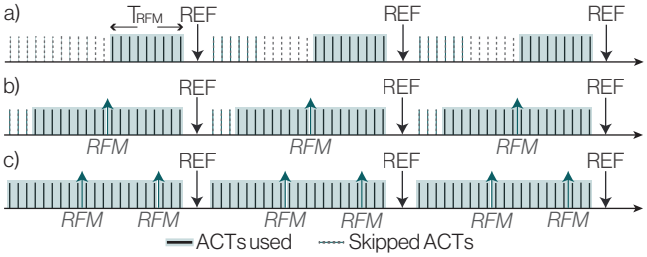

Fig. 7: **Different FEINTING strategies on DDR5.** a) FEINTING-Lite, b) FEINTING-Medium, and c) FEINTING.

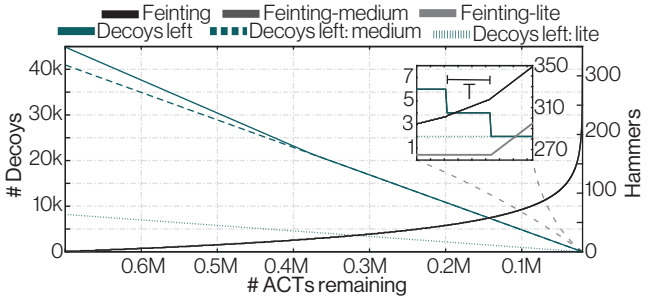

Fig. 8: **Different FEINTING strategies for DDR5.** Example for $\{V; B\} = 2$.

C. *FEINTING on DDR5*

We adapt the previous theorems to DDR5 devices. In DDR5, TRR events happen for both `REF` and `RFM`. For this reason, while keeping the previous definitions, we specify T as follows. We define T_{REF} as the number of activations between two refreshes that perform TRR, and T_{RFM} as `RAAMMT` (=`RAAIMT`$\times m$). We first consider a simple memory controller that generates `RFM` commands every `RAAIMT` activations (i.e., $m = 1$). Later, in §V-D, we relax this assumption to consider postponing `RFMs` (i.e., $m > 1$).

TRR events on DDR5. We calculate the minimum number of possible TRR events generated on a DDR5 device during a `tREFW`. This leads to the minimum number of decoys needed to perform FEINTING. Per DDR5 standard [30], a register in the device indicates whether every `REF` or every second `REF`, a TRR happens (i.e., $d = 1$ or 2). For simplicity, we denote by REF_{TRR} the `REFs` that do TRRs. Depending on d there are 8192 or 4096 REF_{TRRs} in a `tREFW`. These are the minimum numbers of TRR events that happen in a `tREFW`, without including `RFMs`. With FEINTING-Lite, we show how an attacker can perform FEINTING without ever inducing an `RFM`.

FEINTING-Lite. In DDR5, `tREFW` is 32 ms by default, which leads to $T_{REF} = 83$ ($d = 1$). Instead, the maximum value of T_{RFM} is 80. For FEINTING-Lite and the other variants to be introduced later, we always consider an optimized memory controller that does not send an `RFM` if the next command is a REF_{TRR}. For this reason, T_{RFM} activations can always be sent between two REF_{TRR} without causing an `RFM`: as the `RAA` counter becomes `RAAMMT`, it is immediately set to zero with a REF_{TRR}. The FEINTING attack is reproducible without variations by skipping $T_{REF} - T_{RFM}$ activations every REF_{TRR} (Figure 7-a): we refer to such attack as FEINTING-Lite. Because we have already proven FEINTING to be optimal, *this is the optimal attack if no `RFM` command is triggered.*

FEINTING-Medium. If multiple blocks of T_{RFM} activations can fit between two REF_{TRR}, it is straightforward to prove that FEINTING-Lite can be improved by using the complete $T_{REF} - (T_{REF} \ mod \ T_{RFM})$ activations between two REF_{TRR}. T_{REF} can be segmented into blocks of T_{RFM} activations as shown in Figure 7-b. These additional blocks increase the number of intervals used for the attack in a `tREFW`. In the case of FEINTING-Lite, exactly 8192 (or 4096) intervals are used for the attack, each of T_{RFM} activations. In FEINTING-Medium, each additional block performs T_{RFM} activations and requires V (additional) decoys: exactly as if FEINTING-Lite lasted longer. Because of Theorem 2, this strategy improves the attack. In FEINTING-Medium, between two REF_{TRR}, the remaining ($T_{REF} \ mod \ T_{RFM}$) *extra activations* are skipped. FEINTING-Medium is the optimal attack if the remaining extra activations are not used. In the last step, we analyze if it can ever be beneficial for the attacker to use these extra activations.

FEINTING. Starting from FEINTING-Medium, we evaluate if the attack can be improved by causing *some* extra `RFMs` using the remaining extra activations ($T_{REF} \ mod \ T_{RFM}$) between two REF_{TRR}. There is a cost attached when using these extra activations: every extra `RFM` triggered increases the number of decoys needed by V. The attacker needs to use activations to hammer these additional decoys. Unfortunately, these additional decoys are less impactful than the others since they add fewer activations to the attack. This leads to the following question:

> *Considering "FEINTING-Medium", when is it optimal for an attacker to use the extra activations that cause `RFM`?*

Theorem 3 (Optimal number of extra `RFMs`)**.** *If using extra activations ceases to be beneficial for an attacker, then it can never become beneficial again in the same attack.*

Corollary. *If using extra activations at the beginning of the attack is not useful, then it will never be.*

> ▸ *Intuition.* Extra activations will trigger more TRR events, requiring more decoys to be hammered during the attack. As time passes, these decoys must be hammered (Theorem 1) until they are finally refreshed by the extra `RFM`. This can be seen as an expense for the attacker. From an attacker's point of view, it is less expensive to trigger the extra `RFM` earlier, so that the accumulated cost of hammering these decoys is lower.

▸ *Proof.* For simplicity, let us assume that only one full T_{RFM} fits between two REF_{TRR}. We consider two cases that are identical up to REF_{TRR} interval $i - 1$ with victim count $x(\alpha_{i-1}) = y$. Case (1): the attacker uses $\epsilon = T_{REF} - T_{RFM}$ extra activations in the interval i. Case (2): the attacker skips interval i as using extra activations is not useful, and then, in the next interval $i + 1$, these ϵ activations become useful and are used for the attack. We now prove that case (2) is impossible. We start by evaluating the victim hammer count in the two cases, summing the different contributions:

$$x^{(1)}(\alpha_{i+1}) = y + \frac{T_{RFM}\times B}{D(\alpha_i)+V} + \frac{\epsilon\times B}{D(\alpha_i)} + \frac{T_{RFM}\times B}{D(\alpha_i)-V}$$
$$x^{(2)}(\alpha_{i+1}) = y + \frac{T_{RFM}\times B}{D(\alpha_i)+V} + \frac{T_{RFM}\times B}{D(\alpha_i)} + \frac{\epsilon\times B}{D(\alpha_i)-V}$$

We can evaluate when $x^{(1)}(\alpha_{i+1}) > x^{(2)}(\alpha_{i+1})$. This results in $T_{RFM} > \epsilon$ which is always true. Therefore, case (2) can never be more optimal than case (1). This means that is not possible that using the extra activations ceases to be useful in one interval and becomes useful again in a later interval. Likewise, if case (1) was not useful, case (2) would also not be useful. By induction, it cannot become useful in the future: if i is not useful and $i+1$ is not useful, $i+2$ will also not be useful, and so on. Concluding, the attacker can calculate when to stop inducing extra RFMs, deriving the best possible FEINTING. Figure 8 shows the effectiveness of different FEINTING strategies on DDR5. These results show that while FEINTING-Medium improves the attack compared to FEINTING-Lite, in the case of $\{V; B\} = 2$ the improvement of the last optimization does not result in a higher $Hammer_{max}$.

D. FEINTING on DDR5 with RFM postponing

More sophisticated memory controllers may issue RFM commands irregularly, i.e., not always precisely after RAAIMT activations. However, it must never be after $T_{RFM} = m \times$ RAAIMT (i.e., RAAMMT) activations. In case that $T_{RFM} > T_{REF}$, FEINTING can be improved if we assume that the attacker can influence the scheduling of RFM commands. The idea is to leverage extra activations gained by postponing RFMs to build blocks of RAAIMT activations. This causes the RAA counter to increase quickly, and at some point, the memory controller will have to issue multiple, previously postponed RFM commands. It is optimal for the attacker if the L_{tREFW} activations are equally distributed over intervals of size RAAIMT, similarly as for FEINTING-Medium. In the last few intervals, postponed RFMs can be sent after the tREFW, as such, allowing the attacker to further increase the count of the decoys (needed for REFs) and victim in these intervals without causing RFMs. Furthermore, in this setting, the attacker requires fewer decoys since fewer RFMs are issued during the attack. We refer to Appendix B for more details.

VI. PROTRR

An ideal TRR mechanism (TRR_{ideal}) requires a large amount of storage. For example, a single-rank module with 16 banks/rank and 16 bit row addresses needs in total 14 MiB (R_{thresh} = 5 K). Mitigations deployed in the memory controller can use known optimized data structures to detect when a potential victim row reaches a specific threshold. Once this happens, these mitigations can delay the execution of normal DRAM operations to refresh this victim row [16], [19], [20], [45]. As already explained (§V), it is not possible for in-DRAM mitigations to delay DRAM requests due to the synchronous nature of the DRAM protocol.

Park et al. [19] use Misra-Gries summaries [31] that provide deterministic guarantees of finding the most frequently activated (aggressor) rows [49]. Misra-Gries summaries are proven to be optimal in the number of counters they need for detecting frequent items. Unfortunately, these summaries cannot be directly applied to the in-DRAM setting. First, Misra-Gries provides guarantees of finding frequent items occurring more than a fixed threshold in a stream with a specific length. However, an in-DRAM mitigation must protect V rows with the highest count at *any TRR event* without a fixed threshold. It is unclear how many counters are necessary to provide similar guarantees in PROTRR. Secondly, in a proactive in-DRAM setting, the counters of refreshed rows must reset while processing the stream, which is not considered in Misra-Gries.

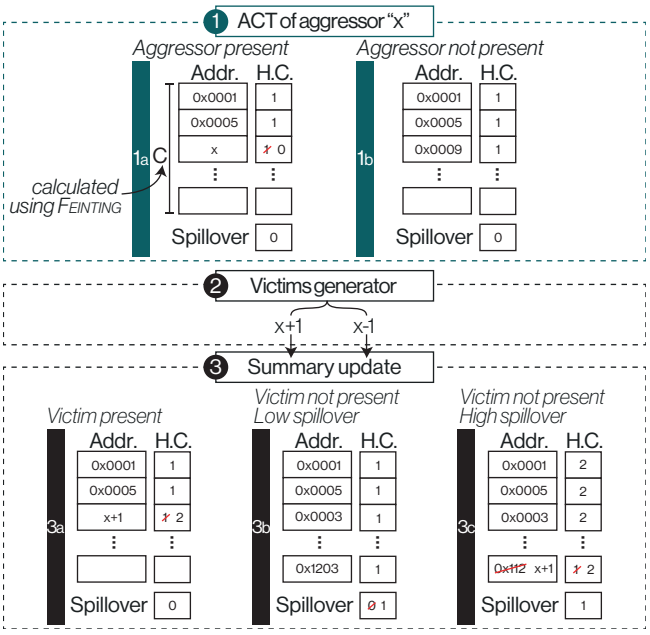

Fig. 9: **Victim counting in PROTRR.** Once a row is activated (1), if its address is contained in the summary it is pruned (1a). Then, the aggressor blast diameter is considered (2, for e.g. $B = 2$) identifying the victim rows. The victim rows are compared with the summary's content, which is updated accordingly (3a,3b,3c).

Our proposed in-DRAM Rowhammer mitigation, PROTRR, uses a new frequent item counting scheme for in-DRAM operation, called PROMG (Proactive Misra-Gries). PROMG operates similarly to the Misra-Gries scheme, but is designed to function in the in-DRAM scenario. In the followings, we show how PROMG is similarly optimal in the number of required counters by leveraging the bounds given by FEINTING. Furthermore, we show how PROMG enables PROTRR to provide an optimal trade-off between the number of required counters and additional refreshes – given a DRAM device with a specific R_{thresh}.

A. Design of PROTRR

PROMG is a proactive version of Misra-Gries summaries with two crucial differences. First, PROMG needs a different number of counters than the original Misra-Gries since it needs to make proactive decisions. We later show how FEINTING can be used to right-size PROMG summaries. Second, PROMG supports pruning entries from its summaries.

Similar to Misra-Gries, a PROMG summary is a table of < ID, count > pairs and a *spillover* counter. Conceptually, the spillover counter represents the upper bound of counts for all rows that are currently not in the summary. For every input,

its ID is compared with all existing table entries; if there is a match, the associated counter is increased. Otherwise, the spillover value is compared with the lowest counter, and if the former is equal to or higher than the latter, the new input replaces that entry and its counter is increased. If every entry has a higher count than the spillover, the spillover is increased. Unlike Misra-Gries, in PROMG, a row that is either activated or refreshed is pruned from the summary, and its victim rows are treated as summary inputs.

Figure 9 shows how PROTRR makes use of PROMG. On each activation, PROTRR updates its summary accordingly by incrementing counters that are associated with victim rows of the activated row. At each TRR event, PROTRR refreshes the V rows with the highest counters in the summary.

Right-sizing the PROMG summary in PROTRR. In the original Misra-Gries scheme, given C counters and an input stream of size L, any entry occurring more than $\frac{L}{C+1}$ times will be included in the summary [19]. In contrast, PROTRR uses PROMG to make proactive decisions without reaching a threshold. To do this securely, we need to find the right number of PROMG counters for PROTRR to be secure against FEINTING. Furthermore, every row will be refreshed in a `tREFW` which we also leverage in PROTRR to ensure that the counters do not grow unbounded. To do this securely, however, we have to adjust the bounds given by FEINTING. We now prove theorems that shows how PROTRR right-sizes PROMG considering these observations.

Theorem 4 (FEINTING optimality against PROTRR)**.** *If the amount of TRR events in an attack is n, given PROTRR with $C = (n-1) \times V + 1$ counters in the summary (excluding the spillover), FEINTING is the optimal attack against PROTRR.*

Corollary. *Given $Hammer_{max}$ obtained with FEINTING for fixed (V, B and n TRR events) and considering PROTRR with $C = (n-1) \times V + 1$ counters (excluding the spillover), PROTRR protects any device less vulnerable than $Hammer_{max}$, i.e., where the Rowhammer threshold $R_{thresh} > Hammer_{max}$.*

▸ *Proof.* Given that $C = (n-1) \times V + 1$, PROTRR behaves exactly like an ideal counter against FEINTING. Therefore, an attacker is able to reach $Hammer_{max}$ as described earlier. We now prove that an attacker forcing the replacement of rows in the summary due to the limited number of counters does not increase $Hammer_{max}$. A replacement happens if a row $\tilde{d}_s$ that is not in the summary is hammered, and the spillover is equal or higher than the minimum count of the summary (row $\tilde{d}_t$). The replacement increases the counter that now refers to $\tilde{d}_s$. The effect on the attack is equivalent as if $\tilde{d}_t$ had been hammered, since for the victim to survive, it does not matter which decoy is TRRed. Note that the replacement can only happen if more than C decoys have already been hammered; otherwise, $\tilde{d}_s$ is added to the summary. Moreover, because $C = D_T$, all the decoys necessary for the attack have already been hammered. Therefore, these replacements cannot improve the attack.

Resetting. Over time, the counters can grow unbounded, thus requiring unlimited storage to avoid overflows. This does not reflect reality where every row is refreshed at least once in a `tREFW`. To handle this, PROTRR resets the entire summary once every `tREFW`. The refresh of a given row, however, is not necessarily synchronized with the summary reset. This loss of information about the past `tREFW` allows an attacker to perform FEINTING across a reset, thus changing the supported R_{thresh}. We address this in Theorem 5.

Theorem 5 (Non-linearity of FEINTING)**.** *In the presence of a summary reset, two independent and shorter back-to-back FEINTING result in a higher $Hammer_{max}$ than a longer one.*

▸ *Intuition.* FEINTING starts after the victim row has been regularly refreshed (REF_I) to maximize the activations available for the attack (L_{tREFW}). However, during the attack, the summary could reset, leading to an information loss that can be exploited to increase $Hammer_{max}$. For example, two attacks of (each) 4096 intervals require half of the decoys than one attack lasting 8192 intervals but using the same number of activations, allowing the victim to be hammered more.

▸ *Proof.* We define the baseline as case (1): FEINTING lasting n intervals, never crossing a summary reset. The number of times the victim will be hammered by the end of these intervals is denoted by $x^{(1)}(\alpha_n)$. In case (2), we consider a summary reset happening σ intervals after FEINTING has started (with $\sigma < n-1$). Once the summary resets, it becomes empty, and a new FEINTING can be initialized, lasting the remaining $i = n - \sigma$ intervals. The cumulative number of times the victim is hammered, after n intervals is $x^{(2)}(\alpha_n)$. We compare these two cases. In case (2), the number of hammers to the victim is obtained by two different contributions, the first attack (σ intervals) and the second attack ($n - \sigma$ intervals): $x^{(2)}(\alpha_n) = \sum_{\phi=1}^{\sigma} \frac{B \times T}{\phi \times V + 1} + \sum_{\phi=0}^{i-1} \frac{B \times T}{1 + \phi \times V}$. Instead, case (1) consists of only one attack: $x^{(1)}(\alpha_n) = \sum_{\phi=0}^{n-1} \frac{B \times T}{1 + \phi \times V}$. The second part of case (2) overlaps with the start of case (1), i.e., their contributions are equal — a direct consequence of Theorem 2. The first part of case (2) is larger than the sum of the σ last terms in case (1), which proves the non-linearity.

Corollary (FEINTING-Split)**.** *Given a summary reset every `tREFW`, two balanced ($\sigma = \frac{n-1}{2}$), independent back-to-back FEINTING attacks are optimal.*

We proved that if $\sigma < n - 1$, it is always better for the attacker to have two distinct and independent FEINTING. Now we prove that the optimal condition for the attacker is when there are two equally long attacks. We start by showing the effect of moving an interval from the attack's second part (i.e., last $n - \sigma$ intervals) to the first part (i.e., first σ intervals). The reader will remember that the sum of the two intervals is fixed by n. Moving an interval from the second to the first part is beneficial for the attacker when $\frac{B \times T}{\sigma \times V + 1} \geq \frac{B \times T}{1 + (i-1) \times V}$ leading to $(i-1) \geq \sigma$. Given that $i = n - \sigma$, it follows that the best case for the attacker is when $\sigma = \frac{n-1}{2}$. Because

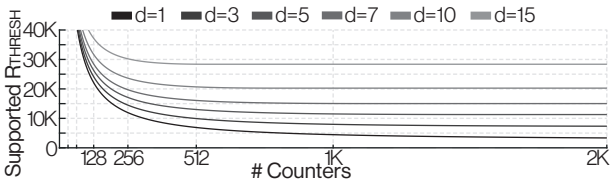

Fig. 10: **Flexibility** in PROTRR. Example for DDR4 (B=2, V=2, tREFW =64 ms). For a fixed storage, TRR distance (d) can be used as trade off.

PROTRR implements summary refresh, we have to consider this adaptation of FEINTING, which we refer to as FEINTING-Split, when right-sizing the PROMG summary. Before finalizing FEINTING-PROTRR, we add flexibility to PROTRR.

B. Optimality and Flexibility

Depending on the DRAM technology, a vendor may afford a maximum number of TRR events (N) to be performed in a tREFW and a certain number of counters (C) to keep track of victim rows. We design PROTRR to be flexible: given any pair of (N, C), the maximum vulnerability protected can be obtained using FEINTING. A DRAM vendor, knowing the R_{thresh} for its own devices, can decide to change N or C as needed. Furthermore, we show that for any given (N, C, R_{thresh}), PROTRR is optimal: there exists no other deterministic in-DRAM TRR that is secure against FEINTING with a smaller number of TRR events than N. Similarly, for a given R_{thresh} and N, the number of counters C is optimal. We first show how PROTRR achieves flexibility and optimality for N, and then we discuss the same for C.

Flexible and optimal TRR events. The bounds given by FEINTING enable vendors to calculate the required TRR events (N) in a tREFW for a device-specific R_{thresh}. The following theorem shows that N is optimal for a given R_{thresh}.

Theorem 6 (TRR events optimality)**.** *For a supported R_{thresh}, PROTRR is optimal in the number of TRR events needed.*

To defend $R_{thresh} = Hammer_{max} + 1$ against FEINTING, the device requires at least $\frac{D_T - 1}{V} + 1$ TRR events in a tREFW. If a smaller number of TRRs are employed, then the decoys for FEINTING will be fewer, and $Hammer_{max}$ will exceed R_{thresh}. Hence, the number of TRR events is optimal. This feature of PROTRR provides it with flexibility on the number of TRR events. We can reduce the number of TRR events if a device has a high R_{thresh}. In practice, a manufacturer can tune the number of TRR events using the distance d (§V-B). This enables configurability of PROTRR according to the DRAM vendors' needs. Figure 10 shows how PROTRR can support devices with different R_{thresh} by appropriately choosing d. We now show how PROTRR provides further flexibility in the number of required counters.

Flexible and optimal number of counters. For a given R_{thresh}, FEINTING gives us the optimal number of TRR events. It follows that D_T counters are needed. Given that Misra-Gries summaries are space-optimal [49], using D_T counters will be optimal against FEINTING. For more flexibility, we show how PROTRR can reduce this number of counters with a slight increase of R_{thresh}.

FEINTING-Ghost. We adapt FEINTING to handle cases where PROTRR has a limited storage, providing a trade-off between the supported R_{thresh} and the number of counters in the summary. With reduced storage, an attacker engaged in FEINTING can create ghost decoys by first saturating the number of counters. Theorem 7 proves the optimal number of decoys for this modified attack.

Theorem 7 (FEINTING-Ghost optimality)**.** *For PROTRR with $C < (n-1) \times V + 1$ counters, where n is the number of TRR events in a tREFW, FEINTING-Ghost with $C + 1$ decoys is optimal.*

▸ *Proof.* We assume $C < D_T$ and prove that $C + 1$ is the maximum number of decoys needed. After C decoys are hammered, the summary is full, and the next (new) hammered decoy turns the spillover counter to one. Now, the rows that are not in the summary are considered already hammered once (i.e., *ghost* decoys) – thus reducing the number of hammers for maintaining them. Likewise, after the next $C + 1$ hammers, each row will be considered hammered twice, and so on. This condition persists until the number of decoys is C. From this point on, all hammers target rows present in the summary, and the attack is the same as the original FEINTING.

Theorem 8 (Counters optimality)**.** *For a supported R_{thresh}, given a number of TRR events, PROTRR is counter-optimal.*

▸ *Proof.* If we remove one counter (i.e., $C - 1$), there would be a ghost decoy for which an attacker does not need to waste activations until there are only $C - 1$ alive decoys left. These extra activations could be used to further increase the victim (and decoys) to exceed R_{thresh}. Hence, the number of counters needed in PROTRR is optimal. Figure 10 shows how this allows PROTRR to massively reduce the number of counters needed, marginally increasing R_{thresh} in most settings.

FEINTING-PROTRR. Summarizing, the optimal attack against PROTRR is the adaptation of FEINTING given two new conditions: summary reset and limited number of counters. We define this attack as FEINTING-PROTRR, which is the implementation of FEINTING-Split, where each part is performing FEINTING-Ghost. We consider $Hammer_{max}$ achieved by FEINTING-PROTRR in different settings in our evaluation in §VII.

C. Implementation of PROTRR

We implemented PROTRR in a popular 12 nm ASIC technology, to confirm its feasibility. In our evaluation (§VII), we assessed the supported vulnerability for the number of counters implemented in current mitigations. Our design, depicted in Figure 11, uses a *decoder* logic (❶) to distribute simple micro-operations over several clock cycles. The *entries update logic* (❷) performs the summary update and, depending on the given micro-operation (❸): removes a row after it has been refreshed *(REF request)*, increases the counters of a victim *(Blast request)*, or resets the summary *(Clear request)*. Within the same cycle, two parallel combinational circuits *(min/max reduction*, ❹a and ❹b*)* determine the rows with the lowest and highest counts for the next summary update (❺a and ❺b). We

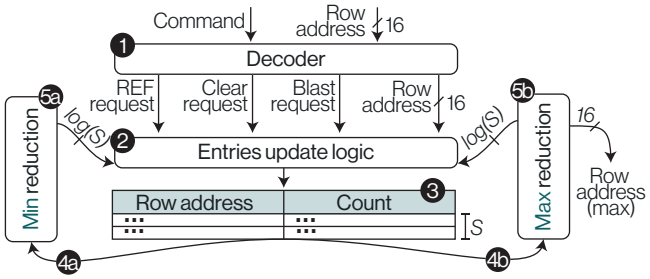

Fig. 11: **PROTRR's ASIC design**. Schematic of PROTRR's mechanisms.

implemented the summary as a standard cell memory to get simultaneous access to all its elements for the reductions.

Integration and placement. PROTRR can replace existing counter-based, in-DRAM TRR schemes [2], [50], [51]. Typically, control logic (excluding array decoders, Figure 1) is placed in the center of the DRAM chip, while the rest of the area is devoted to the DRAM cell blocks [52]–[56]. Instead, for LPDDR devices, the control logic is placed on an edge pad. We received confirmation from a DRAM vendor that the TRR mechanism is placed in the peripheral logic which is part of the control logic. They also confirmed that it is feasible to implement 2K counters in this area in an older technology than the one used by PROTRR. While this is enough for almost all settings we considered in §VII, more recent process technologies are capable of implementing more counters if needed.

VII. EVALUATION

In this section, we present an extensive evaluation of PROTRR. We consider three key aspects that we assess for both DDR4 and DDR5: the impact on performance, storage requirements, and energy consumption. We show that PROTRR is lightweight, incurs negligible energy and performance overhead, and is practical for real-world deployments.

In § VII-A, we show PROTRR's flexibility in supporting different device constraints with a varying number of required counters. To estimate the performance and energy overhead, we run the SPEC2017 benchmark suite [57], as described in §VII-B. We run the benchmarks using full system simulation, allowing us to evaluate the impact of PROTRR under real-world conditions. Additionally, even though PROTRR provides formal guarantees, we verified its implementation against state-of-the-art Rowhammer fuzzers [1], [3] and FEINTING (§VII-D). We provide a confirmation of PROTRR's feasibility with our ASIC implementation (§VII-C), and lastly, we test FEINTING against real DDR4 devices (§VII-E). We point out that PROTRR is the first Rowhammer mitigation that is compatible with the latest DDR standard (DDR5), and this is the first work that evaluates the impact of `RFM`.

A. Storage size and supported vulnerability

The required storage of PROTRR is derived from the number of banks (N_{banks}) and the size of each summary. A summary contains entries ($S_{entries}$), each consisting of a row address and a counter. We consider 16-bit addresses, and $\log_2(Hammer_{max})$ bits for the counter. The total size in bits is $N_{banks} \times S_{entries} \times (16 + \lceil \log_2(Hammer_{max}) \rceil)$.

TABLE I: Hardware settings and DRAM geometries of our gem5 simulations.

CPU *(OoO)*		Memory Controller		DRAM	DDR4 2933	DDR5 4800
Cores	8	Channels	2	Ranks	1	1
CPU Freq.	3 GHz	Page Policy	Open Page	Bankgroups	4	8
L1D/I Cache	32 KiB	Scheduling	FR-FCFS	Banks/Group	4	2
L2 Cache	256 KiB	Queue Structure	Per Bank	Banks/Rank	16	16
L3 Cache	8 MiB	Total Capacity	16 GB	Rows/Bank	$64K$	$64K$

Figure 12 presents the storage size of different DDR4 and DDR5 settings based on the geometries given in Table I. These figures show the required size per rank to support varying levels of device vulnerability to Rowhammer in different setups. The blast diameter of 4 incorporates devices subject to the recently discovered *half-double* attack [34]. These results illustrate how storage can flexibly be traded-off by a higher refresh rate, a lower TRR distance, or RFM postponing in DDR5.

DDR4. We consider a `tREFW` of 64 ms and 32 ms with a TRR volume of 2, for blast diameters of 2 and 4 (Figure 12-a). We also indicate the highest vulnerability degree as reported in previous work [21]. We make two observations using these results: (i) Devices that use LPDDR4 with 64 ms of `tREFW` can no longer be protected against the half-double attack with any possible integrated in-DRAM solution. We need to increase the refresh rate to 32 ms to be able to protect these devices with PROTRR. (ii) The TRR distance has a significant impact on the supported vulnerability. Due to the lack of RFM support in DDR4, this suggests that a TRR distance of one is required for newer process technologies. In Appendix F, we present the same analysis for a TRR volume of 4.

DDR5. Figure 12-b shows the required storage size for DDR5 for the worst possible case with `RFM` postponing of 6. We refer to Figure 20 (Appendix F) for more details. We make the following observations: (i) Thanks to the `RFM` extension, PROTRR can protect DDR5 devices with drastically lower Rowhammer thresholds. (ii) Lowering `RAAIMT` only marginally increases the offered protection, suggesting that the current set of possibilities in the latest JEDEC standard [30] is suboptimal. (iii) All the possible setups can protect against the most recently discovered half-double patterns.

B. Performance and energy overhead

Methodology. We evaluate PROTRR on the *SPEC® 2017* [57] benchmark suite to assess its performance and energy overhead in real-world workloads. We follow the benchmark's guidelines and run each benchmark with eight parallel copies (i.e., number of cores) to maximize the simulated load. We use *gem5* [58], a cycle-accurate hardware simulator, in conjunction with *DRAMsim3* [59], a cycle-accurate memory controller. We implemented PROTRR in DRAMsim3, and due to the lack of publicly available DDR5 simulators, added DDR5 support to DRAMsim3, including the new `RFM` command. For benchmarking, we use the full system simulation mode of gem5 to run Ubuntu 20.04 with the Linux kernel 5.4.49. We follow the SMARTS methodology [60] to obtain 20 equally-spaced checkpoints, each running 10 M instructions, for a total of 200 M instructions in line with previous work [20], [21].

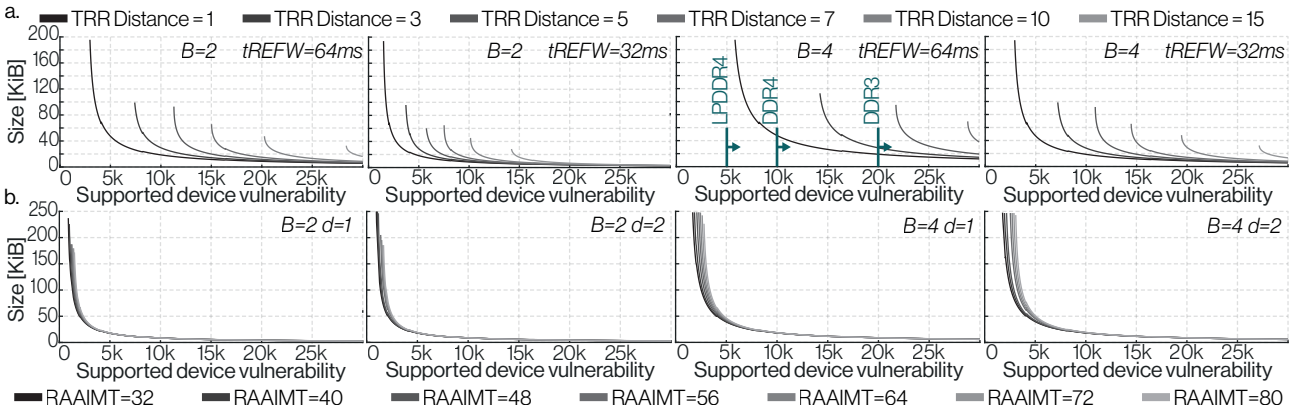

Fig. 12: **Storage size of PROTRR, per-chip values.** The green arrows indicates the worst device vulnerability taken from Kim et al. [21]. **a. DDR4 storage size.** $V = 2$. **b. DDR5 storage size.** $m = 6$, $V = 2$. The results for DDR4 with tREFW = 32 ms also apply for DDR5 without RFM support.

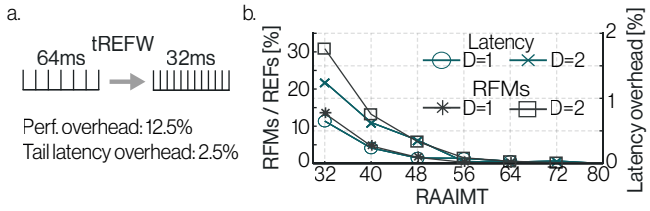

Fig. 13: **Average performance impact on DDR4 and DDR5. (a) DDR4.** Performance and tail latency overhead with tREFW = 32 ms. **(b) DDR5.** Left: percentage of RFM, relative to REFs in a tREFW. Right: tail latency overhead.

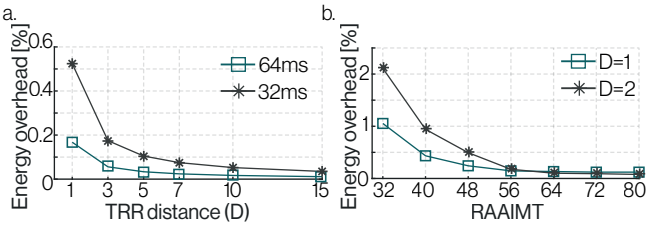

Fig. 14: **Average energy impact on DDR4 and DDR5. (a) DDR4.** Energy overhead of TRRs performed during REF. **(b) DDR5.** Energy overhead due to TRRs performed during REF and RFM.

The simulated hardware setup is listed in Table I. The results are relative to a baseline, which was obtained by running the benchmarks without any active mitigation. The simulations consider varying (a) TRR volumes, (b) TRR distances, and (c) tREFW durations. As recommended by JEDEC [61] to help against Rowhammer, we assume that the REFs cannot be postponed. We still show in § VIII how PROTRR can support postponing REFs. We configure the memory controller to immediately send an RFMsb upon reaching RAAIMT, which is the worst-case scenario for performance. Note that our performance and (dynamic) power measurements are independent of R_{thresh}. A vendor should select the correct TRR distance, tREFW (in case of DDR4), and RAAIMT (in case of DDR5) according to the R_{thresh} for their device. In contrast, the implementation-dependent area and static power overhead depend on R_{thresh} which we report in §VII-C using our ASIC implementation.

Performance. In DDR4, TRRs happen only during REF without any performance overhead. However, as discussed, a default tREFW of 64 ms may not provide adequate protection with low Rowhammer thresholds. For this reason, we evaluated the impact of changing tREFW to 32 ms (Figure 13-a). This not only reduces the time window available for an attack but also increases the frequency of internal TRRs. The result is an average CPI (cycles-per-instruction) overhead of 12.5% while increasing the tail latency of DRAM accesses by 2.5%[2].

In DDR5, TRR events still happen during REF, but, if required, the new RFM command is sent, potentially introducing overhead. To analyze the RFM's impact, we tested all possible combinations of RAAIMT and TRR distances. In all scenarios, the performance overhead is always negligible, never exceeding 0.2%. To better understand the impact of RFM, we present the percentage of RFM compared to REF commands and the increasing tail latency with varying RAAIMT and TRR distance in Figure 13-b (for more details, see Appendix F). We make two observations: (i) For small RAAIMT numbers, we require a substantial number of RFM commands (30.87% increase compared to the baseline REF in the worst case). These RFM commands, however, do not alter the instruction throughput (i.e., CPI) due to the parallelism offered by the out-of-order CPU cores and bank-level parallelism offered by RFM. In DDR4, the REF is a *per-rank* command, blocking the entire rank and substantially increasing the overhead when moving from tREFW of 64 ms to 32 ms. (ii) While CPI (i.e., instruction throughput) remains mostly unaffected, RFM does increase the tail latency of DRAM accesses (1.25% in the worst case).

Energy. We analyze the energy impact of the additional refreshes during TRR events. For each benchmark, we calculate the energy consumption as a sum of the device's plain energy, the energy of the TRRs performed during REF commands, and the energy consumed by RFM commands. To estimate the energy of these extra TRR refreshes, we calculate the energy required to refresh a single row and multiply it by the volume.

Figure 14-a reports the energy overhead of PROTRR in DDR4 for a tREFW of 64 ms and 32 ms with varying TRR distance between 1 and 15. Figure 14-b shows the energy overhead of DDR5 for different RAAIMT and the two possible TRR distances. We make the following observations: (i) The energy overhead in DDR4 is always below 0.6% of the device's total energy. (ii) The energy overhead in DDR5 is generally

[2]Considering DRAM accesses that take longer than 200 cycles.

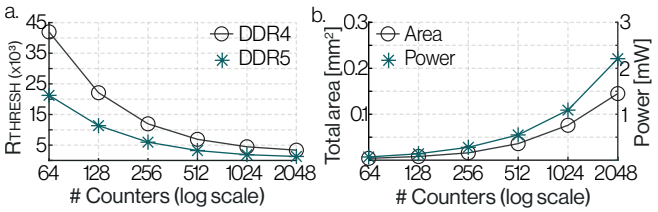

Fig. 15: **PROTRR feasibility, per-chip values. (a)** Required number of counters for different R_{thresh} in DDR4 (tREFW = $64ms$, d = 1) and DDR5 (T_{RFM} = 32, d = 1). **(b)** ProTRR ASIC costs in terms of total area and power consumption.

higher than for DDR4 due to the additional TRRs. However, this is still relatively small and at 2.11% in the worst case. (iii) In DDR5, given the same number of activations, for a TRR distance of 2, a higher number of RFM must be sent to compensate, increasing the energy overhead.

C. Feasibility

We implemented PROTRR in ASIC, using a popular 12 nm technology and the Synopsys Design Compiler. Figure 15-b reports the total area required and power consumption and Figure 15-a shows the R_{thresh} that PROTRR can protect for the number of counters. As results show, having more counters than 1024 does not substantially increase security; therefore, we consider it as the worst-case scenario. We designed the ASIC such that all updates (including lookups) are faster than the time between two consecutive ACTs, allowing PROTRR to execute in parallel. In particular, the operations require V cycles during a refresh, and B+1 cycles during an ACT. We considered 45 ns as the minimum time between two activations, as previously reported [19].

Static power. Although previous work showed that the energy consumption for the mitigation logic is negligible [19], [20], we evaluated it for completeness. The overhead for 1024 counters is at maximum 17.44 mW for 16 banks, obtaining a total of 139.52 mW for 8 chips. This is in line with previously reported values [20]. For a baseline static consumption of 3 W/8 GiB [62], this leads to 4.65% static power overhead. However, given the current technology and consumer DDR4 chips, 512 counters are enough to ensure protection in the worst cases, leading to 2.35% static power overhead.

Area. Chips area depends on process technology, fabrication, and the array size. For our analysis, we consider a common density for DDR4 devices, 0.247 GB/mm^2 [63]. For a chip that uses 16 banks and 1024 counters per bank, this leads to a maximum area overhead of 3.7%. Unfortunately, currently deployed TRR mechanisms are kept secret and there is no open DRAM implementation that can integrate PROTRR. For this reason, to further confirm the feasibility of PROTRR, we contacted a DRAM manufacturer. We obtained confirmation that (i) up to 2K counters have already been deployed in the past, and (ii) given PROTRR's specifications (dimension, as obtained from results), it is reasonable to deploy it.

D. Correctness

We tested PROTRR against FEINTING to check its implementation by running PROTRR in DRAMsim3 with memory traces.

TABLE II: **Result of FEINTING on three DDR4 devices.** We report the best attack's parameters (**Best Params.**) as: attacks duration (in tREFIs), TRR distance, and number of victim hammer repetitions.

DIMM	Mf. Date (yy-ww)	Size (GiB)	Freq. (MHz)	Geom. #R., #B.	Best Params.	Bit Flips Observed
$\mathbb{D}_0$	20-03	8	2666	1, 16	2048, 9, 1	✔
$\mathbb{D}_1$	20-06	32	2666	2, 16	2048, 9, 3	✔
$\mathbb{D}_2$	20-10	8	2400	1, 16	8192, 9, 4	✔

In all the cases with a correct configuration, PROTRR could withstand FEINTING. Instead, in cases where PROTRR was improperly configured, FEINTING could successfully trigger bit flips. We also generated traces from two state-of-the-art Rowhammer fuzzers [1], [3] and executed them against PROTRR for three days without observing any bit flip.

E. FEINTING on real devices

FEINTING assumes a mitigation that counts every activation with an adequate number of counters. Existing TRR schemes are not ideal and may employ multiple concurrent mechanisms to catch aggressors, some completely different from PROTRR [2]. However, we were still interested to see if FEINTING is able to generate bit flips on devices with a deployed counter-based mitigation. To evaluate this, we acquired three DDR4 devices from the same manufacturer previously reported to use a counter-based mitigation [2] (see Table II).

We conducted our experiments on an Intel i7-8700K running on Linux with kernel 4.15.0. We adapted FEINTING based on insights from [2] as follows: we assumed that counters could track at most 16 rows (i.e., 18 decoys needed as part of FEINTING-Ghost), and systematically tested different attack durations (2048× up to 32768× tREFI) as a row could be refreshed multiple times in a tREFW. We tested different TRR distances (1 up to 9) and victim hammer repetitions (1 to 4) while assuming 5 hammering repetitions for decoys.

In Table II, we show the results of running FEINTING-Ghost on our acquired DDR4 devices. An attack trace can be seen in Figure 16, where the duration is 8192× tREFI. Our results show that with minor adaptations, we could successfully trigger bit flips on all three devices using FEINTING. Further, we can see that an attack duration shorter than a tREFW and hammering the victim fewer times (e.g., one time for $\mathbb{D}_0$) can be beneficial because sampling may happen only at specific times as reported in previous work [1], [3]. We tested Blacksmith [3] on the same devices, which could trigger bit flips on all of them, while TRRespass [1] failed to obtain bit flips on DIMM $\mathbb{D}_0$.

VIII. DISCUSSION

We discuss how PROTRR can (i) be adapted to handle postponing and pulling-in of refresh commands, (ii) obtain better bounds by using subarray parallelism, and (iii) generalize to other, yet unknown, Rowhammer effects.

Postponing and pulling-in of REFs. The DDRx standard gives some flexibility in terms of REFs by allowing REF postponing and pulling-in. Attackers can exploit postponing to maximize TRR-free REFs, which reduces the number of decoys needed for both DDR4 and DDR5. For DDR4, the victim can be hammered more often than before, but for DDR5

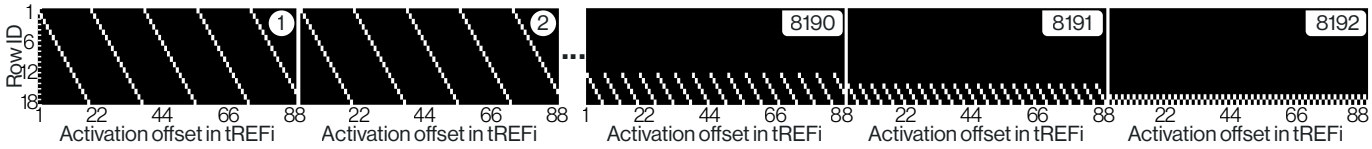

Fig. 16: **Trace of FEINTING-Ghost against DDR4 samples with 16 counters. The attack duration is 8192 `tREFI`.**

nothing changes due to `RFM` still being sent. A more detailed analysis of `REF` postponing/pulling-in is given in Appendix A.

Subarray parallelism. Subarrays enable a bank to refresh multiple rows at each `REF`. PROTRR can potentially leverage this to perform more TRRs when necessary. We provide a detailed description of how FEINTING can be adapted to subarray parallelism in Appendix C. In summary, each bank can perform multiple TRRs at the same time, effectively increasing V. However, the additional TRRs cannot target any row as each subarray can still only refresh V rows at any given TRR event. An adapted FEINTING can exploit this limitation by reducing the number of required decoys to create the optimal attack.

Generalization. FEINTING provides the basis to configure PROTRR to protect against Rowhammer. The only (implicit) requirement for FEINTING is knowing the interaction between an aggressor and its victims. In the case of a basic Rowhammer attack, which we originally considered, an aggressor activation interacts with its direct neighboring victim rows. With new Rowhammer effects, FEINTING should be adjusted to consider new interactions between aggressors and victims. We discuss two of these cases next.

During the development of PROTRR, the half-double pattern was disclosed by Google researchers [34]. To also protect against it, we only needed to consider that on certain devices, an aggressor activation can also interact with victim rows that are more than one row apart (i.e., up to B rows). Currently, a rigorous characterization of the half-double effect is missing. For this reason, we assumed the worst-case in the design of PROTRR, i.e., the same effect on every row in the blast diameter. Once this relationship is better understood, future research can adapt FEINTING accordingly to derive the optimal version of PROTRR when $B > 2$.

Another concurrent discovery shows that rows that are kept active can also influence adjacent rows [64]. PROTRR can easily generalize to this case as well. The only new requirement is to increase the counter for victims of the (aggressor) row that remains active. It remains unclear, however, whether simply keeping a row active is more effective than using the time for additional hammering which should be characterized more rigorously in the future.

IX. SECURITY ANALYSIS OF EXISTING SCHEMES

We first discuss a general limitation when mitigating Rowhammer outside of DRAM. We then present our security analysis of state-of-the-art hardware mitigations, which resulted in the discovery of novel vulnerabilities in four earlier proposed schemes [16], [19], [45], [65].

Internal row remapping. Previous work has observed that bits can flip in rows that are not adjacent to an aggressor [12], [13], [15]. This is due to internal row remapping that does not necessarily map logically-adjacent to physically-adjacent rows inside the DRAM device [66]. This is a major limitation of all existing Rowhammer mitigations that are outside of DRAM, both hardware and software [9], [12], [15], [15]–[19], [22], [24], [25], [45] except Blockhammer [20]. PARA [15] explicitly requires this remapping information to be communicated from the DRAM to the CPU, which has never been implemented. In contrast, the in-DRAM nature of PROTRR allows it to use the correct row mapping that is known by the DRAM chip only.

CBT [45] and CAT-TWO [16]. Both mitigations reset their table after a `tREFW` period. If within this period, an aggressor row reaches R_{thresh} activations, its neighbors are refreshed. An attacker can, however, activate an aggressor $R_{thresh} - 1$ times immediately before and after the `tREFW`, violating the guarantees provided by the mitigation. A second issue concerns CAT-TWO only: it employs trees of counters distributed among the rows of a full rank. However, these trees are blind to victim rows that share aggressor rows across different trees. By hammering each aggressor for $R_{thresh} - 1$ times, the victim can exceed the threshold.

Graphene [19]. Refreshing a row with TRR has a similar effect like an `ACT`, which is used while hammering rows. As a consequence, an attacker could exploit TRRs to hammer rows. While this could be easy to fix, it is not taken into account in the current design of Graphene. We discuss how PROTRR securely handles TRRs in Appendix D.

Panopticon [65]. Concurrent to our work, Panopticon is a new in-DRAM mitigation against Rowhammer that relies on per-row counters stored in DRAM and uses the `ALERT` mechanism to request more time (from the memory controller) to TRR victim rows that reached a threshold. While storing counters inside DRAM itself is cheap, it is insecure as they can similarly be affected by Rowhammer bit flips. Furthermore, overloading the `ALERT` mechanism has multiple undesirable implications. First, not all devices may support `ALERT` as it is optional according to the standard, and the PHY-level errors causing them are very rare. Second, `ALERT` is a signal that blocks the whole device, likely causing significant performance degradation. Finally, it is unclear how the memory controller can tell the difference between a real `ALERT` (to retry commands) and one due to Rowhammer activity. If counters in DRAM can be secured (e.g., with a strong ECC), PROTRR can use these counters to provide a better alternative.

X. RELATED WORK

We summarize existing work on Rowhammer mitigations in Table III and compare the following properties: (i) scalability, i.e., the optimality of resource allocation; (ii) security, i.e.,

TABLE III: **Rowhammer mitigations in hardware and software.**

	Mitigation	Scalability		Security			Support		Integration		
		Flex.	Opt.	Det.	FP	Vuln.	DDR4	DDR5	OS	CPU	DRAM
HW-based	**PROTRR**	●■	●	●	●	—	●◇	●	○	○	●
	Blockhammer [20]	○□	○	●	●	—	●◇	○	○	●	○
	CBT [45]	○□	○	●	○	●	●◆	○	○	●	○
	CAT-TWO [16]	○□	○	●	○	●	●◆	○	○	●	○
	Graphene [19]	◐□	●	●	●	◐	●◆	○	○	●	○
	MRLoc [17]	○■	○	○	○	◐	●◆	○	○	●	○
	Panopticon [65]	○■	○	◐	○	◐	●◆	○	○	◐	◐
	PARA [15]	○■	○	○	○	—	●◆	○	○	●	○
	PRoHIT [48]	○■	○	○	○	●	●◇	○	○	○	●
	TWiCe [18]	○□	○	●	●	◐	●◆	○	○	●	○
SW-based	ALIS [12]	—	—	●	○	◐	—	—	●	○	○
	ANVIL [22]	—	—	●	○	●	—	—	●	○	○
	CATT [24]	—	—	●	○	◐	—	—	●	○	○
	GuardION [9]	—	—	●	○	●	—	—	●	○	○
	ZebRAM [25]	—	—	●	○	◐	—	—	●	○	○

the strength of the provided security guarantees; (iii) support, i.e., the mitigation's supported DRAM standards; (iv) and integration, i.e., the solution's required integration effort.

For **scalability**, we consider if mitigations optimally use counters and refreshes (Opt.); and if these resources can flexibly be traded-off with each other (Flex., ●). For flexibility, we further analyze if the mitigation's required storage size is always the same (□), hence is more wasteful, or scales with the system's connected devices (■). The **security** category includes the mitigations' guarantees, which are either deterministic (Det., ●) or probabilistic (○). Deterministic mitigations provide a stronger guarantee against bit flips. Further, we consider if a mitigation provides a formal proof (FP) for its design (●). Lastly, we highlight those mitigations for which we (or existing work) revealed vulnerabilities (Vuln.), and we distinguish between minor issues (◐) and fundamental flaws in the design (●). An extensive **support** of different DRAM standards (DDR4, DDR5) is essential to ensure practicality and widespread adoption. We further analyze whether mitigations require changes to the DRAM protocol (◆) or not (◇). Finally, we consider the system **integration** effort by describing which components need to be modified. Minimizing the effort is critical for real-world adoption as indirectly affected manufacturers (i.e., CPU/OS vendors) may not be willing to implement complex solutions.

Scalability. Only two mitigations (PROTRR and Graphene) are optimal w.r.t. counters and refresh requirements. PROTRR is the only solution that can flexibly trade-off storage with additional refreshes. PROTRR, ProHIT and Panopticon are the only mitigations that have counters in-DRAM, i.e., their required storage scales per connected device. Panopticon's storage is flexible as the counter table uses DRAM memory. PARA is completely stateless and does not require any storage. Similarly, MRLoc has negligible storage requirements. All other hardware-based mitigations are implemented in the memory controller; hence vendors need to provision enough storage for the system's maximum supported DRAM size.

Security. Few mitigations provide formal security guarantees for protection against Rowhammer attacks. We denote mitigations without known vulnerabilities by "—". Based on our security analysis (§ IX) and previous work [19], most of the hardware-based mitigations suffer from vulnerabilities. PARA's security is probabilistic, and to protect modern devices the overhead can be substantial [21]. Instead, all software mitigations provide a partial protection because of blindness to internal row remapping, and to newer Rowhammer variants like half-double [34]. Previous work has also shown design-level flaws in ANVIL [8], [44], [66] and GuardION [42].

Support. None of the existing hardware-based mitigations are DDR5-ready, except PROTRR, which considers the new RFM extension introduced in the DDR5 standard [30]. Software-based mitigations are agnostic to the DDR technology. PROTRR, ProHIT, and Blockhammer are the only three mitigations that do not require changing the DRAM protocol. TWiCe and Graphene require adding new DRAM commands for refreshing rows adjacent to the aggressors, and PARA requires communicating the mapping of internal rows to the CPU. All other mitigations implicitly assume that there exists a DRAM command for refreshing a specific row – which currently does not exist.

Integration. Our comparison shows that all hardware-based solutions require modifications to the CPU (e.g., memory controller), except for PRoHIT, and PROTRR, which can be fully implemented in-DRAM. PRoHIT is vulnerable to specific patterns [19]. Panopticon [65] requires the CPU's memory controller to handle the `ALERT` signal gracefully, and as discussed in Section IX, some of its security aspects remain unclear. Instead, PROTRR is the only solution with deterministic and formal security guarantees. Software-based solutions are often integrated into the operating system's kernel. None has seen widespread adoption so far.

XI. CONCLUSION

We introduced PROTRR, the first in-DRAM Rowhammer mitigation with formal security guarantees, for which we also proved that it is optimal in terms of storage and refresh overhead for any given DRAM technology. PROTRR is secure against FEINTING, the best possible attack we have formally constructed against a perfect in-DRAM TRR. Moreover, we used insights from FEINTING to provide a flexible trade-off between needed storage and refreshes given a DRAM device with a certain degree of vulnerability to Rowhammer. PROTRR is compatible with DDR4 and leverages the recent RFM extension in DDR5 to support future devices that are more susceptible to Rowhammer. We evaluated PROTRR's space, performance, and power overhead using an ASIC implementation and cycle-accurate simulation. In summary, PROTRR can protect current and future devices while requiring minimal storage and incurring negligible power and performance overhead.

ACKNOWLEDGMENTS

We thank our anonymous reviewers, also Stefan Saroiu and Hans Diesing for their valuable feedback. We thank Kubo Takashi for sharing valuable insights into DRAM technology. This research was supported by the Swiss National Science Foundation under NCCR Automation, grant agreement 51NF40_180545, and in part by the Netherlands Organisation for Scientific Research through grant NWO 016.Veni.192.262.

References

[1] 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 *IEEE S&P*, 2020.

[2] H. Hassan, Y. C. Tugrul, J. S. Kim, V. Van der Veen, K. Razavi, and O. Mutlu, "Uncovering in-dram rowhammer protection mechanisms: A new methodology, custom rowhammer patterns, and implications," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 1198–1213.

[3] P. Jattke, V. van der Veen, P. Frigo, S. Gunter, and K. Razavi, "Blacksmith: Scalable Rowhammering in the Frequency Domain," in *IEEE S&P*, May 2022.

[4] 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.

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

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

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

[8] 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 *ACM SIGSAC*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 1675–1689.

[9] 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.

[10] 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.

[11] 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.

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

[13] M. Lipp, M. Schwarz, L. Raab, L. Lamster, M. T. Aga, C. Maurice, and D. Gruss, "Nethammer: Inducing Rowhammer Faults Through Network Requests," in *EuroS&PW*, 2020, pp. 710–719.

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

[15] 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.

[16] I. Kang, E. Lee, and J. H. Ahn, "CAT-TWO: Counter-Based Adaptive Tree, Time Window Optimized for DRAM Row-Hammer Prevention," *IEEE Access*, vol. 8, pp. 17 366–17 377, 2020.

[17] J. M. You and J.-S. Yang, "MRLoc: Mitigating Row-hammering based on Memory Locality," in *DAC*. IEEE, 2019, pp. 1–6.

[18] 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.

[19] Y. Park, W. Kwon, E. Lee, T. J. Ham, J. H. Ahn, and J. W. Lee, "Graphene: Strong yet Lightweight Row Hammer Protection," in *MICRO*. IEEE, 2020, pp. 1–13.

[20] A. G. Yağlikçi, M. Patel, J. S. 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, pp. 345–358.

[21] J. S. Kim, M. Patel, A. G. Yağlıkçı, H. Hassan, R. Azizi, L. Orosa, and O. Mutlu, "Revisiting RowHammer: An Experimental Analysis of Modern DRAM Devices and Mitigation Techniques," in *ISCA*, 2020, pp. 638–651.

[22] 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.

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

[24] F. Brasser, L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "CAn't Touch This: Software-Only Mitigation against Rowhammer Attacks targeting Kernel Memory," in *USENIX Security*, 2017.

[25] 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 *USENIX OSDI*, 2018.

[26] 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, pp. 28–41.

[27] J.-B. Lee, "Green Memory Solution," 2014.

[28] Micron, "DDR4 SDRAM Datasheet," Tech. Rep., 2016.

[29] JEDEC Solid State Technology Association, "JESD79-4B, DDR4 Specification," 2017.

[30] ——, "JESD79-5, DDR5 Specification," 2020.

[31] J. Misra and D. Gries, "Finding repeated elements," *Science of Computer Programming*, vol. 2, no. 2, pp. 143–152, 1982.

[32] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu, "A case for exploiting subarray-level parallelism (SALP) in DRAM," in *ISCA*. IEEE, 2012, pp. 368–379.

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

[34] G. LLC, "Half-Double: Next-Row-Over Assisted Rowhammer," Google LLC, Tech. Rep., May 2021.

[35] 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.

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

[37] ——, "Advanced Fault Attacks in Software: Exploiting the Rowhammer Bug," in *Fault Tolerant Architectures for Cryptography and Hardware Security*, S. Patranabis and D. Mukhopadhyay, Eds. Singapore: Springer Singapore, 2018, pp. 111–135.

[38] A. P. Fournaris, L. Pocero Fraile, and O. Koufopavlou, "Exploiting Hardware Vulnerabilities to Attack Embedded System Devices: A Survey of Potent Microarchitectural Attacks," *Electronicsweek*, 2017.

[39] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks," in *USENIX Security*, 2016.

[40] D. Poddebniak, J. Somorovsky, S. Schinzel, M. Lochter, and P. Rösler, "Attacking Deterministic Signature Schemes Using Fault Attacks," in *EuroS&P*, 2018.

[41] R. Qiao and M. Seaborn, "A New Approach for Rowhammer Attacks," in *HOST*, 2016.

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

[43] X.-C. Wu, T. Sherwood, F. T. Chong, and Y. Li, "Protecting Page Tables from RowHammer Attacks Using Monotonic Pointers in DRAM True-Cells," in *ASPLOS*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 645–657.

[44] 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 *IEEE S&P*, 2018.

[45] S. M. Seyedzadeh, A. K. Jones, and R. Melhem, "Counter-Based Tree Structure for Row Hammering Mitigation in DRAM," *IEEE Computer Architecture Letters*, vol. 16, no. 1, pp. 18–21, 2016.

[46] M. Kaczmarski, "Thoughts on Intel Xeon E5-2600 v2 Product Family Performance Optimisation Component Selection Guidelines," 2014.

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

[48] M. Son, H. Park, J. Ahn, and S. Yoo, "Making DRAM Stronger Against Row Hammering," in *DAC*, 2017, pp. 1–6.

[49] E. D. Demaine, A. López-Ortiz, and J. I. Munro, "Frequency Estimation of Internet Packet Streams with Limited Space," in *ESA*. Springer, 2002, pp. 348–360.

[50] S. Ayyapureddi and R. Sreeramaneni, "Apparatus and method including analog accumulator for determining row access rate and target row address used for refresh operation," US Patent US10 964 378B2, Mar., 2021.

[51] Y.-C. Lai, P.-H. Wu, and J.-S. Hsu, "Target row refresh mechanism capable of effectively determining target row address to effectively mitigate row hammer errors without using counter circuit," US Patent US10 916 293B1, Feb., 2021.
[52] K. et al., "A 1.2v 38nm 2.4gb/s/pin 2gb ddr4 sdram with bank group and ×4 half-page architecture," 2012, pp. 40–41.
[53] T. Vogelsang, "Understanding the energy consumption of dynamic random access memories," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2010, pp. 363–374.
[54] N. Chatterjee, N. Muralimanohar, R. Balasubramonian, A. Davis, and N. P. Jouppi, "Staged reads: Mitigating the impact of dram writes on dram reads," in *IEEE International Symposium on High-Performance Comp Architecture*. IEEE, 2012, pp. 1–12.
[55] C. et al., "A 16gb lpddr4x sdram with an nbti-tolerant circuit solution, an swd pmos gidl reduction technique, an adaptive gear-down scheme and a metastable-free dqs aligner in a 10nm class dram process," in *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, 2018, pp. 206–208.
[56] S. et al., "A 16gb 1.2v 3.2gb/s/pin ddr4 sdram with improved power distribution and repair strategy," in *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, 2018, pp. 212–214.
[57] J. Bucek, K.-D. Lange, and J. v. Kistowski, "SPEC CPU2017: Next-Generation Compute Benchmark," in *ICPE*, 2018, pp. 41–42.
[58] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH*, vol. 39, no. 2, pp. 1–7, Aug. 2011.
[59] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob, "DRAMsim3: A Cycle-Accurate, Thermal-Capable DRAM Simulator," *IEEE Computer Architecture Letters*, 2020.
[60] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling," in *ISCA*, 2003, pp. 84–97.
[61] JEDEC Solid State Technology Association, "JEP300-1: Near-Term DRAM Level Rowhammer Mitigation," 2021.
[62] M. T. Inc. How Much Power Does Memory Use?
[63] T. Inc. Micron MT40A4G4JC-062E_E 1z nm DDR4 Process Flow Full.
[64] L. Orosa, A. G. Yaglikci, 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-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 1182–1197.
[65] T. Bennett, S. Saroiu, A. Wolman, and L. Cojocar, "Panopticon: A Complete In-DRAM Rowhammer Mitigation," in *DRAMSec*, 2020.
[66] A. Tatar, C. Giuffrida, H. Bos, and K. Razavi, "Defeating Software Mitigations against Rowhammer: A Surgical Precision Hammer," in *RAID*, 2018.

APPENDIX

A. Impact of REF postponing and pulling-in: FEINTING-PostponingREFs

The DDR4 and DDR5 standards [29], [30] allow the memory controller to postpone some REF commands (i.e., under heavy DRAM activity) or to pull in a number REF commands (i.e., under idle DRAM activity). With the standard refresh rate in DDR4, up to 8 REF commands can be postponed or pulled in, and the maximum distance between two consecutive refreshes can be up to 9× tREFI. Likewise, for DDR5 devices, up to 4 REF commands can be postponed or pulled in, and the maximum distance between two consecutive refreshes can be up to 5× tREFI. The JEDEC consortium recommended to disable REF postponing and pulling-in to reduce the impact on in-DRAM Rowhammer mitigations [61]. Nonetheless, we show how PROTRR can securely support REF postponing and pulling-in by slighting modifying FEINTING.

Postponing and pulling-in are a relaxation of *when* REF commands need to be sent to a DRAM device. That said, even with postponing and pulling-in, a certain number of REF commands needs to be sent to the DRAM device in a tREFW. Given that the structure of the FEINTING is agnostic to when REF commands are issued, the only remaining question is whether it enables using fewer decoys. In both DDR4 and DDR5, at the end of the tREFW, the attacker can abuse postponing to maximize the number of tREFIs without any REF commands, which we indicate by P_{max} (1 when there is no postponing).

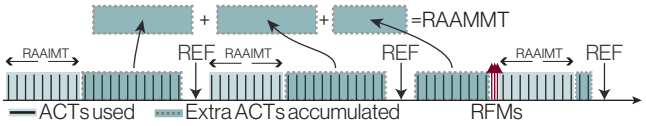

Fig. 17: **FEINTING for large T_{RFM}.** As the attacker sends activations, REF can reduce RAA accounting only for RAAIMT activations. The exceeding value eventually reaches RAAMMT, and m RFMs are sent by the controller.

This configuration has two implications. First, $(P_{max}-1)\times V$ fewer decoys are needed compared to the original FEINTING for both DDR4 and DDR5. Second, for DDR4, an attacker can continuously hammer the victim for $P_{max}\times$tREFI. For DDR5, if RAAIMT is between 32 and 64, this does not change the number of times the victim can be hammered other than what is allowed by RFM postponing (as discussed in §V-D). When RAAIMT is set to 72 or 80, however, the distance between groups of RFM commands can be higher than those of postponed REF commands: $6\times\text{RAAIMT} > 5\times T_{REF}$. Where, in the default tREFW of DDR5 (32 ms), T_{REF} is equal to 83. In these cases, equally as in DDR4, the last round of the attack is extended.

B. RFM postponing: FEINTING-PostponingRFMs

We now consider a more advanced memory controller that postpones RFM commands. This means that RFM commands do not have to be sent exactly after RAAIMT activations. Instead, the controller has the flexibility to choose a better scheduling. As explained in §IV, the only requirement set by the standard [30] is that RAA can never exceed $T_{RFM} = m\times\text{RAAIMT}$. In other words, RFM commands can be postponed up to m times. In a real scenario, it is very hard for an attacker to influence the way RFM commands are scheduled. Nonetheless, we assume the most favorable scheduling from an attacker's perspective. Depending on T_{RFM}, there are two possibilities: (i) $T_{RFM} \le T_{REF}$, where the same as in an earlier section (§V-C) applies and nothing changes from an attacker's point of view; and (ii) $T_{RFM} > T_{REF}$, where FEINTING can further be improved. We now analyze case (ii) considering an attacker who is able to precisely influence the scheduling of RFM commands.

FEINTING for large T_{RFM}. Postponing RFM commands enables a lucky attacker to avoid the situation of costly RFM commands triggered due to only a few extra activations, such as the case in Figure 7-c, §V-C. Instead, RFM postponing can be used to create blocks of RAAIMT activations similar to FEINTING-Medium (Figure 7-b, §V-C). This postponing, however, causes the RAA counter to increase, and at some point, the memory controller will have to issue RFM commands. Overall, two phenomena are happening simultaneously: a slow

accumulation of extra activations that results in a series of RFM, and a fast increase of RAA that is reduced immediately upon refresh as shown in Figure 17. However, given that our L_{tREFW} activations are now equally distributed over intervals of size RAAIMT, this is the optimal scenario for the attacker similar to FEINTING-Medium.

An exception are the last few intervals of FEINTING in this scenario. The last RFM commands can potentially be sent after the tREFW, de facto removing them from the mitigation. These activations can freely be used to increase the count of the last decoys (needed for REF) and the victim. We consider RFM postponing when configuring PROTRR as discussed in §VI and our evaluation in §VII.

C. FEINTING against subarray parallelism: FEINTING-Subarrays

In the following, we describe how FEINTING can be adapted for subarray TRR parallelism. To obtain the highest $Hammer_{max}$, the choice of aggressor rows used for FEINTING must be optimized. We define S as the maximum number of subarrays refreshed at each TRR event. We consider a TRR mechanism that refreshes the highest rows from S different subarrays, resulting in a volume of $S \times V$. The number of rows in each subarray is R_{sb}, typically 512 [32] resulting in 128 subarrays per bank.

Theorem 9 (Optimal aggressor distribution for subarray parallelism)**.** *In the case of subarray TRR parallelism, the aggressor rows have to be distributed equally over all the subarrays in a bank to maximize* $Hammer_{max}$.

▸ *Proof.* The maximum number of decoys in a subarray is given by $R_{sb} \times B/(B+1)$ (remember that B is the blast diameter of an aggressor). If only one subarray is used for FEINTING, all the rows involved would be refreshed after $T_{alive} = R_{sb}/V \times B/(B+1)$. If the aggressors are distributed over a number of subarrays lower or equal to S, the same result would apply as all the rows would be refreshed in parallel. Moreover, as the number of subarrays in the attack increases up to S, the share of activations used for the victim is reduced as this only increases the number of necessary decoys, lowering the final $Hammer_{max}$. Instead, targeting a number of subarrays higher than S means that the parallel refresh will be saturated, and some subarrays will be skipped (Figure 18). Similar to FEINTING without subarray parallelism, we assume that for equal counters, the attacker can control that the victim row to have the lowest refresh priority. That is, a subarray is never picked for a refresh if at least S different subarrays exist with the same maximum row count. Because of FEINTING, all the rows are equally often activated. Considering targeting $S+1$ subarrays and using all the possible decoys, it would mean that in the first T_{alive} TRR events, the S *decoy* subarrays are completely refreshed, and in the last T_{alive} event, the rows from the victim subarrays are refreshed. In the same way as the original FEINTING, any other distribution of activations either induces a refresh on the victim subarray or is a loss because a decoy is refreshed with a higher activation count that could have otherwise been used for the victim. Therefore, the distribution of rows across subarrays should still follow the original theorems for FEINTING (see §V). To exploit the saturation as much as possible, FEINTING must be performed using all available subarrays. Figure 18 shows the structure of FEINTING considering subarrays parallelism.

TABLE IV: Overview of used symbols.

Symbol	Description	Ref. (§)
tREFI	Duration of a refresh interval (tREFW/8192) in μs.	II-A
tREFW	Duration of a REF window, e.g. 64 ms (DDR4).	II-A
B	No. of rows affected by an aggressor (e.g., 2 or 4).	III
R_{thresh}	Number of hammer required to trigger a bit flip.	III
L_{attk}	Number of total activations of an attack.	V-A
m	The value of the MR59 register in OP[7:6].	IV
RAA	Rolling Accumulated ACT.	IV
RAAMMT	Maximum Management Threshold (RAAIMT $\times\, m$).	IV
RAAIMT	Initial Management Threshold.	IV
V	TRR volume: no. of rows refreshed at every TRR event.	V
T	No. of ACTs in between of two consecutive REFs.	V
$\mathcal{A}$	Rowhammer attack: a sequence of row activations	V-A
$Hammer_{max}$	Max. hammer count a victim can reach before refresh.	V-A
L_{tREFW}	Total number of activations in a tREFW.	V-A
D_T	No. of rows used in FEINTING (decoys and victims).	V-B
$D(\alpha)$	No. of decoys that have not been refreshed at ACT a.	V-B
d	Distance of TRR events expressed in REFs.	V-B
T_{REF}	Number of activations between two consecutive REFs that perform a TRR.	V-C
T_{RFM}	Number of activations between two consecutive RFMs.	V-C
N	Maximum number of TRR events in a tREFW.	VI-B
C	No. of counters used to track victim rows.	VI-B
$S_{entries}$	Number of counters in a PROTRR summary.	VII-A
N_{banks}	No. of banks of the system.	VII-A
P_{max}	The max. number of tREFIs without any REFs.	VIII

D. Impact of TRR Events

The TRR mechanism itself performs an activation when refreshing a row. This effect should be considered when deriving $Hammer_{max}$. In our study, the maximum number of activations L_{tREFW} is increased by the times TRR is performed and the TRR volume: $L'_{tREFW} = L_{tREFW} \times (1 + V/T)$. Moreover, because every T activations, V more TRRs are sent, the effective TRR interval T' is calculated as $T' = T + V$.

E. Double-sided Rowhammer versus FEINTING

Double-sided Rowhammer is a technique to hammer a victim row, where both its directly adjacent rows are alternatingly activated. In PROTRR, this technique is avoided as it is not beneficial for the attacker. To model the defender's worst case, we assume a closed-page policy for the DRAM device. This means that a row is automatically precharged after activating it. In other words, in an interval of T activations, a victim row can be hammered T times by accessing only one aggressor. This is the same amount of activations that can be achieved with double-sided Rowhammer but with the difference that the total number of victims affected is higher, and as such, the total number of generated decoys. Consequently, for the same number of TRR events where the victim is not refreshed, a higher number of activations can be used against the victim, resulting in a higher $Hammer_{max}$.

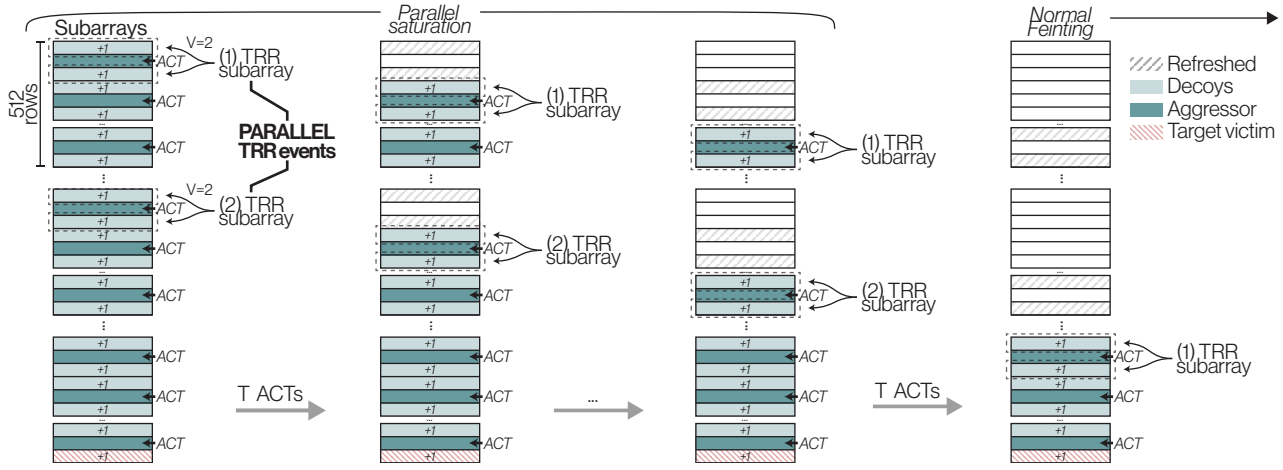

Fig. 18: FEINTING against subarray parallelism.

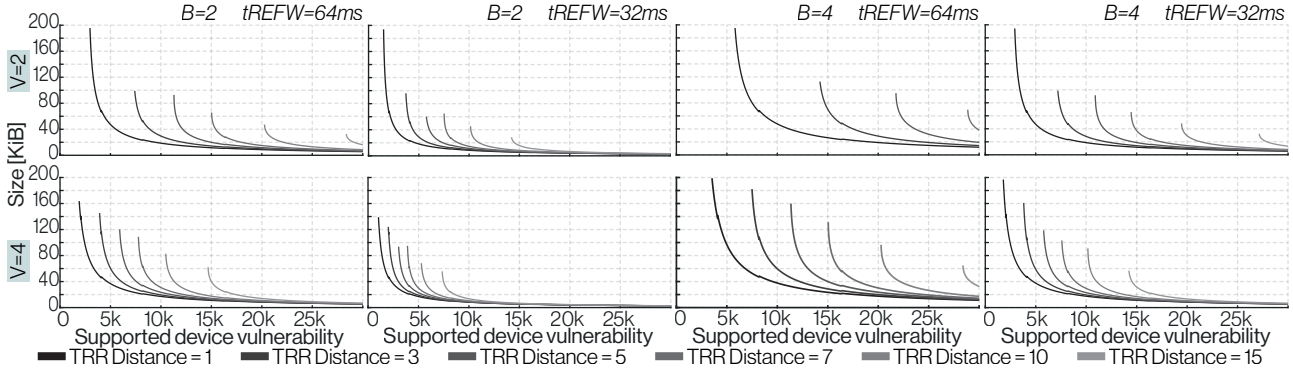

Fig. 19: The storage size for different possible setups and degrees of vulnerability in DDR4. The first line considers volumes of 2 and the second volumes of 4. Setups with `tREFW` of 64 ms 32 ms are alternated to blast diameters of 2 and 4.

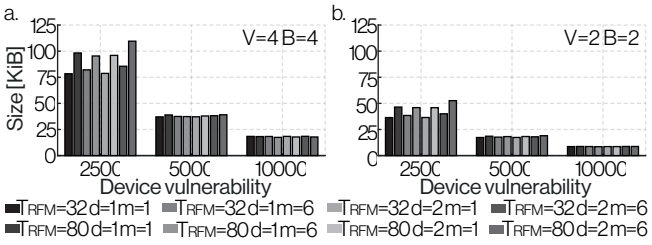

Fig. 20: Required storage size on DDR5 for various possible setups.

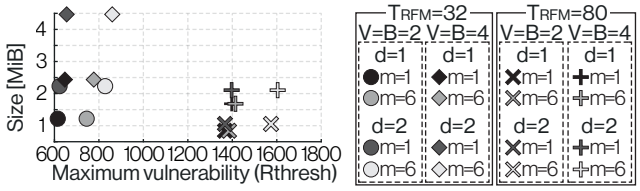

Fig. 21: Maximum vulnerability supported in DDR5.

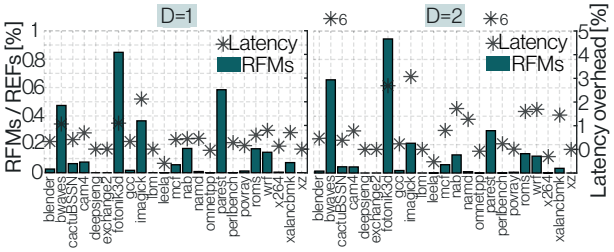

Fig. 22: `RFMs` sent relative to `REFs` and the increase in the tail latency for all SPEC benchmarks. D={2;1}.

F. Extra figures

Extended storage size analysis. Figure 19 and Figure 20 show the storage size required for PROTRR in different settings, including a volume of 4 at each TRR event. Figure 21 reports the maximum vulnerabilities that can be protected, in various DDR5 configurations.

Details of increased RFM sent. Figure 22 shows the increase in RFM sent and the increased tail latency for individual SPEC benchmarks.

Symbols. In Table IV, we present an overview of symbols with references to the section where they were introduced.