Posthammer: Pervasive Browser-based Rowhammer Attacks with Postponed Refresh Commands

Finn de Ridder
ETH Zurich

Patrick Jattke
ETH Zurich

Kaveh Razavi
ETH Zurich

Abstract

Rowhammer attacks are pervasive in client systems when launched natively. The biggest Rowhammer threat for such systems, however, lies in the browser. Our large-scale evaluation of browser-based Rowhammer attacks shows that they can only trigger bit flips on a small fraction of DRAM devices. Postponing refresh commands that trigger in-DRAM mitigations can boost the performance of Rowhammer attacks, but it has never been demonstrated in practice.

We introduce POSTHAMMER, a new Rowhammer attack in JavaScript that forces the CPU's memory controller to postpone refresh commands by creating long durations of intense Rowhammer activity followed by sufficiently long delay windows to allow the memory controller to batch refresh commands. POSTHAMMER features a new abstraction called *lane*, which enables a subset of addresses in a Rowhammer pattern to be accessed more often. Lanes enable POSTHAMMER to support effective *refresh-postponed non-uniform patterns* in the browser for the first time. Our evaluation shows that POSTHAMMER is 2.8 × more effective than the state of the art, triggering bit flips on 86 % of our 28 DDR4 test devices.

1 Introduction

Despite deployed in-DRAM mitigations, recent work shows it is possible to trigger Rowhammer bit flips on all DDR4 devices in a PC [12]. Arguably, the most interesting Rowhammer threat model for such systems is a browser-based attacker. While there has been some work on JavaScript-based Rowhammer attacks, their evaluations have only considered a few DDR4 devices [4, 10, 18]. Are browser-based Rowhammer attacks as pervasive as when launched natively?

Our evaluation using 28 DDR4 devices shows that state-of-the-art attacks can only trigger bit flips on up to 29 % of these devices. To enable pervasive browser-based Rowhammer attacks, we introduce POSTHAMMER that relies on *refresh-postponed non-uniform patterns*. POSTHAMMER generates two particular patterns that trigger bit flips on 86 % of our test devices, enabling Rowhammer exploitation in the wild.

Refresh postponement. Recent DDR standards allow the memory controller to pull-in or postpone refresh commands to improve performance [15, 16]. While it has been suggested that irregular refresh commands might weaken in-DRAM mitigations [17], such attacks have not been demonstrated in practice. The refresh commands are sent periodically by the memory controller, and it is unclear whether an attacker can sufficiently control *when* these refresh commands are postponed. Through a series of experiments, we show that an attacker can indirectly control the postponing of refresh commands by triggering intense periods of memory activity and forcing the memory controller to send postponed refresh in batches at desired times by inserting sufficiently long delays in the access patterns. We use this technique to build POSTHAMMER, which generates *refresh-postponed many-sided patterns*. While this version of POSTHAMMER triggers bit flips on 43 % of the devices, the question is whether the recent native non-uniform patterns [12] could further enhance POSTHAMMER.

Eviction-based non-uniform patterns. A non-uniform pattern hammers certain aggressors less often than others to evade in-DRAM mitigations better. While a non-uniform pattern is straightforward to implement natively, the same does not apply to the browser environment. As the browser-based attacker relies on eviction sets, simply accessing a particular aggressor twice will nudge the cache replacement policy into keeping that aggressor always in the cache, breaking the eviction chain constructed by the attacker. To address this challenge, POSTHAMMER relies on a new abstraction, which we call *lane*. A lane is a minimal eviction set made out of part of a given Rowhammer access pattern. By combining different lanes, some sharing certain aggressor addresses, POSTHAMMER accesses certain aggressors less often while performing evictions correctly. Hence, lanes enable POSTHAMMER to achieve *refresh-postponed non-uniform patterns* with cache evictions for the first time. Such patterns enable POSTHAMMER to trigger bit flips on 86 % of our devices.

Practical JavaScript attacks. While POSTHAMMER is capable of finding effective patterns on many devices, finding these

patterns can take a significantly long time, making Rowhammer attacks impractical. Ideally, the attacker should only need a small set of patterns that work well on a large number of DIMMs. By visualizing the per-DIMM best patterns, we can distill two types of generic patterns with high performance across most DIMMs. These two types of patterns are all the attacker needs to make their browser-based Rowhammer exploit succeed 61 % of the time—without assuming any particular target DIMM and therefore greatly amplifying the impact of browser-based Rowhammer attacks.

Contributions. We make the following contributions:

- We present POSTHAMMER, the first demonstration of a Rowhammer attack that leverages refresh postponement to weaken in-DRAM mitigations.
- We describe a novel approach for generating eviction-based non-uniform patterns using the *lane* abstraction. POSTHAMMER leverages this approach to generate *refresh-postponed non-uniform patterns* that enable pervasive Rowhammer attacks in the browser.
- We reveal two types of patterns that work on most DIMMs and use them to obtain an arbitrary read-write primitive in Firefox.

We will provide more information about POSTHAMMER at https://comsec.ethz.ch/posthammer.

2 Background

We discuss DRAM (Section 2.1), Rowhammer (Section 2.2), and refresh synchronization in recent attacks (Section 2.3).

2.1 DRAM

The *Dynamic Random Access Memory* (DRAM) is the most common type of volatile memory in use today and can be found in all modern computers, smartphones, and other electronic devices. DRAM is organized into *banks*, which are further divided into *rows* and *columns*. Each row contains a number of *cells*, each of which stores a single bit of data. To access a cell, the row it belongs to must be *activated*, which moves the row into the bank's *row buffer*. Once in the row buffer, the cell can be read or written. Importantly, only one row can be in the row buffer at a time and if a row is already in the buffer, it will not be activated again.

Refresh command. Because of its volatile nature, each DRAM cell needs to be refreshed periodically. This process, in which the memory enters a so-called *refresh cycle*, is described by the DDR standard (e.g., DDR4 [15]) and varies across generations [14, 16]. In short, the standard dictates that every *tREFI* (7.8 µs in DDR4), the memory controller ought to send a refresh command over the memory bus. Upon reception, the device initiates a refresh cycle. During such a cycle several rows are refreshed in parallel across banks. Which row or rows is decided by logic internal to the DRAM chip.

Postponing refreshes. The standard also specifies that refresh commands may be *postponed* or *brought forward (pulled-in)* for "improved efficiency in scheduling and switching between tasks" [15]. For example, by postponing a single refresh command, the timeline becomes as follows: (i) a REF, (ii) 15.6 µs without a REF, and (iii) two REFs (one of which was postponed). Up to eight refresh commands may be postponed or pulled-in, which would create a tREFI of nine times 7.8 µs.

2.2 Rowhammer

Whenever a row is accessed in rapid succession, its charge may start interfering with that of neighboring rows. In some cases, this interference is so severe that it causes bits to flip in one of these neighbors, i.e., a zero becomes one or vice versa. This effect is termed Rowhammer [20] with the row that is accessed (or *activated*) often known as the aggressor row, and the neighbors as the victim rows. In general, the more activations made to the aggressor row per time unit, the stronger the Rowhammer effect [19].

Cache line invalidation. To trigger and then exploit Rowhammer, the attacker tries to activate a set of rows as often as possible. To this end, the attacker cannot simply access the same address repeatedly: the data would be stored in the CPU caches immediately, preventing the access from going to DRAM. To solve this problem, native attacks such as [18, 21, 26, 28, 30] have relied on cache line flushing instructions, non-temporal loads/stores [26], DMA [22, 31] or uncacheable memory [33, 34].

Double-sided Rowhammer. In addition to bypassing the caches, the attacker needs to make sure that the access pattern produces a series of activations to the hammered rows, which will not happen as long as the row resides in the bank's row buffer. For this reason, to avoid a row buffer *hit*, the attacker instead alternates between two rows because the row buffer can only hold a single row. Accordingly, accessing two rows in turns means they will be continuously replacing each other in the row buffer and thus activated on every access.

To maximize the disturbance in a single *victim* row, the attacker chooses two *aggressor* rows such that they neighbor the victim row. This *access pattern* is called double-sided Rowhammer and is by far the most commonly used by attacks [2, 18, 21, 25, 28, 30, 32, 37].

Mitigations. To mitigate Rowhammer attacks, since DDR4 DRAM contains additional circuitry that aim to *detect* and then proactively *refresh* the victim row under attack. This type of mitigation has been referred to as Target Row Refresh (TRR). Due to its non-volatile nature, DRAM requires each row to be refreshed periodically. With TRR, however, additional refreshes to suspected victim rows makes sure to minimize the sensitivity of the victim row to the interference of neighboring aggressor rows. Unfortunately, TRR has been shown ineffective [7, 12]: in short, an attacker can by-

pass TRR by using a more complex access pattern, i.e., compared to double-sided Rowhammer. For example, for some DIMMs, cycling through 2-10 (instead of one) double-sided pairs suffices [4, 7] while in other cases more complex access patterns are necessary [11, 12]. Blacksmith [12] has shown that all TRR implementations can be eluded by employing *non-uniform* patterns that unlike double-sided patterns access some aggressors more frequently than others.

2.3 Refresh synchronization

Effective Rowhammer patterns need to synchronize with refresh commands (REFs), sent to DRAM every 7.8 µs [4, 12, 13]. For example, in DDR4, around 160 activations may occur within one tREFI [15]. A Rowhammer pattern that repeats every 160 activations would therefore be synchronized. More generally, a pattern is synchronized if it perfectly fits inside a tREFI. Patterns shorter than a tREFI may be synchronized by repeating them until they fit. It is not necessary to spend all 7.8 µs on activations, however: the attacker may insert periods of no memory activity to stretch a pattern of, say, 40 activations, until its length is 7.8 µs. Such periods have been filled with NOPs [4].

Effectiveness. TRR implementations rely on the periodicity of the refresh command to obtain a reliable sample of the rows that are being activated [7, 11]. With synchronization, however, sampling becomes ineffective, either because (i) the sampler only observes some and thus always the same activations within a tREFI—missing others—or because (ii) the sampler does not sample at every tREFI, but for example, every second tREFI, making it possible to consistently miss certain activations. In case of the former, an attacker may bypass the mitigation by activating decoy rows when the mitigation is sampling within a tREFI, and in the case of the latter, by only activating decoy rows during the tREFI in which the sampler is active.

Refresh scheduling. This supposition, of why synchronization helps the attacker, is supported by a JEDEC publication on Rowhammer mitigations that states [17]:

> Elimination of pulled-in and postponed Refresh commands allows the DRAM the chance to consistently align internal address sampling and perform Rowhammer mitigation at consistent intervals. Additionally, some randomization when issuing of Refresh commands within the interval can also assist Rowhammer mitigation.

This not only confirms that Rowhammer mitigations rely on the *consistency* of the refresh command for reliable samples, but also that pulled-in and postponed refreshes weaken the mitigation further. How the attacker can use refresh postponement to their advantage and to what extent this can weaken the mitigation remains unclear, however.

3 Threat Model

We assume a threat model where the victim runs untrusted JavaScript code in their browser, either by visiting a website under the control of an attacker, or through a malicious advertisement. Moreover, and contrary to previous work [4, 18], we do *not* assume a particular DDR4 DIMM, which would have greatly simplified—and reduced the severity—of the attack, as some DIMMs are much more vulnerable than others. The aim of the attacker is to achieve an arbitrary read/write primitive in the browser.

4 Motivation and Challenges

We aim to demonstrate the *widespread exploitability* of Rowhammer on DDR4 devices. Without native code execution and not knowing anything further about the DIMM, the attacker is forced to build self-evicting Rowhammer patterns that work on any DIMM. These self-evicting patterns have been used in two JavaScript-based Rowhammer attacks: SMASH [4] and SledgeHammer [18].

SMASH. These patterns consist of double-sided aggressor pairs that evict each other from the caches, which makes them *self-evicting*. Given a typical LLC associativity of 16, this would mean each pattern would have to consist of at least 17 aggressor pairs which for some DIMMs is too many to bypass the mitigation [7]. To overcome this limitation, SMASH patterns pin some accesses to the caches, effectively reducing the cache's associativity. This enables patterns of fewer than 17 pairs at the price of a few cache hits only, see further [4].

SledgeHammer. Cached accesses in SMASH might as well be replaced by aggressors that target different banks in DRAM. By having the aggressors share their row but not their bank, a SledgeHammer pattern takes advantage of bank-level parallelism. This means that like SMASH's cache hits, accessing the additional aggressors is fast, while unlike SMASH's hits, SledgeHammer's extra aggressors may improve the pattern's chances of triggering a bit flip [18].

As a first step, we determine the general applicability of both SMASH and SledgeHammer. We construct SMASH and SledgeHammer patterns of different numbers of aggressors and use them to hammer each of the 28 DIMMs listed in Section C. We continuously randomize the aggressors—and therefore the target or victim rows—as well as their number.

The results in Table 1 show that (i) SMASH patterns cover 4× more DIMMs than SledgeHammer patterns, but they still manage (ii) to compromise only 29 % of DIMMs. Moreover, there is no overlap between the DIMMs vulnerable to SMASH and SledgeHammer.[1] Given these numbers, we aim to increase the coverage of browser-based Rowhammer attacks to increase their impact.

[1] More details can be found in Section 8.

Table 1: Coverage of the state of the art. The number of DIMMs for which SMASH and SledgeHammer managed to trigger at least one flip.

Pattern type	Effectiveness (#DIMMs)
SMASH [4]	8/28 (29 %)
SledgeHammer [18]	2/28 (7 %)

4.1 Challenges

To make self-evicting patterns more effective, we investigate if it is possible to weaken the mitigation by abusing the flexibility given to the memory controller for scheduling refresh commands. As explained in Section 2.1, the DDRx standard allows refresh commands to be either postponed or pulled-in to improve performance. Until now, however, it has not been clear if it affects the effectiveness of in-DRAM TRR mitigations and how attackers can use it to their advantage [1].

Challenge 1. Take advantage of refresh scheduling to weaken in-DRAM mitigations.

Previous work shows that in-DRAM mitigations act at *time-of-refresh* [7, 11]. This makes it is worthwhile to investigate (i) whether the attacker is able to manipulate the scheduling of refresh commands and (ii) what consequences this has for in-DRAM mitigations. As we will show, by producing intense memory activity in short durations, the attacker is indeed able to make the memory controller postpone refresh commands. Moreover, while for some DIMMs refresh postponement "only" increases a pattern's effectiveness, for others it is a *necessity*, allowing us to trigger bit flips without *clflush* for the first time. Our implementation of this technique in a new Rowhammer attack, called POSTHAMMER, improves our coverage from 29 % to 43 % (or 12/28) and is therefore a good first step. However, before we may claim widespread exploitability, we need to improve it further.

Previous work [12] shows that for some DIMMs, a *non-uniform* pattern is essential in order to bypass the mitigation. However, while it is relatively straightforward to construct a non-uniform *clflush*-based pattern—simply by accessing some aggressors more frequently than others—the same does not apply to self-evicting patterns. More than for *clflush*-based patterns, the *access order* is crucial for patterns that rely on eviction sets [4]. For this reason, self-evicting patterns make use of pointer chasing [4, 18], which prevents the CPU from reordering the pattern. As we will show, however, it is challenging to construct a *non-uniform* pointer chase.

Challenge 2. Craft patterns that are both self-evicting and *non-uniform*.

We generalize self-evicting patterns by introducing the notion of a *lane*: a subset of a pattern's addresses that form a *minimal* eviction set. Using lanes, POSTHAMMER is able to add a variable amount of non-uniformity to self-evicting patterns without breaking the rather fragile eviction dynamics.

By adding non-uniformity, POSTHAMMER can find a pattern for 86 % (or 23/28) of the DIMMs in our testbed. While all of these patterns are *clflush-free*, they appear rather heterogeneous. This ultimately creates another challenge for the attacker.

Challenge 3. Find a tractable set of patterns that works well on a large number of DIMMs.

Since we assume the attacker does not know the victim's DIMM, POSTHAMMER must try a subset of patterns that has high coverage but is small enough to be tested during the online part of the attack. Through manual analysis, we find that this set consists of only *two types* of patterns—the *SB*- and *LW*-patterns. *SB*-patterns consist of a single eviction set and between 12 and 36 aggressors. *LW*-patterns are more complex and always require exactly two eviction sets. Each of the two types comes with a small but manageable number of parameters, as we will further explain in Section 7. We show the effectiveness of these patterns by using POSTHAMMER to build an end-to-end exploit in Firefox, achieving an arbitrary read-write primitive in its renderer process as discussed in Section 9.

5 Exploiting Refresh Scheduling

We show how the attacker may take advantage of refresh scheduling to weaken TRR. Refresh scheduling, as briefly explained in Section 2.1, improves performance by providing the memory controller with some flexibility w.r.t. scheduling refresh commands. For example, postponing refresh commands can avoid interrupting a dense series of reads and writes. Our underlying idea behind the *exploitation* of refresh scheduling is the following: *if* TRR implementations rely on the periodicity of refresh commands for sampling, as suggested by previous work [7, 11] and JEDEC [17], *then* by postponing refresh commands sampling becomes less reliable. As a consequence, the mitigation is weakened. The experiments in this section indeed show that the attacker is able to induce the postponing of refresh commands (Section 5.1) and that for most DIMMs, but in particular those by Samsung, refresh-postponing patterns are much more effective (Section 5.2).

5.1 Inducing refresh postponement

The first step towards exploiting refresh postponement is to show that the attacker is able to induce it. To this end, we need (i) a way to trigger it and (ii) a way to measure it.

Because flexible refresh scheduling is a performance optimization, we conjecture that the memory controller will try

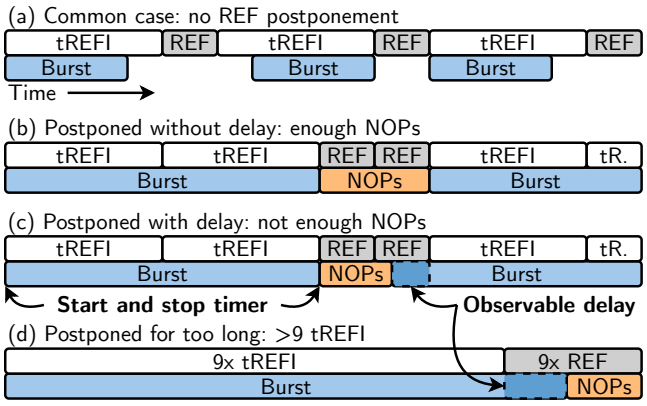


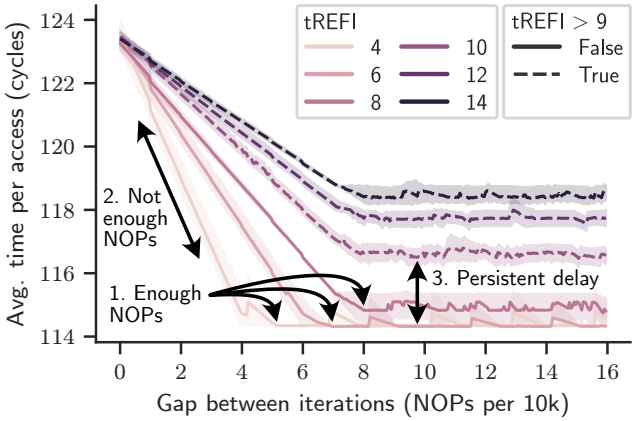
Figure 1: Experiment for refresh postponement detection. **(a)** The common case: DRAM receives a burst of memory requests, but there is plenty of idle time for the memory controller to regularly schedule a REF. **(b)** A longer burst of memory accesses that pushes one REF to right after the burst. The memory controller does not need to interrupt the second burst as it is given sufficient time for both REFs. **(c)** In this case, not enough time (i.e. NOPs) is given for the REFs. The observer will measure a delay of the time it takes to complete the burst. **(d)** Finally, a burst of more than nine tREFI, which would cause more than eight REFs to be postponed and is therefore always interrupted.

to avoid delaying read and write commands—until it has *not* sent a refresh command for nine tREFI and by the specification is forced to send all eight refreshes at once. Whenever this happens, we expect the memory accesses right after the batch of refreshes to experience an observable delay. This is a reasonable expectation, as previous work suggests that it is possible to even detect the delay of a single refresh [7].

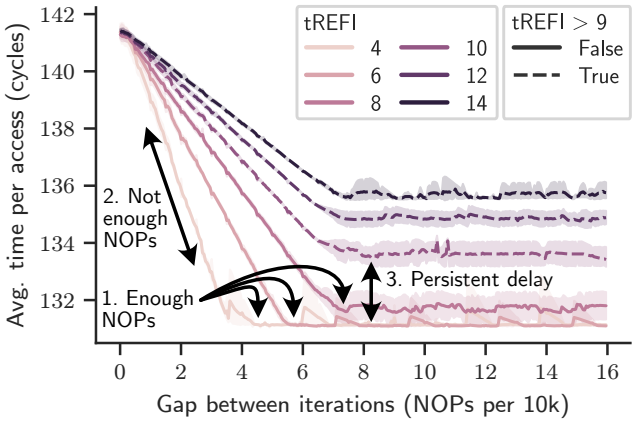
In other words, to observe refresh postponement, we need to create a series of memory accesses that takes *more than* nine tREFI. It is important that this series is *dense* to hinder the memory controller from scheduling the refresh commands it has postponed so far. In that case, we will not be able to observe a delay. Further, to later *exploit* refresh postponement, we should avoid relying on *clflush* to create this dense series of memory accesses, because ultimately, we would like to use it to simplify *clflush*-free Rowhammer patterns.

Experiment. To observe whether we can trigger refresh postponement, we perform the experiment illustrated in Figure 1. For bursts of different lengths, we measure the time it takes until all memory requests have been served. Each burst is a self-evicting (double) pointer chase (similar to [4] where each aggressor evicts the next), and therefore, free of *clflush*. After each burst we execute a series of NOPs, giving the memory controller time to schedule refresh commands.

The experiment is designed to distinguish between three cases, depending on the burst length and size of the NOP gap:



(a) Coffee Lake



(b) Kaby Lake

Figure 2: Results of the experiment in Figure 1. We can clearly distinguish between the three cases explained in the text. We use dashed lines for configurations with more than 9 tREFI and solid lines for all others.

1. **Gap large enough.** The burst is longer than a single tREFI but smaller than nine. The NOP gap is large enough to accommodate all postponed refreshes. See Figure 1-(b).

2. **Gap too small.** As above, except that this time, the NOP gap is too small to squeeze in the postponed refreshes. A delay is expected. See Figure 1-(c).

3. **Gap too late.** The burst is longer than nine tREFI. In this case, we expect a *persistent* delay as the memory controller is forced to interrupt the burst every time, no matter the size of the NOP gap. See Figure 1-(d).

We conduct all our experiments on Intel Core i7-8700K (Coffee Lake) and Intel Core i7-7700K (Kaby Lake) machines. For more details, see Section 8.

Results. The results are shown in Figure 2. We are able to

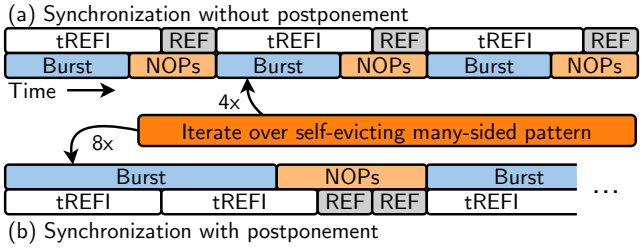


Figure 3: Experiment to measure the effectiveness of refresh postponement. Above **(a)**, synchronization without postponement, and below **(b)**, synchronization with postponement by iterating over the eviction set more often (4× versus 8× in the example).

clearly distinguish between all three cases on both Coffee Lake (Figure 2a) and Kaby Lake (Figure 2b). The annotations inside the figure show exactly which case is responsible for the observed (average) latency on the vertical axis. For example, for a *clflush*-free burst of six tREFI in length in Figure 2a, we find the average latency per access decreasing as more NOPs are added in between successive bursts until the number of NOPs reaches 60k. From then on, the average latency stabilizes. Our explanation is that the memory controller requires the equivalent of 60k NOPs in time to schedule the five refreshes postponed by the burst of six tREFI.

The plots in Figure 2 show a clear difference between bursts of up to and including nine tREFI (solid lines) compared to those of more than nine (dashed lines). We expect the latter to introduce a persistent delay, as the memory controller will always interrupt them. As our data shows, until around 80k NOPs, the average latency for these longer bursts decreases, until it stabilizes well above the equilibrium of the shorter bursts, i.e., less than nine tREFI. We conclude from that:

Observation 1. The attacker is able to induce refresh postponement by generating bursts of memory accesses followed by NOPs.

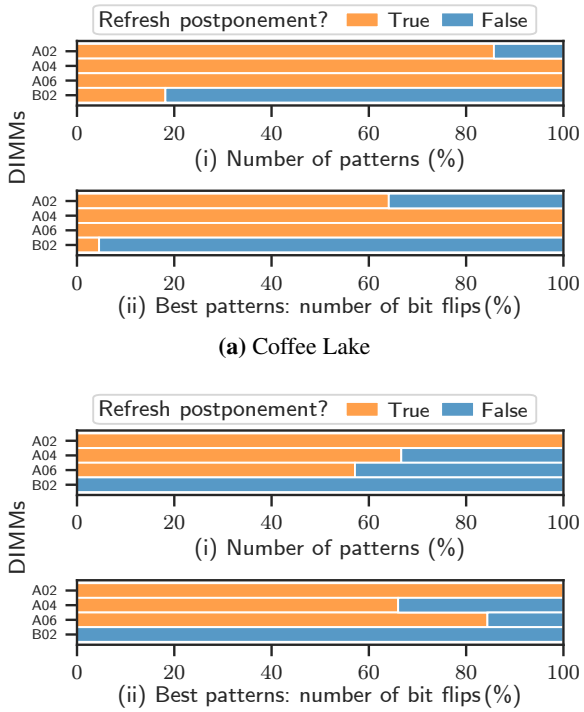
5.2 The effectiveness of refresh postponement

We implement POSTHAMMER to evaluate the effectiveness of refresh postponement by modifying self-evicting patterns [4] to either induce or not induce refresh postponement. To make a pattern refresh postponing, we synchronize it over *more than one* tREFI, as opposed to a single tREFI.

Experiment. To evaluate the potential of refresh postponement for weakening TRR, we perform the experiment illustrated in Figure 3. POSTHAMMER creates self-evicting patterns of different lengths where half of the patterns is made to induce refresh postponement, while the other half is made to avoid it. In other words, we either

(i) measure the time it takes to iterate through the eviction set (and thus hammer) once, calculate how many iterations would fit in a tREFI, and fill up the remaining time with NOPs. This gives us a pattern of a single tREFI, see Figure 3-(a); or we

(ii) choose a random number x between 1 and 15, calculate how many iterations would fit into x tREFI, and fill up the remaining time with NOPs. See Figure 3-(b), where $x = 2$.



(a) Coffee Lake

(b) Kaby Lake

Figure 4: Results of the experiment in Figure 3. Refresh postponement has the potential to make Rowhammer patterns much more effective.

We run this experiment on four different DIMMs (A02, A04, A06, and B02; see Section C for details), which we found to be vulnerable to TRRespass [12]. This experiment allows us to only add a single parameter—whether the pattern uses refresh postponement or not. The self-eviction part of the pattern, however, depends on a few parameters: (i) the number of double-sided aggressor pairs and (ii) the ratio between cache misses and hits. We need these parameters to abstract away the particularities of different DIMMs (e.g., different sampler sizes). For this reason, we try random combinations of all parameters and either make the pattern induce refresh postponement or not (Figure 3). Patterns of the first type will be referred to as in the *postponing* group while patterns that do not induce refresh postponement are *non-postponing*.

Results. The results for Coffee Lake are shown in Figure 4 and answer the following questions:

(a) Patterns: Of all patterns that triggered a bit flip, how many used refresh postponement and how many did not?

(b) Strength: How many bit flips does the best postponing pattern trigger compared to the best non-postponing pattern? This says something about the effectiveness of refresh postponement.

Best pattern. The best pattern is selected as follows: for each pattern that triggers a bit flip, we hammer the pattern again but with a different eviction set. This way, we avoid falsely concluding that a pattern is strong or weak simply because its victim rows—which are implicitly determined by its aggressors and therefore the eviction set—happen to be rather vulnerable or, conversely, resistant to Rowhammer. We repeat this 100 times and make sure to never reuse the same eviction set.[2] Thereafter, we count the total number of bit flips triggered during these 100 attempts and use it to rank the patterns. We create one ranking per parameter value, i.e., we rank the postponing and non-postponing groups individually.

This gives us two values: the number of bit flips triggered by the best postponing and non-postponing patterns. We set their sum to 100% and compute and plot their individual contributions. For example, if these best patterns triggered exactly the same number of bit flips, the figure would show 50% for both true and false.

For Coffee Lake, Figure 4-(a.i) shows that for all DIMMs, except B02, the majority of flip-inducing patterns used refresh postponement. This is a strong indication that—at least for DIMMs of manufacturer A (Samsung)—refresh postponement helps to bypass the mitigation as we equally tested patterns with and without refresh postponement. Moreover, in the evaluation in Section 8, we will show that refresh postponement also helps the attacker for other manufacturers. Figure 4-(a.ii) confirms the results in Figure 4-(a.i): for all DIMMs except B02, the best refresh postponing pattern triggers more bit flips than the best non-postponing pattern. The results for Kaby Lake Figure 4-(b) are very similar: with refresh postponement showing benefits in finding effective patterns on the same DIMMs where we saw benefits on Coffee Lake. We conclude with:

Observation 2. For some DIMMs, refresh postponement helps to bypass the mitigation, and we find more and stronger patterns that use refresh postponement compared to patterns that do not.

Armed with refresh postponement, next we discuss how we add support for non-uniform self-evicting patterns to POSTHAMMER.

[2]Moreover, we build eviction sets such that they never overlap, i.e., map to the same row.

6 Self-evicting Non-uniform Patterns

Previous work [12] has shown that for some DIMMs, a non-uniform pattern is necessary to bypass TRR. In a non-uniform pattern, the aggressors are not hammered uniformly, i.e., some are activated more frequently than others. This way, the attacker may fool mitigations that rely on counting per-row activations and *only* refresh the neighbors of the most frequently activated aggressor(s). As a consequence, less frequently activated aggressors may be unnoticed by the mitigation. When activated sufficiently often, such aggressors will be able to trigger bit flips. In this section, we tackle the problem of constructing self-evicting patterns that are also non-uniform.

6.1 Introducing lanes

We add non-uniformity to self-evicting patterns using *lanes*.

Definition. A *lane* is a minimal eviction set of aggressors.

This means the number of aggressors in a lane equals the associativity of the LLC. Lanes arise from the observation that all self-evicting patterns consist of a series of lanes that continuously *replace* each other in the caches.

Example. Consider two lanes of aggressors, L_A and L_B. As each lane forms a minimal eviction set, we have

$$L_A \xrightarrow{\text{maps to}} c_A \tag{1}$$

which denotes that all aggressors in L_A map to some cache set c_A in the LLC. Similarly, $L_B \to c_B$. To build a pattern, we thus need $c_A = c_B$, or equivalently,

$$L_A \to c \;\text{ and }\; L_B \to c \qquad \text{for a cache set } c. \tag{2}$$

This means, not only should the addresses in each lane map to the same set, but addresses from *all* lanes should map to the same cache set and slice. Without this condition, they will not be able to replace each other to cause eviction. With Equation (2), however, we get *replacement by alternation:*

Replacement by alternation. By first accessing all addresses in L_A, followed by all addresses in L_B, then L_A again, etc. we will continuously replace addresses in c. For example, if sets in the LLC have 16 ways—which means both L_A and L_B contain 16 addresses as they are minimal eviction sets—accessing L_A will cause 16 cache misses, followed by 16 misses for L_B, followed again by 16 misses for L_A, and so on. This means alternating between lanes, as in

$$\cdots L_A, L_B, L_A, L_B \cdots \tag{3}$$

gives us complete replacement of c. The problem with replacement by alternation is that naively, all aggressors will be accessed equally. In other words, $L_A, L_B, L_A, L_B \cdots$ produces a uniform pattern in which the aggressors of L_A are accessed as often as those of L_B.

Non-uniformity by introducing extra lanes. To solve this problem, we introduce (at least) one more lane L_C that like L_A and L_B maps to the same cache set c. This allows us to create non-uniform sequences such as:

$$\cdots L_A, L_B, L_A, L_C \cdots \tag{4}$$

where the aggressors in L_A are accessed twice as often as those in L_B and L_C. Please note that introducing a third (or fourth, fifth, etc.) lane is strictly necessary. It is not possible to create non-uniformity using only two lanes: either the lanes alternate and the pattern is uniform, see again Equation (3), or we repeat a lane (e.g., L_A, L_B, L_B), but then we lose eviction by replacement as the second time we access L_B, the lane is already cached. While using extra lanes gives us non-uniformity, we may not always want to have the entire lane of aggressors go to DRAM. As an example, assuming an LLC with associativity of 16, iterating over a three-lane pattern once involves activating three times 16 aggressors for a total of 48 aggressor pairs which is too many for most DIMMs. Instead, we want the number of aggressors to be a parameter that is (mostly) independent of the pattern's non-uniformity. The solution is to introduce cache hits, as also done in previous work [4], and explained next.

6.2 Pattern construction

We will now explain the details of constructing a non-uniform pattern using lanes. First, because we need several lanes—at least three—that all map to the same cache set and slice, we start from a large eviction set E in which all addresses map to set c. For example, assuming an associativity (and thus lane size) of 16 and three lanes, the size of this large set would be $3 \cdot 16 = 48$. We then split the set E into three groups L_A, L_B, and L_C that form our lanes: $E = \{L_A, L_B, L_C\}$.

Double-sided aggressor pairs. Our goal is to create a pattern composed of double-sided aggressor pairs, inspired by SMASH [4]. To this end, we *fork* the eviction set E above—creating another one, $E*$—as follows: for each address, in every lane, we add or subtract two to its row address (to get the *double side*). On Kaby and Coffee Lake microarchitectures, this means we will change the address' bank as well. That is a problem: double-sided aggressor pairs are one row apart *but also map to the same bank.* To solve it, we toggle another (non-overlapping) bank bit and thereby restore the address' bank [4]. As a consequence, however, the address' cache set index changes from c to $c*$. Fortunately, this applies to all addresses in our forked eviction set $E*$ equally, i.e., all addresses will now map to $c*$, and thus, our forked set is an *eviction set* as well.

Eviction blocks. At this point, we have two eviction sets of 48 addresses each, E and $E*$, of which the cross-set pairs form double-sided aggressor pairs. Moreover, being eviction sets, all addresses in E map to some cache set c while all addresses in $E*$ map to another set $c*$. Together, E and $E*$ form a unit, the building block of every self-evicting pattern. We will refer to such a pair of eviction sets as an *eviction block*. Figure 5 illustrates the concept.

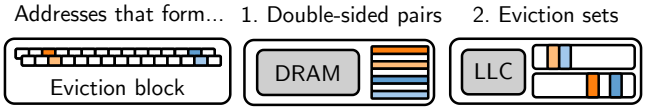


Figure 5: An eviction block is made of two sets of addresses (left), of which the cross-set pairs form double-sided (aggressor) pairs in DRAM (middle), and where the addresses in each set map to the same cache set (right).

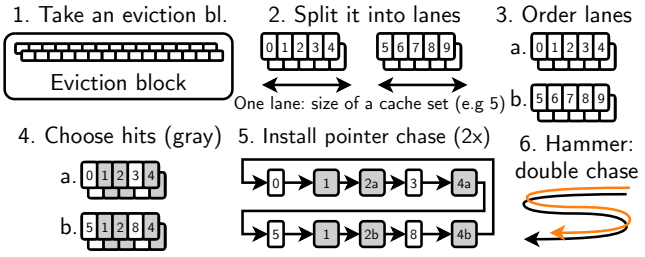


Figure 6: From eviction block to double pointer chase. We start with an eviction block, as in Figure 5. Second, we split it into so-called lanes whose size equals the associativity of the LLC. Third, we order the lanes. Fourth, we decide on the number of recurring addresses (or cache hits, in gray) and their lane-relative positions. Fifth, for each set of congruent addresses, we connect lanes (a) and (b) through a pointer chase. Sixth, we hammer by traversing the pointer chases in an alternating manner.

Conversion to pointer chase. Although all patterns consist of eviction blocks, an eviction block is not a pattern yet. To prepare an eviction block for hammering, it needs to be *converted* into a pointer chase. The pointer chase is essential: it ensures the aggressors, within each lane and between lanes, are accessed in the intended order. We found that without a pointer chase, eviction is not reliable since the expected number of cache misses caused by hammering the pattern would not match the number reported by the CPU's performance counters. To convert an eviction block into a pointer chase, we proceed as shown in Figure 6 and described in the following:

1. We start with an eviction block comprised of two large eviction sets E and $E*$.

2. We split the block into lanes, i.e., we split both eviction sets. For example, $E = \{L_A, L_B, L_C\}$ and $E* = \{L_{A*}, L_{B*}, L_{C*}\}$.

3. We decide on a lane sequence, such as

$$\cdots L_A, L_B, L_A, L_C \cdots$$

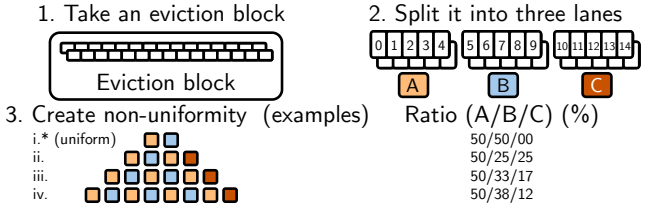


Figure 7: Creating non-uniformity by having three or more lanes. Using a third lane, we are able to create non-uniform patterns as per the ratios shown in the bottom right of the figure. For example, in variant (iii), the aggressors of lane (a) receive 50 % of the activations, those in lane (b) 33 %, while the remaining 17 % go to the aggressors in lane (c). The figure is not exhaustive: further variants with both three lanes and more than three lanes exist.

Note that the lane sequence should be the same for both E and $E*$; otherwise, the aggressors will no longer form double-sided pairs.

4. **Choose hits.** In this step, the attacker decides on the number of aggressors. Even though lanes give us replacement by alternation, *fully* replacing the cache set every time may not be desired, as it directly determines the number of aggressors that go to DRAM. For this reason, the attacker can also opt for *partially* replacing the cache set, which is achieved by creating *overlap between lanes*. This pins some addresses to the caches—they are accessed as part of every lane—and introduces cache hits for a faster pattern execution with fewer aggressors. In Figure 6, the hits are illustrated in gray, while the aggressors are white.

5. At this point, the attacker has two complete sequences of aggressors, one for each eviction set, that form double-sided aggressor pairs. Each sequence is connected through a pointer chase: to connect a to b, the attacker stores the address of b at the location of a. In pseudocode `*a = b;` where both `a` and `b` are pointers.

6. Finally, the attacker hammers the pattern by chasing both chases in alternating fashion. We found that for all DIMMs, alternating is much more effective compared to first iterating over one chase once before iterating over the other.

This algorithm allows the attacker to create patterns that are both self-evicting and non-uniform. Moreover, by varying the number of lanes and their access order, the attacker can tune the pattern's non-uniformity. Figure 7 gives some examples of what the attacker can do with three lanes.

Multi-block patterns. In addition to the multi-lane strategy, we can add non-uniformity to our patterns by using more than one eviction block while distributing the activations unequally among them, see Figure 8. This adds non-uniformity to the pattern at a much coarser granularity than before. While this is not necessarily a problem—and may even be desirable depending on the mitigation—what could make this strategy less effective is the larger number of aggressors required and, consequently, the lower per-aggressor activation rate.

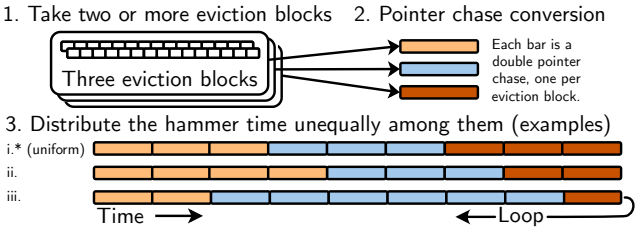


Figure 8: Non-uniformity by creating multi-block patterns. The three-block pattern in the figure gives us three pointer chases (using the steps in Figure 6). By repeating some chases more often than others, we distribute the per-aggressor activations unequally over the pattern.

The multi-lane and -block strategies for adding non-uniformity are easily combined. The former is a block property and the latter a property of the pattern as a whole. Accordingly, while exploring the non-uniform search space below, we test all possible variants: single- and multi-block patterns, and for each for their blocks, single- and multi-lane blocks.

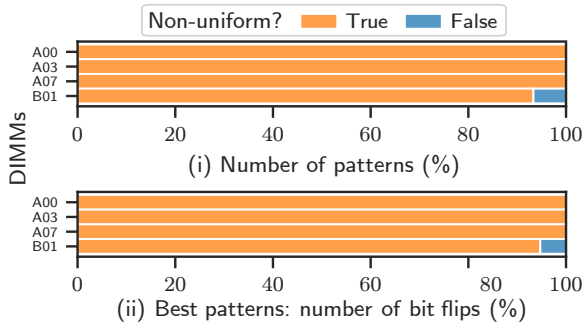
6.3 Effectiveness of non-uniformity

We evaluate the effectiveness of non-uniform patterns constructed as outlined above on four different DIMMs for which TRRespass *did not* manage to trigger a bit flip.[3] We try all possible combinations of: first, the uniform and three non-uniform chases in Figure 7-(3), and second, the multi-block patterns in Figure 8, for patterns of 1 up to and including 8 blocks.

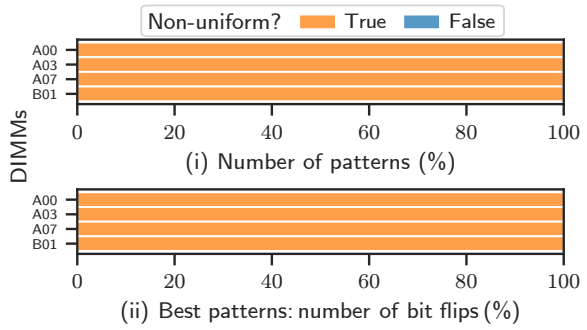
Results. The results in Figure 9 are unambiguous for both microarchitectures: non-uniform patterns are more effective. In fact, for most DIMMs, not a single uniform pattern was found. The exception is DIMM B01, for which a fraction of the effective patterns was uniform, though only on the Coffee Lake machine. This is somewhat surprising, as it means TRRespass should in theory be able to trigger bit flips on this DIMM as well. At the same time, Figure 9-(a.ii) shows that B01's best non-uniform pattern is much stronger compared to its best uniform pattern and accounts for roughly 95% of the bit flips, which makes the latter rather weak and probably not useful. We therefore conclude that at least for these DIMMs, and for both microarchitectures, eviction-based non-uniform patterns are more effective.

Conclusion. We have introduced the concept of a *lane* to

[3]The complete results, for all 28 DIMMs, can be found in Section 8.



(a) Coffee Lake



(b) Kaby Lake

Figure 9: Effectiveness of the best uniform vs. best non-uniform pattern. A pattern is non-uniform if it (i) consists of more than one eviction block or (ii) contains at least one three-lane pointer chase.

generalize self-evicting patterns and make them non-uniform. Additionally, by employing more than one *eviction block*, we can create multi-block patterns of many aggressors and further increase their non-uniformity. Finally, we have shown that non-uniform self-evicting patterns that rely on multiple lanes and/or blocks effectively bypass the more "advanced" in-DRAM mitigations that defend against many-sided but uniform patterns.

7 POSTHAMMER in JavaScript

The attacker's ultimate goal is to trigger a bit flip from inside the browser's JavaScript sandbox. This is more challenging than triggering a *clflush*-free bit flip natively. For example, inside the renderer process, there is (i) no accurate timer available, which would greatly simplify building eviction sets as well as pattern synchronization, (ii) continuous and complex just-in-time compilation, which reduces the reliability of pointer chases, and (iii) no information available on the physical memory assigned to the process, which complicates pattern construction. While these challenges have already been tackled in previous work [4, 18], this time the attacker faces two additional ones. First, building more complex

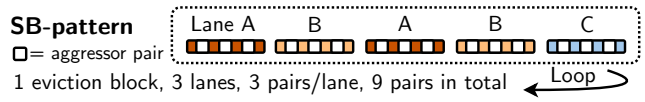


Figure 10: The SB-pattern. A single eviction block. The colors denote different lanes.

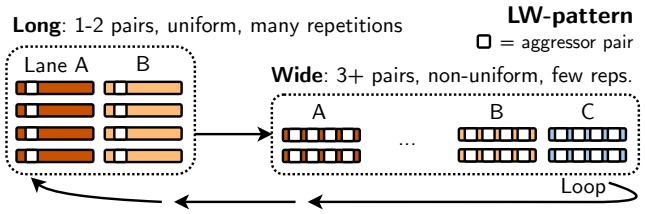


Figure 11: The long-wide (LW) pattern. Two eviction blocks, of which one is *long* (few aggressors, many repetitions), while the other is *wide* (many aggressors, few repetitions). The colors denote different lanes.

patterns—non-uniform self-evicting patterns require larger eviction blocks and synchronizing them is more involved—and second, deciding *which* patterns to build.

Our evaluation in Section 8 shows that using our non-uniform *and* refresh postponing patterns, we are able to trigger a bit flip on 86 % of the 28 DIMMs in our testbed. However, due the large number of possible patterns, it took several weeks of fuzzing to achieve this result. This also means the attacker cannot simply replicate the "native search space" in JavaScript as it would make the attack too slow and therefore unrealistic. Instead, the attacker needs a "JavaScript search space" that is small enough, yet covers most DIMMs. To this end, we manually analyze the per-DIMM top five patterns in the hopes of being able to detect similarities and discover what such a search space looks like.

Experiment. As mentioned in Section 5.2 (see "Best pattern"), the quality of a pattern is determined by the total number of bit flips it is able to produce while replacing the pattern's eviction blocks 100 times. By doing this, we make the pattern target different rows in possibly different banks. We ensure to avoid overlapping eviction blocks and patterns that consist of multiple blocks have *all* of them replaced every $1/100$ time. For each DIMM, we use this method to identify its five most effective patterns. We then *visualize* all $22 \cdot 5 = 110$ best patterns and manually look for similarities.

Results. Among all 110 patterns, we identified two types of patterns, the SB- and LW-patterns (Figures 10 and 11), of which at least one is in the top five of $20/22$ DIMMs (91 %) with native bit flips. The two excluded DIMMs are C02 (for which we only found one hard-to-reproduce pattern in total) and A02 (for which we found the SB-pattern to be the *sixth*-best pattern).

The single block or SB-pattern. The SB-pattern is illustrated

in Figure 10. It is rather simple: it consists of a single eviction block that may be non-uniform (by taking advantage of one of the three-lane compositions shown in Figure 7). Similarly, an SB-pattern *may* take advantage of refresh postponement. This means SB-patterns are sometimes uniform and not postponing. What characterizes the SB-pattern, however, is the number of double-sided pairs per lane ranging from 3–6 for a minimum of 12 and a maximum of 36 aggressors.

The long-wide- or LW-pattern. The other pattern that we identified as prevalent among the top patterns consists of two eviction blocks and is shown in Figure 11. Characteristic of this long-wide pattern is that one eviction block's pointer chase is *long*, i.e., it consists of relatively few double-sided pairs (2 in the figure) that are repeated often, while the second chase is *wide*, i.e., it consists of many different double-sided pairs (12 in the example), is typically non-uniform, but its aggressors are not accessed as often. For all LW-patterns, we found the ratio between the long and wide patterns to be in the range 2–5. That is, on the vertical axis, which means, for example, that the long chase might be 5× longer than the wide chase. As for the widths, the number of double-sided pairs per lane for the long block ranges from 1–3 (rather low) while those of the wide block range from 3–6. Furthermore, LW-patterns *always* use refresh postponement, but due to their relatively large size are made to fit inside *at least* 9 tREFIs. Finally, while the wide block is almost always non-uniform, the long block does not have to be, as also shown in the example. We hypothesize that in LW-patterns, the wide block acts as a distractor of the long block.

Together, the SB- and LW-patterns form only a small parameter space—small enough for the attacker to explore during the online part of the attack. Moreover, and as we will show in the next section, using only these patterns we are able to trigger bit flips in JavaScript on the majority (61 %) of DIMMs in our test pool.

8 Evaluation

In this section, we examine the search space of non-uniform, refresh-postponing, and self-evicting patterns more closely. In particular, we search for effective patterns on the 28 DIMMs listed in Section C. The result: for 86 % (24/28) of the devices, we are able to trigger bit flips without *clflush*. However, for 6 of them we were unable to make their best LW- and SB-patterns trigger bit flips in JavaScript as well. This means that we arrive at 61 % (17/28) of devices for which we managed to trigger bit flips in JavaScript using the LW- or SB-patterns discussed in the previous section. The complete results are shown in Table 2 and will now be further explained.

Benchmarking platforms. All *native* experiments, including those in this section, were performed on ordinary desktop machines equipped with either an Intel Core i7-7700K (Kaby Lake) or Intel Core i7-8700K (Coffee Lake) CPU. The JavaScript experiments, however, were only conducted on the Kaby Lake microarchitecture due to the complexity of the Coffee Lake slice addressing functions. Although these functions have been reverse engineered [5,8], we were unable to build eviction sets using the results reported in [5]. The systems' BIOSes were set to their default configurations, including the refresh rates. Our selection of 28 DDR4 DIMMs includes DIMMs from all three major DRAM vendors (Samsung, Micron, and SK Hynix) and of varying sizes and clock frequencies. Section C provides further details on the DIMMs.

Research question. We consider the following question: for how many DIMMs are we able to find an effective *clflush*-free pattern? We will divide this question into three parts:

(a) For which DIMMs are we able to find a self-evicting pattern *natively*?

(b) For which DIMMs are we able to find a self-evicting pattern *in JavaScript*? That is, using an SB- or LW-pattern.

(c) To what extent do refresh postponement and non-uniformity contribute to the results?

Experiment 6. To answer the first question, we configure POSTHAMMER to test self-evicting patterns with different levels of non-uniformity and refresh postponement. We further vary the number of aggressors by employing cache hits as discussed in Section 6.2.

On our Coffee Lake cluster, we fuzz each DIMM until either (i) we have found 30 patterns, or (ii) tested 50 000. Since we have significantly fewer Kaby Lake machines available, on these machines, we stopped the experiment after either 10 patterns were found or 5000 were tested.

The results are given in Table 2. We also show for which DIMMs we were able to trigger a bit flip using SMASH [4] and SledgeHammer [18] (both natively) and using POSTHAMMER in JavaScript while relying exclusively on the SB- and LW-patterns presented in Section 7.

Results 6a: coverage. First and foremost, the results show that we are able to find a self-evicting pattern for the majority of devices (24/28), though there are large differences between vendors. In particular, while for manufacturer A we have been able to find a pattern for every device, for manufacturer C we could only find a pattern on about half of the devices. This suggests that either (i) these DIMMs have more advanced in-DRAM mitigations or (ii) are simply less susceptible to the Rowhammer effect.

Second, while Table 2 shows a difference between the Kaby and Coffee Lake microarchitectures (e.g. C00 was found vulnerable on Coffee Lake but not on Kaby Lake, while the opposite applies to C01), in general, the rates at which the fuzzer was able to find patterns are comparable. For this reason, and because our earlier experiments (Figure 2, Figure 4, and Figure 9) did not show a noteworthy difference between the microarchitectures, we assume the difference displayed

Table 2: Fuzzing results. We report if a pattern was found with SMASH (SM), SledgeHammer (SH) (✔), or POSTHAMMER (✔). The KL and CL columns show the average number of effective patterns found every six hours on our Kaby Lake and Coffee Lake machines, respectively. The second last column shows if we also triggered bit flips in JavaScript (✔) and the final column (p_1) using which pattern.

	clflush-free (native)					JavaScript	
DIMM	SM	SH	**Posth.**	KL/6h	CL/6h	**Posth.**	p_1
A00	–	–	✔	0.96	1.7	✔	LW
A01	✔	–	✔	69.0	56.0	✔	LW
A02	✔	–	✔	1.0	0.49	✔	SB
A03	–	–	✔	0.42	1.4	✔	LW
A04	–	✔	✔	1.9	1.6	✔	LW
A05	✔	–	✔	1.6	1.5	✔	LW
A06	–	✔	✔	2.2	2.0	✔	LW
A07	–	–	✔	0.7	2.3	✔	LW
A08	–	–	✔	0.45	1.9	✔	LW
A09	–	–	✔	1.0	2.7	✔	LW
A10	✔	–	✔	56.0	46.0	✔	LW
A11	–	–	✔	0.28	0.97	✔	LW
A12	–	–	✔	0.98	2.1	✔	LW
B00	–	–	✔	–	0.0044	–	–
B01	✔	–	✔	0.17	0.21	✔	SB
B02	✔	–	✔	0.47	0.17	✔	SB
B03	–	–	–	–	–	–	–
B04	–	–	✔	–	0.041	–	–
B05	✔	–	✔	0.13	0.12	✔	SB
B06	–	–	✔	–	0.032	–	–
B07	–	–	–	–	–	–	–
B08	✔	–	✔	–	0.15	–	–
C00	–	–	✔	–	0.94	✔	LW
C01	–	–	✔	0.037	–	–	–
C02	–	–	✔	–	0.0044	–	–
C03	–	–	–	–	–	–	–
C04	–	–	✔	–	0.028	–	–
C05	–	–	–	–	–	–	–
Total	29%	7%	86%	61%	82%	61%	

in the table is exclusively due to (i) the fuzzer randomly selecting a pattern in a vast search space and (ii) the roughly 10× smaller data set gathered on the Kaby Lake machines, as mentioned above.

Third, for modules of the same vendor, we sometimes find a difference of two orders of magnitude in the rate at which POSTHAMMER was able to find effective patterns. For example, while for DIMMs A01 and A10 we would find more than 40 different patterns every 6 h (Coffee Lake), for other DIMMs by A this number is rather low and in the range 1–2. This shows not only that the same manufacturer may implement different mitigations, but also that—at least since DDR4—a Rowhammer attack demonstrated on a single DIMM [18] cannot be assumed to work on other devices as well.

Table 3: Pattern categories. The four categories ("Both", "Postponing only", etc.) of Figure 12.

	Refresh postponing	**Non-postponing**
Non-uniform	Both	Non-uniform only
Uniform	Postponing only	Neither[1]

[1] SMASH [4] and SledgeHammer [18] belong to this category.

Results 6b: JavaScript. The second last column of the table shows that for 17/28 (61 %) of devices we managed to reproduce the *clflush*-free bit flips in JavaScript on a Kaby Lake machine. The experiment consists of fuzzing a rather small search space that consists of LW- or SB-patterns only, as explained in Section 7. We also report which type of pattern—LW or SB—triggered a bit flip first in the p_1 column.

The results in the JavaScript columns show that (i) the DIMMs for which we did not manage to trigger a bit flip in JavaScript are also relatively hard to break natively, which is not surprising, and (ii) that DIMMs of manufacturer A seem more vulnerable to LW-patterns (though we also found some SB-patterns to be effective) while SB-patterns work well on DIMMs of manufacturer B. The latter suggests that adding non-uniformity by means of multiple eviction blocks (see Section 6.2) does not work well for manufacturer B. Moreover, it would explain why the (native) fuzzer, which randomly chooses a number of blocks in the range 1–3, required much more time to find patterns for manufacturer B compared to A, as shown in Table 2. We will report on the time it takes to find exploitable bit flips in Section 9.

Results 6c: contribution. To determine the contribution of both refresh postponement and non-uniformity to the results in Table 2 we analyze each of the patterns found more closely in Figure 12. Specifically, we focus on the native Coffee Lake results (sixth column in Table 2) because it is our largest data set. Each effective pattern is assigned to one of the categories shown in Table 3.

Based on Figure 12-(a), we make the following observation:

Observation 3. About half (11/23) of the DIMMs is vulnerable to a pattern that does *not* use refresh postponement or non-uniformity, i.e. is in the "Neither" category. *However, all (23/23) DIMMs are vulnerable to a pattern that uses either or both.*

In other words, without non-uniformity or refresh postponement, the attacker's impact is limited. Furthermore, Figure 12-(b) shows that:

Observation 4. The *strongest* patterns use both refresh postponement *and* non-uniformity.

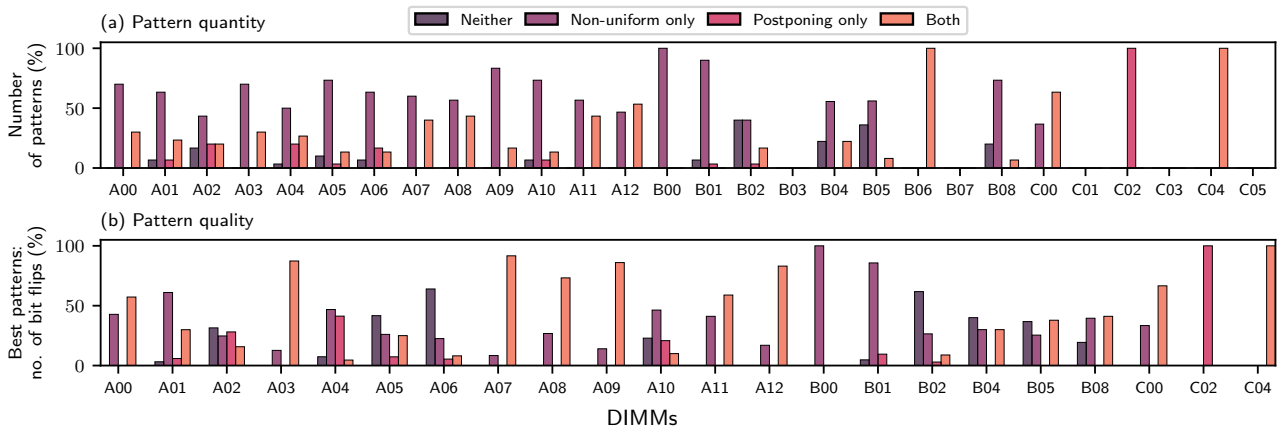


Figure 12: Contribution of refresh postponement and non-uniformity. The quantity (top figure) and quality (bottom) of patterns that rely on refresh postponement and/or non-uniformity compared to patterns that do not. For the meaning of the colors, see also Table 3.

This observation is based on the fact that for 9/23 (29 %) of the DIMMs, the best refresh postponing *and* non-uniform pattern (i.e. "Both") triggers at least 50 % of the bit flips triggered by that DIMM's four best patterns combined—one of each category. Conversely, in only 2/23 cases (A06 and B02), the "Neither" category's pattern is the strongest and triggers at least half of the bit flips. In all other cases, either *exclusively* non-uniform (A01, A04, and B01) or refresh postponing (C02) patterns win, or there is no clear winner.

Conclusion. First, the results affirm the need for non-uniformity and support our earlier findings that hinted at the positive effect (from the attacker's perspective) of postponing refreshes. They further suggest that non-uniformity is essential in order to bypass in-DRAM mitigations while refresh postponement rather *boosts* the effectiveness of a pattern than that it serves as a means to circumvent the mitigation independently. Second, with 61 % of the devices vulnerable to a bit flip in JavaScript, we have established the pervasiveness of Rowhammer-based browser attacks on DDR4-based systems.

9 Browser Exploitation

We demonstrate how our non-uniform and refresh-postponing patterns can be leveraged to build a practical JavaScript-based Rowhammer attack in a modern browser. We build on previous work [4, 6, 18] that demonstrates how to do a type-flipping attack in JavaScript. This attack exploits *NaN-boxing* in JavaScript Arrays, which allows storing heterogeneous data types (e.g., pointers and floating-point numbers) in the *same* array. By triggering bit flips in an Array that is vulnerable to Rowhammer, it is possible to modify type information and thereby convert pointers into floating-point numbers ($1 \rightarrow 0$ bit flip) and vice versa ($0 \rightarrow 1$ bit flip). We refer the interested reader to previous work [4, 6, 18] and our implementation for details. We use the latest stable version of 64-bit Firefox (130.0) running on a Kaby Lake system.

Contiguous memory. As shown before [4, 18], we exploit the buddy allocator in Linux to obtain contiguous memory, but unlike previous work we do not assume the availability of transparent huge pages (THPs). In order to be able to free the victim row and have it reallocated later, we need to work with small arrays. Otherwise, if our buffers are larger than 32 pages of 4 kB (depending on the size of the DIMM), the victim and aggressor rows will be part of the same array, making it impossible to release the victim without also releasing the aggressors and thus breaking the pattern.

Looking at allocation patterns, we find that Firefox creates virtual memory areas of 252 pages (just below a megabyte) to store buffers up to and including that size. We refer to these regions as *slabs*. For example, an array of 14 times 4 kB reside inside a slab, while a larger 1 MB array (256 pages) ends up in its own 1 MB virtual memory area instead of a slab. Furthermore, we find that it is possible to obtain physically contiguous slabs simply by first exhausting all fragmented memory. This makes slabs a suitable building block for constructing patterns: first, they can be made to be physically contiguous, and second, because they combine several smaller buffers, we are able to selectively free parts of them as part of memory massaging [28]. To give an example, assuming a DIMM of 8 GB, we allocate 50% of the system's memory (i.e., 4 GB) through 1 GB ArrayBuffer allocations to exhaust all lower-order (buddy allocator) pages, before allocating around 37.5% of the system's remaining memory (i.e., 2.95 GB) as slabs of 252 times 4 kB, where each slab consists of 18 ArrayBuffers of 14 pages each. We then use a cache side channel to color these slabs, using the same method used to color THPs in [4, 6].

Memory alignment. Coloring a slab means finding the page

(one out of 252) that is physically aligned to a megabyte. Once this page is found, we are able to calculate which offsets map to which slice, set, bank, and row, enabling us to build the patterns described in Section 6.2. To find this megabyte-aligned page, we simply guess that if this page is megabyte-aligned, then certain offsets within the slab are congruent. Two addresses are congruent if they map to the same set and slice. This strategy, combined with amplified cache eviction, allows us to color a slab within a few seconds.

Templating. For every bit flip that we find, we assess its exploitability by comparing the bit flip's direction (i.e., $0\rightarrow1$ vs. $1\rightarrow0$) and location (i.e., byte offset). Our attack needs both a $0\rightarrow1$ and a $1\rightarrow0$ bit flip. Further, we need the bit flips to be in the tag bits of the Array elements, which limits us to 15 out of 64 exploitable bits [4]. We continue with the next step once we found two patterns producing the required bit flips.

Memory massaging. We need to massage the memory layout to place an Array in the physically vulnerable location. For this, we release the pages containing our victim rows and spray 10 360 000 Array objects. We experimentally determined that this amount of memory results in the allocation of an Array at the vulnerable location. To release the memory, we use the same technique as reported earlier [4] and pass all references to a web worker that we terminate.

Crafting an arbitrary read/write primitive. These steps are described in detail in previous work [4, 6]. In summary, we retrieve the ArrayBuffer's virtual address ($1\rightarrow0$ bit flip), read out the ArrayBuffer's header ($0\rightarrow1$ bit flip), and craft an arbitrary read/write primitive ($0\rightarrow1$ bit flip) by creating a nested fake ArrayBuffer that the attacker controls from within the outer ArrayBuffer.

Results. We conclude with a detailed evaluation of the exploit. It takes a median of 11 seconds to build the first eviction set and 12 seconds to color 8 slabs (enough to build another eviction set), respectively. On DIMMs A01 and A10, it took us 11.4 and 7.0 minutes on average, respectively, to find a pattern that could trigger exploitable $0\rightarrow1$ and $1\rightarrow0$ bit flips in the slabs. Once we find the exploitable bit flips, the massaging stage takes around 8 seconds, and it is successful in 79.9% of the trials on average. The attacker can retry with another slab in case massaging is unsuccessful.

10 Discussion

We discuss the relevance of refresh postponement for mitigations and DDR5 devices.

Impact on deployed mitigations. We discussed why refresh postponement can benefit an attacker in Section 2.3. To make this more concrete, assume a device from vendor A, as analyzed in Section V-C of TRRespass [7] and Section III-B of Blacksmith [12], that samples α activations after receiving a refresh command. Assume N is the number of activations in a tREFI and P is the number of postponed refresh commands. Without refresh postponement, $\frac{\alpha}{N}$ activations are subject to sampling, while with refresh postponement, $\frac{\alpha}{N\times P}$ are subject to sampling. This shows the impact of refresh postponement on reducing the effectiveness of sampling in devices of vendor A, as we also empirically showed in Figure 4.

Impact on state-of-the-art mitigations. ProTRR [23] analyzed the impact of refresh and refresh management (RFM) postponement on deterministic in-DRAM mitigations with calculated bounds for secure operations. MINT [27] uses a queue to handle the impact of refresh and RFM postponement in its probabilistic tracker. We are not aware of other academic mitigations that consider refresh postponement in their design which will likely reduce their effectiveness. This impact can be derived analytically or using PosTHammer inside a simulator that implements the target mitigations. This is an interesting direction for future work.

DDR5 devices. The DDR5 standard [16] supports refresh postponement, but we currently lack effective patterns that bypass mitigations on newer DDR5 devices which is an orthogonal research direction to PosTHammer. It will be interesting to evaluate PosTHammer on DDR5 once effective patterns for such devices have been discovered.

11 Related work

Browser-based microarchitectural and Rowhammer attacks face three challenges not faced by the native attacker: first, accurate timers are unavailable. Second, *clflush* is not available either. Third, due to the lack of pointers and the fact that the attacker's script runs in a JavaScript engine, the physical memory layout of the attacker-controlled process is unknown.

Previous work shows how to bypass timer mitigations [9, 29, 36]. Major obstacles for Rowhammer-based browser attacks are the lack of *clflush* and the unknown memory layout. The former prevents DRAM access and therefore hammering. Microarchitectural attacks in the browser use CPU caches to build a timing side channel [3, 24, 35] but do not pursue DRAM accesses. Both classes of attacks use eviction sets to achieve their objectives, however.

While before DDR4 devices with mitigations, Rowhammer-based browser attacks could rely on double- or even single-sided patterns [2, 6, 10], recent patterns have become more complex [4] which incentives optimizing the access patterns to improve activation rates [18] and increases the need for knowledge of the physical memory layout [4, 18]. Accordingly, previous work [4, 6, 18] used allocator exhaustion to "force" the memory allocator to hand out contiguous memory, to form double-sided aggressor pairs. In comparison to PosTHammer, all these attacks create uniform Rowhammer

Table 4: Chronological overview of Rowhammer-based browser attacks.

Attack	Consequence	Novelty	Memory layout	Tested DIMMs
Rowhammer.js [10]	PTE exploit of [30]	Rowhammer without *clflush*	Huge pages	3x DDR3, 1x DDR4
Dedup Est Machina [2]	Arbitrary read/write	Leak data via deduplication	No assumptions	1x DDR3
GLitch [6]	Arbitrary read/write	Rowhammer through WebGL	Allocator exhaustion	1x LP-DDR3
SMASH [4]	*Variant of [6]*	Refresh synchronization	Huge pages	2x DDR4
SledgeHammer [18]	*Variant of [6]*	Multi-bank hammering	Allocator exhaustion	1x DDR4
POSTHAMMER	*Variant of [6]*	Refresh postponement	Allocator exhaustion	17x DDR4

access patterns that do not allow for pervasive Rowhammer attacks in the browser. Table 4 summarizes the contributions of related work and POSTHAMMER.

12 Conclusion

We built POSTHAMMER, a pervasive browser-based Rowhammer attack on DDR4 systems. POSTHAMMER leverages refresh postponement to weaken in-DRAM mitigations and a new abstraction called *lane* to add non-uniformity to self-evicting patterns. The *refresh-postponed non-uniform patterns* generated by POSTHAMMER can trigger bit flips on 86 % of our 28 DDR4 test devices. We found that these patterns share significant similarities and used this insight to reduce the pattern search space in JavaScript for practical end-to-end browser exploitation, which we also demonstrated.

Acknowledgements

We thank the anonymous reviewers and shepherd for their valuable feedback. This research was supported by a grant from the ETH Future Computing Laboratory (EFCL).

Appendices

A Ethics considerations

This work presents an attack on real systems. However: first, Rowhammer (on DDR4) is a known problem. Previous work [4, 12, 18] has already made DRAM vendors as well as browser developers aware of the practicality of Rowhammer-based browser attacks. By establishing the pervasiveness of this problem, we indirectly remind all parties involved of the need for a solution. Second, while using our attack we are able to escape (partially) from the browser's sandbox, thereby breaking through a security boundary, as such it does not allow the attacker to leak confidential information or otherwise harm end-users. This means we do *not* foresee any potential negative outcomes associated with the publication of this work.

B Open science

The native fuzzer, JavaScript fuzzer, and exploit are available at https://doi.org/10.5281/zenodo.14738153 and will also appear at https://github.com/comsec-group/posthammer.

C DDR4 DIMMs

In Table 5, we provide a detailed overview of the 28 DIMMs in our test pool.

References

[1] Tanj Bennett, Stefan Saroiu, Alec Wolman, Lucian Cojocar, and Avant-Gray Llc. Panopticon: A Complete In-DRAM Rowhammer Mitigation. *DRAMSec '21*. https://dramsec.ethz.ch/papers/panopticon.pdf.

[2] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *IEEE S&P '16*, pages 987–1004, 2016. https://ieeexplore.ieee.org/abstract/document/7546546.

[3] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking Data on Meltdown-resistant CPUs. In *ACM CCS '19*, 2019. https://dl.acm.org/doi/abs/10.1145/3319535.3363219.

[4] Finn de Ridder, Pietro Frigo, Emanuele Vannacci, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. SMASH: Synchronized Many-sided Rowhammer Attacks from JavaScript. In *USENIX Security '21*, pages 1001–1018, August 2021. https://www.usenix.org/conference/usenixsecurity21/presentation/ridder.

[5] Guillaume Didier and Clémentine Maurice. Calibration Done Right: Noiseless Flush+Flush Attacks.

Table 5: Tested DDR4 DIMMs. We list the details of the DDR4 UDIMMs used throughout our work. We abbreviate the DRAM vendors Samsung (A), Micron (B), and SK Hynix (C). We report the DIMM's manufacturing date (Mf. Date), frequency (Freq.), size; and like in previous work [13], the number of ranks (RK), bank groups (BG), banks per bank group (BA), and rows (R).

DIMM	Mf. Date [ww-yyyy]	Freq. [MHz]	Size [GiB]	DIMM Geometry (RK, BG, BA, R)
A00	03-2020	2666	8	$(1, 4, 4, 2^{16})$
A01	06-2020	2666	32	$(2, 4, 4, 2^{17})$
A02	51-2020	2132	4	$(1, 4, 4, 2^{15})$
A03	45-2020†	2132	8	$(1, 4, 4, 2^{16})$
A04	45-2020†	2132	8	$(1, 4, 4, 2^{16})$
A05	45-2020†	2132	8	$(1, 4, 4, 2^{16})$
A06	45-2020†	2132	8	$(1, 4, 4, 2^{16})$
A07	45-2020†	2132	16	$(2, 4, 4, 2^{16})$
A08	45-2020†	2132	16	$(2, 4, 4, 2^{16})$
A09	45-2020†	2132	16	$(2, 4, 4, 2^{16})$
A10	23-2020	2666	32	$(2, 4, 4, 2^{17})$
A11	45-2020†	2666	8	$(1, 4, 4, 2^{16})$
A12	16-2020	2666	16	$(2, 4, 4, 2^{16})$
B00	38-2019	2400	16	$(2, 4, 4, 2^{16})$
B01	45-2020†	2132	8	$(1, 4, 4, 2^{16})$
B02	43-2019	2400	4	$(1, 4, 4, 2^{15})$
B03	05-2020	2666	8	$(1, 4, 4, 2^{16})$
B04	07-2020	2400	8	$(1, 4, 4, 2^{16})$
B05	51-2019	2400	16	$(2, 4, 4, 2^{16})$
B06	45-2020†	2132	32	$(2, 4, 4, 2^{17})$
B07	09-2020	2134	8	$(2, 4, 4, 2^{15})$
B08	45-2020†	2400	8	$(1, 4, 4, 2^{16})$
C00	45-2020†	2132	16	$(2, 4, 4, 2^{16})$
C01	38-2020	2400	8	$(1, 4, 4, 2^{16})$
C02	38-2020	2400	8	$(1, 4, 4, 2^{16})$
C03	38-2020	2400	8	$(1, 4, 4, 2^{16})$
C04	38-2020	2400	8	$(1, 4, 4, 2^{16})$
C05	48-2017	2400	4	$(1, 4, 4, 2^{15})$

† Purchase date used as manufacturing date not reported by SPD.

In Leyla Bilge, Lorenzo Cavallaro, Giancarlo Pellegrino, and Nuno Neves, editors, *DIMVA '21*, volume 12756, pages 278–298. Springer International Publishing, Cham, 2021. https://inria.hal.science/hal-03267431/file/dimva21_didier.pdf.

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

[7] Pietro Frigo, Emanuele Vannacc, Hasan Hassan, Victor Van Der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TRRespass: Exploiting the Many Sides of Target Row Refresh. In *IEEE S&P '20*, pages 747–762, 2020. https://ieeexplore.ieee.org/abstract/document/9152631.

[8] L. Gerlach, S. Schwarz, N. Faroß, and M. Schwarz. Efficient and generic microarchitectural hash-function recovery. In *IEEE S&P '24*, pages 32–32, Los Alamitos, CA, USA, May 2024. https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00028.

[9] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the Line: Practical Cache Attacks on the MMU. In *NDSS*, February 2017. https://www.ndss-symposium.org/wp-content/uploads/2017/09/ndss2017_09-1_Gras_paper.pdf.

[10] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez, editors, *Springer DIMVA '16*, Lecture Notes in Computer Science, pages 300–321, Cham, 2016. https://link.springer.com/chapter/10.1007/978-3-319-40667-1_15.

[11] Hasan Hassan, Yahya Can Tugrul, Jeremie S. Kim, Victor van der Veen, Kaveh Razavi, and Onur Mutlu. Uncovering In-DRAM RowHammer Protection Mechanisms:A New Methodology, Custom RowHammer Patterns, and Implications. In *IEEE/ACM MICRO '21*, pages 1198–1213, New York, NY, USA, October 2021. https://dl.acm.org/doi/abs/10.1145/3466752.3480110.

[12] Patrick Jattke, Victor Van Der Veen, Pietro Frigo, Stijn Gunter, and Kaveh Razavi. BLACKSMITH: Scalable Rowhammering in the Frequency Domain. In *IEEE S&P '22*, pages 716–734, May 2022. https://ieeexplore.ieee.org/abstract/document/9833772.

[13] Patrick Jattke, Max Wipfli, Flavien Solt, Michele Marazzi, Matej Bölcskei, and Kaveh Razavi. ZenHammer: Rowhammer Attacks on AMD Zen-based Platforms. *USENIX Security '24*. https://www.usenix.org/conference/usenixsecurity24/presentation/jattke.

[14] JEDEC Solid State Technology Association. DDR3 SDRAM (JESD79-3F). https://www.jedec.org/sites/default/files/docs/JESD79-3F.pdf, July 2012.

[15] JEDEC Solid State Technology Association. DDR4 SDRAM (JESD79-4). https://www.jedec.org/sites/default/files/docs/JESD79-4.pdf, September 2012.

[16] JEDEC Solid State Technology Association. DDR5 SDRAM (JESD79-5). https://www.jedec.org/

sites/default/files/docs/JESD79-5.pdf, July 2020.

[17] JEDEC Solid State Technology Association. Near-Term DRAM Level Rowhammer Mitigation (JEP300-1). https://www.jedec.org/standards-documents/docs/jep300-1, March 2021.

[18] Ingab Kang, Walter Wang, Jason Kim, Stephan van Schaik, Youssef Tobah, Daniel Genkin, Andrew Kwong, and Yuval Yarom. SledgeHammer: Amplifying Rowhammer via Bank-level Parallelism. *USENIX Security '24*. https://www.usenix.org/conference/usenixsecurity24/presentation/kang.

[19] Jeremie S Kim, Minesh Patel, A Giray Yağlıkçı, Hasan Hassan, Roknoddin Azizi, Lois Orosa, and Onur Mutlu. Revisiting RowHammer: An Experimental Analysis of Modern DRAM Devices and Mitigation Techniques. page 14. https://ieeexplore.ieee.org/abstract/document/9138944.

[20] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. *ACM SIGARCH Computer Architecture News*, 42(3):361–372, June 2014. https://dl.acm.org/doi/abs/10.1145/2678373.2665726.

[21] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. RAMBleed: Reading Bits in Memory Without Accessing Them. In *IEEE S&P '20*, pages 695–711, San Francisco, CA, USA, May 2020. https://ieeexplore.ieee.org/document/9152687/.

[22] Moritz Lipp, Michael Schwarz, Lukas Raab, Lukas Lamster, Misiker Tadesse Aga, Clémentine Maurice, and Daniel Gruss. Nethammer: Inducing Rowhammer Faults through Network Requests. In *EuroS&PW '20*, pages 710–719, September 2020. https://ieeexplore.ieee.org/document/9229701.

[23] Michele Marazzi, Patrick Jattke, Flavien Solt, and Kaveh Razavi. ProTRR: Principled yet Optimal In-DRAM Target Row Refresh. In *IEEE S&P '22*, pages 735–753, San Francisco, CA, USA, May 2022. https://ieeexplore.ieee.org/abstract/document/9833664.

[24] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In *ACM CCS '15*, pages 1406–1418, Denver, Colorado, USA, October 2015. https://dl.acm.org/doi/abs/10.1145/2810103.2813708.

[25] Lois Orosa, Ulrich Rührmair, A Giray Yaglikci, Haocong Luo, Ataberk Olgun, Patrick Jattke, Minesh Patel, Jeremie Kim, Kaveh Razavi, and Onur Mutlu. SpyHammer: Understanding and exploiting RowHammer under fine-grained temperature variations. *IEEE Access*, 2024. https://ieeexplore.ieee.org/abstract/document/10547262.

[26] Rui Qiao and Mark Seaborn. A New Approach for Rowhammer Attacks. In *HOST '16*, pages 161–166, May 2016. https://ieeexplore.ieee.org/document/7495576/?arnumber=7495576.

[27] Moinuddin Qureshi, Salman Qazi, and Aamer Jaleel. MINT: Securely Mitigating Rowhammer with a Minimalist in-DRAM Tracker . In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 899–914, Los Alamitos, CA, USA, November 2024. IEEE Computer Society.

[28] Kaveh Razavi, Ben Gras, Cristiano Giuffrida, Erik Bosman, Bart Preneel, and Herbert 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.

[29] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In Aggelos Kiayias, editor, *Financial Cryptography and Data Security*, Lecture Notes in Computer Science, pages 247–267, Cham, 2017. Springer International Publishing. https://link.springer.com/chapter/10.1007/978-3-319-70972-7_13.

[30] Mark Seaborn and Thomas Dullien. Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges. https://www.blackhat.com/docs/us-15/materials/us-15-Seaborn-Exploiting-The-DRAM-Rowhammer-Bug-To-Gain-Kernel-Privileges.pdf, 2015.

[31] Andrei Tatar, Radhesh Krishnan Konoth, Cristiano Giuffrida, Herbert Bos, Elias Athanasopoulos, and Kaveh Razavi. Throwhammer: Rowhammer Attacks over the Network and Defenses. In *USENIX ATC '18*, page 14, 2018. https://www.usenix.org/conference/atc18/presentation/tatar.

[32] Youssef Tobah, Andrew Kwong, Ingab Kang, Daniel Genkin, and Kang G Shin. Spechammer: Combining spectre and rowhammer for new speculative attacks. In *IEEE S&P '22*, pages 681–698, 2022. https://ieeexplore.ieee.org/abstract/document/9833802.

[33] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *ACM CCS '16*, pages 1675–1689, Vienna Austria, October 2016. https://dl.acm.org/doi/10.1145/2976749.2978406.

[34] Victor van der Veen, Martina Lindorfer, Yanick Fratantonio, Harikrishnan Padmanabha Pillai, Giovanni Vigna, Christopher Kruegel, Herbert Bos, and Kaveh Razavi. GuardION: Practical Mitigation of DMA-based Rowhammer Attacks on ARM. In *DIMVA '18*, June 2018. https://link.springer.com/chapter/10.1007/978-3-319-93411-2_5.

[35] Stephan van Schaik, Alyssa Milburn, Sebastian Osterlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue In-Flight Data Load. In *IEEE S&P '19*, pages 88–105, San Francisco, CA, USA, May 2019. https://ieeexplore.ieee.org/abstract/document/8835281.

[36] Johannes Wikner, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Spring: Spectre Returning in the Browser with Speculative Load Queuing and Deep Stacks. In *WOOT*, May 2022.

[37] Zhi Zhang, Wei He, Yueqiang Cheng, Wenhao Wang, Yansong Gao, Dongxi Liu, Kang Li, Surya Nepal, Anmin Fu, and Yi Zou. Implicit Hammer: Cross-Privilege-Boundary Rowhammer through Implicit Accesses. *IEEE Transactions on Dependable and Secure Computing*, pages 1–18, 2022. https://ieeexplore.ieee.org/abstract/document/9919335.

USENIX Security '25 Artifact Appendix: Posthammer: Pervasive Browser-based Rowhammer Attacks with Postponed Refresh Commands

Finn de Ridder
ETH Zurich

Patrick Jattke
ETH Zurich

Kaveh Razavi
ETH Zurich

A Artifact Appendix

A.1 Abstract

Posthammer shows that browser-based Rowhammer attacks are pervasive. In particular, we show that the majority of DDR4 devices are vulnerable to *clflush*-free Rowhammer patterns that the attacker can launch from client-side JavaScript.

Posthammer includes the following artifacts: first, an experiment for inducing and measuring refresh postponement. Second, a fuzzer for triggering Rowhammer bit flips natively using *clflush*-free and *refresh-postponing* patterns in a large search space. Third, a JavaScript version of the fuzzer that searches for effective patterns in a reduced search space. Fourth, an exploit that obtains an arbitrary read-write primitive in the address space of the JavaScript process.

A.2 Description & Requirements

To reproduce our results, the evaluator needs:

1. The hard- and software dependencies listed under A.2.3 and A.2.4.
2. The artifacts themselves, which are available at `https://doi.org/10.5281/zenodo.14738152` and `https://github.com/comsec-group/posthammer` (preferred).

After unpacking the artifacts, execute the following commands:

1. For the first two artifacts, the experiment and native fuzzer, in `./native-fuzzer`, execute `./main.sh |& tee dump`. Simply run it again if you get an error about `mem.o` missing. The refresh postponement experiment requires the `SPLIT_DETECT` macro to be defined.
2. For the JavaScript fuzzer, in `./js-dbg-hugepages`, execute `make clean && make; make`. We run `make` twice to force execution despite the transpiler warnings.
3. For the exploit, in `./js-exploit`, also execute `make clean && make; make`.

The exploit may fail either due to a segfault or an assertion failing (e.g. because it cannot find an eviction set). In these cases, please try again. Similarly, the native fuzzer may fail to find its first eviction set, but should otherwise not crash.

The exploit will run until it has (intentionally) segfaulted at `0x1337` while the native fuzzer will fuzz indefinitely while writing its results to `./pattern/flip.csv`. Human-friendly output can be found in `dump` (if captured as suggested above). Search for `->` to view the bit flips that have been triggered.

A.2.1 Security, privacy, and ethical concerns

The artifacts are safe. As mentioned above, upon successful execution, the exploit causes a harmless segmentation fault at address `0x1337`. This segmentation fault will be reported in the kernel log, see `dmesg`. Unsuccessful runs will trigger arbitrary but equally benign segmentation faults in the address space of the script.

A.2.2 How to access

The latest version of the artifacts is available at `https://github.com/comsec-group/posthammer`. The directory structure suggests where which artifact can be found: the exploit is contained in `js-exploit`, the native fuzzer in `native-fuzzer`, and the JavaScript fuzzer in `js-dbg-hugepages`.

A.2.3 Hardware dependencies

1. A desktop machine with an Intel Core i7-7700K (Kaby Lake) processor. *This is a strict requirement for the exploit.* The native fuzzer also works on Intel Core i7-8700K (Coffee Lake) CPUs.
2. A (single) vulnerable DDR4 DIMM. The specifics of the DIMMs used in the paper are given in the paper's appendix. For testing, we recommend a Samsung (A) DIMM, as they are most vulnerable to Posthammer.

A.2.4 Software dependencies

1. A Debian-based operating system. We have used Ubuntu 18.04.6 under Linux 5.4.0-150-generic. Although we recommend to use exactly these versions, newer ones might also work. For example, the native fuzzer also works on Ubuntu 22.04.5 under Linux 5.15.0-130-generic.
2. Certain software packages. The details are given in the README. The installation should be straightforward.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

We refer the evaluator to the README for the installation instructions.

A.3.2 Basic Test

The following tests can be used to verify the environment.

1. The availability and version of the TypeScript transpiler:

```
$ tsc --version
Version 2.7.2
```

2. The availability and version of the JavaScript shell:

```
$ ./jsshell-130/js --version
JavaScript-C130.0
```

A.4 Evaluation workflow

A.4.1 Major Claims

C0: Rowhammer patterns crafted as specified in the paper (Sections 5 and 6) induce refresh postponement. This is proven by Figure 2 of the paper, which can be reproduced using E0, see below.

C1: On the majority of DDR4 devices, these non-uniform and/or refresh postponing *clflush*-free Rowhammer patterns trigger bit flips while the self-evicting patterns used in previous work do not. This is proven by Experiment 6 (Table 2) in the paper.

C2: It is possible to trigger these bit flips from JavaScript with a reduced search space. See also Experiment 6 and Table 2.

C3: The bit flips triggered by these patterns can be used to obtain an arbitrary read-write primitive in the JavaScript runtime.

A.4.2 Experiments

The experiments below map linearly to the claims in A.4.1.

E0: *Refresh postponement*: produces Figure 2 of the paper and therefore shows that our patterns induce refresh postponement.

1. Navigate to `./native-fuzzer/pattern/pattern.c`. Open the file and enable the `SPLIT_DETECT` macro.
2. Execute `./native-fuzzer/main.sh`. This should take around 30 minutes. Plot the data it has written to `./native-fuzzer/split.csv`.

E1: *Native fuzzer:* explores the *clflush*-free, non-uniform, and refresh-postponing pattern space. Triggers bit flips on most DDR4 devices, see again Table 2.

1. Make sure the `SPLIT_DETECT` macro is *undefined* (default).
2. Execute `./native-fuzzer/main.sh |& tee dump`. Depending on the vulnerability of the DIMM, it may take several hours until the first bit flip. As Table 2 shows, however, for most DIMMs, 6 hours should suffice.

As mentioned A.2, we recommend piping the output to a file and grepping it for the arrow symbol `->` to check for bit flips.

E2: *JavaScript fuzzer*: shows that the native patterns translate to JavaScript. Relies on huge pages for convenience.

1. Navigate to `./js-dbg-hugepages` and execute `make clean && make; make`.

The `Makefile` should automatically enable transparent huge pages (THPs).

E3: *Exploit*: the exploit. Does *not* rely on huge pages. Uses two bit flips to obtain an arbitrary read-write primitive in the JavaScript runtime. To showcase the primitive, we write to virtual address `0x1337` and segfault.

1. Navigate to `./js-exploit` and execute `make clean && make; make`.

The exploit may take up to an hour to complete. Moreover, it may segfault before completion due to *unwanted* bit flips. It will print `About to segfault at 0x1337...` just before segfaulting as planned, which may be verified by inspecting `dmesg`.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.