SMASH: Synchronized Many-sided Rowhammer Attacks from JavaScript

Finn de Ridder
ETH Zurich
VU Amsterdam

Pietro Frigo
VU Amsterdam

Emanuele Vannacci
VU Amsterdam

Herbert Bos
VU Amsterdam

Cristiano Giuffrida
VU Amsterdam

Kaveh Razavi
ETH Zurich

Abstract

Despite their in-DRAM Target Row Refresh (TRR) mitigations, some of the most recent DDR4 modules are still vulnerable to many-sided Rowhammer bit flips. While these bit flips are exploitable from native code, triggering them in the browser from JavaScript faces three nontrivial challenges. First, given the lack of cache flushing instructions in JavaScript, existing eviction-based Rowhammer attacks are already slow for the older single- or double-sided variants and thus not always effective. With many-sided Rowhammer, mounting effective attacks is even more challenging, as it requires the eviction of many different aggressor addresses from the CPU caches. Second, the most effective many-sided variants, known as *n*-sided, require large physically-contiguous memory regions which are not available in JavaScript. Finally, as we show for the first time, eviction-based Rowhammer attacks require proper synchronization to bypass in-DRAM TRR mitigations.

Using a number of novel insights, we overcome these challenges to build SMASH (Synchronized MAny-Sided Hammering), a technique to succesfully trigger Rowhammer bit flips from JavaScript on modern DDR4 systems. To mount effective attacks, SMASH exploits high-level knowledge of cache replacement policies to generate optimal access patterns for eviction-based many-sided Rowhammer. To lift the requirement for large physically-contiguous memory regions, SMASH decomposes *n*-sided Rowhammer into multiple double-sided pairs, which we can identify using slice coloring. Finally, to bypass the in-DRAM TRR mitigations, SMASH carefully schedules cache hits and misses to successfully trigger *synchronized many-sided Rowhammer* bit flips. We showcase SMASH with an end-to-end JavaScript exploit which can fully compromise the Firefox browser in 15 minutes on average.

1 Introduction

Transistor scaling has been continuously improving the performance and capacity of modern DRAM devices. Security has instead been lagging behind. In particular, rather than addressing the root cause of the Rowhammer bug, DDR4 introduced a mitigation known as Target Row Refresh (TRR). TRR, however, has already been shown to be ineffective against many-sided Rowhammer attacks from native code, at least in its current in-DRAM form [12]. However, it is unclear whether TRR's failing also re-exposes end users to TRR-aware, JavaScript-based Rowhammer attacks from the browser. In fact, existing many-sided Rowhammer attacks require frequent cache flushes, large physically-contiguous regions, and certain access patterns to bypass in-DRAM TRR, all challenging in JavaScript.

In this paper, we show that under realistic assumptions, it is indeed possible to bypass TRR directly from JavaScript, allowing attackers to exploit the resurfaced Rowhammer bug inside the browser. In addition, our analysis reveals new requirements for practical TRR evasion. For instance, we discovered that activating many rows in rapid succession as shown in TRRespass [12] may not always be sufficient to produce bit flips. The scheduling of DRAM accesses also plays an important role.

Target Row Refresh. The discovery of the Rowhammer bug in 2014 [18] has led to an entire new class of attacks that all take advantage of the bug's promise: bit flips across security boundaries. In particular, researchers have demonstrated practical attacks on browsers, virtual machines, servers, and mobile systems, launched from native code, JavaScript, and even over the network [5,9,11,14,15,24,33,36,38,41,42,44]. In addition to these memory corruption attacks, Rowhammer may also serve as a side channel to leak information [21].

In response to the onslaught, manufacturers enhanced DDR4 chips with in-DRAM TRR—a Rowhammer "fix" which monitors DRAM accesses to mitigate Rowhammer-like activities. TRR consists of two components: a *sampler* and an *inhibitor*. The sampler is responsible for sampling memory requests to detect potentially Rowhammer-inducing sequences before they do harm. The inhibitor seeks to avert attacks by proactively refreshing the victim rows. Unfortu-

nately, TRRespass [12] recently showed that the mitigation is inadequate and can be bypassed by moving from double-sided to many-sided Rowhammer, i.e., activating not just two but up to 19 rows depending on the particular TRR implementation.

The crux of the problem is the memory chips' sampler. A reliable sampler would take enough samples to provide the inhibitor with sufficiently accurate information to refresh all the necessary victim rows in the case of a Rowhammer attack. Unfortunately, common sampler implementations monitor a limited number of aggressors and always at the same time [12], implicitly relying on the assumption that memory requests will arrive in an uncoordinated, chaotic fashion. However, given precise control over the rows to hammer by means of large physically-contiguous memory regions and by aggressively hammering multiple rows through explicit cache flushing (using the `CLFLUSH` instruction), many-sided Rowhammer can overwhelm the sampler and trigger bit flips even in TRR-enabled DRAM [12].

Bypassing TRR from JavaScript. While the resurrection of native-code Rowhammer on modern DDR4 systems is certainly serious, especially in clouds and similar environments, it does not immediately affect Web users exposed to attacks from JavaScript. In the absence of `CLFLUSH` and control over physically-contiguous memory, hammering a large number of rows by means of cache evictions, at a rate that is high enough to induce bit flips while still bypassing in-DRAM TRR, is not possible without new insights. Note that compared to double-sided Rowhammer, many-sided Rowhammer patterns exacerbate these challenges, as they require even *more* physically-contiguous memory and even *more* evictions.

A first key insight that helps us simplify the access patterns and greatly reduce the demand for physically-contiguous memory is that many-sided Rowhammer is equivalent to *many times double-sided* Rowhammer. As we will see, we can easily identify suitable double-sided pairs using slice coloring. Our slice-coloring strategy exploits slice-collision side channels and uses amplification techniques [27] to boost the signal and operate with the (unmodified) low-resolution, jittery timers available to JavaScript in modern browsers [35].

Our second key insight is that we can use high-level knowledge of cache replacement policies to improve the efficiency of eviction-based many-sided Rowhammer. In particular, rather than common eviction sets, we carefully construct access patterns such that every single cache miss itself leads to an aggressor row access and thus contributes to the hammering activity. Since all the misses are now useful, we minimize the number of "useless" memory accesses for eviction to a small number of highly efficient cache hits. Interestingly enough, such *self-evicting* patterns can be even more efficient than traditional flush-based patterns.

Even these optimized hammering patterns do not trigger bit flips without our third key insight. Where previous work on Rowhammer focuses on blindly generating as many access patterns per time unit as possible, we discovered that carefully *scheduling* the accesses plays an important role in bypassing TRR. Specifically, we show for the first time that an attacker needs to ensure that the sampler consistently samples the accesses to the same set of rows, allowing the (unsampled) accesses to the other rows to hammer away without hindrance from TRR's refresh operations. As we shall see, we will do so by synchronizing the access patterns with the refresh commands. Using our three novel insights, we build SMASH (Synchronized MAny-Sided Hammering), a technique to mount TRR-aware, JavaScript-based Rowhammer attacks. To demonstrate the practicality of SMASH, we present an end-to-end exploit to fully compromise the Firefox browser without software bugs in 15 minutes on average.

Summarizing, we make the following contributions:

1. A demonstration of the first end-to-end (synchronized many-sided) Rowhammer attack on TRR-enabled DDR4 in modern browsers, providing first evidence that Rowhammer continues to threaten Web users.

2. An automated approach to generate optimal access patterns for many-sided Rowhammer without relying on cache flushing instructions.

3. A further analysis of TRR, revealing synchronization as an additional requirement for successful attacks.

2 Background

We briefly discuss DRAM, Rowhammer, and caches to help readers understand the challenge of performing Rowhammer from JavaScript in the browser on modern DDR4.

2.1 DRAM

Since the introduction of synchronous DRAM or SDRAM, main memory is organized in *banks*, partitioned among the DRAM chips attached to the *Dual Inline Memory Module (DIMM)* or simply module. To access data in memory, the CPU's memory controller selects a bank and broadcasts its requests to all chips sharing the same *rank*, where a rank corresponds to the chips on one side of the module. One or more DIMMs may be connected to the processor's memory interface by the memory bus or *channel*. The number of available channels depends on the microarchitecture. For example, Intel's Kaby Lake processors have two channels, each supporting up to two modules.

The memory controller uses parts of the physical memory address to select the corresponding channel, DIMM, rank, and bank. A bank is a two-dimensional array of cells storing bits. To read their content, the memory controller further uses parts of the physical address to bring a row of information into the bank's *row buffer*. Once the row is in the row buffer, the CPU's memory controller can read and write to different

offsets within this buffer using column bits in the physical address of the target location.

To prevent data loss from the slow but continuous leakage of charge from the capacitors that make up the DRAM cells, the cells need to be refreshed periodically. Each full refresh consists of transferring the rows to the row buffer and writing them back. The DDR4 SDRAM standard specifies that under normal conditions, the per-row refresh or `REF` command should be issued every 7.8 µs.

2.2 Rowhammer

Kim et al. [18, 45] show that by repeatedly activating a row (the *aggressor* row) at high frequency, it is possible to cause a disturbance error in one of the neighboring (*victim*) rows. The disturbance manifests as a bit flip, where 0 becomes 1 or vice versa. The Rowhammer effect may be amplified in a particular row by activating *both* adjacent rows in alternating fashion, an access pattern known as double-sided Rowhammer. Researchers have shown Rowhammer attacks on numerous targets in various scenarios [5, 9, 11, 14–16, 21, 32, 33, 36–38, 41, 42, 44].

2.3 Target Row Refresh

Target Row Refresh (TRR) is the deployed industry solution against the Rowhammer vulnerability. TRR tries to *detect* Rowhammer-inducing access patterns to then prevent a bit from flipping, e.g., by refreshing the victim rows. The recent investigation into TRR by Frigo et al. [12] concluded that almost all recent DDR4 devices advertise themselves as Rowhammer-free by implementing the TRR mitigation inside the DRAM device itself. The in-DRAM TRR features a sampling mechanism to detect the aggressor rows and an inhibitor mechanism that refreshes the victim rows. They also showed that it is often possible to bypass in-DRAM TRR. By moving from double-sided Rowhammer to many-sided Rowhammer, where not just two but many rows (up to 19) are hammered repeatedly, it is still possible to trigger bit flips. Experiments suggest that the reason behind the effectiveness of many-sided patterns is due to the limited space in the *sampler*. This means that if there are more victims (because there are more aggressors) than the sampler can track, some will go unprotected.

2.4 CPU caches

To perform Rowhammer attacks in the browser, the attacker needs to flush aggressor addresses from different levels of CPU cache. Modern Intel processors have three levels of set-associative caches. In an N-way set-associative cache, each set accommodates N cache lines. Once a set is full but a new line for that set arrives, a replacement policy determines which line to evict. The replacement policies for many microarchitectures have already been reverse engineered [3, 43].

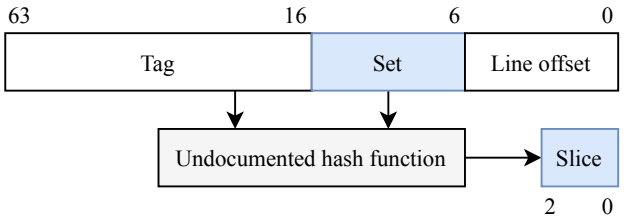

Figure 1: Displayed is a physical address and the mappings its bits are used for. Only the LLC is partitioned into slices. For efficiency reasons, the L1 cache uses the virtual instead of physical address bits for set addressing.

Many modern processors have an *inclusive* last-level cache (LLC): any 64-byte cache line stored in the upper levels also resides in the LLC. Reasoning about cache behavior becomes simpler with an inclusive LLC, in particular because eviction from the LLC implies eviction from the entire cache hierarchy.

The LLC set to which a cache line belongs is determined by the *set index bits* of the physical address, as shown in Figure 1. The width of the set index is determined by the number of sets of the LLC. For example, the Kaby Lake LLC has 1024 sets per *slice*. Slices first appeared in the Sandy Bridge microarchitecture in 2011, and further partition the LLC in 4, 8, or 16 chunks [26]. Originally, the number of slices was equal to the number of CPUs, but in recent microarchitectures it instead equals the number of hyperthreads [7]. Intel uses undocumented but reverse engineered [10, 26, 46] hash functions to map cache lines to slices. As we shall see later, cache slicing poses challenges for the creation of access patterns for many-sided Rowhammer attacks in the browser.

3 Threat Model

We assume an attacker who controls a malicious website (or a malicious ad on a benign website) that is visited by the victim. The attacker does not rely on any software bug but only exploits Rowhammer bit flips triggered from within the JavaScript sandbox to gain control over the victim's browser. We assume the victim's system deploys all the state-of-the-art mitigations against Rowhammer and side-channel attacks, i.e., modern in-DRAM Rowhammer mitigations [12, 22, 28] and browser mitigations against microarchitectural attacks, including low-resolution, jittery timers and mitigations against speculative execution attacks [29, 31, 35, 40]. Finally, similar to [15], SMASH relies on transparent huge pages (THP) for crafting its access patterns. See Appendix C for an overview of the default THP settings on popular Linux distributions.

4 Rowhammering DDR4 in the Browser

Carrying out a Rowhammer attack from inside JavaScript has never been trivial [15]. The attacker needs to find a way to flush the aggressors from the cache without relying on cache flushing instructions. Lack of memory addressing information in JavaScript further complicates such attacks. The arrival of the in-DRAM TRR mitigation only exacerbated such challenges. Because of the mitigation, ordinary double-sided Rowhammer will no longer suffice. To attack TRR-enabled DDR4, the attacker needs a many-sided access pattern. This poses our first challenge:

> **Challenge 1:** to build a many-sided access pattern, the attacker needs a large chunk of physical memory, which is hard to acquire in JavaScript.

Many-sided patterns consist of many adjacent rows. Since DRAM row addresses are determined by high physical address bits, collecting adjacent rows requires a relatively large amount of physical memory. As shown later, SMASH addresses this challenge by applying a new insight about many-sided Rowhammer that allows it to collect the required aggressor addresses without the need for a large contiguous chunk of physical memory.

The next hurdle faced by the attacker is: how to make sure every memory access goes to DRAM (and not one of the caches)? The attacker could try to adopt a known solution such as Rowhammer.js [15] or the technique presented by Aweke et al. [4]. These methods use eviction sets to create `CLFLUSH`-free access patterns. Aweke et al. [4] take advantage of the LLC's replacement policy and introduce two additional cache misses and a series of hits to ensure the eviction of double-sided or single-sided pairs, while Rowhammer.js searches for efficient eviction strategies that also introduce at least two additional misses and many more hits.

The problem with these approaches is that, when applied to many-sided access patterns, they cause an intolerable slowdown. We evaluated the effects of the method from Aweke et al. [4] on our test bed $\mathcal{S}_0$ (specified in Section 8) for creating `CLFLUSH`-free 18-sided access patterns. In this experiment, the 18 aggressor rows end up in different cache sets. Consequently, 270 additional cache hits (15 cache hits multiplied by 18 aggressors) and 18 additional cache misses are necessary for ensuring that the 18 aggressors are evicted from the LLC. We chose the method from Aweke et al. [4] since it has less overhead compared to Rowhammer.js. Still, we found this pattern to be too slow to trigger bit flips. This brings us to our second challenge:

> **Challenge 2:** the attacker needs to find a strategy to produce patterns that can efficiently perform many-sided Rowhammer without introducing too many additional cache hits and misses.

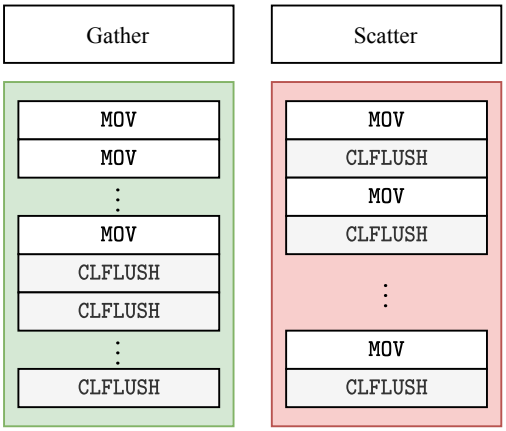

Figure 2: The order of memory requests and `CLFLUSH` instructions matters. The "gather" 18-sided pattern on the left does produce bit flips while the "scatter" 18-sided pattern on the right does not.

As shown later, SMASH addresses this challenge by crafting optimal access patterns that ensure all the cache misses land on the aggressor rows and contribute to hammering.

The next important observation we make is about the order of cache hits and cache misses. The common belief is that as long as enough requests are sent to memory in a given period of time, it is possible to trigger Rowhammer bit flips. Another experiment, summarized in Figure 2, suggests that this does not hold for DDR4 devices with in-DRAM TRR. In one case, we send 18 memory requests in a batch for an 18-sided pattern, followed by the `CLFLUSH` instructions that flush the aggressors from the cache. We confirm that this pattern triggers bit flips as shown in previous work [12]. However, if we interleave the `CLFLUSH` instructions with memory requests to the aggressor rows, we can no longer trigger bit flips despite sending the same number of requests in a given period of time. As we show later, this is due to the properties of the TRR mitigation. This observation leads to our third and final challenge:

> **Challenge 3:** the attacker must carefully schedule the sequence of cache hits and misses to bypass the in-DRAM TRR mitigation successfully.

As shown later, SMASH addresses this challenge by synchronizing DRAM accesses with the TRR mitigation.

4.1 Overview

In the remainder of the paper we tackle the aforementioned three challenges. In Section 5 we discuss how we can relax the requirements of large memory allocations (C_1) imposed by TRRespass [12] showing how one of the parameters discussed by Frigo et al. [12] (i.e., row location) does not always play a role when trying to bypass in-DRAM mitigations. We then show (Section 6) how we can increase the hammering

Table 1: Minimum contiguous allocations. The table reports the index of the physical address bit that maps to the LSB of the DRAM row address, for different memory configurations. The final column shows how much contiguous physical memory is needed to control three adjacent rows, computed as $2^{(LSB+\log_2 3)}$ B.

Organization				LSB row address	Min. alloc.
Ch.	DIMMs/Ch.	Ranks	Banks		
2	2	2	16	20	3.0 MB
2	1	2	16	19	1.5 MB
1	2	2	16	19	1.5 MB
1	1	2	16	18	0.75 MB
2	1	1	16	18	0.75 MB
1	2	1	16	18	0.75 MB
1	1	1	16	17	0.38 MB

throughput (C_2) by building gray-box self-evicting Rowhammer patterns that rely on the known cache replacement policies of modern Intel CPUs [3]. Finally, in Section 7 we discuss why maximizing throughput alone does not yet allow us to trigger bit flips (Section 4) and that we need to carefully order our memory accesses to bypass TRR (C_3).

After addressing these challenges, we will be able to trigger bit flips from the browser on modern DDR4 systems. We then evaluate the effectiveness of our self-evicting patterns on different memory modules and configurations in Section 8 and finally show how we can exploit the bit flips to compromise the latest version of the Firefox browser without relying on any software bug (Section 9).

5 Minimal Rowhammer Patterns

As discussed in Section 4, TRRespass requires contiguous memory allocations that are bigger than what is provided by modern operating systems to control the location of each aggressor. This is a consequence of the mapping between physical memory and DRAM addresses——and the way the operating system provides physical memory ranges to user space applications (see Figure 3). For example, in order to build a 19-sided pattern, one that TRRespass found to be effective against one of our test beds (see Section 8), the attacker requires $2^{(17+\log_2 37)} = 4.63$ MB of contiguous physical memory since the DRAM row address starts at the high order physical address bit 17 and because we need $2 \cdot 19 - 1 = 37$ rows in total, including victim rows, to form a 19-sided pattern. Obtaining such allocations is not trivial from the restricted environment of the JavaScript sandbox.

Contiguous memory in JavaScript. In order to gain control over DRAM row addresses from JavaScript prior work relies on two techniques to obtain contiguous allocations: (i) 2 MB Transparent Huge Pages (THP) [15] or (ii) massaging the buddy allocator in order to obtain high order allocations (max. 4 MB) [11]. Neither technique allows us to obtain the 4.63 MB of contiguous physical memory necessary to perform 19-sided Rowhammer.

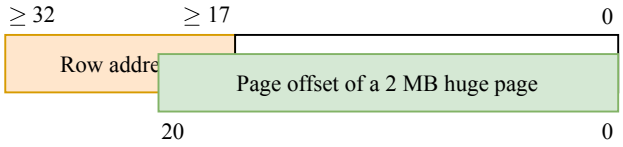

Figure 3: Row address control. The high order bits of a physical address determine the DRAM row address. In this example, where the LSB of the row address is 17, a 2 MB huge page allows the attacker to control $2^{(20-17+1)} = 16$ rows.

Relaxing the constraints. In one of our initial experiments we tried to understand the impact of row location when trying to bypass in-DRAM TRR. Starting from the assisted double-sided pattern described by Frigo et al. [12] (i.e., a double-sided pair "escorted" by an arbitrary row) we implemented its generalization: N-assisted double-sided[1]. That is, a pattern where a single double-sided pair is accompanied by N dummy rows. This means a 19-sided pattern becomes double-sided with $N = 19 - 2 = 17$ dummies. While Frigo et al. [12] observed that on a specific DIMM the locations of these dummies matter, in the experiments on our test beds we did not observe any noticeable difference in the number of flips with dummy rows at arbitrary locations within the same bank. This means that the attacker only needs to form a single double-sided pair and N dummy rows mapping to the same bank—a requirement that we will show is easier to fulfill.

Minimum viable allocations. The DRAM row address is determined by the outcome of linear functions applied to the physical address. These functions simply map high order physical address bits to the row address (see Figure 3). Thanks to the discovery of N-assisted double-sided we now need to control only three adjacent DRAM rows: two aggressor rows and a victim row in the middle. In other words, we need to control only the two LSBs of the DRAM row address.

To find out how much contiguous physical memory we need (or where in the physical address these two LSBs are), we reverse engineered the DRAM addressing functions for most modern Intel CPUs.[2] They can be found in Table 5 of Appendix A.

Given these particular functions, Table 1 shows how much contiguous physical memory is required to control one double-

[1] We could not trigger bit flips when using only unpaired rows à la single-sided Rowhammer.

[2] This task was made easier by the discovery that these functions have not changed for any of the successors of the Skylake microarchitecture (e.g., Kaby Lake, Coffee Lake, Coffee Lake Refresh).

sided pair. As mentioned before, huge pages give us 2 MB of contiguous physical memory and by massaging the buddy allocator we may be able to obtain 4 MB. We therefore conclude that in most cases, huge pages will suffice. They fall short only for large configurations such as systems using more than two DRAM channels (not included in Table 1). In Section 9 we discuss the advantages and limitations, with respect to exploitation, of using either huge pages or buddy allocator massaging to acquire contiguous physical memory.

6 Self-Evicting Rowhammer

The effectiveness of Rowhammer heavily depends on the optimality of the aggressors' activation rate (i.e., number of activations within a fixed time interval) [4, 8, 17]. As we have explained in Section 4, the eviction-based Rowhammer techniques described in prior works [4, 15], while effective on DDR3 systems, do not generate enough memory accesses to trigger bit flips on DDR4 systems where long many-sided Rowhammer patterns are required. This calls for newer eviction strategies that maximize hammering throughput.

Aweke et al.'s method [4] for eviction-based Rowhammer introduces only one cache miss per aggressor by exploiting the LRU-like replacement policy of the LLC [3]. This translates to two extra accesses to DRAM in the case of double-sided Rowhammer. However, each additional aggressor will introduce another extra access and therefore the approach does not scale to many-sided patterns.

In Section 5 we explained that the location of the dummy rows (i.e., rows used to distract the TRR mitigation) does not matter. In this section we will show that it is possible to eliminate all extra DRAM accesses by using the dummy rows for eviction too. We first explain our strategy for selecting these dummy rows in Section 6.1. We then discuss how we create access patterns that handle the replacement policy of the LLC in Section 6.2. Finally, in Section 6.3 we show how we can make these patterns faster using parallelization.

6.1 Selecting double-sided pairs

The goal of our self-evicting Rowhammer patterns is to employ the dummies to bypass the in-DRAM TRR sampler while also evicting the caches. To ease the discussion we first define the terminology we will use throughout the remainder of the paper.

Terminology. As mentioned before, a double-sided pair is a pair of addresses that map to two rows (i.e., the aggressors) surrounding a third row (i.e., the victim) in the same bank. Let (a,b) denote a double-sided pair with virtual addresses a and b. A virtual address d is a dummy of (a,b) if it co-located in the same bank as a, and therefore also with b, and not equal to either. With these definitions established, an N-assisted double-sided pattern, in access order, looks as follows:

$$a,b \quad d_0 \ldots d_{N-1} \tag{1}$$

with d_i denoting the $(i+1)$th dummy of (a,b). After d_{N-1} we again access a. We will use the term aggressor to refer to either a, b, or a dummy address. Furthermore, we use A and B to refer to the cache sets of respectively a and b.

We are now able to specify our intentions more accurately. We want to employ the dummies d_i for the eviction of a and b from the CPU caches. In order to do so, we split them into two groups of equal size as follows: dummies d_{2k} map to A while dummies d_{2k+1} map to B, with k an integer from 0 to $N/2$. We are basically creating a zebra-like pattern in which every other address maps to the same set.

Building eviction sets. In order to achieve our goal of self-eviction we need to make sure the dummy addresses are not only co-located with (a,b) in the same bank but are also *congruent* with a or b, i.e., they map to the same cache set. Unfortunately an attacker capable of allocating 2 MB of contiguous physical memory does not control higher order physical address bits (i.e., the bits above bit 20) used to index the CPU cache slices (recall Figure 1). We solve this problem with the help of a page coloring algorithm that allows us to discover the seemingly unreachable high order slice bits, similar to Liu et al. [25].

Huge page coloring. Consider an attacker with a set of 2 MB huge pages at their disposal. The color of a huge page, then, is given by the result of the slice hash function applied only to the slice bits above the huge page offset. Since the attacker already controls the slice bits within the page offset, with known page colors the attacker has full control of slice indices.

To reveal a huge page's color the attacker exploits a side channel based on cache eviction. Suppose, by way of illustration, that the associativity of the LLC is $W = 1$. We are given two huge pages P and Q and would like to know whether their colors are equal or different. To find out we choose an arbitrary page offset f and create two addresses p and q, one from each page but both at page offset f to make sure that their set indices and slice bits within the page are equal. We then access p, followed by q, and again by p. If our second access to p causes a (slow) cache miss, then the hash of the high order slice bits or equivalently page colors of P and Q are equal, otherwise they are different.

In practice the LLC's associativity is larger than one, say $W = 16$ and the number of page colors (i.e., slices) is eight on modern quad-core CPUs. As a result brute forcing all possible permutations to find $W + 1 = 17$ same-color pages quickly bloats. Fortunately, there is a faster way: given the slice hash functions on Intel processors [10, 26, 46], each huge page contains precisely four cache lines that are congruent

(i.e., they have equal set index and slice bits). As a result, we only need to search for five huge pages of an identical color assuming a 16-way LLC (i.e., because $4 \cdot 5 > 16$).

Given this property of huge pages, our aim is to color each huge page based on how it shares cache slices with other huge pages. If two huge pages have the same color, they map similarly to the LLC. Our coloring algorithm works as follows. We take five random huge pages and extract from each four congruent addresses (i.e., in total 20 addresses). Using the eviction-based side channel, we test for equal page colors. If the page colors are the same, repeatedly accessing these 20 addresses will take long due to evictions. If the page colors differ, then repeatedly accessing these addresses execute quickly. In that case, we change these 20 addresses by permuting the slice bits under our control (i.e., within the huge pages). Assuming eight different slices, as can be found modern quad-core CPUs, we have eight possible colors for each huge page. In other words, we are searching for eight "valid" (i.e., five pages of the same color) permutations among $8^5 = 32768$.

Coloring more huge pages After we have identified five pages of the same color, we can quickly reveal the color of any other page as follows. We remove one of the huge pages with a known color from the access sequence by removing its four associated addresses, and replacing them with four addresses from the new page with an unknown color. We then proceed as before, but this time only permute the new page, not changing the other addresses. As soon as the four new addresses are congruent with the other already congruent addresses we observe eviction and are able to deduce the new page's color.

In order to distinguish between a cache miss and hit, we need to address the limited resolution of timers in modern browsers [19, 34, 40]. We do so by amplifying our measurements: to "test" a permutation, we repeatedly, say, a 1000 times, perform the associated sequence of 20 accesses and measure the total time this takes.

Address selection. Using the page coloring algorithm the attacker can reveal the page color of 512 huge pages (i.e., 1 GB) in seconds. With the colors known, we can start creating N-assisted double-sided Rowhammer patterns as described in Equation 2.

We first select the double-sided pair (a,b). To find a, the attacker chooses an arbitrary offset within one of the known color huge pages. Then, to find b, we add two to (or subtract two from) the row address of a. We also change a few additional bits in b to make sure a and b still map to the same bank after the addition, for the actual bits used in our experiments see Table 6 in Appendix B. Next, we select dummy addresses at the same page offsets as (a,b) but from different huge pages of the same color. Using the same offsets on pages of the same color, we ensure that the dummies at the same offset as a map to A, and dummies at the offset of b map to B. In addition, the dummies will automatically be co-located in the same bank as (a,b).

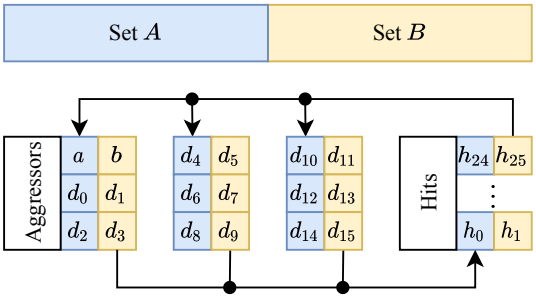

Figure 4: A self-evicting Rowhammer pattern. A self-evicting 16-assisted double-sided Rowhammer pattern with $W = 16$, $W' = 3$ and therefore $2(W - W') = 26$ hits. Each address maps to either set A or B. Aggressor i is evicted by aggressor $(i+6) \pmod{18}$. The arrows show the order of access.

6.2 Handling the replacement policy

We now have our double-sided pair (a,b) and the dummies. In principle, given that our dummies map to either A or B, the N-assisted double-sided pattern is self-evicting. In practice, however, the dummies will only evict each other or the double-sided pair if they do not all fit inside their cache sets. In particular, an N-assisted double sided pattern of length $L = N+2$ will only be self-evicting if $L/2 > W$ where W is the associativity of the LLC. This would severely limit SMASH to only very large numbers of N.

Introducing hits. We therefore have to introduce yet another kind of address, which we refer to as the *hits* (as in cache hits). The hits are addresses that ought to never leave the LLC and like the dummies, they either map to A or B. That is, h_{2k} is congruent to a and dummies d_{2k} while h_{2k+1} is congruent to b and dummies d_{2k+1} for some integer k. Hits are used to effectively reduce the LLC's perceived associativity to $W' < W$. With the hits, for example, a 6-assisted double-sided pattern (with four aggressors per set) can also be self-evicting even if the LLC's associativity is larger than four, which is the case in practice.

With hits, such a 6-assisted double-sided pattern may look as follows:

$$\begin{array}{lll} a,b & d_0,d_1 & h_0 \ldots h_{2(W-2)-1} \\ d_2,d_3 & d_4,d_5 & h_0 \ldots h_{2(W-2)-1} \end{array} \quad (2)$$

Each line consists of exactly $2W$ accesses that fill up both A and B. The first four accesses on each line go to DRAM and evict the first four on the other line. In Equation 2, for example, d_2 evicts a in A, d_3 evicts b in B, d_4 evicts d_0 in

A again, etc. Figure 4 shows another example, a 16-assisted double-sided pattern with $W' = 3$.

With the introduction of hits we also introduce a new parameter, namely how many of them we introduce in our patterns. At a minimum, we need to make sure the pattern does not fit in A and B (otherwise there will be no evictions) and therefore have to add at least $W - L/2$ to a pattern of length L. Second, it does not make sense to introduce more than $2(W-1)$ as with more there is no space left in A and B for aggressors. Third, please note that if W' does not divide $L/2$ (which it does in Equation 2, where $W' = 2$ and $L/2 = 4$), the order of access becomes a bit more complicated, for example if $W' = 3$ we get

$$\begin{array}{cccc} a,b & d_0,d_1 & d_2,d_3 & h_0 \dots h_{2(W-3)-1} \\ d_4,d_5 & a,b & d_0,d_1 & h_0 \dots h_{2(W-3)-1} \\ d_2,d_3 & d_4,d_5 & a,b & h_0 \dots h_{2(W-3)-1} \end{array} \quad (3)$$

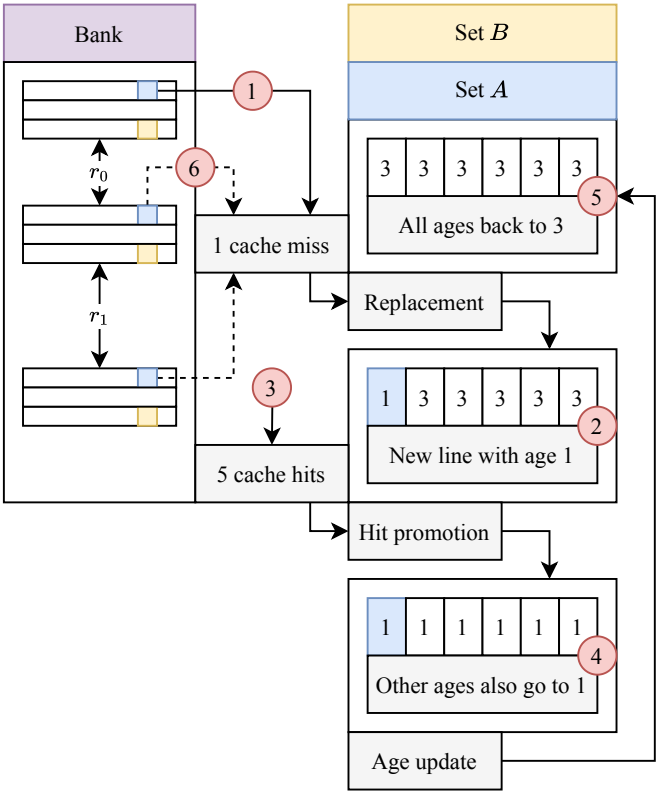

Figure 5: The evolution of a 6-way cache under self-eviction. Displayed on the left are three pairs of aggressors (incl. dummies) at random distances r_0 and r_1 from each other. The blue upper half of each pair maps to set A, shown on the right, the yellow lower half to B, which is not shown in its entirety but which evolves equally. Each of the six steps shown is explained in the text.

Handling QLRU. Until now we have implicitly assumed an LRU-like replacement policy. We now demonstrate how we can relax this assumption to create patterns that self-evict with Quad-age LRU (QLRU), the actual replacement policy used by the LLC of modern Intel CPUs. Figure 5 shows the evolution of cache set A under self-eviction, illustrating the QLRU behavior[3] under the reduced associativity $W' = 1$, which means after accessing one aggressor per set we immediately move on to the hits. We will start at ① and end at ⑤, with ⑥ denoting the start of the next round.

① First, we access a, which is brought into set A and replaces the oldest and leftmost cache line. ② As all lines have age three at this point (the oldest possible age) a ends up in the leftmost slot. Since a is new, its age becomes one. ③ We continue by causing five cache hits, accessing each of the other cache lines currently in the cache but whose age is three. ④ Accessing them makes their ages go back to one. ⑤ Finally, because all ages are now one, the replacement policy says that all of them should become three, which is done through the age update. ⑥ We are now back at the beginning, with a in the LRU position, ready to be evicted upon the attacker accessing the first dummy mapping to A, namely d_0.

6.3 Double pointer chase

Finally, to make our patterns even faster we use a double pointer chase to perform the accesses as opposed to the more common single pointer chase. In a single (register) pointer chase, the memory location pointed to by an aggressor provides the address of the next aggressor (or sometimes a hit, as in our case). This approach, however, does not maximize the memory throughput since every second memory access needs to wait until the first has completed, reducing parallelism at the memory controller level.

For this reason, instead of using one register, we have used two. We naturally split the pattern in two halves, with the addresses in each half mapping to either A or B. When then chain both halves in two single pointer chases that we intertwine. The result is something like

```
mov rax, (rax)
mov rbx, (rbx)
mov rax, (rax)
mov rbx, (rbx)
```

where each instruction loads from memory, not stores to. Our experiments showed that the double pointer chase improves memory throughput by 80 % compared to using a single pointer chase.

[3] To be precise, the replacement policy shown in Figure 5 is QLRU_H11_M1_R0_U0, which is the policy employed by the LLC of the Intel i7-7700K CPU used for our experiments. See [3] for more details.

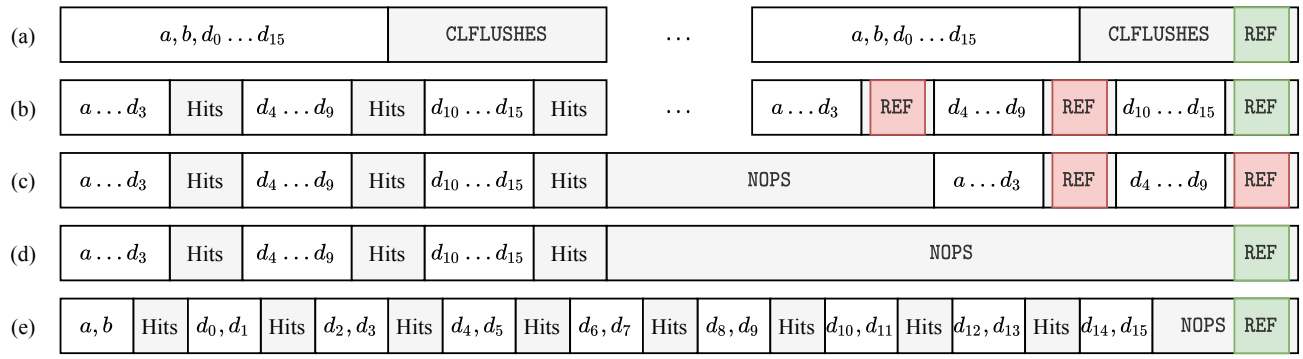

Figure 6: Five different variants of 18-sided Rowhammer: the baseline pattern that uses `CLFLUSH` (a), a naive self-evicting pattern (b) and a desynchronized self-evicting pattern with `NOP`s (c) both of which are detected by the sampler, a strongly synchronized self-evicting pattern that does trigger bit flips (d), and finally a weakly synchronized self-evicting pattern that also produces bit flips (e).

7 Synchronized Rowhammer

The self-evicting pattern presented in Section 6 is not directly able to trigger bit flips, not even when the double-sided pair encloses a known-to-be vulnerable row. This implies that many-sided Rowhammer alone is not always enough to bypass TRR. To investigate this, we have to understand the difference between the "gather" `CLFLUSH`-based pattern of Figure 2 (i.e., the baseline pattern) that can trigger bit flips and the self-evicting pattern that cannot.

In this section we clarify how an attacker may craft a Rowhammer-inducing, synchronized self-evicting pattern taking $\mathcal{S}_0$ as case study. One system configuration is usually vulnerable to several many-sided patterns. This fact simplifies the final implementation because the attacker can select the pattern with the optimal number of aggressor rows in order to improve the eviction process. Hence, we chose to employ the 18-sided (or rather, 16-assisted double-sided) pattern instead of the original 19-sided reported in [12]. As shown in Figure 6-a, the baseline pattern iterates over the aggressors, issuing memory requests in succession and it then flushes these addresses with the x86_64 `CLFLUSH` instruction. Our test machine is equipped with an Intel i7-7700K CPU with a 16-way set-associative LLC (i.e., $W = 16$). We chose to set the reduced associativity $W' = 3$. As an example, our self-evicting pattern looks as follows (the double-sided pair, eight pairs of dummies, and 26 hits, see Figure 6-b):

$$\begin{array}{cccc} a,b & d_0,d_1 & d_2,d_3 & h_0 \dots h_{26-1} \\ d_4,d_5 & d_6,d_7 & d_8,d_9 & h_0 \dots h_{26-1} \\ d_{10},d_{11} & d_{12},d_{13} & d_{14},d_{15} & h_0 \dots h_{26-1} \end{array} \quad (4)$$

Given that this pattern is about 30 % faster than the baseline pattern, it is surprising that it does not generate bit flips. We hence tried to slow this pattern down to make it execute at the same speed as the baseline pattern.

7.1 Self-eviction with hard synchronization

To investigate this phenomenon, we slow down the pattern through the addition of additional `NOP`s in front of (a,b). In this way, the activation interval increases. The outcome is shown in Figure 7 providing us with two important insights:

1. First, we are able to *synchronize our memory requests with the refresh commands sent to DRAM*. Exactly when t (i.e., the time to iterate over the pattern of Equation 4 once) or $2t$ divides `tREFI`, both patterns stop slowing down and the curve flattens despite the increasing number of `NOP`s.

2. Second, if the pattern is too fast, i.e., $\texttt{tREFI}/t > 5$, it does not trigger bit flips.

Frigo et al. [12] reports that in-DRAM TRR "acts on every refresh command" and that the sampler "can sample more than one aggressor per refresh interval". In all experiments, *we only found bit flips in victim rows adjacent to the first $n-S$ aggressors* where n is as always the total number of aggressors and S the suspected capacity of the sampler, in terms of the number of aggressors it keeps track of. We learned S simply by decreasing the number of aggressors until we were no longer able to reproduce a particular bit flip.

The memory controller needs to schedule refresh commands on average once every $\texttt{tREFI} = 7.8\,\mu s$. Modern memory controllers try to improve performance by opportunistically sending a refresh command when there is no DRAM activity. To successfully trigger Rowhammer bit flips, the pattern needs to repeat for tens of thousands of times during which many refresh commands have to be issued by the memory controller. When there are no `NOP`s (i.e., pattern in Figure 6-b), the memory controller will try to schedule a refresh command during one of the regions with many cache hits. This means that the TRR mechanism will be able to successfully sample and refresh each of the 18 aggressor rows

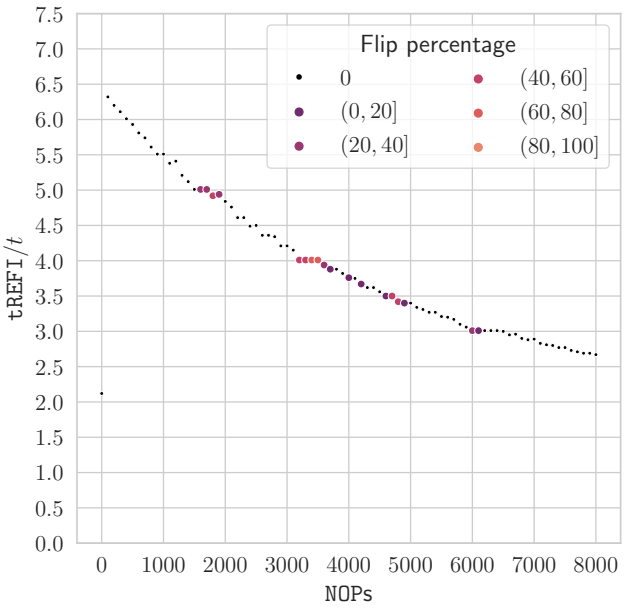

Figure 7: The self-evicting pattern using a double pointer chase and structured as in Figure 4 and Equation 4. The vertical axis reports the number of times we could fit our pattern (taking t ns) inside a refresh interval, while the horizontal axis reports the number of NOPs in front of our access pattern. We monitor the percentage of times we observed a of bit flip at a vulnerable memory location when repeating the experiment for 30 times.[4]

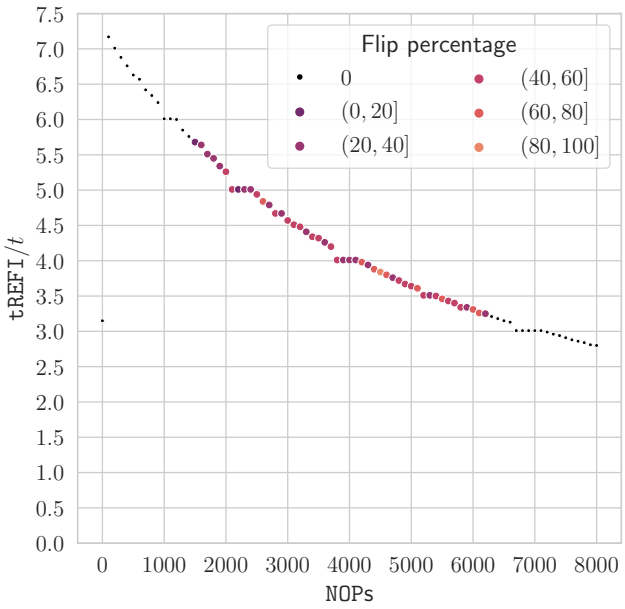

Figure 8: The self-evicting pattern with an increasing number of NOPs in front but arranged as in Equation 5 to minimize the inter-pattern gaps.

in the pattern of Equation 4 when the refresh command lands in the three different regions with cache hits.

When inserting NOPs in front of the pattern three different scenarios can happen as shown in Figure 7. In the first scenario, as shown in Figure 6-c, with a small number of NOPs, the memory controller may still choose to send the refresh command in the regions with cache hits, resulting in no bit flips. In the second scenario, a very large number of flips in front of each pattern would make the pattern too slow to trigger bit flips. In the third scenario, as shown in Figure 6-d, with the right number of NOPs, the pattern synchronizes with when the memory controller intends to send a refresh command. This results in the first aggressors (a,b) *escaping* the TRR mechanism to successfully hammer memory and trigger bit flips.

While the strategy of adding NOPs to the beginning of the self-evicting pattern is effective in triggering bit flips, it requires the attacker to very precisely synchronize with the refresh command and find out the correct number of NOPs for a successful attack. While this is a plausible strategy, as we will show in our evaluation, it is not always trivial to precisely synchronize with the refresh interval in JavaScript. Instead, we will describe another strategy for creating patterns that only softly synchronize with the memory controller's refresh command and lift the requirement of finding the exact number of NOPs for effective hammering.

7.2 Self-eviction with soft synchronization

To make sure that the memory controller does not sneak in a refresh at the moment that is not desired, we have to make sure that the regions with cache hits are sufficiently small. To this end, we slightly modify our self-evicting access pattern from Section 6 to more evenly distribute our cache hits between our cache misses, creating the following self-evicting pattern:

$$\begin{array}{lllll} a,b & h_0 \ldots h_7 & d_0,d_1 & h_8 \ldots h_{17} & d_2,d_3 & h_{18} \ldots h_{25} \\ d_4,d_5 & h_0 \ldots h_7 & d_6,d_7 & h_8 \ldots h_{17} & d_8,d_9 & h_{18} \ldots h_{25} \\ d_{10},d_{11} & h_0 \ldots h_7 & d_{12},d_{13} & h_8 \ldots h_{17} & d_{14},d_{15} & h_{18} \ldots h_{25} \end{array} \quad (5)$$

Figure 8 shows the results of executing this pattern with a variable number of NOPs in front. As shown in Figure 6-e, given that the few cache hits do not provide a large-enough window for the memory controller to schedule refreshes, it opportunistically uses the single available NOP gap instead for scheduling refresh commands. The pattern of Equation 5 can hence synchronize with the refresh command more softly without requiring a precise number of NOPs. This makes the

[4] Note that with a small number of NOPs we observe a massive slowdown. The processor's performance counters indicate that, in this case, many more LLC misses occur than we expect. We verified that these additional misses are not caused by the L1 and L2 prefetchers. We hence think that this effect is due to a different behavior of the cache replacement policy triggered by a high access density in the absence of the delay caused by many NOPs.

Table 2: The test beds used in our evaluation.

System	CPU	Organization			
		Ch.	DIMMs/Ch.	Ranks	Banks
$\mathcal{S}_0$	i7-7700K	1	1	1	16
$\mathcal{S}_1$	i7-7700K	2	1	2	16
$\mathcal{S}_2$	i7-7700K	2	1	2	16

search for bit flips much more convenient from JavaScript as we show in the next section.

Adjusting for cache slices. During these experiments we realized that accesses to different slices may take a variable amount of time. This makes it harder to generate synchronizing patterns where A and B, the two sets to which the addresses map, reside in different slices. Therefore, by adjusting each address' column bits, we make sure A and B map to the same slice (see Table 6 in Appendix B).

8 Evaluation

The previous section shows that we can successfully generate self-evicting hammering patterns that are able to bypass TRR mechanisms using a soft synchronization technique with refresh operations. In Section 6, we also showed that the ability to properly select aggressor addresses depends on the virtual-to-DRAM addressing functions and how to select addresses that map into a given cache set and slice.

In this section, we evaluate the constraints under which the attacker can successfully create effective self-evicting patterns. We evaluate the feasibility of constructing self-evicting patterns on three setups with different memory configurations and memory modules from two of the major memory vendors (see Table 2). All systems feature an Intel Core i7-7700K CPU which employs a Kaby Lake microarchitecture. Since Skylake, Kaby Lake, Coffee Lake (R) microarchitectures all use the same DRAM addressing function for a given memory configuration, as we discovered in Section 5, we focus on the feasibility of constructing self-evicting synchronizing patterns on different memory configurations without lack of generality in terms of the CPU's microarchitecture.

8.1 Practicality of self-evicting patterns

Whether it is possible to extract self-evicting patterns from huge pages only, depends on the DRAM addressing functions employed by the memory controller. To a lesser degree, it also depends on the complex addressing scheme used for slice addressing, which has not changed since Sandy Bridge, apart from the fact that as of Skylake the number of slices equals the number of hyperthreads instead of cores [3, 7].

Table 3: The self-evicting patterns for our three tested setups (Table 2).

System	Best TRRespass	Self-evicting pattern ($W = 16$)			
		L^*	W'	Hits	Total length incl. hits
$\mathcal{S}_0$	19-sided	18	3	26	96
$\mathcal{S}_1$	10-sided	10	3	26	160
$\mathcal{S}_2$	3-sided	4	1	30	64

* A pattern's length $L = N+2$ where N is the number of dummies in the corresponding N-assisted double-sided pattern.

Table 4: The table shows, for each test bed, whether we were able to produce bit flips natively in C and in JavaScript, and if so, with how many `NOP`s and `XOR`s, respectively, using the patterns in Table 3.

System	Native flips	`NOP`s	JavaScript flips	`XOR`s
$\mathcal{S}_0$	✓	1500-6200	✓	300-900
$\mathcal{S}_1$	✓	100-1900, 3500-3700	✓	0-400, 700*
$\mathcal{S}_2$	✓	100	✗	—

* We also found a tiny number of bit flips with respectively 500 and 600 `XOR`s (i.e., less than 1 % of all flips triggered by this pattern).

We have reverse engineered the physical-to-DRAM addressing functions for several memory configurations using the software-based method of DRAMA [30]. Table 5 in Appendix A shows the results of our reverse engineering and Table 6 in Appendix B the bits that need to change for obtaining double-sided pairs in the form of (a, b). Assuming THP is enabled, in six out of the seven possible configurations we could successfully find double-sided pairs inside a huge page. In the remaining case, the row bits start after the huge page boundary so we cannot find a b that is two rows apart from a.

8.2 Ability to produce bit flips

We first used the open-source TRRespass fuzzer [12] to find the most effective many-sided access pattern for each of our three test beds. We then made each pattern self-evicting and synchronized according to the strategies described in Sections 6 and 7. Table 3 summarizes some distinct properties of the resulting patterns (e.g., reduced associativity W').

Table 4 shows that all three self-evicting patterns were able to trigger bit flips using our native C implementation. At the same time, we observe a clear difference in the number of `NOP`s required for different systems. In particular, the "effective `NOP` range" is more narrow on systems $\mathcal{S}_1$ and $\mathcal{S}_2$ compared to system $\mathcal{S}_0$. We suspect that this is caused by the relative slowness (compared to the flushing patterns) of the self-evicting patterns that we built for $\mathcal{S}_1$ and $\mathcal{S}_2$. As shown in Table 3, the TRRespass patterns for $\mathcal{S}_1$ and $\mathcal{S}_2$ are smaller

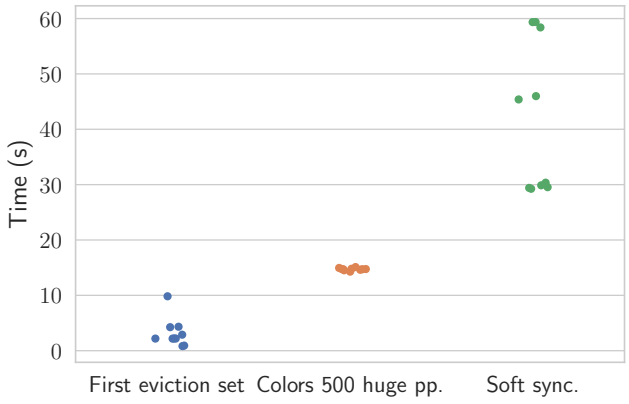

Figure 9: The time spent on each of the different parts of the initialization. Measurement are repeated $n = 10$ times. Most time is spent on synchronization.

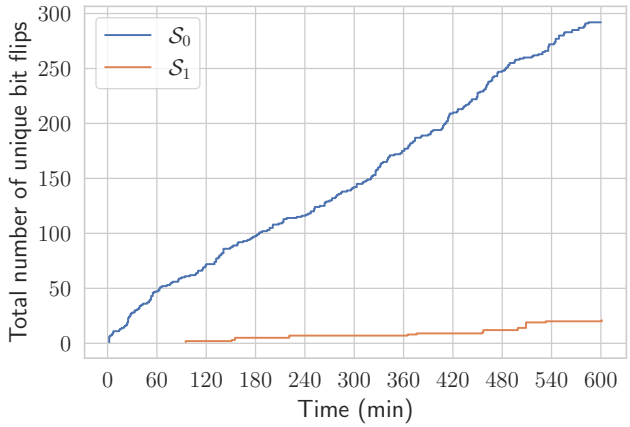

Figure 10: The cumulative number of unique bit flips on respectively systems $\mathcal{S}_0$ and $\mathcal{S}_1$ during a single run of 10 hours, using our implementation in JavaScript and the patterns of Table 3. The horizontal axis shows time passed in minutes, excluding the one minute initialization. In Section 9 we consider time until first *exploitable* bit flip.

(i.e., fewer dummies necessary) compared to $\mathcal{S}_0$ which means the introduction of hits will have a larger relative effect on the time in between activations of (a,b). As a consequence, using too many `NOP`s make these patterns too slow to trigger bit flips.

Our implementation in JavaScript is able to trigger bit flips on systems $\mathcal{S}_0$ and $\mathcal{S}_1$. We observed that the occurrence of bit flips on $\mathcal{S}_1$ is less frequent compared to $\mathcal{S}_0$. Although we observe this too using our native implementation, it is reasonable to expect that the difference is exaggerated by the more stringent synchronization requirements (i.e., the smaller `NOP` ranges) for systems $\mathcal{S}_1$ and $\mathcal{S}_2$, compared to $\mathcal{S}_0$. Since `NOP`s are not available in JavaScript, our implementation uses `XOR`s instead. Both instructions are cheap, yet the `XOR`-loop in JavaScript has more overhead and therefore introduces a delay with coarser granularity. This makes it harder to target the sweet spot of systems $\mathcal{S}_1$ and $\mathcal{S}_2$.

8.3 JavaScript implementation benchmarks

We now evaluate the performance of our JavaScript implementation, running on the latest version of Mozilla's JavaScript runtime SpiderMonkey. In particular, we consider the program's initialization phase (e.g. time spent on detecting page colors) and hammering phase.

Initialization. The attacker starts by running the slice-coloring algorithm to reveal the page color of 500 huge pages backing their `ArrayBuffer`. Next, the attacker uses a subset (the size of which depends on the pattern's length) of huge pages to assemble the first self-evicting pattern. Finally, the attacker needs to take care of synchronization and does so by varying the number of `XOR`s in front of the pattern.

Figure 9 reports the time spent on each of these steps: "First eviction set" and "Colors 500 huge pp." together report the time required by the slice-coloring algorithm to find five huge pages of the same color and subsequently using them to reveal the color of 500 other pages, respectively. Lastly, "Soft sync." reports the time spent on finding the right number of `XOR`s for soft synchronization. Each measurement has been repeated 10 times. On average, it takes an attacker one minute to complete the initialization. Note that the soft synchronization step takes the longest. In our implementation, we use amplified time measurements to estimate how many times the pattern fits inside the refresh interval `tREFI` of 7.8 µs before using it for hammering. If the pattern fits four times, then it is a good candidate for hammering as shown in Figures 7 and 8.

Hammer time. With the initialization complete, the attacker starts hammering in search of an exploitable bit flip. To hammer different rows, the attacker simply changes the subset of huge pages used for pattern assembly. Figure 10 shows the cumulative number of unique bit flips over time during a single 10 hour experiment on $\mathcal{S}_0$ and $\mathcal{S}_1$. Section 9 shows how we can use these bit flips to compromise SpiderMonkey.

8.4 Discussion

To perform SMASH successfully, the attacker needs to be aware of the victim's memory configuration. In particular, without knowing the DRAM addressing functions and at least one n-sided pattern that bypasses TRR, it is not possible to build a self-evicting pattern. While fingerprinting is possible to detect a specific system, the attacker can also try different configurations until one is successful.

9 Exploitation

After harvesting all the primitives to re-enable Rowhammer from the browser on modern DDR4 systems, we can now use Rowhammer bit flips to build an exploit. For our proof-of-concept exploit, we target the latest version of the Firefox browser at time of writing (v. 81.0.1) running on Ubuntu 18.04 with the latest updates and Linux kernel 4.15.0-111-generic installed. Our exploitation techniques mimic the ones of the original GLitch exploit [11]. However, GLitch takes advantage of WebGL and the GPU to exploit the browser on ARMv7 (32-bit) systems. As a consequence, we cannot rely on the same GPU-triggered bit flips, vulnerable templates, or memory massaging techniques.

9.1 Memory massaging

In Section 5 we discussed how an attacker can obtain physically-contiguous memory in the browser using THP or by massaging power-of-two allocators (e.g., Linux's buddy allocator [2]). We now discuss these techniques and how we use them for our exploit.

THP. Transparent Huge Pages (THP) need to be enabled in the operating system as the Firefox browser does not explicitly request the use of huge pages when performing large allocations. However, it does carry out MB-aligned allocations for large objects e.g. when allocating a large `ArrayBuffer`. As a result, with THP enabled the operating system will transparently back these objects with huge pages, which the attacker can use to template memory for vulnerable locations. Unfortunately, THP are hard to split for exploitation which means that we can only trigger bit flips on memory we already control. This means that we still need to massage the operating system's allocator to land a 4 kB page on a vulnerable template after releasing the huge page back to the operating system.

Massaging buddy. In order to release the huge pages backing our vulnerable `ArrayBuffer`s to the operating system we need to force the browser to `munmap` the associated mapping. To do so we trick SpiderMonkey, the browser's JavaScript runtime, into releasing all the references to it by transferring the `ArrayBuffer` to a Web Worker and then killing the Web Worker. Finally, we massage the buddy allocator in Linux [2] into providing us with the same 4 kB page frames that previously formed a huge page.

The buddy allocator used by Linux distributions tries to first serve applications with all the available 4 kB pages before fragmenting larger memory blocks. This means that in order to split the 2 MB page containing a vulnerable template we first need to exhaust all the smaller power-of-two contiguous allocations i.e., 4 kB, 8 kB, 16 kB, etc. Depending on the amount of memory available to the system, this approach can reach near-out-of-memory situations which may cause the operating system to abort the application. In our experiments we always managed to get the huge page split and reused by smaller objects before the operating system would kill the application.

Once Firefox reuses the vulnerable 2 MB page, we can now identify which 4 kB objects are backed by this contiguous chunk of memory by exploiting a timing side channel on the self-evicting hammering pattern discussed in Section 6. This technique is similar to what was described for the page coloring algorithm. Indeed, if the pages being used are 2 MB-aligned, the self-evicting pattern will again reach DRAM with every "alleged" cache miss and also cause a bank conflict. But when this is not the case, memory accesses will generate row hits at best but most likely cache hits since the addresses will map to different cache sets.

9.2 Vulnerable templates

The GLitch exploit relies on a technique known as type flipping. This exploits the fact that modern browsers [1,5] encode pointers in "invalid" double-precision floating point numbers (i.e., NaN values). These NaN values defined in the IEEE 754 double-precision floating point encoding cannot store any useful information for mathematical computations. The Firefox browser uses some of these $2^{53} - 1$ unused values to store pointers in it. The type flipping technique exploits this "abuse" of the NaN value to turn pointers into numbers and vice versa. In other words we can break ASLR and craft arbitrary pointers by simply triggering Rowhammer bit flips on values stored inside JavaScript `Array`s. The outcome of the operation depends on the direction of the bit flips (i.e., if it is a 1-to-0 or 0-to-1 bit flip).

The exploit chain. The end-to-end exploit gives the attacker an arbitrary read/write primitive inside the browser which can then be escalated to remote code execution using different strategies [13, 39]. We first allocate a small (inlined) `ArrayBuffer`. These objects store metadata and data consecutively in memory and allow byte-granular memory reads. This property makes them a perfect target for browser exploitation. Then the exploit chain unfolds in three stages.

1. We store the pointer to this `ArrayBuffer` in a vulnerable `Array` cell and then trigger a 1-to-0 bit flip on this pointer in order to derandomize the location of the object (and consequently its data).

2. With knowledge of the location of this buffer, we now need to leak its header's metadata in order to craft a counterfeit object that we fully control. To leak the header's metadata we rely on a fake `JSString`. `JSString`s are easy to craft since they simply contain the pointer to the data and some constant metadata. Unfortunately, they are immutable which means that they provide us only

with an arbitrary read primitive. We use the pointer we leaked in the previous step to craft the fake `JSString` and leak the `ArrayBuffer`'s metadata.

3. Finally, we craft a fake `ArrayBuffer` using the metadata leaked in the previous step and reuse the 0-to-1 bit flip to create a reference to it. These nested `ArrayBuffers` allow us to overwrite the pointer of the inner buffer from the outer one. This provides us with the arbitrary read/write primitive we were seeking.

Firefox 64-bit. Firefox implements two different NaN-boxing techniques for 32 and 64-bit architecture. The abstract design is similar, but the kinds of bit flips that can be used for exploitation differs. If the value stored in the `Array` is greater than a special tag value ($\texttt{0xfff80000} \ll 32$ on 64-bit) the value is stripped of the type-casting metadata and then used as a canonical pointer. Due to the larger pointer size on 64-bit systems, the number of exploitable bit flips is reduced compared to 32-bit systems. Frigo et al. [11] report 25 out of 64 bits to be exploitable on every `Array` entry. In our case, we can exploit only 15 out of 64 bits, making the exploitation more cumbersome.

Evaluation. We evaluated our exploit on test bed $\mathcal{S}_0$. The exploit chain runs in a matter of seconds. Finding a first exploitable bit flip, however, takes more time. We run our experiments 40 times looking for an exploitable 1-to-0 and 0-to-1 bit flip. The median times to first exploitable bit flip (after initialization) are 703.5 s (about 13 min) and 857 s (about 16 min) respectively for 1-to-0 and 0-to-1 bit flips. However, these values have a large variance. In fact, in some of our runs we could detect exploitable bit flips in as little as 6 s for 1-to-0 and 44 s for 0-to-1 flips.

10 Mitigations

We briefly discuss three possible directions for mitigating Rowhammer in general and SMASH in particular.

Mitigating Rowhammer in hardware. Rowhammer is a vulnerability in DRAM hardware and it is sensible to expect that it should be fixed in hardware. Unfortunately it will take many years for newer and more effective mitigations to reach end users. Furthermore, given that future DRAM devices will feature even smaller transistors, it remains to be seen whether it is possible to build effective mitigations for such devices. Nevertheless, there are three directions in which the security of future DRAM devices can be improved: first, hardware manufacturers can build more precise samplers at a higher cost, either inside the DRAM device or at the memory controller. Second, more aggressive error correction than existing solutions [9] can be deployed to reduce the probability of triggering bit flips. Third, the number of potential activations can be limited depending on the access patterns and the vulnerability of a given DRAM device. All three directions come with either performance or storage overhead, not to mention an additional power consumption.

Mitigating Rowhammer in software. There have been many proposals for mitigating Rowhammer in software while hardware mitigations become available. There are however issues with their security, compatibility or performance. CATT [6] proposes to protect kernel memory from getting hammered by user memory using a guard page. Unfortunately kernel memory may directly be exposed to user memory through common mechanisms such as the page cache, leaving the system exposed [14]. ALIS [38] and GuardION [42] try to protect the rest of the system against memory regions that may be hammered, but these solutions require changes to each software and furthermore do not protect the rest of the system against attacks. ZebRAM [20] tries to partition a VM's memory into safe and unsafe regions using odd and even rows. The safe region can be directly accessed by the VM, while the unsafe region is used as an ECC-protected swap cache. ZebRAM's design, while secure against known attacks, has non-trivial performance overhead with memory-intensive workloads.

Mitigating SMASH. We now discuss more pragmatic mitigations that make it harder to exploit browsers with SMASH without addressing the underlying Rowhammer vulnerability. The current version of SMASH relies on THP for the construction of efficient self-evicting patterns. Disabling THP, while introducing some performance overhead, would stop the current instance of SMASH. Furthermore, our exploit relies specifically on corrupting pointers in the browser to break ASLR and pivot to a counterfeit object. Protecting the integrity of pointers in software or in hardware (e.g., using PAC [23]) would stop the current SMASH exploit.

11 Conclusion

We showed that Internet users are still affected by the Rowhammer vulnerability in modern DDR4 devices. These devices require many-sided Rowhammer patterns for bypassing their TRR mitigation. Efficiently executing such patterns in JavaScript without access to cache flushing instructions and contigous physical memory is specially challenging. We discovered a new property of the TRR mitigation that in combination with a careful selection of hammering addresses allowed us to create efficient many-sided Rowhammer patterns in the browser. Triggering bit flips in JavaScript, however, required us to go one step further and carefully schedule cache accesses with respect to the refresh commands issued by the CPU's memory controller. Our end-to-end exploit, called SMASH,

can fully compromise the Firefox browser with all the mitigations enabled in 15 minutes on average. We discussed future directions for mitigating Rowhammer attacks in general and SMASH in particular.

Acknowledgements

We thank our shepherd Vasileios Kemerlis and the anonymous reviewers for their valuable feedback. This work was supported by the European Union's Horizon 2020 research and innovation programme under grant agreements No. 786669 (ReAct) and No. 825377 (UNICORE), by Intel Corporation through the Side Channel Vulnerability ISRA, by the Netherlands Organisation for Scientific Research through grants NWO 639.021.753 VENI "PantaRhei", and NWO 016.Veni.192.262. This paper reflects only the authors' view. The funding agencies are not responsible for any use that may be made of the information it contains.

References

[1] Value.h. https://searchfox.org/mozilla-central/source/js/public/Value.h (commit 9c72508f) (visited on 2021-02-09).

[2] Physical Page Allocation, 2019. https://web.archive.org/web/20190306040105/https://www.kernel.org/doc/gorman/html/understand/understand009.html (visited on 2021-02-09).

[3] Andreas Abel and Jan Reineke. nanoBench: A Low-Overhead Tool for Running Microbenchmarks on x86 Systems. In *ISPASS '20*. IEEE.

[4] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd M. Austin. ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks. In *ASPLOS '16*. ACM.

[5] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *S&P '16*. IEEE.

[6] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. CAn't Touch This: Software-only Mitigation against Rowhammer Attacks targeting Kernel Memory. In *USENIX Security '17*. USENIX Association.

[7] Samira Briongos, Pedro Malagón, José Manuel Moya, and Thomas Eisenbarth. RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks. In *USENIX Security '20*. USENIX Association.

[8] Lucian Cojocar, Jeremie S. Kim, Minesh Patel, Lillian Tsai, Stefan Saroiu, Alec Wolman, and Onur Mutlu. Are We Susceptible to Rowhammer? An End-to-End Methodology for Cloud Providers. In *S&P '20*. IEEE.

[9] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks. In *S&P '19*. IEEE.

[10] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr, and Dejan Kostic. Make the Most out of Last Level Cache in Intel Processors. In *EuroSys '19*. ACM.

[11] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In *S&P '18*. IEEE.

[12] Pietro Frigo, Emanuele Vannacci, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TRRespass: Exploiting the Many Sides of Target Row Refresh. In *S&P '20*. IEEE.

[13] Samuel Groß. Exploiting a Cross-mmap Overflow in Firefox, 2017. https://web.archive.org/web/20200915100228/https://saelo.github.io/posts/firefox-script-loader-overflow.html (visited on 2021-02-09).

[14] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another Flip in the Wall of Rowhammer Defenses. In *S&P '18*. IEEE.

[15] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *DIMVA '16*. Springer.

[16] Sanghyun Hong, Pietro Frigo, Yigitcan Kaya, Cristiano Giuffrida, and Tudor Dumitras. Terminal Brain Damage: Exposing the Graceless Degradation in Deep Neural Networks Under Hardware Fault Attacks. In *USENIX Security '19*. USENIX Association.

[17] Jeremie S. Kim, Minesh Patel, Abdullah Giray Yaglikçi, Hasan Hassan, Roknoddin Azizi, Lois Orosa, and Onur Mutlu. Revisiting RowHammer: An Experimental Analysis of Modern DRAM Devices and Mitigation Techniques. In *ISCA '20*. IEEE.

[18] Yoongu Kim, Ross Daly, Jeremie S. 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. In *ISCA '14*. IEEE.

[19] David Kohlbrenner and Hovav Shacham. Trusted Browsers for Uncertain Times. In *USENIX Security '16*. USENIX Association.

[20] Radhesh Krishnan Konoth, Marco Oliverio, Andrei Tatar, Dennis Andriesse, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. ZebRAM: Comprehensive and Compatible Software Protection Against Rowhammer Attacks. In *OSDI '18*. USENIX Association.

[21] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. RAMBleed: Reading Bits in Memory Without Accessing Them. In *S&P '20*. IEEE.

[22] Jung-Bae Lee. Green Memory Solution. *Samsung Electronics' Investor's Forum*, 2014.

[23] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. PAC it up: Towards Pointer Integrity using ARM Pointer Authentication. In *USENIX Security '19*. USENIX Association.

[24] 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&P Workshops '20*. IEEE.

[25] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *S&P '15*. IEEE.

[26] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. In *RAID '15*. Springer.

[27] Ross McIlroy, Jaroslav Sevcík, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. *CoRR*, 2019.

[28] Micron. DDR4 SDRAM Datasheet. page 380, 2016.

[29] Microsoft Edge Team. Mitigating speculative execution side-channel attacks in Microsoft Edge and Internet Explorer, 2018.

[30] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security '16*. USENIX Association.

[31] Filip Pizlo. What Spectre and Meltdown Mean For WebKit, 2018. https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/ (visited on 2021-02-09).

[32] Damian Poddebniak, Juraj Somorovsky, Sebastian Schinzel, Manfred Lochter, and Paul Rösler. Attacking Deterministic Signature Schemes Using Fault Attacks. In *EuroS&P '18*. IEEE.

[33] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip Feng Shui: Hammering a Needle in the Software Stack. In *USENIX Security '16*. USENIX Association.

[34] Tom Ritter. Fuzzy Timers Changes, 2018. https://hg.mozilla.org/mozilla-central/rev/920270da576f (visited on 2021-02-09).

[35] Tom Ritter. Set Timer Resolution to 1ms with Jitter, 2018. https://bugzilla.mozilla.org/show_bug.cgi?id=1451790 (visited on 2021-02-09).

[36] Mark Seaborn and Thomas Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges. In *Black Hat USA*, 2015.

[37] Andrei Tatar, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Defeating Software Mitigations Against Rowhammer: A Surgical Precision Hammer. In *RAID '18*. Springer.

[38] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Throwhammer: Rowhammer Attacks over the Network and Defenses. In *USENIX ATC '18*. USENIX Association.

[39] argp. OR'LYEH? The Shadow over Firefox, 2016. http://www.phrack.org/issues/69/14.html (visited on 2021-02-09).

[40] The Chromium Projects. Mitigating Side-Channel Attacks, 2018. https://www.chromium.org/Home/chromium-security/ssca (visited on 2021-02-09).

[41] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *CCS '16*. ACM.

[42] 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*. Springer.

[43] Pepe Vila, Pierre Ganty, Marco Guarnieri, and Boris Köpf. CacheQuery: learning replacement policies from hardware caches. In *PLDI '20*. ACM.

[44] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation. In *USENIX Security '16*. USENIX Association.

[45] Thomas Yang and Xi-Wei Lin. Trap-Assisted DRAM Row Hammer Effect. *IEEE Electron Device Letters*, 2019.

[46] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. Mapping the Intel Last-Level Cache. *IACR Cryptol. ePrint Arch.*, 2015.

A DRAM Addressing Functions

Table 5: The DRAM addressing functions for different memory configurations on Intel Skylake, Kaby Lake, Coffee Lake, and Coffee Lake Refresh microarchitectures.

Organization				Addressing			Min. alloc.
Ch.	DIMMs/Ch.	Ranks	Banks	LSB row	Bank	Ch.	
2	2	2	16	20	7-14, 15-20, 16-21, 17-22, 18-23, 19-24	8-9-12-13-18-19	3.0 MB
2	1	2	16	19	7-14, 15-19, 16-20, 17-21, 18-22	8-9-12-13-15-18	1.5 MB
1	2	2	16	19	6-13, 14-19, 15-20, 16-21, 17-22, 18-23	—	1.5 MB
1	1	2	16	18	6-13, 14-18, 15-19, 16-20, 17-21	—	0.75 MB
2	1	1	16	18	7-14, 15-18, 16-19, 17-20	8-9-12-13-15-16	0.75 MB
1	2	1	16	18	6-13, 14-18, 15-19, 16-20, 17-21	—	0.75 MB
1	1	1	16	17	6-13, 14-17, 15-18, 16-19	—	0.38 MB

B Address Selection

Table 6: Start with an arbitrary huge page offset a. To find the offset b that together with a forms a double-sided pair mapping to different sets but the same slice, take a and change the bits given in the table. For example, on $\mathcal{S}_0$ we have $b = a \oplus (1 \ll 18) \oplus (1 \ll 15) \oplus (1 \ll 11) \oplus (1 \ll 10)$.

System	Row addition or subtraction	Same bank	Same slice
$\mathcal{S}_0$	18	15	11, 10
$\mathcal{S}_1$	20	16	13, 9
$\mathcal{S}_2$	20	16	13, 9

C Default THP Setting

Table 7: The default THP setting on popular Linux distributions. SMASH requires `always`.

Linux distribution	Version	Default `/sys/kernel/mm/transparent_hugepage/enable`
Ubuntu	Desktop 20.04 LTS	`madvise`
Fedora	33 Workstation	`madvise`
Linux Mint	20.1	`madvise`
Manjaro	20.2.1	`madvise`
Debian	10.7.0	`always`
CentOS	8	`always`
Kali Linux	2021 W02	`always`
openSUSE	Leap 15.2	`always`