BLACKSMITH: Scalable Rowhammering in the Frequency Domain

Patrick Jattke
ETH Zurich
pjattke@ethz.ch

Victor van der Veen
Qualcomm Technologies Inc.
vvdveen@qualcomm.com

Pietro Frigo
VU Amsterdam
p.frigo@vu.nl

Stijn Gunter
ETH Zurich
sgunter@ethz.ch

Kaveh Razavi
ETH Zurich
kaveh@ethz.ch

***Abstract*—We present the new class of *non-uniform* Rowhammer access patterns that bypass undocumented, proprietary in-DRAM Target Row Refresh (TRR) while operating in a production setting. We show that these patterns trigger bit flips on *all* 40 DDR4 DRAM devices in our test pool. We make a key observation that all published Rowhammer access patterns always hammer "aggressor" rows *uniformly*. While uniform accesses maximize the number of aggressor activations, we find that in-DRAM TRR exploits this behavior to *catch* aggressor rows and refresh neighboring "victims" before they fail. There is no reason, however, to limit Rowhammer attacks to uniform access patterns: smaller technology nodes make underlying DRAM technologies more vulnerable, and significantly fewer accesses are nowadays required to trigger bit flips, making it interesting to investigate less predictable access patterns.**

The search space for non-uniform access patterns, however, is tremendous. We design experiments to explore this space with respect to the deployed mitigations, highlighting the importance of the *order*, *regularity*, and *intensity* of accessing aggressor rows in non-uniform access patterns. We show how randomizing parameters in the frequency domain captures these aspects and use this insight in the design of Blacksmith, a scalable Rowhammer fuzzer that generates access patterns that hammer aggressor rows with different *phases*, *frequencies*, and *amplitudes*. Blacksmith finds complex patterns that trigger Rowhammer bit flips on all 40 of our recently-purchased DDR4 DIMMs, 2.6× more than state of the art, and generating on average 87× more bit flips. We also demonstrate the effectiveness of these patterns on Low Power DDR4X devices. Our extensive analysis using Blacksmith further provides new insights on the properties of currently-deployed TRR mitigations. We conclude that after almost a decade of research and deployed in-DRAM mitigations, we are perhaps in a worse situation than when Rowhammer was first discovered.

I. INTRODUCTION

A dangerous mistake when designing a mitigation is assuming that attackers will operate the same way after the deployment of the new mitigation. This is especially true for in-DRAM Target Row Refresh (TRR), a selection of defense mechanisms for stopping the ever-worsening Rowhammer effect in the DRAM substrate. Proprietary, undocumented in-DRAM TRR is currently the only mitigation that stands between Rowhammer and attackers exploiting it in various scenarios such as browsers, mobile phones, the cloud, and even over the network [1]–[11]. In this paper, we show how deviations from known *uniform* Rowhammer access patterns allow attackers to flip bits on all 40 recently-acquired DDR4 DIMMs, 2.6× more than the state of the art [12]. The effectiveness of these new *non-uniform* patterns in bypassing TRR highlights the need for a more principled approach to address Rowhammer.

Existing Rowhammer patterns. Data in DRAM is stored in rows of cells. These cells consist of capacitors that leak charge over time. For preserving the data, the charge needs to be restored by refreshing the cells regularly. However, it is possible to leak charge from these cells with the Rowhammer vulnerability before they have a chance to get refreshed [13]. Existing approaches trigger Rowhammer by selecting one to many different "aggressor" rows to hammer [1], [12], [14]. These aggressor rows are repeatedly accessed in a short duration before cells get refreshed, causing bit flips in "victim" rows adjacent to these aggressors. As an example, the double-sided Rowhammer access pattern sandwiches a victim row with two aggressor rows, maximizing charge leakage in the victim row. To leak as much charge from victim rows as possible, such patterns hammer aggressors as often as possible before their victims have a chance to get refreshed.

Target Row Refresh. Target Row Refresh (TRR) is an umbrella term for hardware mitigations against the Rowhammer vulnerability, with recent variants operating entirely inside DRAM chips [12]. At a high level, TRR aims to detect rows that are frequently accessed (i.e., hammered) and refresh their neighbors before their charge leak results in data corruptions. The challenge is finding the frequent items in a stream of DRAM accesses. However, as precise frequent item counting is expensive in hardware, TRR implementations try to estimate the frequent items (i.e., the aggressors). Recent work shows that by increasing the number of aggressors, certain implementations of TRR are unable to keep track of all aggressors and corruptions resurface [12]. A majority of TRR implementations (roughly 70%), however, remain secure since they can detect all aggressors given that they are hammered *frequently enough*.

Non-uniform Rowhammer patterns. We make the key observation that prior Rowhammer attacks always access aggressors *uniformly*. From a frequent item counting perspective, this is a straightforward case for estimating frequent items. However, there is, of course, no need for attackers to hammer in the space where TRR implementations operate effectively. Given the increasing (physical) susceptibility of DRAM to Rowhammer [15], aggressors no longer need many accesses: attackers are free to choose from many hammering strategies between the times a victim row is refreshed. While this provides many possibilities to fool the TRR's estimation of the frequent

https://t.me/learningnets

items, at the same time, it creates a problem for attackers since the search space for *non-uniform* patterns is huge.

We design a series of experiments that start by fully randomizing the patterns and gradually discovering the essential properties that make them successful. This exploration ultimately results in a set of parameters for constructing non-uniform patterns that can effectively explore the weaknesses in existing TRR mechanisms. Notably, we find three *temporal* properties, namely *order*, *regularity*, and *intensity*, play a crucial role in constructing non-uniform patterns that can escape various TRR mechanisms.

Rowhammering in the frequency domain. To capture these temporal parameters, we propose constructing non-uniform patterns in the frequency domain. Signal properties such as *phase*, *frequency* and *amplitude* conveniently map to the parameters that are important in exploring the blind spots of TRR. Based on this insight, we build *Blacksmith* — a scalable Rowhammer fuzzer capable of generating access patterns by randomizing parameters in the frequency domain for randomly-selected aggressors. In contrast to previous work [12], our novel patterns are highly complex, making it difficult for humans to explore manually. Furthermore, our scalable fuzzing-based approach makes it easy to test a large number of DRAM devices against Rowhammer, without the need for time-consuming reverse engineering. On top of generating non-uniform patterns, we can distinguish interesting DRAM-dependent temporal properties by analyzing patterns.

Our evaluation shows that Blacksmith can generate patterns that can trigger bit flips on all 40 recently-purchased DRR4 DIMMs from the three major DRAM vendors (Samsung, Micron, and Hynix), a factor of 2.6× more than state-of-the-art many-sided patterns [12]. We also demonstrate the effectiveness of these patterns on 16 out of 19 Low Power DDR4X devices. These results show that instead of obscure TRR mitigations, we need to invest in principled mitigations with clear guarantees. To gain more insights into these non-uniform patterns, we systemically evaluate how Blacksmith converges to the specific values of the different spatial and temporal parameters. Using the bit flips triggered by these patterns, we uncover interesting new properties of deployed TRR mitigations such as the number of aggressors that they track, the importance of the aggressors' addresses and significant differences in the number of triggered bit flips on different chips of the same device. Furthermore, we reverse-engineer properties of the TRR implementation on one of the Low Power DDRX devices that Blacksmith could not trigger bit flips and show how a different configuration of Blacksmith could trigger bit flips on these devices.

Contributions. We make the following contributions:

(1) We present novel non-uniform Rowhammer patterns that make it difficult for TRR to estimate the potential aggressor rows accurately.
(2) We design Blacksmith, a new Rowhammer fuzzer that can effectively explore the important parameters of these non-uniform patterns by hammering in the frequency domain.

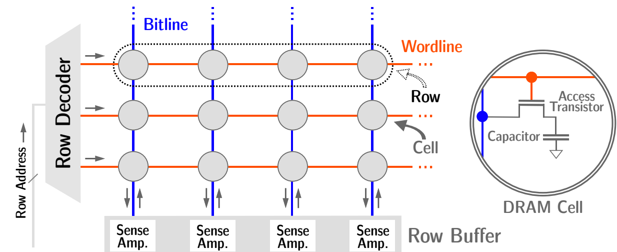


Fig. 1: **DRAM structure.** Low-level view on a DRAM bank.

(3) We evaluate Blacksmith on 40 DDR4 DIMMs from all three major DRAM vendors, showing that it is possible to trigger bit flips on 100% of them by using non-uniform patterns. We also show Blacksmith's ability to trigger bit flips on 16 out of 19 LPDDR4X DRAM chips.
(4) We conduct an extensive analysis of the effective patterns and bit flips found by Blacksmith to gain insights on patterns and deployed mitigations. Furthermore, we reverse-engineer the TRR mechanism of one of the LPDDR4X devices where Blacksmith could not trigger any bit flips to show how it can better be configured.

Reproducibility. To enable reproducibility, we publish the source code of Blacksmith on this URL: https://github.com/comsec-group/blacksmith.git.

Responsible disclosure. We reported our findings to affected parties by following a responsible disclosure process. In Q1-2021, we initiated the process with the NCSC Switzerland (NCSC-CH). In Q2-2021, NCSC-CH informed affected parties and shared our results with DRAM vendors, OEMs, and cloud providers. In Q3-2021, NCSC-CH sent affected parties an updated version of our work and announced the public disclosure date. In Q4-2021, we have been assigned a CVE (CVE-2021-42114) and publicly disclosed Blacksmith on November 15, 2021. The three DRAM manufacturers (Samsung, SK Hynix, and Micron), Intel, AMD, Microsoft, Oracle, and Google confirmed the receipt of our findings. SK Hynix got in touch with us to discuss the LPDDR4X results. We discussed a possible mitigation with Intel and our findings more in detail with Google. None of the contacted parties informed us of their mitigation plans.

II. Background

This section gives an overview of DRAM, including its internal organization and interaction with the memory controller. We also introduce the Rowhammer attack, widely-deployed mitigations against it, and describe common access patterns.

A. DRAM Organization

While there are different DRAM types for PCs, servers, and laptops, they share a common organization discussed here.

Addressing & Geometry. A DRAM address is composed of a channel, bank, rank, row, and column. Each *channel* is connected to one or multiple DIMMs, of which each can operate independently. A DIMM is equipped with multiple DRAM *chips* that are grouped into *ranks* and these, in turn,

https://t.me/learningnets

consist of multiple *banks* that can operate in parallel [16]. A bank is made of many DRAM cells, of which each contains a capacitor, which stores a single data bit as electrical charge, and an access transistor. These cells are arranged in a two-dimensional grid (see Figure 1) and connected row- and column-wise by a *word*- and a *bit line*, respectively. Every bank has a *row buffer*, an array of sense amplifiers connected to the bit lines involved in reading/writing data from/to rows.

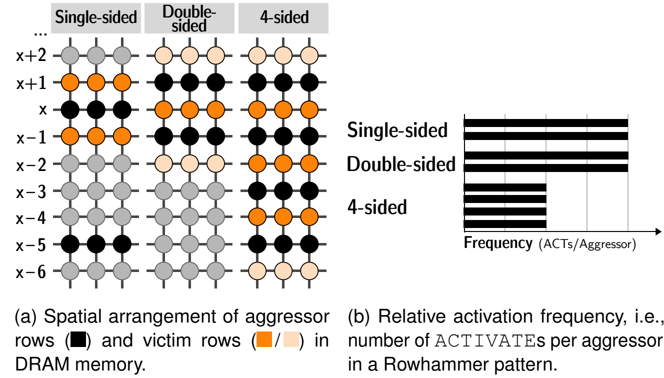
DRAM Commands [16]. Before reading or writing data to a DRAM address, the memory controller (MC) puts the associated bank in a precharged state by issuing the `PRECHARGE` command to DRAM, deactivating the row buffer. Next, the MC issues an `ACTIVATE` command, after which the requested row is loaded into the row buffer. Now, data can be read (`READ`) or written (`WRITE`); both require specifying the targeted column(s) of the loaded row. Additionally, the MC must issue `REFRESH` commands regularly, on average every 7.8 µs (the *refresh interval* or `tREFI`) [17], to preserve a cell's value since the capacitors leak charge over time [18]. The `REFRESH` only refreshes a small subset of rows at a time, which are determined by the DRAM chip, based a row's last refresh time. Related to that is the *retention time*, typically 64 ms in DDR4 [18], [19], the minimum time that DRAM cells must be able to hold data without losing information.

B. Rowhammer

While the industry has been aware of the Rowhammer vulnerability in DRAM since at least 2012 [20], Kim et al. [13] study the problem rigorously for the first time in their seminal paper in 2014. They observed that commodity DRAM chips from all major vendors suffer from disturbance errors induced by repeatedly opening (`ACTIVATE`) and closing (`PRECHARGE`) a DRAM row (i.e., *aggressor* row) in a short period of time. This action causes some cells in neighboring rows (i.e., *victim* rows) to leak charge at a faster pace than usual. Consequently, these cells can no longer retain their charge for the period they are supposed to before the cell is refreshed, resulting in their bits flipping.

The Rowhammer attack attracted much attention due to its devastating impact on systems security. Follow-up research showed how Rowhammer can be used to compromise users via JavaScript [2], [3], [8], [11], in the cloud [4], [5] on mobile phones [6], [7], and even over the network [9], [10].

Target Row Refresh. The industry has responded to Rowhammer by deploying a mitigation known as Target Row Refresh (TRR). Frigo et al. [12] analyze TRR to find that it refers to a variety of different solutions with the recent variants all operating inside the DRAM chips. They further show that in-DRAM TRR tries to detect which rows are being hammered using a *sampling* mechanism and internally refresh their victims before these receive their regular refresh. An ideal TRR sampler needs to keep track of every row that receives an `ACTIVATE` command but doing so is expensive in hardware. Instead, existing TRR mechanisms estimate the rows that are activated most often. The TRRespass fuzzer [12] shows gaps in this estimation by increasing the number of aggressor rows, causing Rowhammer bit flips to resurface on roughly 30% of modern DDR4 DIMMs. The question that we are trying to answer in this paper is *whether there are more effective ways of discovering gaps in the estimation of aggressor rows.*



(a) Spatial arrangement of aggressor rows (■) and victim rows (■/■) in DRAM memory.

(b) Relative activation frequency, i.e., number of `ACTIVATE`s per aggressor in a Rowhammer pattern.

Fig. 2: **Common Rowhammer access patterns.** Overview of the most common Rowhammer access patterns from prior work.

Rowhammer Access Patterns. We use the term *pattern* to describe memory access sequences and denote patterns as being *effective* when they can trigger bit flips. In search of effective patterns for more DIMMs, we must understand how existing instances work. Figure 2a shows the three common Rowhammer access patterns. In the original work [13], the authors used two far apart aggressor rows for hammering, later termed as *single-sided* because, from the victim row's point of view, their charge is being leaked from one side. Later, Seaborn and Dullien [1] showed that if a victim row is sandwiched by two aggressors, it increases the chance of bit flips (i.e., *double-sided*). Frigo et al. [12] introduced n-sided Rowhammer, where n refers to $n-1$ victims being hammered by n aggressors from both sides. Figure 2a shows an example with $n = 4$. The recent SMASH attack [11] shows that it can trigger bit flips in JavaScript by synchronizing n-sided patterns with the DRAM `REFRESH` command. Our experiments with SMASH patterns, as discussed in Appendix A, show that while aligning with `REFRESH` increases the number of effective patterns found on certain DIMMs, overall, it does not compromise TRR on more devices than the original n-sided patterns.

We make a key observation that the aggressors in all these previous patterns are hammered *uniformly* as shown in Figure 2b. While hammering uniformly maximizes the chance of triggering a Rowhammer bit flip, since it maximizes the frequency in which the aggressors are hammered, it is also the easiest case for TRR to estimate the rows that are accessed the most (i.e., hammered). Given the increasing degree of vulnerability to Rowhammer, the aggressors no longer need to be hammered as frequently as possible, and a significantly smaller number of accesses is enough to trigger Rowhammer [15]. This provides an opportunity to better exercise the TRR's estimation of aggressor rows by hammering *non-uniformly*. This paper explores the design of non-uniform patterns against in-DRAM TRR.

https://t.me/learningnets

III. Properties of Effective Non-uniform Patterns

While non-uniform access patterns will likely make it more challenging for TRR estimating the aggressors, at the same time, they are challenging to craft due to the large design space. Let us consider the possible number of activations in a `tREFI` ($\approx$100 accesses), so we end up with $\approx 819\,\text{k}$ possible activations between two (victim) row refreshes, where each could potentially be used to hammer our aggressors. Assuming that we need to hammer $10\,\text{k}$ times, it gives us more than 6.7×10^{23447} possibilities to distribute our double-sided aggressor accesses (see Appendix C for details). As this is impractically large, we explore the important properties of effective non-uniform patterns to reduce the size of this search space.

One possibility is to reverse-engineer specific details of various TRR implementations, as has been done in concurrent [21] and earlier work [12]. This is a time-consuming process and needs to be repeated on new devices given that vendors tend to change their implementations [12]. Instead, our goal here is to determine the generic properties of existing TRR implementations. For this purpose, we conduct a series of experiments on DIMMs $\mathcal{A}_{10}$ and $\mathcal{B}_2$ of the two major vendors in our test pool. We later show how these discovered properties can help in triggering bit flip on other DIMMs from the same vendors as well in devices from vendor $\mathcal{C}$, and from unknown vendors.

We start exploring non-uniform patterns by fully randomizing the number of aggressors being hammered and their location (Section III-A). To limit the search space, we try to answer questions such as *when* we should hammer an aggressor and for *how long*. We first answer these questions for patterns that fit within a `REFRESH` interval (Section III-B) and later extend our search to larger patterns (Section III-C). After we understand the properties of successful patterns, we discuss how we can capture these properties when generating effective non-uniform access patterns (Section III-D).

A. Can non-uniform access patterns bypass mitigations?

We design an experiment to explore the effectiveness of non-uniform patterns. In this experiment, we assess the importance of non-uniformity by considering two extremes in the design space: (i) adding *some* some randomness to n-sided patterns and (ii) creating randomized patterns.

In the first experiment (i), we introduce non-uniform aggressor accesses (i.e., accesses at random times) into common n-sided patterns by accessing selected aggressors more or less often than all others. This means, we access a randomly picked double-sided aggressor pair at random times during the regular accesses of an n-sided pattern[1].

The naive approach for implementing such random accesses would be using conditional branching based on some random value. However, the CPU might speculatively execute the wrong branch, leading to unwanted memory accesses. Therefore, we rewrite our branching into a statement that targets different memory locations depending on the condition's value. As

[1] with a randomly picked number of aggressors $n \in [2, 32]$, an aggressor intra-distance $d \in [0, 16]$, and an aggressor intra-distance $v \in [1, 4]$.

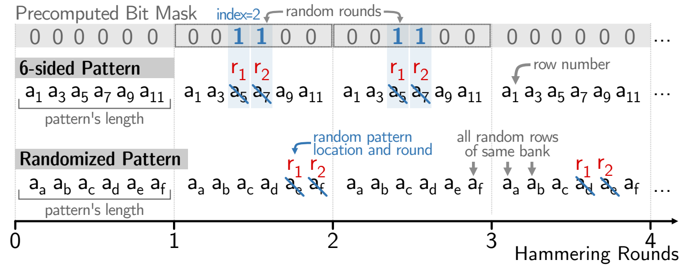


Fig. 3: **Non-uniform patterns experiment.** (i) We take a n-sided pattern (e.g., $n = 6$) and based on a precomputed bit mask, randomly replace accesses to a randomly picked double-sided aggressor pair r_1, r_2. (ii) We create a fully-randomized pattern and hammer a randomly-picked double-sided aggressor pair r_1, r_2 at random times.

depicted in Figure 3, we precompute a bit mask that decides when and how often our aggressor pair should be hammered. This bit mask is computed based on existing work [15] that showed between $10\,\text{k}$ and $147.5\,\text{k}$ `ACTIVATE`s (*Hammer Count*) are required on modern DDR4 devices to trigger bit flips. Ideally, this value should be as small as possible to reduce the chance of detection by TRR, yet large enough to cause a bit flip. As we cannot determine this value for our PC-DDR DIMMs, we randomly pick a value in between $10\,\text{k}$ and $147.5\,\text{k}$ for each pattern. While hammering the pattern, we then use the bit mask to offset an array that points to part of our n-sided pattern or our randomly-picked double-sided aggressor pair.

In experiment (ii), we follow the same methodology to access a selected double-sided aggressor pair non-uniformly; however, instead of a n-sided pattern as a basis, we now fully randomize the pattern's accesses. Note that these random accesses are spread over the same bank as our aggressors, i.e., there are no fixed distances in-between aggressors like in n-sided patterns. Similarly as in experiment (i), we use patterns of length $n \in [2, 32]$ but we replace aggressors by our double-sided aggressor pair at random locations of the pattern. This makes all aggressor accesses in our pattern non-uniform.

We extended TRRespass [22] by these two new ways of creating patterns and try these patterns as well as the original n-sided patterns on all DIMMs of our test pool (see Appendix B) for $6\,\text{h}$. To see if a pattern is successful, we check all rows next to accessed rows for bit flips. The fully randomized approach was the most successful and could trigger bit flips on 37.5% of all devices in our test pool, followed by n-sided patterns (35%), and n-sided patterns with random accesses (27.5%). Considering all three approaches together, we observed bit flips on 20 of 40 DIMMs (50%). From these 20 DIMMs, there are 8 DIMMs where all three approaches triggered bit flips and 6 DIMMs where one (or both) of the two non-uniform approaches succeeded. Table VII in Appendix D provides more detailed results from these experiments.

These experiments confirm our assumption that there are DIMMs where we need non-uniform patterns to bypass the mitigation. This shows that non-uniformity is a promising concept for finding effective Rowhammer access patterns on more devices.

https://t.me/learningnets

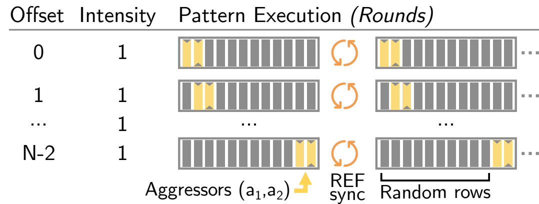


Fig. 4: **Offset & intensity experiment.** Systematic probing of aggressor offsets $0 \ldots N-2$ for a pattern of length N.

> **Observation (O1).** Non-uniform accesses can find effective patterns on DIMMs where previous n-sided patterns could not trigger any bit flips.

However, there are also three opposite cases where only pure n-sided patterns are effective; this indicates that these simple approaches are not effective enough. Besides that, we observe that our pattern search space is not optimal yet: using n-sided patterns as a base seems to be too restrictive, whereas the fully random approach creates an enormous search space that cannot be explored in sufficient depth within a reasonable time. Therefore, we aim to identify parameters of effective patterns that allow us to guide pattern generation and, as such, reduce the search space.

B. When should we hammer an aggressor and for how long?

Prior work [11], [12] suggests that in-DRAM TRR acts at the same time of a REFRESH. Based on this, we aim to explore the parameters of effective non-uniform patterns within two consecutive REFRESH commands, i.e., a refresh interval.

To verify *when* we should hammer, we design an experiment where we randomly choose a double-sided aggressor pair (a_1, a_2) and generate a pattern of length N, where N corresponds to the number of memory accesses that fit inside a refresh interval (determined experimentally beforehand). For each possible offset $t = 0, \ldots, N-2$ in that we can place the two aggressors, we craft a pattern as follows: the aggressors a_1 and a_2 are placed at position t and $t+1$ in the pattern, respectively, and the remaining $N - |\{a_1, a_2\}| = N - 2$ accesses, (i.e., positions $0 \leq i < N$ for $i \notin \{t, t+1\}$) are filled up by accesses to random rows in the same bank as a_1 and a_2. This is depicted in Figure 5: the pattern's aggressor accesses are highlighted in yellow and the random accesses in grey. We repeat hammering each pattern for one million rounds, i.e., long enough to see bit flips even with *strong* DRAM cells [15]. We note that the rows are randomly picked for each offset (including the aggressors) in each iteration of the experiment. For improving the reliability, we repeat the experiment ten times on different locations (i.e., DRAM rows). To ensure that these patterns remain inside the refresh interval, at the end of each round, we access two random rows from the same bank repeatedly until we observe a peak in the access time, which signals that a REFRESH happened.

Figure 5 depicts the results of our experiment for $\mathcal{A}_{10}$, aggregated over ten DRAM locations. The *best pattern*, i.e., the pattern that triggered the highest number of bit flips (red bar), starts at offset 91 and generates 140 bit flips. We can

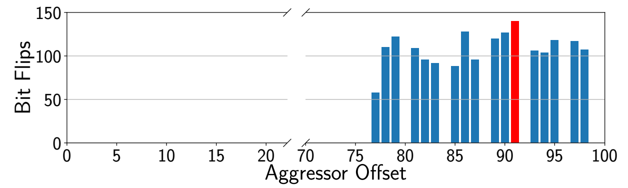


Fig. 5: **Aggressor offset.** Observed bit flips on $\mathcal{A}_{10}$, over ten probed locations, at which we place aggressors at different offsets in the pattern ($N = 100$). Using an offset of 91 (red) triggers the most (140) bit flips.

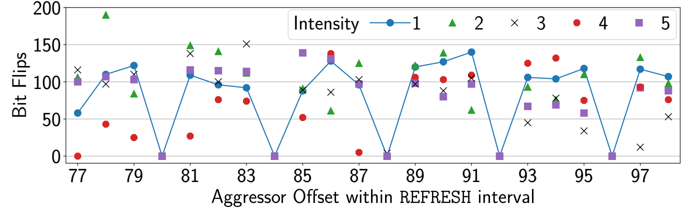


Fig. 6: **Hammering intensity.** Number of observed bit flips when repeating hammering the aggressors with different intensity (1–5), accumulated over 10 different locations on $\mathcal{A}_{10}$. Hammering with an intensity of two, starting from offset 78, triggers the most (190) bit flips.

see that an arbitrarily chosen aggressor offset may lead to no bit flips because the TRR sampler on this device considers the first accesses in a refresh interval, similar to what has been reported in earlier work [12]. These results suggest that towards the end of the refresh interval, only certain accesses (at offsets $80, 84, \ldots, 96$) are sampled. Hence, we can trigger bit flips by hammering at specific times in the last ≈23% of the refresh interval (i.e., offset $77 - 98$). The number of bit flips that we observe in this range is, on average, higher than for all other possible offsets within a REFRESH interval. From that we conclude that our assumption is correct: carefully choosing *when* to access aggressors is significant for maximizing effectiveness.

> **Observation (O2).** Inserting aggressors at the "right" location in a pattern enables them to bypass the mitigation.

A natural follow-up question from this result is whether hammering our aggressor pair with greater intensity (i.e., more than only once successively) increases the number of observed bit flips. More bit flips are favorable for attacks as they typically require bit flips at specific page offsets. Hence, more bit flips increase the attack's success rate. However, accessing an aggressor pair successively too often will likely result in a TRR. To investigate this, we extend our last experiment by repeating hammering each possible pattern offset up to five times for one million rounds in total. This experiment is depicted in Figure 4. We limit the intensity to five because higher intensities do not trigger bit flips anymore.

In Figure 6, we show the results of this experiment. We report observed bit flips within aggressor offset $77-98$ (derived from the previous experiment, see Figure 5). We can see that for some offsets, an increased hammering intensity leads to more bit flips. For example, starting from offset 78 and successively hammering two times is more effective (190 bit flips) than only a single time (110 bit flips) and also outperforms the best offset hammered only a single time (offset 91, 140 bit

https://t.me/learningnets

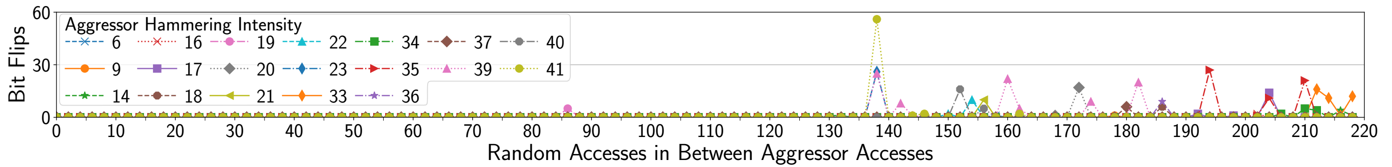


Fig. 8: **Hammering duration.** Observed bit flips on $\mathcal{B}_2$ for patterns up to three `REFRESH` intervals, a varying number of random accesses and aggressor hammering intensities. Choosing an offset of 138 with intensity of 41 triggers the most (56) bit flips. We omit intensities without any bit flips.

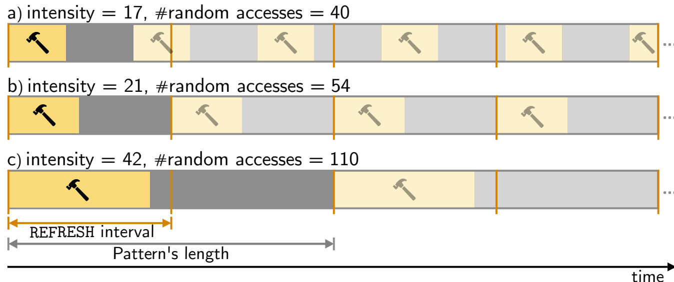


Fig. 7: **Regularity experiment.** Examples of tested patterns with different intensity and number of random accesses: a pattern smaller than (a) and of equal length to (b) a refresh interval, and (c) one covering two refresh intervals. Opaque regions show the pattern's repetition during execution.

flips). As expected, hammering the aggressors for too long triggers a TRR, which results in fewer or no bit flips at all. This strongly indicates that TRR sampling happens at specific offsets $(80, 84, 88, \ldots)$, but it is not enough for an aggressor row to get sampled only at one of them. For example, we can see that at offset 80 with an intensity of 5, our aggressors are sampled by the mitigation; however, if we use an intensity of 4 starting at offset 79, we also do access an aggressor at offset 80 but we do trigger bit flips. This suggests that the TRR mechanism on this device deploys a counter and we need to get sampled multiple times before a TRR. We conclude from this that there is a *sweet spot* up to which we can increase the intensity to induce more bit flips.

Observation (O3). Up to a specific point (*sweet spot*), increasing the hammering intensity leads to more bit flips.

These two properties of effective non-uniform patterns allow us to reduce the search space because the pattern's length of one refresh interval implicitly limits possible offsets and hammering intensities for our aggressors. But when we ran the same experiment on $\mathcal{B}_2$, it required a significantly higher hammering intensity to trigger bit flips. We tried intensities up to a whole refresh interval and could trigger only 5 bit flips with an intensity of 19. Not to risk limiting our search space by too much, we will also explore whether larger patterns can be more effective in bypassing certain TRR variants, such as the one in $\mathcal{B}_2$.

C. Should our patterns be longer than one refresh interval?

To answer the question of the pattern's length, we design the experiment presented in Figure 7. We first hammer two randomly picked double-sided aggressors with a given intensity and then issue a varying number of alternating accesses to two randomly picked rows. In our experiment, we cover intensities from 1 up to 64 and between 1 and 384 random accesses because they result in patterns of up to $64 \times 2 + 2 \times 192 = 512$ accesses, which covers five full refresh intervals. Again, we repeat the experiment for each combination ten times on different DRAM locations and check the rows around the double-sided aggressors for bit flips. Unlike before, we do not synchronize with the `REFRESH` anymore since our patterns now grow beyond a single refresh interval. This approach allows us to investigate how access intensity and regularity affect a pattern's effectiveness.

Figure 8 shows the experiment's result for intensities where we observed bit flips. As the number of observed bit flips decreased if we issued more than 200 random accesses in-between, we focus here on two refresh intervals only. In contrast to the earlier observation on $\mathcal{A}_{10}$ (Section III-B), the DIMM $\mathcal{B}_2$ considered here requires a higher intensity (≥ 6) to trigger any bit flips due to its different TRR implementation. The plot shows notable differences in the number of bit flips for specific pattern lengths. Interestingly, there are cases where we hammered almost the whole refresh interval (≈ 85 accesses) without being captured by the mitigation. For example, with hammering intensity of 41 and offset of 138, we first issue 41×2 aggs. $= 78$ aggressor accesses (i.e., almost a whole `REFRESH` interval), followed by 138 random accesses.

We conclude with two points from these findings. For an aggressor pair to successfully trigger bit flips, (1) it should not be hammered in certain (long) periods, and (2) when it is hammered, it should be with high intensity, even up to a whole refresh interval. These results naturally imply that we need to consider patterns larger than a single refresh interval.

Observation (O4). Hammering aggressors with a high intensity at specific points inside multiple refresh intervals allows us to bypass the mitigation more effectively.

D. How can we generate new patterns based on these insights?

In this section, we showed that non-uniformity allows finding effective patterns where previous approaches failed (**O1**) and that it is crucial to carefully choose when, within the pattern, to issue memory accesses to the aggressors (**O2**). We further discovered that the number of successive hammering repetitions can increase the number of bit flips (**O3**) and that long patterns, covering multiple refresh intervals, are necessary to discover patterns triggering bit flips on certain DIMMs (**O4**).

We leveraged these four observations to design and implement Blacksmith, a new blackbox Rowhammer fuzzer. Blacksmith generates patterns consisting of aggressors that are placed in the pattern using concepts from the frequency domain, such as *phase*, *amplitude*, and *frequency*. This enables us to

https://t.me/learningnets

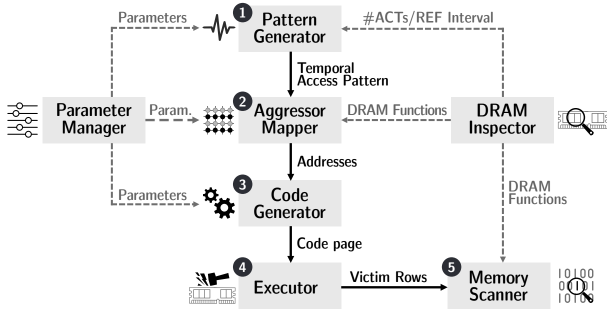


Fig. 9: **Blacksmith's architecture.** Overview of Blacksmith's main components, their interaction, and execution order (❶–❺).

distinct aggressors by expressing when we access them (*phase*), how often we repeat accessing them successively (*amplitude*), and how their accesses are distributed over time (*frequency*). By fuzzing these properties, we can compose patterns that stress TRR mitigations to trigger bit flips successfully. Our approach finds parameters efficiently by probing multiple {phase, amplitude, frequency} sets for different aggressors in a single pattern. This eliminates the need to explicitly select aggressors given that now the entire pattern is comprised of potential aggressors, some fooling the mitigations while the others effectively hammering. To the best of our knowledge, Blacksmith is the first fuzzer that uses this novel strategy for generating non-uniform Rowhammer patterns.

IV. Blacksmith

We now describe the design and implementation of Blacksmith. We first give a high-level overview of Blacksmith's architecture (Section IV-A), followed by describing how Blacksmith generates Rowhammer patterns, including a formalization of the underlying concepts (Section IV-B). After that, we introduce Blacksmith's parameter-tracking mode that uses bit flips as a feedback mechanism to learn parameters of effective patterns (Section IV-C). Finally, we provide selected implementation details (Section IV-D).

A. High-Level Overview

Figure 9 depicts Blacksmith's components. The *Pattern Generator* ❶ implements our non-uniform access patterns, which randomizes the temporal aspects the aggressors inside the pattern (i.e., *when* within a pattern, *for how long successively*, and *how often* aggressors are accessed). The *Aggressor Mapper* ❷ maps aggressors to DRAM locations, i.e., assigns each aggressor of a temporal pattern to a DRAM address by using known bank/rank address functions [11], [23]. In this step, aggressors can either be distributed equidistantly over the same DRAM bank (i.e., same number of rows in between) or randomly placed with one row in between aggressors that target the same victim. These mapping parameters are also randomized during fuzzing. The mapper then derives the virtual addresses corresponding to all hammered rows and passes them to the *Code Generator* ❸ to just-in-time (JIT) compile the hammer instructions into an executable code page. For the same reason as in Section III-A, we compile access patterns to avoid conditionals (e.g., if-else) during pattern execution as branches can be executed speculatively, resulting in unwanted memory accesses, and thus "break" our pattern's access order. Also, it allows us to determine where we need to serialize memory reads and flushes using fences. We follow a flush-early and fence-late strategy by flushing aggressors from the cache immediately after accessing them and fence immediately before accessing them again to minimize the performance impact of serialization. The *Executor* ❹ then runs the compiled code page to execute the pattern for multiple refresh windows (i.e., multiple 64 ms). To ensure that we keep accessing rows with their defined frequency, we synchronize accesses with the `REFRESH` at the end of each pattern's repetition (similar to [11]). Finally, the *Memory Scanner* ❺ verifies if the random data pattern, written before to memory, changed during hammering. Because all pattern's aggressors can potentially trigger bit flips, the Memory Scanner checks two rows around each of them for flipped bits; and if it finds any flips, it reports them and restores the original data pattern. We then either (i) hammer the same pattern with the same mapping again on a different DRAM location (❸–❺), (ii) hammer the same pattern with a new mapping (❷–❺), (iii) or generate a new pattern and repeat the whole procedure (❶–❺). Probing different locations is required because we could have been unlucky and tried a pattern on a memory region where cells are less vulnerable, thus resulting in no bit flips. The Parameter Manager and the DRAM Inspector are two supporting components. The *Parameter Manager* defines fuzzing parameters, their valid value ranges, and samples values from these ranges. The *DRAM Inspector* loads the proper DRAM address functions (derived from a DIMM's number of ranks as all our evaluation systems are equal) and determines required DIMM-specific information, such as the number of possible `ACTIVATE`s in a refresh interval.

B. Frequency-Based Patterns

Blacksmith crafts access pattern by considering their two dimensions separately: the temporal dimension, which describes *when* we access a row, and the spatial dimension, which defines *where* in memory we hammer (i.e., bank and row). Our *non-uniform access patterns* focus on the temporal dimension discussed next. We consider the spatial dimension by testing a crafted frequency-based pattern on three different (randomly chosen) memory locations as the vulnerability of different DRAM cells may vary [15].

Capturing temporal properties. We use terminology inspired by the frequency domain as composing signals with different frequencies can be used as analogy to crafting a Rowhammer access pattern with aggressors of different frequencies.

First, we generalize the idea of aggressors by defining the notion of an ***aggressor tuple*** $\mathbb{A}_k = (a_1, a_2, \ldots, a_k)$, i.e., an ordered access sequence of k aggressors. Our pattern's aggressors are not associated with specific DRAM locations but we *map* them later to specific DRAM rows. For example, in the case of $\mathbb{A}_2$, we could map them like a double-sided aggressor pair. Multiple such aggressor tuples fill up a Blacksmith access

https://t.me/learningnets

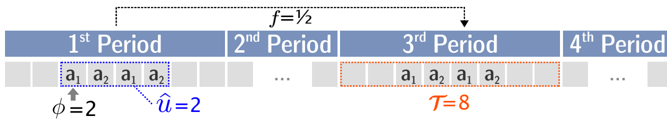


Fig. 10: **Parameters of pattern generation.** Example showing an aggressor tuple $\mathbb{A}_2 = (a_1, a_2)$ with (f,ϕ,$\hat{u}$) = ($\frac{1}{2}, 3, 2$) and $\mathcal{T} = 8$.

pattern to improve the fuzzer's efficiency while exploring the parameter space.

Each aggressor tuple $\mathbb{A}_k$ has three characteristics: a *frequency*, a *phase*, and an *amplitude*. The ***frequency*** f defines how often the aggressor tuple $\mathbb{A}_k$ is accessed within a pattern. The ***phase*** ϕ defines *when* from the start of the pattern a specific aggressor tuple $\mathbb{A}_k$ will be executed. The ***amplitude*** $\hat{u}$ describes for *how long* we should hammer a specific tuple, i.e., the number of consecutive hammering repetitions of $\mathbb{A}_k$.

Building a pattern. Blacksmith combines multiple aggressor tuples $\mathbb{A}_k$ to form an access pattern. For intertwining these $\mathbb{A}_k$ effectively, we define a global parameter that aids the construction: the *base period*. The ***base period*** $\mathcal{T}$ defines (and limits) the frequency of an aggressor tuple.

We depict the pattern creation in Figure 10. Before starting to fuzz, assume we determined that we can issue 64 accesses in a refresh interval, and we want our pattern to cover four refresh intervals (i.e., $4 \times 64 = 256$ accesses). As a result, we can choose any of $\{2, 4, \ldots, 256\}$ as the base period. Let us pick 8 so that the frequency f of any aggressor tuple is now a multiple (or divisor) of $\mathcal{T} = 8$. For instance, if $f = 1$ we execute the aggressor tuple once every base period, while if $f = 2/8$ we execute it every 4 ($= 8/2$) base periods. In Figure 10 we fill the pattern with an aggressor tuple $\mathbb{A}_2$ with ($f = 1/2$, $\phi = 3$, $\hat{u} = 2$) meaning that $\mathbb{A}_2$ is executed every two base periods ($f = 1/2$), it is displaced by 3 from the start of the pattern ($\phi = 3$), and the aggressor tuple is always hammered two times sequentially ($\hat{u} = 2$).

Once a tuple is inserted, other aggressor tuples are inserted following the same logic avoiding access slots that are already occupied by previously declared aggressor tuples. For instance, after adding $\mathbb{A}_2$ above, we cannot introduce another $\mathbb{A}_2$ with $\phi = 5$ since such time slot is already filled. We refer the interested reader to Appendix E for a more detailed description of the pattern generation algorithm.

Unlike in previous work (e.g., [12], [13], [24], [25]), all accesses in our patterns can potentially trigger bit flips. That means all rows are treated as *aggressors* as we do not distinguish the rows that are only accessed to bypass TRR. After hammering a pattern, we can measure the distance between accessed rows and flipped rows to estimate the *effective* aggressors, i.e., the ones that most likely caused the bit flip. This property also implies that we need to check for bit flips around every accessed row of a pattern.

C. Parameter-Tracking Mode

To understand how Blacksmith parameters impact a pattern's effectiveness, we implemented a parameter-tracking mode. This feature uses a pattern's effectiveness (bit flip count) and rarity (how hard it is to find) in a feedback mechanism to learn which parameter sets are most successful. The parameter-tracking mode starts with a uniform distribution for each parameter and gradually learns, based on the aforementioned indicators, which parameter values work best for specific DIMMs. It uses the feedback to modify the parameter distributions by increasing the probabilities of parameter outcomes that were successful. Using this, we can learn what parameters and values are most important to bypass mitigations. Furthermore, it allows us to derive interesting insights, as we show in Appendix F.

Table I: **Blacksmith's parameter setup.** For each pattern, we choose a number of aggressor tuples and refresh intervals (which results in the pattern's length N). For each aggressor tuple, we pick a number of aggressors, a phase, an amplitude, and derive a frequency from the base period. The amplitude is limited by $\mathtt{ACT}_{\mathtt{tREF}}$, the number of possible activations in a `REFRESH` interval.

Parameter	Range	Sampling Unit
#Aggressor tuples	$[8, 96]$	Pattern
#Refresh intervals	$[1, 16]$	Pattern
#Aggressors	$[1, 2]$	Aggressor tuple
Base period	$[4, N]$	Aggressor tuple
Phase	$[1, N]$	Aggressor tuple
Amplitude	$[1, \mathtt{ACT}_{\mathtt{tREF}}]$	Aggressor tuple
– for $\mathcal{B}_{2,8,9}$	$[1, 4 \times \mathtt{ACT}_{\mathtt{tREF}}]$	

We used our parameter-tracking mode to determine a *golden* set of parameter ranges that can find effective patterns on 37/40 DIMMs of our test pool. For three DIMMs ($\mathcal{B}_{2,8,9}$), we had to slightly increase the amplitude from (up to) one to four refresh intervals. To determine these generic parameters, we performed a 24 h run using large parameter ranges to determine the common ranges based on the discovered effective patterns. Table I shows the final ranges used in our evaluation.

D. Implementation

Our Blacksmith fuzzer was implemented from scratch in C++11 in around 6.7 k lines of code. It uses several open-source libraries such as *asmjit* [26] for JIT compiling a pattern's accesses and *nlohmann/json* [27] for im- and exporting JSON data (e.g., parameters) needed for analyzing and *replaying* effective patterns, and also for analyzing bit flips. The source code can be found on https://github.com/comsec-group/blacksmith.git.

V. Evaluation

In this section, we evaluate the qualities of non-uniform access patterns. In Section V-A, we describe our test devices and infrastructure. After that, we present our large-scale analysis results on 40 DDR4 DIMMs in Section V-B. In Section V-C, we evaluate how our Blacksmith-generated patterns facilitate Rowhammer exploitation. For completeness, we also evaluate the effectiveness of non-uniform patterns on LPDDR4X in Section V-D. Lastly, we provide concrete examples of Blacksmith patterns in Section V-E.

A. Hardware and Fuzzer Setup

Our DDR4 DRAM test pool (Appendix B) consists of 40 DIMMs acquired in July 2020 with varying sizes, module speeds, and timings. We cover all three major DRAM vendors, abbreviated by $\mathcal{A}$ (20×), $\mathcal{B}$ (10×), and $\mathcal{C}$ (6×). DIMMs denoted

https://t.me/learningnets

by $\mathcal{D}$ (4×) do not report their DRAM vendor properly. To show that Blacksmith works in a real-world setup, we do not directly interface with DRAM devices (e.g., FPGA), but we use a traditional PC setup: ten machines equipped with an Intel i7-8700K and running Ubuntu 18.04 LTS (4.15.0). We evaluate LPDDR4X DRAM chips using an in-house, JEDEC-compliant development board that allows us to test DRAM chips from vendors A (6×), B (5×), and C (8×) while operating at 1.5GHz. Similar to previous work [12], we use a pseudo-random, non-repeating data pattern in all our evaluation runs.

B. Blacksmith Results on DDR4

We aim to evaluate the generality and effectiveness of Blacksmith by answering the question: *Is our approach better at finding effective patterns on DIMMs where the state-of-the-art cannot trigger any bit flip?* To answer this question, we perform a large-scale Rowhammer test and compare Blacksmith results against the data that we obtained using TRRespass [12]. We use the following evaluation methodology: (1) we run Blacksmith for $12\,\mathrm{h}$ on each DIMM, i.e., we generate patterns and try each on three different DRAM locations to determine if it triggers bit flips, (2) we "sweep" each effective pattern over (the same) contiguous memory region of $2\,\mathrm{MB}$ to determine the *best* pattern (i.e., most effective) based on the number of observed bit flips, (3) we "sweep" the best pattern over a contiguous memory region of $256\,\mathrm{MB}$ to report the best pattern's effectiveness. By "sweeping" we refer to repeatedly moving each row of a pattern by one, hammering the pattern, and checking for flipped bits. For TRRespass, we skip step (2) and use its own definition of the best pattern based on the number of triggered bit flips during the fuzzing run. We remark that the optimality of the *best* pattern is relative to a fuzzing run, and it might be that there are better patterns that Blacksmith could not find within 12 hours.

Table II shows the results of our large-scale evaluation run. TRRespass found effective patterns on 15 of 40 tested DIMMs (37.5%), similar to the results from prior work (13 of 42 DIMMs, $\approx 31\%$) [12]. In contrast, Blacksmith found effective Rowhammer patterns on all of our 40 DIMMs (100%).

These results demonstrate Blacksmith's effectiveness and scalability in triggering corruptions — answering our initial question positively. Blacksmith could find effective patterns that trigger, on average, 87× more bit flips than TRRespass. We show how this massive increase in the number of bit flips allows for more practical exploitation in Section V-C. Table II also suggests that while there is a trend in DRAM devices from different vendors, there are also outliers.

C. Exploitation with Non-Uniform Patterns

We discuss the consequences of these better access patterns found by Blacksmith by analyzing their effect on three existing Rowhammer exploits. For this purpose, we followed prior work [12], [25] and analyzed (i) the first Rowhammer exploit targeting page tables to gain a kernel read/write primitive [1]; (ii) the exploit from Razavi et al. [4] triggering bit flips in public RSA $2048\,\mathrm{bit}$ keys to allow their factorization and private key recovery; and (iii) the exploit by Gruss et al. [14] flipping bits on the `sudoers.so` library to avoid root permission checks.

In our analysis, we briefly summarize each exploit; we refer to the original descriptions [1], [4], [14] for more details. We measure the number of exploitable bit flips when sweeping over a $256\,\mathrm{MB}$ chunk of memory and report the mean time to find them by relying on a port of the Hammertime framework [28]. We show the results for all DIMMs in Table III.

In the attack from Seaborn and Dullien [1], the aggressor triggers a bit flip on a page frame number (PFN) in a page table page, "hoping" to pivot its pointer to another (attacker-controlled) page table page. This gives an attacker read/write access to their page tables, i.e., full access to all physical memory. On a system with $16\,\mathrm{GB}$ memory, this results in 23 out of every $64\,\mathrm{bit}$ words to be possibly exploitable (i.e., $\log_2 16\,\mathrm{GB} - \log_2 4\,\mathrm{kB}$). This large number of exploitable bits makes it possible to carry out an attack even on a module that manifests very few bit flips; e.g., $\mathcal{B}_3$ with only 111 bit flips can be exploited in around 1 hour. The time to find an exploitable bit flips then dramatically decreases for more vulnerable modules, e.g., $22\,\mathrm{s}$ on average on $\mathcal{D}_3$. The exploit from Razavi et al. [4] gains SSH access to a co-hosted VM by flipping bits on the modulus n of a RSA-2048 public key and factoring the much easier factorable n' ($\neq n$) to recover the private key. We could identify exploitable bit flips on 30 out of our 40 DIMMs (75%). Finally, Gruss et al. [14] exploit specific bit flips on code pages of the `sudoers.so` library, stored in the page cache, to gain root privileges. Their *opcode flipping* technique induces bit flips in cached binary files that often lead to valid opcodes with a different semantic. This technique can break the password verification logic in the `sudoers.so`. Only $29/(4096 * 8)$ bits in a $4\,\mathrm{kB}$ page are exploitable for this attack. Still, 15 out of our 40 DIMMs (37.5%) are susceptible to such attack within at most $38\,\mathrm{min}\,35\,\mathrm{s}$ ($\mathcal{A}_{12}$). These results show how non-uniform patterns largely ease exploitation. In fact, even when considering the more difficult attack (i.e., sudo [14]) we could still build an end-to-end exploit on 15 / 40 DIMMs, which is the total number of DIMMs that TRRespass could trigger bit flips on (see Table II).

Given the large number of bit flip on some devices, we would have expected to see more exploitable bit flips, e.g., in the PTE attack. We investigated this further in Section VI-A, where we show that this is due to the large variance in the number of flips in different chips from the same DIMM.

D. Blacksmith on LPDDR4X

We evaluate the impact of our non-uniform patterns on LPDDR4X memory. Due to power and die area restrictions, there are key differences compared to regular DDR DRAM that make LPDDR an interesting target for Rowhammer analysis: (i) LPDDR's default refresh window is $32\,\mathrm{ms}$, compared to $64\,\mathrm{ms}$ for standard DDR4; (ii) it supports dynamic temperature-based refresh changing through the MR4 Mode Register [29]; and (iii) recent devices deploy on-die ECC [15].

We applied the test methodology outlined in Section V-B to evaluate Blacksmith on 19 LPDDR4X devices. As our

https://t.me/learningnets

Table II: **Blacksmith results for DDR4 DRAM compared to TRRespass**. For each DIMM, we report the number of effective patterns found ($|\mathbb{P}^{+}|$), i.e., patterns that triggered any bit flip during fuzzing; and the total number of bit flips found during fuzzing ($|\mathbb{F}^{\text{total}}_{\text{fuzz}}|$). For a DIMM's best pattern, we do a sweep over $256\,\text{MB}$ and report the same ($|\mathbb{F}^{\text{total}}_{\text{swp}}|$), plus the number of zero-to-one bit flips ($|\mathbb{F}^{0\to1}_{\text{swp}}|$). On three DIMMs, marked by †, we used an amplitude of up to 4 refresh intervals, see Table I.

DIMM	Blacksmith				TRRespass [12]			
	$\lvert\mathbb{P}^{+}\rvert$	$\lvert\mathbb{F}^{\text{total}}_{\text{fuzz}}\rvert$	$\lvert\mathbb{F}^{\text{total}}_{\text{swp}}\rvert$	$\lvert\mathbb{F}^{0\to1}_{\text{swp}}\rvert$	$\lvert\mathbb{P}^{+}\rvert$	$\lvert\mathbb{F}^{\text{total}}_{\text{fuzz}}\rvert$	$\lvert\mathbb{F}^{\text{total}}_{\text{swp}}\rvert$	$\lvert\mathbb{F}^{0\to1}_{\text{swp}}\rvert$
$\mathcal{A}_0$	47	330	82,183	41,471	0	–	–	–
$\mathcal{A}_1$	116	876	12,134	6,095	12	12	5	5
$\mathcal{A}_2$	462	3,543	134,702	68,801	715	16,054	7,404	4,563
$\mathcal{A}_3$	82	480	1,746	890	326	852	114	58
$\mathcal{A}_4$	460	2,988	5,132	2,602	78	105	22	9
$\mathcal{A}_5$	42	294	113,190	57,655	0	–	–	–
$\mathcal{A}_6$	102	771	98,425	49,296	4	11	4	4
$\mathcal{A}_7$	66	450	32,090	15,988	0	–	–	–
$\mathcal{A}_8$	83	657	92,660	46,914	0	–	–	–
$\mathcal{A}_9$	349	2,256	4,889	2,461	14	844	1	1
$\mathcal{A}_{10}$	350	2,193	3,051	1,532	367	961	505	280
$\mathcal{A}_{11}$	202	1,239	3,171	1,630	261	479	38	25
$\mathcal{A}_{12}$	74	576	43,581	22,149	0	–	–	–
$\mathcal{A}_{13}$	72	564	59,721	30,320	0	–	–	–
$\mathcal{A}_{14}$	51	360	64,083	32,543	1	1	4	0
$\mathcal{A}_{15}$	67	516	52,580	26,483	0	–	–	–
$\mathcal{A}_{16}$	372	2,826	99,552	51,029	688	5,499	1,450	983
$\mathcal{A}_{17}$	425	3,189	138,601	70,902	711	12,196	3,871	2,690
$\mathcal{A}_{18}$	126	936	80,601	40,876	14	14	1	1
$\mathcal{A}_{19}$	107	750	11,599	5,736	0	–	–	–
$\mathcal{B}_0$	9	66	63	22	0	–	–	–
$\mathcal{B}_1$	7	39	506	256	0	–	–	–
$\mathcal{B}_2$†	9	45	15	7	7	8	5	3
$\mathcal{B}_3$	1	6	111	58	0	–	–	–
$\mathcal{B}_4$	101	606	1,107	577	0	–	–	–
$\mathcal{B}_5$	19	105	14	6	0	–	–	–
$\mathcal{B}_6$	18	105	78	46	0	–	–	–
$\mathcal{B}_7$	4	33	70	34	0	–	–	–
$\mathcal{B}_8$†	4	21	258	131	0	–	–	–
$\mathcal{B}_9$†	40	234	1,223	625	0	–	–	–
$\mathcal{C}_0$	1	6	26	16	0	–	–	–
$\mathcal{C}_1$	16	105	28	8	0	–	–	–
$\mathcal{C}_2$	82	531	2,551	1,242	0	–	–	–
$\mathcal{C}_3$	6	36	636	296	0	–	–	–
$\mathcal{C}_4$	31	213	769	385	0	–	–	–
$\mathcal{C}_5$	23	150	1,028	516	0	–	–	–
$\mathcal{D}_0$	26	186	10,646	5,329	0	–	–	–
$\mathcal{D}_1$	37	249	6,655	3,406	3	3	0	–
$\mathcal{D}_2$	3	12	2,030	1,008	0	–	–	–
$\mathcal{D}_3$	41	276	6,797	3,475	8	8	1	1
$\sum$	**4,133**		**1.168 M**		**3,209**		**13,425**	

Table III: **Analysis of exploitation of our DRAM modules**. Given the bit flips found by Blacksmith's best pattern, we evaluate how many of these bit flips are exploitable (**#Expl.**) when considering three exploits. For each DIMM, we then computed the average time to find an exploitable bit flip (**Time**). We mark (*) values where a single measurement is available only.

DIMM	PTE [1]		RSA-2048 [4]		`sudo` [14]	
	#Expl.	Time	#Expl.	Time	#Expl.	Time
$\mathcal{A}_0$	7604	4s	210	30s	17	5m
$\mathcal{A}_1$	–	–	28	4m 12s	–	–
$\mathcal{A}_2$	9198	6s	306	21s	13	6m 43s
$\mathcal{A}_3$	73	2m 21s	3	47m 37s	–	–
$\mathcal{A}_4$	214	33s	7	13m 16s	–	–
$\mathcal{A}_5$	99	1m 27s	269	34s	12	11m 41s
$\mathcal{A}_6$	52	2m 12s	220	32s	9	11m 55s
$\mathcal{A}_7$	6043	6s	69	2m 5s	8	11m 11s
$\mathcal{A}_8$	64	2m 24s	184	54s	15	10m 5s
$\mathcal{A}_9$	136	28s	6	9m 45s	–	–
$\mathcal{A}_{10}$	216	24s	7	12m 4s	–	–
$\mathcal{A}_{11}$	197	2m 8s	13	23m 21s	–	–
$\mathcal{A}_{12}$	6596	7s	116	55s	2	38m 35s
$\mathcal{A}_{13}$	4520	8s	144	49s	7	13m 44s
$\mathcal{A}_{14}$	5172	8s	151	44s	7	14m 19s
$\mathcal{A}_{15}$	4567	8s	105	1m 3s	7	14m 7s
$\mathcal{A}_{16}$	6572	6s	231	27s	13	6m 30s
$\mathcal{A}_{17}$	9775	3s	324	11s	10	5m 1s
$\mathcal{A}_{18}$	11124	5s	182	44s	23	5m 28s
$\mathcal{A}_{19}$	832	3s	20	1m 18s	3	6m 21s
$\mathcal{B}_0$	–	–	–	–	–	–
$\mathcal{B}_1$	1	1h 44m*	1	2h 31m*	–	–
$\mathcal{B}_2$	–	–	–	–	–	–
$\mathcal{B}_3$	3	1h 16m	–	–	–	–
$\mathcal{B}_4$	2	1h 27m	4	34m 7s	–	–
$\mathcal{B}_5$	–	–	–	–	–	–
$\mathcal{B}_6$	–	–	–	–	–	–
$\mathcal{B}_7$	–	–	–	–	–	–
$\mathcal{B}_8$	–	–	1	26m 50s*	–	–
$\mathcal{B}_9$	3	1h 3m	–	–	–	–
$\mathcal{C}_0$	1	2h 8m*	–	–	–	–
$\mathcal{C}_1$	–	–	–	–	–	–
$\mathcal{C}_2$	1	1h*	3	59m 39s	–	–
$\mathcal{C}_3$	–	–	–	–	–	–
$\mathcal{C}_4$	4	59m 19s	2	2h 5m	–	–
$\mathcal{C}_5$	–	–	1	4h 2m*	–	–
$\mathcal{D}_0$	5202	4s	23	3m 43s	4	19m 56s
$\mathcal{D}_1$	4	19m 33s	15	5m 25s	–	–
$\mathcal{D}_2$	135	40s	6	11m 41s	–	–
$\mathcal{D}_3$	760	22s	32	5m 49s	–	–

LPDDR4X platform is fragile, which makes it difficult to perform longer runs, we had to reduce the run time to 6 h; even then, we had to restart multiple times until we accumulated in total 6 h. Table IV summarizes our results. We observe that Blacksmith can trigger up to two orders of magnitude more bit flips on LPDDR4X compared to DDR4 DRAM, often finding multiple bit flips in every row of every bank. This confirms previous results [15] that indicated the lower Rowhammer tolerance of LP devices, likely a direct result of the area and power restrictions. However, in contrast to DDR4, for some LPDDR4X DRAM modules from vendor $\mathcal{B}$ Blacksmith was unable to trigger any bit flip. These dram devices were produced recently (in 2020), likely deploying an improved mitigation scheme. To understand why Blacksmith failed to find any effective patterns on the devices B_{0-3}, we reverse-engineered the TRR mechanism of one of them (B_0) in Section VI-C.

E. Pattern's Complexity

We analyzed the effective patterns discovered by Blacksmith on the tested DIMMs. In Figure 11, we present three examples to show that patterns have significant differences in their parameters. Considering the complexity of these patterns, we argue that it is difficult to come up with them manually. We note that these patterns all have only one effective aggressor tuple, but we have also observed instances with more than one.

The best pattern of $\mathcal{B}_2$, given in Figure 11a, consists of 6 aggressor tuples that all share the same period (104) but $a_{5,6}$

https://t.me/learningnets

Table IV: **Blacksmith results for LPDDR4X DRAM.** We report for each chip (**DRAM**) the no. of effective patterns found ($|\mathbb{P}^+|$, or $\max$ and the elapsed time to find the first 128 effective patterns), and for the best pattern, we report the total no. of observed bit flips (**#Flips**) and the no. of zero-to-one flips ($|\mathbb{F}^{0\to1}_{\text{swp}}|$) for a sweep over $16\,\text{MB}$. Additionally, we report the total capacity (**GB**) and refresh rate changes during the experiment; e.g., 4x→2x indicates a refresh interval of 4x tREFI (4x 3.904μs $\approx$ 15.6μs) during test initialization and early fuzzing, but an increasing temperature eventually resulted in a lower refresh interval of 2x tREFI ($\approx$ 7.8μs). For C_4, the refresh interval kept alternating between 2x and 4x. All DRAM devices are from 2018, except for B_0–B_2 from 2020 (marked with †).

DRAM	GB	$\lvert\mathbb{P}^+\rvert$ (mm:ss)	#Flips	$\lvert\mathbb{F}^{0\to1}_{\text{swp}}\rvert$	Rate
A_0	6	max (17:24)	361 K	209 K	4x
A_1	6	max (13:57)	946 K	604 K	4x
A_2	8	max (21:54)	993 K	572 K	4x
A_3	8	max (15:04)	1.633 M	963 K	4x
A_4	12	max (14:53)	844 K	531 K	4x
A_5	12	max (15:05)	1.207 M	752 K	4x
B_0†	6	0	–	–	4x
B_1†	6	0	–	–	4x→2x
B_2†	6	0	–	–	4x→2x
B_3	8	max (29:27)	225 K	119 K	4x→2x
B_4	8	max (11:28)	1.516 M	797 K	4x→2x
C_0	4	max (51:18)	140 K	78 K	4x→2x
C_1	4	max (05:44)	6.560 M	3.050 M	4x
C_2	6	max (05:22)	363 K	239 K	4x
C_3	6	max (05:24)	12.242 M	5.092 M	4x
C_4	8	max (05:11)	3.125 M	1.423 M	4x→2x/4x
C_5	10	24	1,447	1,022	2x
C_6	10	5	14,386	8,689	2x
C_7	10	53	2,623	1,649	2x

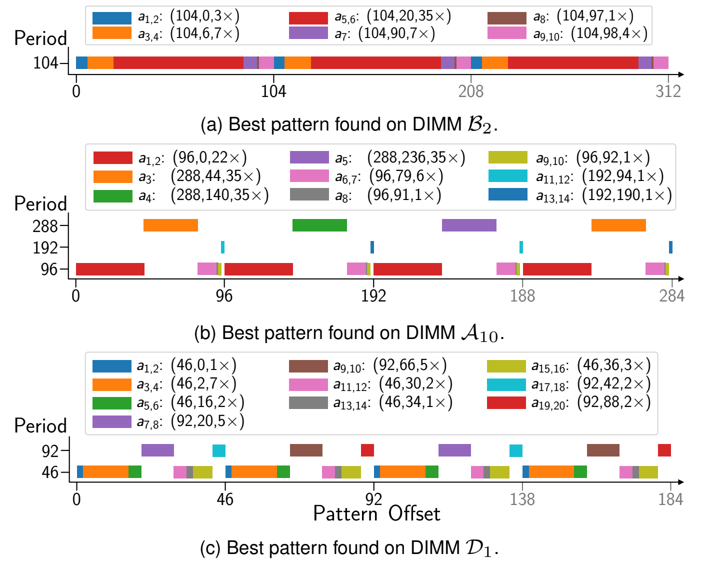
that caused the bit flips has a significantly higher intensity (35×). This very well represents how one would expect a Rowhammer pattern: the most hammered aggressors trigger bit flips. However, this is not always the case. The effective pattern in Figure 11b from $\mathcal{A}_{10}$ consists of 9 aggressor tuples, and the aggressors $a_{1,2}$ causing the bit flips are hammered with a lower intensity (22) than the pattern's highest (35) but more often (period of 96). This agrees with the observation made in our experiments (see Section III-C), showing hammering for too long (high intensity) might be counter-productive. Lastly, we show the best pattern from $\mathcal{D}_1$ in Figure 11c. This shows how intermixing our effective aggressors with other aggressors allows us to evade TRR in this instance.

F. Blacksmith on Devices From Another Vendor

A fourth vendor contacted us to test its DRAM devices against Rowhammer after the responsible disclosure. Although we have not studied these devices before, Blacksmith was able to trigger the first bit flips on them after 13m 19s, 28m 8s, and 3h 43m. This shows the strength of our scalable black-box fuzzing approach for testing DRAM devices compared to traditional reverse-engineering, which would take many weeks if not months to yield effective results.

G. Other Insights

We also investigated other properties of effective patterns like their temporal properties (Appendix F), their portability between different DIMMs (Appendix G), and the reproducibility of bit flips triggered by these patterns (Appendix H).



(a) Best pattern found on DIMM $\mathcal{B}_2$.

(b) Best pattern found on DIMM $\mathcal{A}_{10}$.

(c) Best pattern found on DIMM $\mathcal{D}_1$.

Fig. 11: **Best patterns.** The best patterns of DIMMs $\mathcal{B}_2$, $\mathcal{A}_{10}$, $\mathcal{D}_1$ with (frequency, phase, amplitude) for each aggressor tuple. After a pattern's end, we show how the pattern is repeated during its execution (grey x-axis values). The aggressor tuple that triggers bit flips is depicted in red (■).

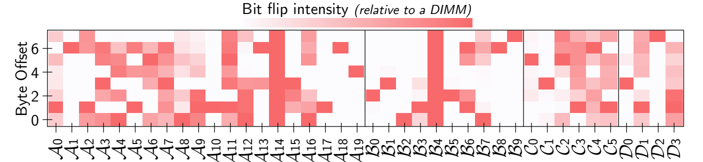


Fig. 12: **Chip dependence.** The distribution of bit flips over byte offsets (0-7) based on the DIMM's sweep with its best pattern.

VI. Insights on TRR

In this section, we investigate properties of TRR using effective patterns found by Blacksmith. In Section VI-A we start by looking into the low exploitability of some devices despite many triggered bit flips. As we wanted to understand better how TRR implementations differ across devices, we studied two characteristic properties in Section VI-B: the TRR sampler size and the TRR's dependence on DRAM addresses. Lastly, we reverse-engineered certain aspects of TRR on B_0 in Section VI-C to find out why Blacksmith could not trigger bit flips on some of the LPDDR4X devices.

A. Chip Dependence

Motivated by the low number of exploitable bit flips on some devices, despite that the best pattern triggered many bit flips, we started looking more into the bit flips from our fuzzing. An analysis of them revealed that on certain DRAM devices, some offsets show significantly more bit flips than others, as depicted in Figure 12. As an example, on $\mathcal{A}_1$ we observe that the best pattern can trigger bit flips exclusively in byte offset 6 during our sweep. Further experiments showed that using effective patterns other than the best pattern leads to bit flips on other DRAM chips but not nearly as many as when using the best pattern. Given that, we conclude that this

https://t.me/learningnets

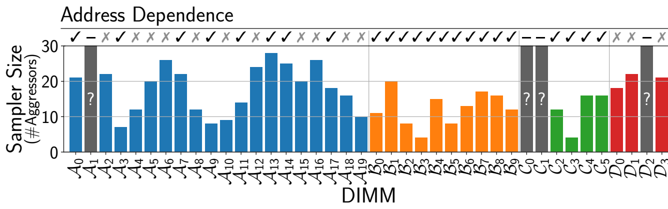


Fig. 13: **TRR Sampler size & address dependence.** We used effective patterns found by Blacksmith to estimate the TRR sampler size (bars) and to detect if TRR takes DRAM addresses into account (✓) or not (✗). A question mark (?) indicates an inconclusive sampler size.

It.	Reduction step		Bit Flips
	best pattern	randomly picked row	
1	p_0: a_a a_b a_c a_d a_e a_f	→ p_1: a_x a_b a_c a_d a_e a_f	✓
2	p_1: a_x a_b a_c a_d a_e a_f	→ p_2: a_x a_x a_c a_d a_e a_f	✓
3	p_2: a_x a_x a_c a_d a_e a_f	→ p_3: a_x a_x a_x a_d a_e a_f	✗
4	p_2: a_x a_x a_c a_d a_e a_f	→ p_4: a_x a_x a_c a_x a_e a_f	✗
5	p_2: a_x a_x a_c a_d a_e a_f	→ p_5: a_x a_x a_c a_d a_x a_f	✓
6	p_5: a_x a_x a_c a_d a_x a_f	→ p_6: a_x a_x a_c a_d a_x a_x	✓

p_6 ⇒ Sampler size equals 3 rows.

Fig. 14: **Sampler size estimation.** An example showing the sampler size estimation methodology over 6 iterations (It.). The pattern p_6 at the end has the minimum number of distinct rows to trigger bit flips.

effect is likely due to TRR rather than the chip's underlying Rowhammer vulnerability. The analysis based on the bit flips from our fuzzing shows that this effect is present on 65% of devices in our test pool. The existence of this chip-dependent variation is confirmed by concurrent work [30].

B. TRR Sampler Size and Address Dependence

To learn more about how TRR implementations differ across DIMMs in our test pool, we use the best pattern found by Blacksmith to determine the number of rows that a sampler can track at any point in time (i.e., sampler size) and if the sampler is sensitive to the DRAM address of the rows inside an effective pattern (i.e., address dependence). For increasing the reliability of our experiments, we repeat hammering each pattern ten times, each for 5 M activations.

We estimate the sampler size using a reduction process as shown in Figure 14: we iteratively replace aggressors of the best pattern by *one* randomly selected row of the same bank until any further replacement would no longer trigger bit flips anymore. The number of distinct rows at the end is an overestimation of the sampler's size. The results in Figure 13 show the sampler size varies across DRAM devices from just 4 to up to 28 rows. We report the sampler size as *inconclusive* in case that our methodology did not lead to a reliable result.

To identify any address dependence on a given DRAM device, we replace all accessed rows that do not trigger bit flips in neighboring rows (i.e., all except the effective aggressors) by randomly selected rows of the same bank. Since these aggressors in the pattern do not contribute to bit flips, replacing them should not affect the ability to trigger bit flips. Hence, if we do not observe bit flips anymore, it indicates that the sampler is address-dependent. Our results in Figure 13 show that 55% of samplers in our DIMMs are address-dependent.

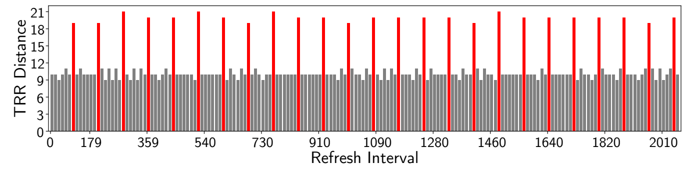


Fig. 15: **TRR distance experiment.** The TRR distance in refresh intervals for the LPDDR4X device B_0. The x-axis shows the accumulated refresh intervals since the beginning of the experiment.

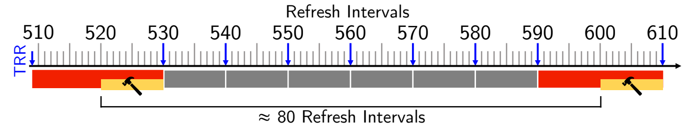


Fig. 16: **Attack strategy.** An extract of the refresh interval range 509-610 of Figure 15 in which we show the distance between TRR-free segments (■) where we target hammering our aggressors (■).

Algorithm 1: Experiment to determine the TRR distance.

1. $\mathbb{A}_2 \leftarrow$ PickRandomAggressorPair();
2. AC $\leftarrow 1.5\times$DetermineRhThreshold($\mathbb{A}_2$);
3. DisableRefresh();
4. **for** $round \leftarrow 0$ **to** 8192 **do**
5. PrepareVictimRow($\mathbb{A}_2$); `// restore data, refresh victim`
6. **for** $i \leftarrow 0$ **to** AC$/2$ **do**
7. Hammer($\mathbb{A}_2$);
8. IssueRefresh(); `// issues a single REFRESH`
9. **for** $i \leftarrow 0$ **to** AC$/2$ **do**
10. Hammer($\mathbb{A}_2$);
11. CheckBitflips($\mathbb{A}_2$);
12. EnableRefresh();

C. Understanding Blacksmith's (In)Effectiveness

Our results show that Blacksmith is able to find effective patterns on all PC-DDR DIMMs of our test pool (see Table II). There are, however, three LPDDR4X devices where Blacksmith could not find any effective pattern (see Table IV). To better understand why this is the case, we reverse-engineer aspects of the TRR implementation on device B_0. For the following experiments, we make use of our LPDDR4X-based test platform where we have control over refresh commands.

TRR distance. In the first experiment, we verify if the distance between TRRs is regularly repeating on B_0. The experiment uses the fact that a TRR-triggered refresh masks bit flips. That means, if we know how often we need to hammer a location to induce a bit flip, we can determine which `REFRESH`es trigger TRRs. The experiment, given in Algorithm 1, works as follows: we randomly pick a double-sided aggressor pair to determine its hammer count (HC), i.e., the number of accesses needed to trigger a bit flip. The HC can be determined by disabling refreshes and repeating hammering while counting the number of activations until we observe bit flips. We then define our target activation count $\mathrm{AC} = 1.5 \times \mathrm{HC}$ and hammer the aggressors for half of the times ($\mathrm{AC}/2$), issue a single refresh, and again hammer for half of the times ($\mathrm{AC}/2$) before checking for bit flips. We repeat this experiment for one `tREFW`, i.e., 8192 refresh intervals to observe the distance (in units of refresh intervals) between TRRs to our victim row.

Figure 15 shows the results for the first 2070 refresh intervals.

https://t.me/learningnets

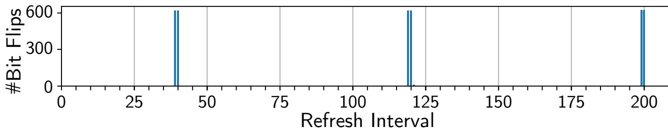


Fig. 17: **Attack result of B_0.** Our manually-crafted pattern can successfully trigger bit flips once synchronized with the proper `REFRESH`.

We can see, on average, there is a TRR happening every 10th `REFRESH`; however, there are periods where TRRs happen less frequently — roughly every 70th `REFRESH` does not trigger a TRR event, resulting in 19–21 consecutive TRR-free intervals. We conclude from the result that Blacksmith, if configured properly, should be able to bypass this TRR implementation.

Building an effective pattern. Our goal is to demonstrate that we could use the TRR-free intervals to craft an effective pattern for B_0. Our attack assumes that we are aligned with the proper refresh interval. Based on our previous observation (Figure 15), we access randomly selected rows in the short segments of 9–11 refresh intervals, and hammer our aggressors in the long TRR-free segments of 19–21 refresh intervals.

However, we first need to align with the right `REFRESH` before we start to hammer. By analyzing Figure 15 carefully, we find out that there are always around 80 refresh intervals in between the two `REFRESH`es where TRRs are skipped. To make this more clear on an example, we focus on the refresh interval range 509–610 in Figure 16. Here, we can see that the two `REFRESH`es without TRR events are around 80 refresh intervals apart. Consequently, by shifting our pattern's `REFRESH` alignment every repetition by one, we need at most 79 repetitions to find the proper start interval.

For our pattern, we choose two random rows to hammer during the short segments (i.e., for 70 intervals) and a double-sided aggressor pair (of the same bank) to hammer during the long TRR-free segments for 10 intervals. In theory, we could hammer our double-sided aggressor pair even longer but we determined that 10 intervals are enough to trigger bit flips.

Figure 17 shows the result of our hand-crafted pattern. We can see that it took 40 tries to align with the proper `REFRESH`, after which we are synchronized and can trigger bit flips. By repeating this method a few times, we can see that the distance between successful offsets (i.e., 39/40, 119/120/121, and 199/200) matches the measured distance of 80 in Figure 15.

Effective configuration of Blacksmith for LPDDR. Our experiment in Section VI-C shows that the TRR distance is regular, which means that our frequency-based patterns should be able to bypass this TRR implementation. Compared to our fuzzing parameter ranges (see Table I), the distance between where we hammer (80 refresh intervals) is within our range of 1 to 128 refresh intervals. But we need to allow hammering an aggressor tuple for at least 10 consecutive refresh intervals. This is currently not covered by our parameter range as our amplitude is limited to only one refresh interval. This explains why Blacksmith could not find any effective patterns on this device. However, even if we consider a proper configuration of Blacksmith, an effective pattern needs to start at the right refresh command. This is possible but can take a long time given the larger parameter space.

VII. Future Work

We discuss the impact of our new findings on future attacks and mitigations.

Improving the fuzzer's approach. Our work shows that with blackbox fuzzing and some assumptions about a pattern's structure, we can efficiently generate patterns bypassing TRR mitigations on a wide range of DIMMs. Although this approach is scalable and outperforms previous work [12], on certain DIMMs we could only find very few bit flips. This leaves improvements to our fuzzing strategy as an attractive direction for future research.

One possibility is tweaking the parameters of effective patterns found by Blacksmith to discover new effective patterns that can trigger more bit flips. This, however, assumes Blacksmith has already found effective patterns.

In situations where Blacksmith does not find effective patterns, reverse-engineering can provide an alternative. As adequate reverse-engineering of a DIMM is time-consuming and does not scale, an interesting approach can combine automated reverse engineering to guide Blacksmith in a grey-box manner. As an example, reverse-engineering can provide the distance between TRRs (Section VI-C). This information, in turn, can be used by Blacksmith to significantly reduce the size of the search space.

Making TRR more secure. Blacksmith enables scalable and effective fuzzing of a given DRAM device. Since our initial disclosure, major companies have already started using Blacksmith to test their devices and evaluate the effectiveness of their mitigations. We are confident that this adoption will directly result in improved future mitigations.

The properties of effective Blacksmith patterns can also guide the design of better mitigations. Blacksmith can trigger bit flips on our DRAM devices since their TRR implementations do not accurately capture aggressor rows. In deterministic mitigations with strong security guarantees, *every access* needs to be considered, unlike existing in-DRAM mitigations. Recent work [31] shows how this can be achieved with a reasonably small number of counters. Our measurements show that currently deployed mitigations keep track of significantly fewer aggressors than needed for complete protection. Probabilistic mitigations (e.g., PARA [13]) can also be used as secure in-DRAM mitigation, but recent work shows that additional refreshes have become prohibitively expensive in recent devices [15].

VIII. Related Work

In this section, we provide an overview of existing work on Rowhammer attacks and defenses.

Attacks. While initially considered an exotic attack vector, Rowhammer has since emerged as an effective means to build a plethora of exploits [32] on a great variety of platforms: on personal computers [1]–[3], [14], mobile platforms [6]–[8], and co-located cloud servers [4], [5], [25]. Attacks were not

https://t.me/learningnets

only demonstrated using native code [1], [4]–[7], [14], [25] but also from the restricted JavaScript sandbox running in modern browsers [2], [3], [8], [11] and even over the network [9], [10]. While TRRespass [12] showed the Rowhammer issue still affects some DDR4 systems, the patterns generated by Blacksmith expose how every DDR4 system is still vulnerable to it — even more so in the case of LPDDR4. Such results make the case for better mitigations more significant.

Typical Rowhammer attacks consist of three phases [4]: (i) memory templating, (ii) memory massaging, and (iii) exploitation. During (i) memory templating, an attacker aims to find a pattern that triggers a bit flip at an attack-dependent offset of a page (*template*). This is where Blacksmith comes into play and can help to find an effective pattern. Thereafter, (ii) memory massaging is used to trick the victim into mapping the target data into one of the attacker's templates in which the attacker can trigger a bit flip during the (iii) exploitation.

Concurrent work [21] uses a new reverse engineering technique based on data retention failures for studying mitigations and crafting patterns that effectively bypass TRR. The methodology leads to very effective patterns but is time-consuming as it is not automated. Similar to our insights on mitigations (see Section VI), recent work [30] studied the Rowhammer sensitivities such as DRAM chip temperature and the Rowhammer effects of keeping aggressor rows active for a longer time. Among others, they make a similar observation regarding the different Rowhammer bit flip distributions across different DRAM chips on the same device as shown in Section VI-A.

Defenses. In the past, systems vendors have made several attempts to mitigate Rowhammer practically, such as an increased (e.g., doubled) DRAM refresh rate [33], [34] to reduce the available time to hammer. Besides this being insufficient [12], [35], it also increases power consumption and lowers system performance [32]. It has long been believed that servers with integrity-protected error checking and correction (ECC) DRAM are safe against Rowhammer, until Cojocar et al. [25] showed that this is not always the case.

More recent proposals use tailored solutions against Rowhammer. For example, Intel's proprietary MC-based implementation pseudo-TRR (pTRR) [36] that is available on selected server systems [12] and requires pTRR-compliant DIMMs. Little is known about its implementation, but it promises a negligible performance impact [37]. There have been ongoing standardization efforts for mitigations, such as in the latest generation of LPDDR (LPDDR5). In LPDDR5, TRR is replaced by Refresh Management [38], [39] — a mechanism that keeps track of activations in a bank and issues selective refreshes to highly activated rows once a threshold has been reached. However, this requires supported DRAM modules and coordination between DRAM and memory controllers [38].

There has been extensive research on novel software- and hardware-based defenses that try to implement a more effective TRR. Software-based defenses may be deployed on systems with DRAM modules that are already in production [7], [35], [40], [41]. However, they require support by the OS, do not always provide complete protection [14], [42], can waste memory [7], [35], and potentially impact performance more negatively [41]. In comparison, hardware-based solutions have a lower performance overhead [13], [31], [43]–[46], but they require hardware adoption that can take many years.

IX. Conclusion

Deployed in-DRAM TRR mitigations against Rowhammer estimate hammered rows and aim to prevent bit flips by issuing extra refreshes to their neighbors. Motivated by the observation that all existing Rowhammer patterns hammer their aggressors uniformly, and given that this is likely an easy case to catch by TRR, we explored the novel class of *non-uniform* Rowhammer access patterns by randomizing parameters in the frequency domain, obtained using a number of carefully-crafted experiments. Our scalable Rowhammer fuzzer *Blacksmith*, is capable of crafting complex non-uniform patterns that trigger bit flips on all 40 recently-acquired DDR4 DIMMs, 2.6× more than state-of-the-art Rowhammer patterns. We used results obtained by Blacksmith to gain insight into the properties of effective patterns and existing mitigations. Our findings highlight an urgent need for the deployment of more principled mitigations against Rowhammer.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. 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] S. Mark and T. Dullien, "Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges: How to cause and exploit single bit errors," https://www.youtube.com/watch?v=0U7511Fb4to, Black Hat USA, Las Vegas, NV, Aug. 2015.

[2] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript," in *DIMVA '16*. Berlin, Heidelberg: Springer-Verlag, Jul. 2016, pp. 300–321, https://doi.org/10.1007/978-3-319-40667-1_15.

[3] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida, "Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector," in *S&P '16*. San Jose, CA: IEEE, May 2016, pp. 987–1004, http://ieeexplore.ieee.org/document/7546546/.

[4] K. Razavi, B. Gras, C. Giuffrida, E. Bosman, B. Preneel, and H. Bos, "Flip Feng Shui: Hammering a Needle in the Software Stack," in *USENIX Security '16*, 2016, https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/razavi.

[5] 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 '16*, Austin, TX, Aug. 2016, pp. 19–35, https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/xiao.

[6] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic Rowhammer Attacks on Mobile Platforms," in *CCS '16*. Vienna Austria: ACM, Oct. 2016, pp. 1675–1689, https://dl.acm.org/doi/10.1145/2976749.2978406.

[7] V. van der Veen, M. Lindorfer, Y. Fratantonio, H. Padmanabha Pillai, G. Vigna, C. Kruegel, H. Bos, and K. Razavi, "GuardION: Practical Mitigation of DMA-Based Rowhammer Attacks on ARM," in *DIMVA*, Jun. 2018, https://link.springer.com/chapter/10.1007/978-3-319-93411-2_5.

https://t.me/learningnets

[8] P. Frigo, C. Giuffrida, H. Bos, and K. Razavi, "Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU," in *IEEE S&P '18*, May 2018, pp. 195–210, https://ieeexplore.ieee.org/abstract/document/8418604.

[9] A. Tatar, R. K. Konoth, C. Giuffrida, H. Bos, E. Athanasopoulos, and K. Razavi, "Throwhammer: Rowhammer Attacks over the Network and Defenses," in *USENIX ATC '18*, 2018, p. 14, https://www.usenix.org/conference/atc18/presentation/tatar.

[10] 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&P Workshops '20*, Sep. 2020, pp. 710–719, https://ieeexplore.ieee.org/abstract/document/9229701/.

[11] 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 '21*, Aug. 2021, https://www.usenix.org/conference/usenixsecurity21/presentation/ridder.

[12] P. Frigo, E. Vannacc, H. Hassan, V. v. der Veen, O. Mutlu, C. Giuffrida, H. Bos, and K. Razavi, "TRRespass: Exploiting the many sides of target row refresh," in *IEEE S&P '20*, 2020, pp. 747–762, https://ieeexplore.ieee.org/abstract/document/9152631.

[13] 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 '14*. Minneapolis, MN, USA: IEEE, Jun. 2014, pp. 361–372, http://ieeexplore.ieee.org/document/6853210/.

[14] 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 '18*, May 2018, pp. 245–261, https://ieeexplore.ieee.org/abstract/document/8418607.

[15] J. S. Kim, M. Patel, A. G. Yaglikci, H. Hassan, R. Azizi, L. Orosa, and O. Mutlu, "Revisiting RowHammer: An Experimental Analysis of Modern DRAM Devices and Mitigation Techniques," in *ISCA '20*. Valencia, Spain: IEEE, May 2020, pp. 638–651, https://ieeexplore.ieee.org/document/9138944/.

[16] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu, "A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM," in *ISCA '12*, p. 12, https://ieeexplore.ieee.org/abstract/document/6237032.

[17] "JEDEC Standard: DDR4 SDRAM (JESD79-4B)," https://www.jedec.org/sites/default/files/docs/JESD79-4.pdf, Jun. 2017.

[18] J. Liu, B. Jaiyen, Y. Kim, C. Wilkerson, and O. Mutlu, "An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms," in *ACM SIGARCH '13*, p. 12, http://ieeexplore.ieee.org/document/6853210/.

[19] W.-K. Cheng, P.-Y. Shen, and X.-L. Li, "Retention-Aware DRAM Auto-Refresh Scheme for Energy and Performance Efficiency," *Micromachines*, vol. 10, no. 9, p. 590, Sep. 2019, https://www.mdpi.com/2072-666X/10/9/590.

[20] K. S. Bains, J. B. Halbert, C. P. Mozak, T. Z. Schoenborn, and Z. Greenfield, "Row Hammer Refresh Command," Patent, Jun., 2012, https://patents.google.com/patent/US20140006703A1/en.

[21] 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 '21*. Virtual Event Greece: ACM, Oct. 2021, pp. 1198–1213, https://dl.acm.org/doi/10.1145/3466752.3480110.

[22] P. Frigo, E. Vannacci, H. Hassan, O. Mutlu, C. Giuffrida, H. Bos, and K. Razavi, "TRRespass," https://github.com/vusec/trrespass, 2020.

[23] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks," in *USENIX Security '16*, p. 18, https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/pessl.

[24] L. Cojocar, J. Kim, M. Patel, L. Tsai, S. Saroiu, A. Wolman, and O. Mutlu, "Are We Susceptible to Rowhammer? An End-to-End Methodology for Cloud Providers," in *IEEE S&P*. San Francisco, CA, USA: IEEE, May 2020, pp. 712–728, https://ieeexplore.ieee.org/document/9152654/.

[25] 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 '19*. San Francisco, CA, USA: IEEE, May 2019, pp. 55–71, https://ieeexplore.ieee.org/document/8835222/.

[26] P. Kobalicek, "AsmJit: A lightweight library for X86/X64 machine code generation written in C++," https://asmjit.com/, 2011.

[27] L. Niels, "JSON for Modern C++," https://github.com/nlohmann/json, 2011.

[28] A. Tatar, C. Giuffrida, H. Bos, and K. Razavi, "Defeating Software Mitigations against Rowhammer: A Surgical Precision Hammer," in *RAID '18*, Sep. 2018, https://link.springer.com/chapter/10.1007/978-3-030-00470-5_3.

[29] "JESD209-4A: Low Power Double Data Rate 4 (LPDDR4)," https://www.jedec.org/sites/default/files/docs/JESD79-4A.pdf, Aug. 2014.

[30] 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 '21*. Virtual Event Greece: ACM, Oct. 2021, pp. 1182–1197, https://dl.acm.org/doi/10.1145/3466752.3480069.

[31] Y. Park, S. N. University, W. Kwon, S. N. University, E. Lee, S. N. University, T. J. Ham, S. N. University, J. H. Ahn, S. N. University, J. W. Lee, and S. N. University, "Graphene: Strong yet Lightweight Row Hammer Protection," in *MICRO '20*, 2020, p. 13, https://ieeexplore.ieee.org/abstract/document/9251863.

[32] O. Mutlu and J. S. Kim, "RowHammer: A Retrospective," *IEEE TCADICS '19*, vol. 39, no. 8, pp. 1555–1571, Aug. 2020, https://doi.org/10.1109/TCAD.2019.2915318.

[33] "About the security content of Mac EFI Security Update 2015-001," https://support.apple.com/en-us/HT204934.

[34] "Row Hammer Privilege Escalation - Lenovo Support CH," https://support.lenovo.com/ch/en/product_security/row_hammer.

[35] 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 '16*. Atlanta, Georgia, USA: ACM Press, 2016, pp. 743–755, http://dl.acm.org/citation.cfm?doid=2872362.2872390.

[36] M. Kaczmarski, "Thoughts on Intel® Xeon® E5-2600 v2 Product Family Performance Optimisation – component selection guidelines," http://infobazy.gda.pl/2014/pliki/prezentacje/d2s2e4-Kaczmarski-Optymalna.pdf, Aug. 2014.

[37] S. Mandava, B. S. Morris, S. Sah, R. M. Stevens, T. Rossin, M. W. Stefaniw, and J. H. Crawford, "Techniques for determining victim row addresses in a volatile memory," US Patent US9 824 754B2, Nov., 2017, https://patents.google.com/patent/US9824754B2/en.

[38] A. Hastings and S. Sethumadhavan, "WaC: A new doctrine for hardware security," in *ASHES '20*, ser. ASHES'20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 127–136, https://doi.org/10.1145/3411504.3421217.

[39] "JEDEC Standard: LPDDR5 (JESD209-5)," https://www.jedec.org/sites/default/files/docs/JESD209-5.pdf, 2019.

[40] 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 '17*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 117–130, https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/brasser.

[41] 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 Security '18*, p. 15, https://www.usenix.org/conference/osdi18/presentation/konoth.

[42] Z. Zhang, Y. Cheng, D. Liu, S. Nepal, Z. Wang, and Y. Yarom, "PThammer: Cross-User-Kernel-Boundary Rowhammer through Implicit Accesses," in *MICRO '20*, Oct. 2020, pp. 28–41, https://ieeexplore.ieee.org/abstract/document/9251982.

[43] E. Lee, I. Kang, S. Lee, G. E. Suh, and J. H. Ahn, "TWiCe: Preventing row-hammering by exploiting time window counters," in *ISCA '19*. Phoenix Arizona: ACM, Jun. 2019, pp. 385–396, https://dl.acm.org/doi/10.1145/3307650.3322232.

[44] A. G. Yağlıkçi, 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 '21*, 2021, pp. 345–358, https://ieeexplore.ieee.org/abstract/document/9407238.

[45] M. Son, H. Park, J. Ahn, and S. Yoo, "Making DRAM Stronger Against Row Hammering," in *DAC '17*. Austin TX USA: ACM, Jun. 2017, pp. 1–6, https://dl.acm.org/doi/10.1145/3061639.3062281.

[46] J. M. You and J. Yang, "MRLoc: Mitigating Row-Hammering based on memory Locality," in *DAC '19*, Jun. 2019, pp. 1–6, https://ieeexplore.ieee.org/abstract/document/8806778.

[47] "4Gb: X16 DDR4 SDRAM Features, EDY4016A - 256Mb X16," https://www.micron.com/-/media/client/global/documents/products/data-sheet/dram/ddr4/4gb_ddr4_dram_2e0d.pdf.

https://t.me/learningnets

Table V: **Synchronized n-sided patterns.** Number of effective patterns ($|\mathbb{P}^+|$) and bit flips ($|\mathbb{F}^{\text{total}}_{\text{fuzz}}|$) found during fuzzing using n-sided patterns with `REFRESH` synchronization (TRRespass + Sync.) compared to regular n-sided patterns (TRRespass). DIMMs where we could not find any patterns with `REFRESH` synchronization are omitted.

DIMM	TRRespass + Sync. (SMASH)		TRRespass	
	$\|\mathbb{P}^+\|$	$\|\mathbb{F}^{\text{total}}_{\text{fuzz}}\|$	$\|\mathbb{P}\|$	$\|\mathbb{F}^{\text{total}}_{\text{fuzz}}\|$
$\mathcal{A}_2$	2,233	8,131	777	3,279
$\mathcal{A}_3$	24	77	53	79
$\mathcal{A}_4$	5	15	5	5
$\mathcal{A}_9$	40	121	47	65
$\mathcal{A}_{10}$	54	165	57	72
$\mathcal{A}_{11}$	16	48	26	27
$\mathcal{A}_{16}$	25	87	310	499
$\mathcal{A}_{17}$	1,312	4,299	593	1,574
$\mathcal{B}_2$	5	16	0	–

Appendix A
Synchronized n-sided Hammering

Recent work [11] showed that synchronizing with the `REFRESH`es while hammering facilitates bypassing Rowhammer mitigations. To investigate whether adding synchronization to n-sided patterns is enough to find effective patterns on more devices than previous work [12] did, we added synchronization to the open-source Rowhammer fuzzer TRRespass.

In Table V, we present the results for a 30 minutes run: we found effective patterns on only 9 of 40 DIMMs (22.5%) of our test pool (see Appendix B), which indicates that synchronization alone is insufficient to find effective patterns on more DIMMs. Our results show that although we do not always find more effective patterns, the effective patterns we found trigger a higher number of bit flips. This matches observations from previous work [11].

Appendix B
PC-DRAM Test Pool

In Table VI, we provide an overview on the DIMMs in our test pool based on the DIMM's reported Serial Presence Detect (SPD) data. We group DIMMs by their DRAM chip vendor (e.g., $\mathcal{A}$) and assign to each a sequentially chosen number (e.g., $\mathcal{A}_0$, $\mathcal{A}_1$, …, $\mathcal{A}_{19}$) to uniquely identify them.

We check whether DIMMs report being Rowhammer-safe, by reading out their maximum activate count (`MAC`): the maximum number of `ACTIVATE`s that a row can resist in an interval of less or equal to the maximum activate window (MAW) without causing flips in neighboring rows [47]. All modules claim to be safe against Rowhammer (*unlimited* `MAC` value).

Appendix C
Search Space Estimation

We following present a simple back-of-the-envelope calculation showing the number of possible combinations for the most simple case of a Rowhammer pattern. We assume the standard DDR4 parameters: a `tREFI` of 7.8125 µs and a retention time (*refresh window*) of 64 ms, see Section II-A for details. This gives us $64\,\text{ms}/7.8125\,\mu\text{s} = 8192$ refresh intervals in each of which we can issue roughly 100 activations, and thus, in total around 819200 activations in a refresh window.

Table VI: **Data of the DIMMs in our testpool.** DIMMs are grouped by their vendor ($\mathcal{A} - \mathcal{D}$). If a DIMM's SPD chip does not report a manufacturing date (†), we instead report its purchase date.

Module	Date (*yy-ww*)	Freq. (*MHz*)	Size (*GB*)	Organization		
				#Ranks	#Banks	#Pins
$\mathcal{A}_0$	20-03	2666	8	1	16	×8
$\mathcal{A}_1$	20-07†	2400	8	1	16	×8
$\mathcal{A}_2$	20-06	2666	32	2	16	×8
$\mathcal{A}_3$	20-10	2400	8	1	16	×8
$\mathcal{A}_4$	16-51	2132	4	1	16	×8
$\mathcal{A}_{5-10}$	20-07†	2132	8	1	16	×8
$\mathcal{A}_{11}$	20-07†	2400	8	1	16	×8
$\mathcal{A}_{12-15}$	20-07†	2132	16	2	16	×8
$\mathcal{A}_{16}$	20-23	2666	32	2	16	×8
$\mathcal{A}_{17}$	20-08	2666	32	2	16	×8
$\mathcal{A}_{18}$	20-07†	2666	8	1	16	×8
$\mathcal{A}_{19}$	20-16	2666	16	2	16	×8
$\mathcal{B}_0$	19-38	2400	16	2	16	×8
$\mathcal{B}_1$	20-07†	2132	8	1	16	×8
$\mathcal{B}_2$	19-34	2400	4	1	16	×8
$\mathcal{B}_3$	20-05	2666	8	1	16	×8
$\mathcal{B}_4$	20-07	2400	8	1	16	×8
$\mathcal{B}_5$	19-51	2400	16	2	16	×8
$\mathcal{B}_6$	20-07†	2132	32	2	16	×8
$\mathcal{B}_7$	20-09	2134	8	2	16	×8
$\mathcal{B}_8$	20-07†	2400	4	1	16	×8
$\mathcal{B}_9$	20-07†	2400	8	1	16	×8
$\mathcal{C}_0$	20-07†	2132	16	2	16	×8
$\mathcal{C}_{1-4}$	20-38	2400	8	1	16	×8
$\mathcal{C}_5$	17-48	2400	4	1	16	×8
$\mathcal{D}_0$	20-15	2400	8	1	16	×8
$\mathcal{D}_1$	20-19	2400	16	2	16	×8
$\mathcal{D}_2$	20-20	2400	16	2	16	×8
$\mathcal{D}_3$	20-20	2400	8	1	16	×8

Next, we derive the number of distinct patterns that we can build given these constraints. We assume a Rowhammer threshold of 10 k activations based on the findings in previous work [15]. This means there are in total 10 k accesses to aggressors needed to trigger a bit flip. For simplifying the calculation, we allow aggressor accesses to be intermixed with other accesses of the pattern (e.g., accesses required to bypass TRR). In the case of a double-sided aggressor pair, this translates to $\binom{819200}{10000} = 6.79322 \times 10^{23447}$ possible combinations to distribute these 10 k accesses.

This shows that by considering only basic constraints, we end up with an impractically large pattern design space that cannot be explored in a reasonable time. Therefore, we need to define properties that allow us to reduce the search space while still being general enough to generate patterns that are effective on many different DIMMs. As discussed in Section III, examples of such properties are the number of aggressors to hammer, how often we hammer each of them, and over how many intervals we spread our hammering effort.

https://t.me/learningnets

Table VII: **Results of our non-uniform accesses experiment**. We compare common n-sided patterns (n-sided) with n-sided patterns where non-uniform accesses are injected (n-sided + Rnd.) and fully random patterns (Fully Rnd.). We report for each DIMM if any effective patterns were found (✓) or not (✗). DIMMs without any effective patterns in all three experiments are omitted for brevity.

Module	n-sided	n-sided + Rnd.	Fully Rnd.	Module	n-sided	n-sided + Rnd.	Fully Rnd.
$\mathcal{A}_1$	✓	✓	✗	$\mathcal{A}_{16}$	✓	✓	✓
$\mathcal{A}_2$	✓	✓	✓	$\mathcal{A}_{17}$	✓	✓	✓
$\mathcal{A}_3$	✓	✓	✓	$\mathcal{A}_{18}$	✓	✗	✗
$\mathcal{A}_4$	✓	✓	✓	$\mathcal{B}_1$	✓	✗	✗
$\mathcal{A}_6$	✗	✓	✗	$\mathcal{B}_2$	✓	✗	✗
$\mathcal{A}_7$	✗	✗	✓	$\mathcal{B}_9$	✗	✗	✓
$\mathcal{A}_9$	✓	✓	✓	$\mathcal{C}_0$	✓	✗	✓
$\mathcal{A}_{10}$	✓	✓	✓	$\mathcal{D}_0$	✓	✗	✓
$\mathcal{A}_{11}$	✓	✓	✓	$\mathcal{D}_1$	✗	✓	✓
$\mathcal{A}_{14}$	✗	✗	✓	$\mathcal{D}_3$	✗	✗	✓
					35%	**27.5%**	**37.5%**

Appendix D
Random Accesses Experiment

We assess three different approaches to generate Rowhammer access patterns: n-sided patterns from previous work [12], n-sided patterns with random accesses in between, and patterns where all except aggressor accesses are fully random. The results of this experiment are presented in Table VII.

Appendix E
Pattern Generation

In this appendix, we explain the technicalities involved in building patterns. We describe how we determine *harmonic frequencies* that respect pattern repetitions, we explain how matching frequencies can fill up a pattern, and finally, we present our algorithm for combining different aggressors with different parameters into a single pattern.

Harmonic Frequencies. There are constraints in the choice of an aggressor tuple's frequency. As the whole pattern is repeated during hammering, we must design it in a way to maintain frequencies over repetitions. For example, given a pattern of 4 periods (like the one in Figure 10), then choosing $f = \frac{1}{3}$ for an aggressor tuple $\mathbb{A}_2$, would lead to accessing the tuple in the first and fourth period. However, repeating the pattern leads to accessing $\mathbb{A}_2$ again in the subsequent (5th) period, thus deviating from its defined frequency.

As a solution, we define a subset of compatible frequencies, namely *harmonic frequencies*. For that, we first define $\mathbb{F} = \{\frac{1}{i} : i \in \mathbb{N}\}$ as the set of all frequencies. Then, let N be the number of periods the pattern is composed of and let $\mathbb{P} = \{2^x : x \in \mathbb{N}_0\}$ be the powers-of-two. Next, we determine the largest $p_i \in \mathbb{P}$ such that p_i divides N. Consequently, all elements $\{p_0, \dots, p_{i-1}\} \in \mathbb{P}$, smaller than p_i, must also divide N. Then we define the set of harmonic frequencies as $\mathbb{F}' = \{\frac{1}{p_0}, \frac{1}{p_1}, \dots, \frac{1}{p_i}\}$. For example, for $N = 40$ we obtain the set of harmonic frequencies $\mathbb{F}' = \{\frac{1}{1}, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}\}$ where all s^{-1} for $s \in \mathbb{F}'$ are divisors of N.

Iteration \ Period	1		2		3		4		5		6		7		8	
①	a_1	a_2							a_1	a_2						
②	a_1	a_2	a_3	a_4					a_1	a_2						
③	a_1	a_2	a_3	a_4	a_5	a_6			a_1	a_2			a_5	a_6		
④	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_1	a_2			a_5	a_6		
⑤	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_1	a_2	a_9	a_{10}	a_5	a_6		
⑥	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_1	a_2	a_9	a_{10}	a_5	a_6	a_{11}	a_{12}

Fig. 18: **Pattern generation over time.** Example showing the pattern generation over six iterations, where an iteration i is marked by ⓘ.

Matching Frequencies. Another difficulty is that we cannot arbitrarily combine different frequencies in a pattern because this may lead to overlapping accesses. By means of illustration, consider a pattern of length 16 and period $\mathcal{T} = 2$, as given in Figure 18. Before starting to fill up the pattern, we compute the harmonic frequencies $\mathbb{F}' = \{\frac{1}{1}, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}\}$. In iteration ①, we can choose any frequency $s \in \mathbb{F}'$, for example, $f = \frac{1}{4}$ for the aggressor tuple (a_1, a_2). For the next aggressor tuple (a_3, a_4), to be placed in the second period, there are fewer options available as we cannot choose $f = \frac{1}{1}$ anymore without overlapping with aggressors (a_1, a_2). As an example, we choose $f = \frac{1}{8}$ for this tuple (see ②), resulting in two remaining compatible frequencies (i.e., $f = \{\frac{1}{4}, \frac{1}{8}\}$), for the next aggressor tuple (a_5, a_6). If we choose $f = \frac{1}{8}$, we end up in iteration ③ with three unfilled periods $(4, 6, 8)$ that only support the frequency $f = \frac{1}{8}$. In iterations ④-⑥, we show how these accesses would be filled up by other aggressor tuples.

To automate selecting only from suitable frequencies, we proceed as follows: a random frequency $s_0 \in \mathbb{F}'$ is picked in the iteration ①. In following iterations (ⓘ > 0), where we add another tuple, we restrict ourselves to frequencies $\mathbb{F}'' = \{s \in \mathbb{F}' : s \leq s_{i-1}\}$, i.e., the current frequency is the upper bound for the available frequencies in the next iteration.

Building Patterns. The next step is to combine multiple aggressor tuples in a pattern (at different phases) to increase the probability that one of the aggressor tuples can successfully bypass the mitigation. We could include only one aggressor tuple with a randomly picked set of parameters $(f, \phi, \hat{u})$ and randomize all other accesses. Our chosen approach, however, allows us to simultaneously try out different parameter sets $(\{(f_1, \phi_1, \hat{u}_{,1}), (f_2, \phi_2, \hat{u}_{,2}), \dots\}$ and DRAM locations; as such, we expect it to find effective patterns more quickly.

However, combining aggressor tuples with different parameters in a pattern brings up new challenges: we need to ensure accesses do not overlap and the parameters of each aggressor tuple are respected. Additionally, we want to make sure that we exhaust but not exceed the possible number of accesses in each period to stay synchronized with the `REFRESH`.

To solve this, we implemented a pattern building algorithm. It uses the fact that a pattern can be expressed as a $A \times B$ matrix, where A refers to the pattern's number of periods and B to the base period $\mathcal{T}$. Each index $(a, b) \in A \times B$ refers to a single access in the pattern, which we will refer to as a *slot*. The algorithm fills up the free slots of an access pattern by adding an aggressor tuple to the first period (lines 3 to 8)

https://t.me/learningnets

Alg. 2: Frequency-based pattern generation.

Input : period $\mathcal{T}$, pattern's length L
Output : access pattern P

1. $\mathbb{F}' \leftarrow$ ComputeHarmonicFrequencies();
2. $P \leftarrow$ CreatePattern(L);
3. **for** $\phi \leftarrow 0$ **to** $\mathcal{T}$ **do** `// fill 1st period at phase` ϕ
4. $n \leftarrow$ PickRandomN($\mathcal{T} - \phi$);
5. $\mathbb{A} \leftarrow$ PickRandomAggressors(n);
6. $\hat{u} \leftarrow$ PickRandomAmplitude($\lfloor(\mathcal{T} - \phi)/n\rfloor$);
7. $f \leftarrow$ PickRandomFrequency($\mathbb{F}'$);
8.   FillPatternByAggressors($\mathbb{A}, f, \phi, \hat{u}, \phi, P$);
9. $\mathbb{F}'' \leftarrow \mathbb{F}'$; `// Copy` $\mathbb{F}'$ `to preserve its value`
    `/* fill remaining periods at offset` ϕ `using same values for` n, $\hat{u}$, ϕ `*/`
10.   **while** *not every slot at* ϕ *in period* $i > 1$ *is filled* **do**
11. $i \leftarrow$ GetNextUnfilledPeriod(ϕ);
12. $\Phi \leftarrow (i \times \mathcal{T}) + \phi$;
13. $\mathbb{A} \leftarrow$ PickRandomAggressors(n);
14. $\mathbb{F}'' \leftarrow$ RemoveFrequenciesLargerThan($\mathbb{F}'', f$);
15. $f \leftarrow$ PickRandomFrequency($\mathbb{F}''$);
16.   FillPatternByAggressors($\mathbb{A}, f, \phi, \hat{u}, \Phi, P$);
17. $\phi \leftarrow n + \hat{u}$; `// update iteration variable`
18. **return** P

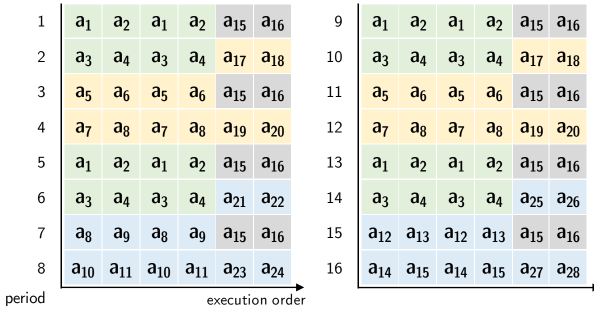


Fig. 19: **A frequency-based pattern.** Example of a frequency-based pattern with $\mathcal{T} = 6$, pattern length 96, and 16 periods. Aggressors are colored based on their frequency f: $\frac{1}{2}$ ( ), $\frac{1}{4}$ ( ), $\frac{1}{8}$ ( ), $\frac{1}{16}$ ( ).

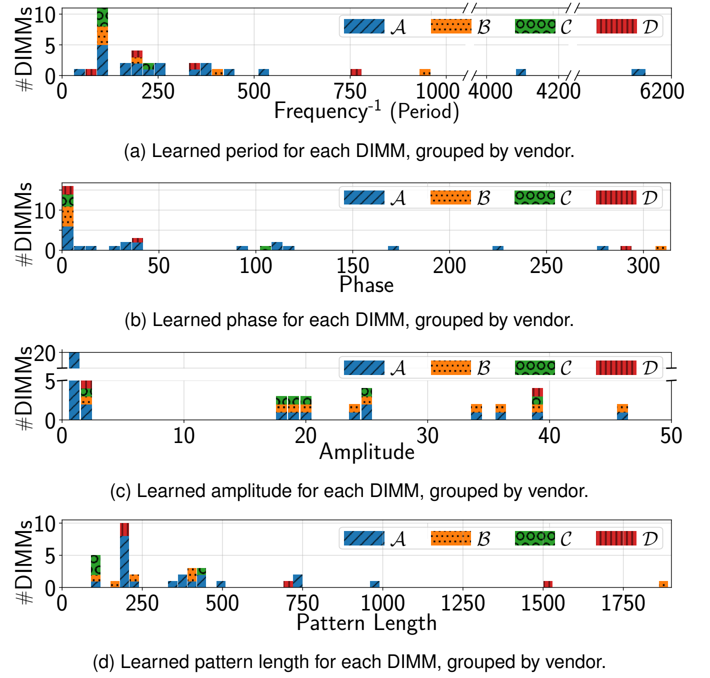
with a randomly picked set of $(f, \phi, \hat{u})$, and then (lines 10 to 16), fills up the same phase ϕ in all other periods with another aggressor tuple with the same amplitude $\hat{u}$ but a second, randomly picked, compatible frequency f. Reusing the same amplitude for aggressor tuples in all other periods (at the same phase only) is a limitation that facilitates patterns' construction. However, we believe that it does not impose a severe limitation because our patterns already consist of aggressor tuples with potentially many different amplitudes.

Pattern Example. In Figure 19, we provide a complete example of a pattern with 96 accesses and a period of 6.

Appendix F
Temporal Properties

We extended our Blacksmith fuzzer with a parameter-tracking mode, as described in Section IV-C. Using this mode, we want to answer whether a correlation between specific parameter values and vendors exists. In Figure 20, we show for each DRAM vendor how the temporal properties converged to certain values. The properties we consider include (a) period, (b) phase, (c) amplitude, and (d) pattern length.

Looking at the **frequencies** (Figure 20a) shows that $\mathcal{A}$ tends towards a high frequency (i.e., low period). We can see for $\mathcal{A}$ a



(a) Learned period for each DIMM, grouped by vendor.

(b) Learned phase for each DIMM, grouped by vendor.

(c) Learned amplitude for each DIMM, grouped by vendor.

(d) Learned pattern length for each DIMM, grouped by vendor.

Fig. 20: **Temporal properties.** We show for each DIMM, grouped by DRAM vendor, the learned values of the temporal properties frequency, phase, amplitude, and pattern length.

period of around 110 (about one refresh interval) up to ≈ 400 (about four refresh intervals). $\mathcal{D}$ is similar, although both have a few outliers. The **phase** plot (Figure 20b) shows for $\mathcal{A}$ and $\mathcal{D}$ a strong preference towards a very low phase, i.e., hammering at the beginning of a refresh interval. However, this does not contradict our earlier findings in Section III-B: Blacksmith also found effective patterns with effective aggressors at the end of a refresh interval (i.e., high offset). We can see in the **amplitude** plot (Figure 20c) that there is a clear preference for $\mathcal{A}$ to an amplitude of one, whereas $\mathcal{B}$ favors an amplitude in the range 18-25. Lastly, the **length** of effective patterns (Figure 20d) shows that there is a tendency towards shorter patterns (≤ 500 accesses) for all vendors. Three ranges clearly stand out: the peak by $\mathcal{A}$ with pattern length 190-220, the peak by $\mathcal{C}$ with 100-130 accesses, and the two instances with very long patterns (≈ 1500 by $\mathcal{D}$ and ≈ 1800 by $\mathcal{B}$).

We can summarize that for some parameters, there is a clear preference for DIMMs of the same vendor. It is possible to use this knowledge in future work to tweak the fuzzer's parameter search space further. We discuss this more in Section VII.

Appendix G
Portability of Blacksmith Patterns

Our data analysis raised the question if effective patterns are *portable*, i.e., can be transferred between DIMMs. Because a pattern inherently encodes information to bypass the mitigation, a pattern working on different DIMMs would suggest that their deployed mitigations work similarly. From an attacker's perspective, portability is of interest as it allows to perform templating on another machine (*offline*) and later, during the attack, use the *golden patterns* found on the victim's host. This

https://t.me/learningnets

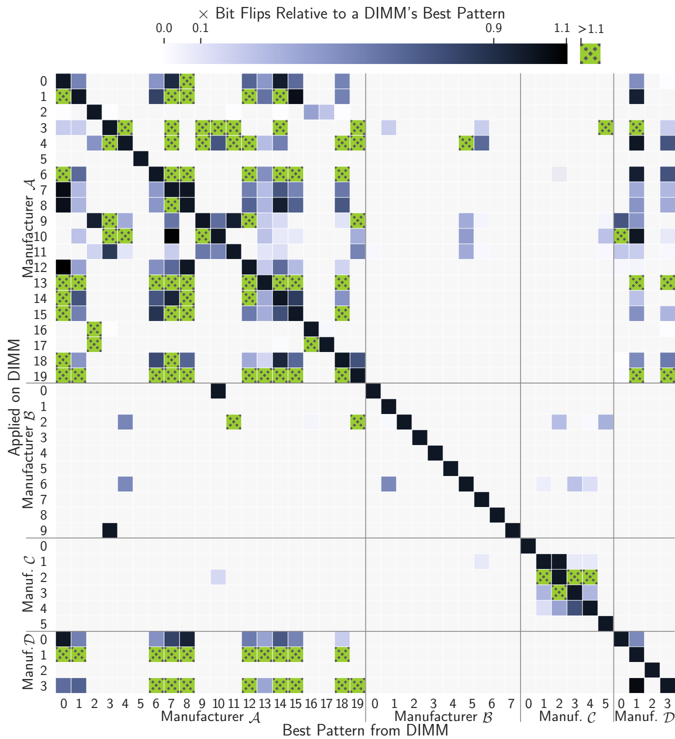


Fig. 21: **Portability results.** We run each DIMM's best pattern (x-axis) on every other DIMM (y-axis) and report the factor of more observed bit flips compared to the DIMM's best pattern (e.g., 3× for 3 times more bit flips).

can drastically reduce the attack execution time as templating is the most time-consuming step.

We aggregated the best patterns from all DIMMs and performed a sweep with each of these patterns on each module over 8 MB of contiguous memory. We report the results of this experiment in a heatmap in Figure 21. The plot shows that the effective patterns from $\mathcal{A}$ are portable: for 17 of 20 DIMMs we could even find a better pattern by taking an effective pattern that we found previously on another DIMM. Given that Blacksmith is performing a randomized search, likely some executions do not necessarily find the best possible access patterns. This explains why patterns discovered on certain DIMMs trigger more bit flips than on others. We observed that effective patterns from vendor $\mathcal{A}$ are more efficient on $\mathcal{D}_{1,3}$ than the best one we found on these DIMMs. Based on that, we believe that these DIMMs, for which we cannot tell the DRAM chip vendor, have chips from vendor $\mathcal{A}$. The DIMMs from vendor $\mathcal{B}$, $\mathcal{C}$ generally show a low portability. This could be because mitigations use DIMM-specific properties that are not adjusted when porting a pattern from another DIMM.

Appendix H
Bit Flip Reproducibility

We investigated the reproducibility of bit flips for exploitation. For this, we considered the best pattern of each DIMM and tried to retrigger bit flips ten times while measuring the number of trials needed and the elapsed time. To limit the total time of our experiment, we limited the maximum number of trials in each round to 1000, i.e., in total 10×1000 trials. As target DRAM location in the experiment, we use the location where the best pattern triggered bit flips during fuzzing.

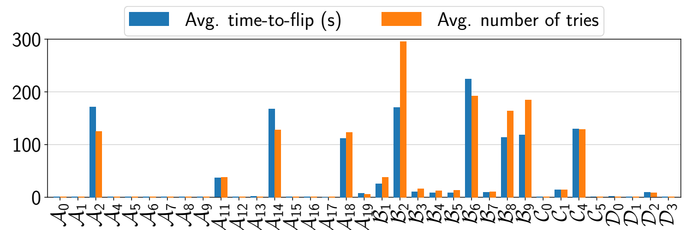


Fig. 22: **Bit flip reproducibility.** The average time-to-flip (in seconds) and the average number of hammering repetitions needed (limit: 1000) to retrigger bit flips with a DIMM's best pattern. We omit DIMMs where we could not retrigger bit flips successfully ($\mathcal{A}_{3,10}$, $\mathcal{B}_0$, $\mathcal{C}_{2,3}$).

While validating our experiment, we observed that the starting time of hammering plays a crucial role. In some cases, our data suggested that our pattern has only been effective in bypassing TRR because we started executing it at the right `REFRESH`. Hence, we improve the chance to reproduce a bit flip by waiting between 0 ms–1 ms in between retries. We argue that this is negligible as it only adds at most 1 s (for 1000 repetitions) to the total time needed to retrigger a bit flip. We do not try to optimize for the optimal `REFRESH` where we start hammering during fuzzing as there exist so many possibilities. We think it is more efficient to try out more different patterns considering the limited fuzzing time.

Figure 22 presents the results of our measurements. We can see that for DIMMs of $\mathcal{A}$ a very small number of repetitions $(1-2)$ are needed to retrigger a bit flip successfully. Other DIMMs (e.g., $\mathcal{A}_{2,11}$, $\mathcal{B}_{4-7}$), in particular those of vendor $\mathcal{B}$, require much more repetitions (e.g., up to 198 for $\mathcal{B}_2$) until we succeed. However, there are 5 out of 40 DIMMs ($\mathcal{A}_{3,10}$, $\mathcal{B}_0$, $\mathcal{C}_{2,3}$) where we could not retrigger any bit flips over all repetitions. On $\mathcal{B}_0$ we succeeded by increasing the number of hammering repetitions to 10 (i.e., we hammer longer). We think that the reason for non-reproducibility on these four DIMMs is that they require special conditions to retrigger bit flips (e.g., proper `REFRESH` alignment), which are hard to reproduce.

For the DIMMs where we could retrigger bit flips, their reproducibility allows practical exploitation. Assuming a bit flip in an exploitable page offset, since retriggering of the bit flip happens after the memory massaging step in all presented attacks, and given that the retriggering is on 90% of our DIMMs successful, the only impact to the end-to-end attack time is an increase by the average time-to-flip as shown in Figure 22.

https://t.me/learningnets